ISSN 0105-8517

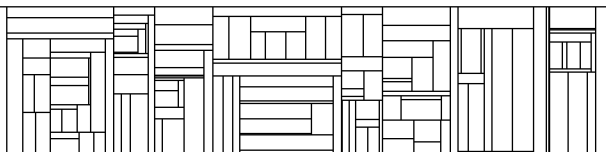
Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools

Aarhus, Denmark, October 24-26, 2005

Kurt Jensen (Ed.)

DAIMI PB - 576
October 2005

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF AARHUS
IT-parken, Aabogade 34
DK-8200 Aarhus N, Denmark



Preface

This booklet contains the proceedings of the Sixth Workshop on Practical Use of Coloured Petri Nets and the CPN Tools, October 24-26, 2005. The workshop is organised by the CPN group at Department of Computer Science, University of Aarhus, Denmark. The papers are also available in electronic form via the web pages: http://www.daimi.au.dk/CPnets/workshop05/

Coloured Petri Nets and the CPN Tools are now used by nearly 3000 users in 106 countries. The aim of the workshop is to bring together some of the users and in this way provide a forum for those who are interested in the practical use of Coloured Petri Nets and their tools.

The submitted papers were evaluated by a programme committee with the following members:

Wil van der Aalst, Netherlands
Jonathan Billington, Australia
Jörg Desel, Germany
Joao M. Fernandes, Portugal
Jorge de Figueiredo, Brazil
Nisse Husberg, Finland
Kurt Jensen, Denmark (chair)
Ekkart Kindler, Germany
Lars M. Kristensen, Denmark
Charles Lakos, Australia
Tadao Murata, USA
Daniel Moldt, Germany
Laure Petrucci, France
Karsten Schmidt, Germany
Robert Valette, France
Rüdiger Valk, Germany
Lee Wagenhals, USA
Jianli Xu, Finland
Wlodek Zuberek, Canada

The programme committee has accepted 16 papers for presentation. Most of these deal with different projects in which Coloured Petri Nets and their tools have been put to practical use – often in an industrial setting. The remaining papers deal with different extensions of tools and methodology.

The papers from the first five CPN Workshops can be found via the web pages: http://www.daimi.au.dk/CPnets/. After an additional round of reviewing and revision, some of the papers have also been published as a special section in the International Journal on Software Tools for Technology Transfer (STTT). For more information see: http://sttt.cs.uni-dortmund.de/

Kurt Jensen

Table of Contents

Guy Edward Gallasch, Somsak Vanit-Anunchai, Jonathan Billington, and Lars Michael Kristensen
Checking Language Inclusion On-The-Fly with the Sweep-line Method 1

M.H. Jansen-Vullers and H.A. Reijers
Business Process Redesign at a Mental Healthcare Institute: A Coloured Petri Net Approach 21

Nataliya Mulyar and Wil M.P. van der Aalst
Towards a Pattern Language for Colored Petri Nets 39

Lin Liu and Johathan Billington
Enhancing the CES Protocol and its Verification 59

B. Zouari and S. Zairi
Synthesis of Active Controller for Resources Allocation Systems 79

Irene Vanderfeesten, Wil van der Aalst, and Hajo A. Reijers
Modelling a product based workflow system in CPN tools 99

Michael Westergaard and Kristian Bisgaard Lassen
Building and Deploying Visualizations of Coloured Petri Net Models Using BRITNeY Animation and CPN Tools 119

Cong Yuan, Jonathan Billington, and Jörn Freiheit
An Abstract Model of Routing in Mobile Ad Hoc Networks 137

M. Pesic and W.M.P. van der Aalst
Modeling Work Distribution Mechanisms Using Colored Petri Nets 157

A.K. Alves de Medeiros and C.W. Günther
Process Mining: Using CPN Tools to Create Test Logs for Mining Algorithms 177

C. Lakos and L. Petrucci
Distributed and Modular State Space Exploration for Timed Petri Nets 191

Christian W. Günther and Wil M.P. van der Aalst
Modeling the Case Handling Principles with Colored Petri Nets 211

Ricardo J. Machado, Kristian Bisgaard Lassen, Sérgio Oliveira, Marco Couto, and Patrícia Pinto
Execution of UML Models with CPN Tools for Workflow Requirements Validation 231

Mariska Netjes, Wil M.P. van der Aalst, and Hajo A. Reijers
Analysis of resource-constrained processes with Colored Petri Nets 251

Panagiotis Katsaros, Vasilis Odontidis, and Maria Gousidou-Koutita
Colored Petri Net based model checking and failure analysis for E-commerce protocols 267

Katrin Winkelmann
Application of Coloured Petri Nets in Cooperative Provision of Industrial Services 285

Checking Language Inclusion On-The-Fly with the Sweep-line Method*

Guy Edward Gallasch[1], Somsak Vanit-Anunchai[1], Jonathan Billington[1], and Lars Michael Kristensen[2]**

[1] Computer Systems Engineering Centre
School of Electrical and Information Engineering
University of South Australia
Mawson Lakes Campus, SA 5095, AUSTRALIA
Email: guy.gallasch@postgrads.unisa.edu.au, jonathan.billington@unisa.edu.au, somsak.vanit-anunchai@postgrads.unisa.edu.au
[2] Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N, DENMARK
Email: kris@daimi.au.dk

Abstract. The sweep-line state space method allows states to be deleted from memory during state exploration, thus alleviating state explosion. Properties of the system (such as the absence of deadlocks) can then be verified on-the-fly. This paper presents an extension to the sweep-line method that allows on-the-fly checking of language inclusion, which is useful for protocol verification. This has been implemented in a prototype Sweep-line library for Design/CPN. We evaluate the prototype by applying it to the connection management procedures of the Datagram Congestion Control Protocol, a new Internet transport protocol.
Keywords: On-The-Fly Protocol Verification, Sweep-line Method, Language Inclusion, Internet Transport Protocols.

1 Introduction

The collection of methods forming the paradigm of analysis techniques that involve generation of all or part of the set of reachable states of a system are known as state space methods. Generation of the state space, along with all possible transitions between states (i.e. the occurrence graph) allows many verification questions to be answered by model checking techniques. State space methods have an advantage over theorem proving techniques [31] in that they are relatively easily automated in software tools such as SPIN [15, 28] or Design/CPN [6, 9] for Coloured Petri nets (CPNs) [17, 20].

One disadvantage that has been the focus of much research is that of the *state explosion problem* [31]. Even for relatively simple systems, the number of reachable states can be very large. The unfortunate result of state explosion is that in many cases the entire occurrence graph is too large to fit into computer memory. This has led to the development of a number of *reduction* techniques to alleviate the problem.

A good survey of reduction techniques is provided in [31]. The sweep-line exploration method [7] is a reduction technique that uses the notion of *progress* to determine which states to delete and guarantees full exploration of the occurrence graph. By defining a *progress mapping* for the states of the model being analysed, it is possible to exploit *progress* in the model to identify states that are guaranteed not to be reached again [7] or are unlikely to be reached again [22]. The sweep-line method can be used to determine a number of properties on-the-fly, such as reachability and termination properties (dead markings). Sweep-line has been used to verify termination properties of CPN models of the Internet Open Trading Protocol (IOTP) [12] and parts of the Wireless Application Protocol (WAP) [14] in addition to analysis of dead transitions in the WAP model. The sweep-line method was combined with equivalence classes in [4] and used to verify an infinite-state system in [25]. Several papers on the sweep-line method

* Supported by Australian Research Council Discovery Grants DP0210524 and DP0559927.
** Supported by the Carlsberg Foundation.

have presented extensions to the original algorithm [7] either to extend the range of systems to which the sweep-line can be applied [22], improve the potential for reduction [21], or add useful facilities for debugging models [23]. Note that there may be a trade-off between space and time savings; using sweep-line with a progress mapping that gives a large saving in memory may induce more re-exploration of parts of the reachability graph and thus take longer to complete. However, analysis of the CPNs mentioned above indicates that, generally, the sweep-line achieves a reduction in both time and space.

Of particular relevance to protocol verification is the area of language analysis [3]. In brief, formal models are created of the *service* that the protocol should provide and also of the protocol itself, using e.g. CPNs. The occurrence graphs of these models contain all possible sequences of user-observable events, called *service primitives*, exhibited by both the service and the protocol. The occurrence graphs (OGs) can be interpreted as Finite State Automata (FSAs) representing the *service language* and *protocol language* respectively. If the service and protocol languages are identical (language equivalent) then the protocol is a faithful refinement of the service. If the protocol language is contained within the service language (language inclusion) then the protocol may still be a faithful refinement of the service, as it may implement an acceptable subset of the service. If the protocol language is not contained within the service language, and given that the service is correctly defined, this indicates erroneous behaviour on the part of the protocol. The check for language inclusion involves computing the language accepted by the parallel composition of the FSA representation of the protocol OG and the complement of the FSA representation of the service OG. If this language is empty, language inclusion holds. Strong parallels can be drawn between this problem and model checking temporal logic [2, 24, 34].

For protocol models with a large number of states, it may be necessary to apply reduction techniques such as the sweep-line method to alleviate the state explosion problem. Unfortunately, the basic language analysis methods described in [3] and briefly mentioned above require the full protocol model occurrence graph to be present in memory, a situation the sweep-line method, by its nature, aims to avoid. Algorithms for temporal logic model checking have been developed that are conceptually similar to the language inclusion problem [34] with the slight difference that temporal logic propositions are formulated using Büchi automata accepting infinite words. Algorithms for performing temporal logic model checking on-the-fly have also been developed [8, 13].

This paper presents a method of on-the-fly language inclusion checking, similar in spirit to on-the-fly temporal logic model checking, using the sweep-line method. This allows the language inclusion property of a protocol model to be checked even when the full occurrence graph is too large to be stored in computer memory. A prototype sweep-line library incorporating language inclusion has been developed for the computer tool Design/CPN for CPNs. Our technique is applied to a CPN model of the connection management procedures of the Datagram Congestion Control Protocol (DCCP) [18].

The paper includes both theoretical and application-oriented contributions. The functionality of the sweep-line method has been extended to allow the checking of language inclusion on-the-fly. This development allows checking the conformance of a system to any property expressable as a deterministic ϵ-free FSA and is thus far more general than the protocol verification context described above. Application of the prototype implementaton has allowed us to confirm the language inclusion property of DCCP's connection management procedures to its service for configurations of the DCCP model that were previously unattainable by the conventional protocol verification techniques in [3].

The rest of this paper is organised as follows. Section 2 provides a brief introduction to the sweep-line method. Section 3 introduces the idea of language inclusion and describes the procedure for determining whether or not the property of language inclusion holds for a given

protocol. Section 4 describes our method for checking language inclusion on-the-fly with the sweep-line method and a proof of correctness is given. We demonstrate our method on DCCP's connection management procedures in Section 5. Finally, some concluding remarks and future work are presented in Section 6. We assume that the reader is familiar with the basic concepts of reachability analysis and formal language theory.

2 The Sweep-line Method

The sweep-line method is based on the notion of *progress* within the system being modelled. Systems exhibit progress in different ways. One example is found in transaction protocols, where interacting protocol entities move through a series of interactions towards a final completed state. Many communication protocols exhibit progress through sequence numbers and retransmission counters. The key concept behind the sweep-line method is that if we can quantify the progress of a system in each state, then we can identify the states with a lower *progress value* that cannot be (or are unlikely to be) reached from states with a higher progress value. When states are no longer reachable we do not need to keep them in memory for comparison with each newly generated state.

The notion of progress is captured formally in a *progress measure* [7, 22]. Importantly a progress measure specifies a *progress mapping* ψ from states to progress values that are ordered. In this paper we shall use the natural numbers $\mathbb{N}$ as the set of progress values and their usual order relations (e.g. $\leq, <, >$). We begin by defining an occurrence graph as a formalism-independent labelled directed graph, a progress mapping from states of an OG to progress values and then explain how the sweep-line method works by way of an abstract example.

Definition 1 (Occurrence Graph).
The occurrence graph OG of a finite state system can be represented as a labelled directed graph $OG = (S, L, E, s_0)$ *where*

- S *is the finite set of states accessible from the initial state* s_0*;*
- L *is a set of labels;*
- $E \subseteq S \times L \times S$ *is the set of labelled directed edges; and*
- $s_0 \in S$ *is the initial state.*

Definition 2 (Progress Mapping).
A progress mapping ψ *from the states of an OG to progress values is a function* $\psi : S \to \mathbb{N}$ *where*

- S *is the set of states of OG; and*
- $\mathbb{N}$ *is the set of natural numbers.*

If, for all $s \in S$, all successors of s have progress values equal to or greater than s, then we say ψ is *monotonic*, i.e. ψ is *monotonic iff* $\forall (s, l, s') \in E, \psi(s') \geq \psi(s)$. If this condition does not hold, i.e. if $\exists s' \in S$ such that $(s, l, s') \in E$ and $\psi(s') < \psi(s)$, then (s, l, s') is a *regress edge*, and ψ is *non-monotonic*. The monotonicity of the progress mapping can be checked during OG generation as all arcs in the OG are traversed by the sweep-line method.

We may consider that the mapping ψ induces an ordered partition on the set of reachable states. When generating the occurrence graph, once all successors of all states with a particular (minimum) progress value have been generated, all the states (that are not of interest) with this progress value can be deleted, freeing up memory, and reducing the time spent comparing new states with those already generated. The overhead is calculating the progress value for each state, and ensuring that states are explored in a least-progress-first order.

Figure 1 (from [23]) shows three snapshots of the OG during OG exploration. Arc labels have been omitted to simplify the diagram. The states are arranged from left to right in ascending progress order. Nodes that have been explored and deleted are represented as empty circles. Nodes currently in memory (but that have not yet been explored) are solid black circles. Nodes yet to be discovered are grey circles. In this example we assume that states are markings of a CPN. In Fig. 1 (a) the states M_0 and M_1 have been explored and subsequently deleted because they have a smaller progress value than the minimal progress value among the unexplored states M_2, M_3 and M_4. The conceptual sweep-line is shown as a vertical dashed line, immediately to the left of the unprocessed states.

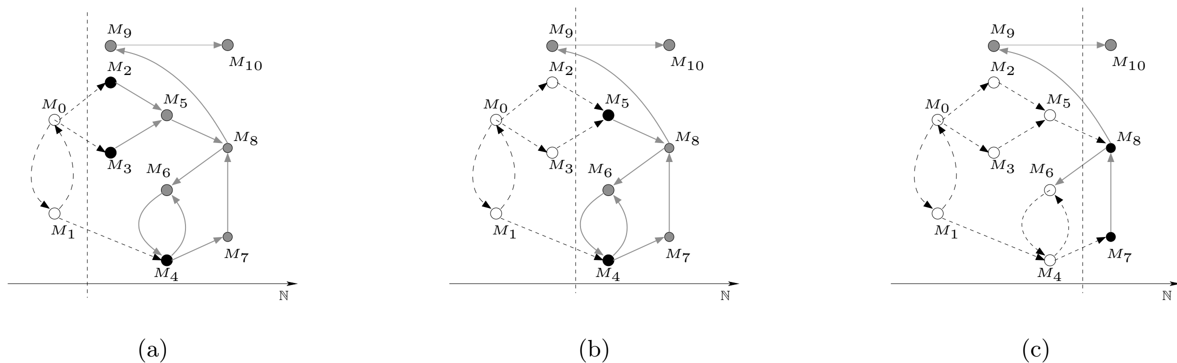
Exploring in least-progress-first order means that either M_2 or M_3 will be explored next. When both have been explored, the sweep-line moves to the right and M_2 and M_3 are deleted, giving the situation shown in Fig. 1 (b). M_4, M_5 and M_6 will be explored and eventually the situation shown in Fig. 1 (c) will be obtained. When M_8 is explored two *regress edges* are identified, one going to the previously explored state M_6 and the other to the unexplored state M_9. Note that the algorithm does not know that M_6 has already been explored, as it was deleted. The Sweep-line method has no way of distinguishing between these two types of regress-edge destinations and so must treat them both as though they have not yet been explored. This is done by marking both M_6 and M_9 as *persistent*, meaning that they cannot be deleted. Exploration does not continue along regress edges, but M_6 and M_9 are flagged as *roots* (initial states) for a subsequent sweep. In the subsequent sweep, M_{10} is discovered, along with the re-exploration of M_4, M_7 and M_8. Because M_6 and M_9 are persistent, the regress edges to M_6 and M_9 discovered in the re-exploration of M_8 do not induce a further sweep. The correctness of the sweep-line algorithm (both termination and full OG coverage) was proved in [22].

An algorithm for the sweep-line method that uses non-monotonic progress mappings was presented in [22]. We presented a modified version in [12] that used set notation, which we reproduce in Fig. 2. For a detailed description of this algorithm, the reader is referred to [12].

One drawback of the sweep-line method is that users need to define and supply their own progress mapping. Steps have been taken towards automatic generation of ψ for low-level Petri nets [30] and compositional systems [21].

3 Language Analysis in Protocol Design and Verification

Language analysis is commonly used in the area of protocol design and verification. A *service specification* captures the requirements of a protocol, such as absence of deadlock and livelock,



(a) (b) (c)

Fig. 1. Snapshots of sweep-line occurrence graph exploration.

1: $\textsc{Roots} \leftarrow \{s_0\}$
2: $\textsc{Persistent} \leftarrow \emptyset$
3: $\textsc{Unexplored} \leftarrow \emptyset$
4: $\textsc{Explored} \leftarrow \emptyset$
5: $\textsc{Successors} \leftarrow \emptyset$
6: **while** $\textsc{Roots} \neq \emptyset$ **do**
7: $\quad\textsc{Unexplored} \leftarrow \textsc{Roots}$
8: $\quad\textsc{Roots} \leftarrow \emptyset$
9: $\quad$**while** $\textsc{Unexplored} \neq \emptyset$ **do**
10: $\quad\quad$(* Generate the successors of a node in $\textsc{Unexplored}$ that has the lowest progress value *)
11: $\quad\quad$Select $s \in \{s' \in \textsc{Unexplored} \mid \forall s'' \in \textsc{Unexplored}, \psi(s') \leq \psi(s'')\}$
12: $\quad\quad\textsc{Unexplored} \leftarrow \textsc{Unexplored} \setminus \{s\}$
13: $\quad\quad\textsc{Explored} \leftarrow \textsc{Explored} \cup \{s\}$
14: $\quad\quad\textsc{Successors} \leftarrow \{s' | (s, l, s') \in E\}$
15: $\quad\quad$**if** $\textsc{Successors} \neq \emptyset$ **then**
16: $\quad\quad\quad\textsc{Roots} \leftarrow \textsc{Roots} \cup \{s' \in \textsc{Successors} \mid \psi(s') < \psi(s) \text{ and } s' \notin \textsc{Persistent}\}$
17: $\quad\quad\quad\textsc{Persistent} \leftarrow \textsc{Persistent} \cup \{s' \in \textsc{Successors} \mid \psi(s') < \psi(s)\}$
18: $\quad\quad\quad\textsc{Unexplored} \leftarrow \textsc{Unexplored} \cup \{s' \in \textsc{Successors} \mid \psi(s') \geq \psi(s) \text{ and } s' \notin \textsc{Explored}\}$
19: $\quad\quad$**end if**
20: $\quad\quad$(* Delete states that have a progress value less than those in $\textsc{Unexplored}$ *)
21: $\quad\quad\textsc{Explored} \leftarrow \textsc{Explored} \setminus \{s \in \textsc{Explored} | \forall s' \in \textsc{Unexplored}, \psi(s) < \psi(s')\}$
22: $\quad$**end while**
23: **end while**

Fig. 2. The Generalised Sweep-line Algorithm, based on the algorithm from [22].

and specifies the service that the protocol must provide to its users. An important part of this is the specification of the allowable sequences of user-observable events, called *service primitives*. A service primitive represents an interaction between the user of the service and the provider of that service. The allowable sequences of these service primitives form the *service language*, which we denote $\mathcal{L}_S$ in this paper.

The behaviour of the protocol itself is captured in a *protocol specification*. Whereas the service specification defines the 'what', the protocol specification defines the 'how'. The protocol specification captures, among other things, a set of sequences of user-observable events (the service primitives) referred to as the *protocol language*, which we denote $\mathcal{L}_P$. The descriptions in this section are initially at a high level, i.e. assuming that we know our service and protocol languages. The method by which we can obtain the service and protocol languages from the service and protocol specifications is discussed later in this section.

The test for *language equivalence* is conducted in two parts. The first is checking whether $\mathcal{L}_P \subseteq \mathcal{L}_S$, i.e. that the protocol language is contained in the service language. This step is known as *language inclusion*. (It is similar in spirit to the notion of *trace preorder* [31] when using trace equivalence.) If this is true, we say that the protocol implements (a subset of) the service and we can guarantee that the protocol does not exhibit any user-observable behaviour that is not in the service. Whether or not the subset of the service implemented by the protocol is an *acceptable* subset depends very much on the protocol itself and is not within the scope of this paper, however [29] provides a good example and discussion. If $\mathcal{L}_P \not\subseteq \mathcal{L}_S$ (and given that the service specification is correct) this indicates that the protocol exhibits unexpected or erroneous behaviour.

The second part of language equivalence involves checking whether $\mathcal{L}_S \subseteq \mathcal{L}_P$, i.e. that everything contained in the service language is contained in the protocol language. If this is true, then all of the behaviour specified by the service is implemented by the protocol. Checking $\mathcal{L}_S \subseteq \mathcal{L}_P$ does not, however, say anything about erroneous behaviour of the protocol. Checking $\mathcal{L}_S \subseteq \mathcal{L}_P$ on-the-fly is outside the scope of this paper but is a topic of future research.

3.1 Finite State Automata Representations of Protocol and Service Languages

Finite State Automata (FSAs) [1] are a useful formalism for the representation and manipulation of service and protocol languages.

Definition 3 (Finite State Automaton).
$FSA = (V, \Sigma, A, i, F)$ is a Finite State Automaton, where

- *V is a finite set of states;*
- *Σ is a finite set of symbols called the alphabet;*
- *$A \subseteq V \times \Sigma \cup \{\epsilon\} \times V$ is the set of actions (transition relation) of the FSA and ϵ is the empty action;*
- *$i \in V$ is the initial state of the FSA; and*
- *$F \subseteq V$ is the set of final states of the FSA.*

The language represented by an FSA, denoted $\mathcal{L}(FSA)$, is the set of all (finite) sequences of symbols from the alphabet Σ recognised by the FSA and that end in a final state. This is sometimes referred to as the language accepted [1] or marked [5] by the FSA. Formal definitions can be found in [1, 5]. In the context of protocol engineering, such a sequence of primitives (called a string) represents a sequence of actions as defined by the service (for $\mathcal{L}_S$) or protocol (for $\mathcal{L}_P$) specification.

Given a deterministic, ϵ-free FSA representing the language $\mathcal{L}(FSA)$ it is easy to find an FSA representing its complement. We denote the complement of FSA as $\overline{FSA}$ and the complement of $\mathcal{L}(FSA)$ as $\overline{\mathcal{L}(FSA)}$. The following definition is based on [5].

Definition 4 (Complement of a Deterministic ϵ-free FSA).
Let $FSA = (V, \Sigma, A, i, F)$ be a deterministic ϵ-free FSA. The complement of FSA (with respect to Σ) is denoted $\overline{FSA} = (\overline{V}, \Sigma, \overline{A}, i, \overline{F})$ where

- *$\overline{V} = V \cup \{Trap\}$;*
- *$\overline{A} = A \cup \{(v, l, Trap) \mid v \in \overline{V}, l \in \Sigma$ and $\forall v' \in V, (v, l, v') \notin A\}$; and*
- *$\overline{F} = \{v \in \overline{V} \mid v \notin F\}$.*

Throughout the rest of this paper let us denote the FSA representing the service language as FSA_S and the equivalent ϵ-free deterministic FSA as $DFSA_S = (V_S, \Sigma_S, A_S, i_S, F_S)$. Let us also denote the FSA representing the protocol language as $FSA_P = (V_P, \Sigma_P, A_P, i_P, F_P)$ and its deterministic ϵ-free equivalent as $DFSA_P$. This paper deals with FSA_P directly and not $DFSA_P$ because the FSA obtained when interpreting an occurrence graph as an FSA, as described shortly, will in general contain ϵ transitions and therefore be nondeterministic. In the conventional situation, however, it is usual for $DFSA_P$ to be obtained and used [3]. For our purposes, there is no technical need for $DFSA_S$ or FSA_P to be minimal, but minimisation can provide space and time benefits in the manipulation of large service or protocol languages.

The occurrence graph of a protocol (or service) model contains all possible sequences of actions that can be performed by the protocol (or service). The occurrence graph therefore contains all sequences of service primitives in the protocol (or service) language, although not necessarily in a form that is easy to analyse or manipulate. Fortunately, an occurrence graph can be interpreted as a FSA simply by designating an initial state and a set of halt states [3,31]. This FSA thus provides a neat way to encapsulate and manipulate the protocol (or service) language, with well-known algorithms for determinisation and minimisation [1].

In the context of protocol verification, we can map from the labels on the arcs of an OG to the set of service primitives, or to ϵ for actions that do not correspond to service primitives. For practical examples we also define a mapping to enumerate the states of the OG (i.e. a mapping from states to integers) for ease of manipulation by computer tools such as the FSM Library [10]. Conceptually, however, this is unnecessary.

Definition 5 (Arc Label Mapping).
Let SP be the set of service primitives *of a given service and protocol. Let $Prim : L \rightarrow SP \cup \{\epsilon\}$ be a mapping from arc labels in an OG to service primitive names or the empty action ϵ.*

Application of this mapping to an OG results in an *abstract occurrence graph* [3] which can be interpreted as a FSA:

Definition 6 (Finite State Automaton of an Abstract OG).
Let $OG = (S, L, E, s_0)$ be an occurrence graph as per Definition 1. By applying the mapping $Prim$ from Definition 5 to OG, let $FSA_{OG} = (V_{OG}, SP, A_{OG}, i_{OG}, F_{OG})$ be a Finite State Automaton interpretation of the abstract OG, where

- *$V_{OG} = S$ is the set of states of the FSA;*
- *SP is the set of service primitive names of the system of interest (the alphabet of the FSA);*
- *$A_{OG} = \{(s, Prim(l), s') \in V \times SP \cup \{\epsilon\} \times V \mid (s, l, s') \in E\}$ is the set of transitions labelled with service primitives or epsilons for internal events;*
- *$i_{OG} = s_0$ is the initial state of the FSA; and*
- *$F_{OG} \subseteq V_{OG}$ is the set of final states of the FSA.*

The FSA interpretation requires knowledge of the legitimate final states of the system prior to analysis. For most protocols, this is not an unreasonable assumption. Legitimate halt states may be known *a priori* or can be determined with an iterative analysis process. If one cannot inspect the state space (e.g. because it is too large to be generated) it is usual in practice to determine halt states on-the-fly using e.g. a predicate function.

Interpretation of the service and protocol OGs as FSAs results in FSAs representing the service (denoted FSA_S) and protocol (denoted FSA_P) languages. FSA_S is usually already deterministic and ϵ-free although, if not, must undergo ϵ removal and determinisation procedures [1] to obtain $DFSA_S$, as the procedure for language inclusion checking calculates the complement of $DFSA_S$. FSA_P does not need to be ϵ-free or deterministic for the purposes of language inclusion checking. This is critical for combining language inclusion checking with the Sweep-line method, as traditional algorithms for ϵ removal and determinisation [1] require the whole FSA to be in memory, something the sweep-line method aims to avoid.

3.2 Checking Language Inclusion

The procedure for checking language inclusion follows the narrative descriptions in [31] for using finite *test* automata representing regular languages to verify properties of a system. For language inclusion to hold, all sequences of service primitives recognised by FSA_P must also be recognised by $DFSA_S$, i.e. $\mathcal{L}(FSA_P) \subseteq \mathcal{L}(DFSA_S)$, or $\mathcal{L}(FSA_P) \cap \mathcal{L}(DFSA_S) = \mathcal{L}(FSA_P)$. Conversely, no sequences recognised by FSA_P can be recognised by $\overline{DFSA_S}$, i.e. $\mathcal{L}(FSA_P) \cap \mathcal{L}(\overline{DFSA_S}) = \emptyset$. In practice, algorithms reason on automata [2], not on regular expressions or (possibly infinite) sets of sequences.

The notion of parallel composition [5, 19] provides a convenient way of finding the common behaviour of two FSAs and thus the common sequences of shared symbols, i.e. the intersection of their languages. A common event (in our case a service primitive) can only be executed in the parallel composition if both FSAs execute it simultaneously, and so the two FSAs are *synchronised* on the common events [5]. All non-common events (in our case the ϵ transitions in FSA_P) can execute without constraint. In our case the two FSAs share a common alphabet, and thus the set of shared sequences of service primitives is the language accepted by the parallel composition. If this set is empty, language inclusion holds. We formalise this below.

We define the parallel composition of two FSAs as follows (based on the definitions in [5]). Note that unlike [5] we use l rather than σ to denote a single symbol from alphabet $\Sigma \cup \{\epsilon\}$. We use σ more conventionally to denote a sequence of symbols from Σ.

Definition 7 (Parallel Composition).
Let $FSA_1 = (V_1, \Sigma_1, A_1, i_1, F_1)$ and $FSA_2 = (V_2, \Sigma_2, A_2, i_2, F_2)$ be two FSAs. The parallel composition, denoted $||$, of FSA_1 and FSA_2 is defined as $FSA_1 \mid\mid FSA_2 = (V, \Sigma, A, i, F)$ where

- *$V \subseteq V_1 \times V_2$ is the set of reachable states of the parallel composition;*
- *$\Sigma = \Sigma_1 \cup \Sigma_2$ is the set of actions of the parallel composition ($\Sigma_1 \cap \Sigma_2$ is the set of common actions on which the two FSAs are synchronised);*
- *$A = \{((v_1, v_2), l, (v_1', v_2')) \mid l \neq \epsilon, (v_1, l, v_1') \in A_1$ and $(v_2, l, v_2') \in A_2\}$ $\cup\ \{((v_1, v_2), l, (v_1', v_2)) \mid (v_1, l, v_1') \in A_1$ and $l \notin \Sigma_2\}$ $\cup\ \{((v_1, v_2), l, (v_1, v_2')) \mid (v_2, l, v_2') \in A_2$ and $l \notin \Sigma_1\}$ is the set of transitions labelled with actions;*
- *$i = (i_1, i_2) \in V$ is the initial state; and*
- *$F \subseteq \{(v_1, v_2) | v_1 \in F_1, v_2 \in F_2\}$ is the set of final states of the parallel composition.*

We are now ready to state the key theorem for language inclusion checking based on [19].

Theorem 1. *Let $DFSA_S = (V_S, \Sigma_S, A_S, i_S, F_S)$ be a deterministic and ϵ-free FSA representing the service language of a system and $FSA_P = (V_P, \Sigma_P, A_P, i_P, F_P)$ be a (nondeterministic) FSA representing the protocol language of the same system, where $\Sigma_S = \Sigma_P$ is the set of service primitives. For $\mathcal{L}_S = \mathcal{L}(DFSA_S)$ and $\mathcal{L}_P = \mathcal{L}(FSA_P)$,*

$$\mathcal{L}_P \subseteq \mathcal{L}_S \text{ iff } \mathcal{L}(FSA_P \mid\mid \overline{DFSA_S}) = \emptyset$$

where $\overline{DFSA_S}$ is the complement (with respect to Σ_S) of $DFSA_S$, $||$ is the parallel composition operator and $\emptyset$ is the empty set of strings.

Proof. Denote the complement of $\mathcal{L}_S$ (with respect to Σ_S) by $\overline{\mathcal{L}_S}$ such that $\overline{\mathcal{L}_S} = \mathcal{L}(\overline{DFSA_S})$. Using the intersection results from [19] we know that the parallel composition of $\overline{DFSA_S}$ and FSA_P is an automaton for the intersection of $\overline{\mathcal{L}_S}$ and $\mathcal{L}_P$. It follows from basic set theory that $\mathcal{L}_P$ is a subset of $\mathcal{L}_S$ if and only if the intersection of FSA_P and $\overline{DFSA_S}$ is empty. □

The *fsmdifference* routine from the FSM tool suite [10] uses this approach to provide automated language inclusion checking and forms part of the protocol engineering methodology presented in [3].

3.3 A Simple Illustrative Example

A simple example will be used to illustrate the language inclusion checking process. We define, using FSAs, a simple service, a simple protocol, and a second simple but erroneous protocol.

Our simple service FSA representation is shown in Fig. 3 (a), denoted $DFSA_S$. We have two service primitives, namely Send and Receive. The service consists of a single Send event followed by a single Receive event, reflected in the FSA in Fig. 3 (a). The initial state is state 0, represented by the bold circle, and the terminal (accepting, halt) state is state 2, represented by the double circle. The sequence (Send, Receive) is the only sequence accepted by this service.

Our simple protocol FSA representation is shown in Fig. 3 (b), denoted FSA_P. Note that some arcs are labelled with ϵ, the empty move. This symbolises that the protocol performs actions that are not service primitives and are thus not externally visible. This protocol accepts two sequences of actions, (Send, ϵ, Receive) and (ϵ, Send, Receive). When abstracting from internal protocol actions, both of these are the sequence (Send, Receive). Thus by inspection the protocol language is the same as the service language, and thus $\mathcal{L}_P \subseteq \mathcal{L}_S$ holds.

Our erroneous protocol, however, accepts the sequence of primitives (Send, Send, Receive). This sequence corresponds to *loss*, where two Send operations are followed by only one Receive operation. This is shown in Fig. 3 (c) and is denoted FSA_{Perr}. The sequence through states $0 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5$ is erroneous, as when abstracting from internal protocol actions, this sequence is not contained in the service language.

Calculating the Service Language Complement
The service language complement contains all strings over the alphabet of service primitives that are not in $\mathcal{L}_S$. Obtaining the complement FSA, $\overline{DFSA_S}$, as per Definition 4, is shown in two steps in Fig. 4. Figure 4 (a) reproduces the service from Fig. 3. Figure 4 (b) shows the introduction of the Trap state and the *completion* of the FSA [5]. Figure. 4 (c) shows the inversion of the halt states and the resulting $\overline{DFSA_S}$.

Calculating the Parallel Composition
Applying Definition 7 to FSA_P (Fig. 5 (a)) and $\overline{DFSA_S}$ (Fig. 5 (b)) the parallel composition is obtained, as shown in Fig. 5 (c). The initial state is the composite state (0,0). The synchronised action Send can occur from both node 0 in FSA_P and node 0 in $\overline{DFSA_S}$ and this is reflected in the arc from (0,0) to (1,1) in Fig. 5 (c). The action ϵ represents a non-synchronised action and is thus represented by the arc from (0,0) to (2,0) in Fig. 5 (c). The rest of the parallel composition can be described in a similar way. From Definition 7 a state of the parallel composition is designated a final state only if the corresponding states in both FSA_P and $\overline{DFSA_S}$ are designated as final states. No final states are reachable in the parallel composition, and thus $\mathcal{L}(FSA_P||\overline{DFSA_S}) = \emptyset$ and language inclusion holds.

Detecting Erroneous Sequences
Our simple example FSA_P contains no erroneous sequences, as $\mathcal{L}(FSA_P \mid\mid \overline{DFSA_S}) = \emptyset$. This can be easily confirmed by inspection of FSA_P and FSA_S for the single sequence allowable by $DFSA_S$, but in general this is far from trivial.

What if $\mathcal{L}_P \not\subseteq \mathcal{L}_S$? Figure 6 shows the parallel composition of the erroneous protocol, FSA_{Perr}, from our running example. FSA_{Perr} is reproduced in Fig. 6 (a). Recall that the erroneous protocol accepts a sequence comprising two Send primitives followed by a single Receive primitive (after abstracting from internal protocol actions). The resulting parallel composition of this and $\overline{DFSA_S}$ in Fig. 6 (b) is shown in Fig. 6 (c). It accepts the erroneous string, thus $\mathcal{L}(FSA_{Perr} \mid\mid \overline{DFSA_S}) \neq \emptyset$ and this indicates an error in the protocol.

The inverse projection of $\mathcal{L}(FSA_{Perr} \mid\mid \overline{DFSA_S})$ onto the alphabet L_{Perr} of the OG from which FSA_{Perr} was derived allows error traces in the original OG, and thus in the protocol, to be obtained. While a procedure for obtaining error traces is beyond the scope of this paper, it provides a topic for future research.

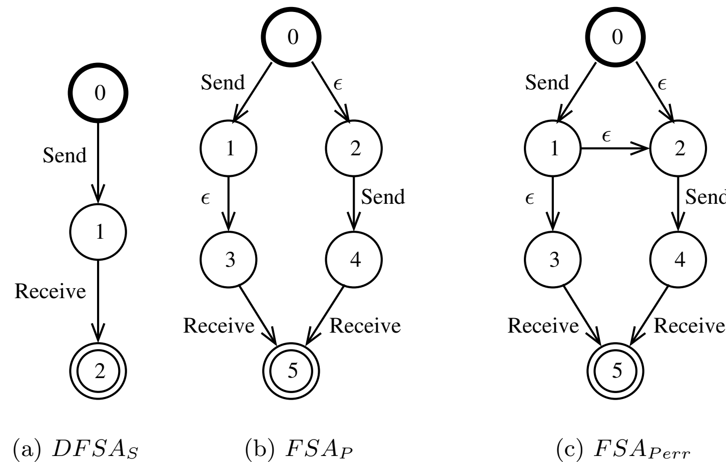
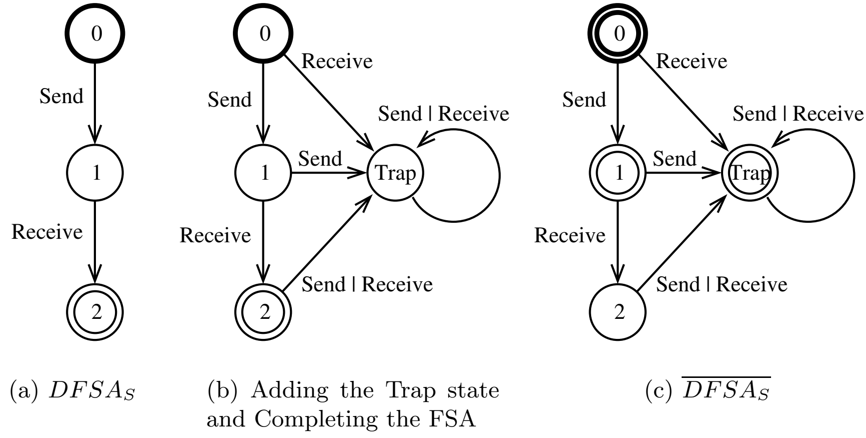
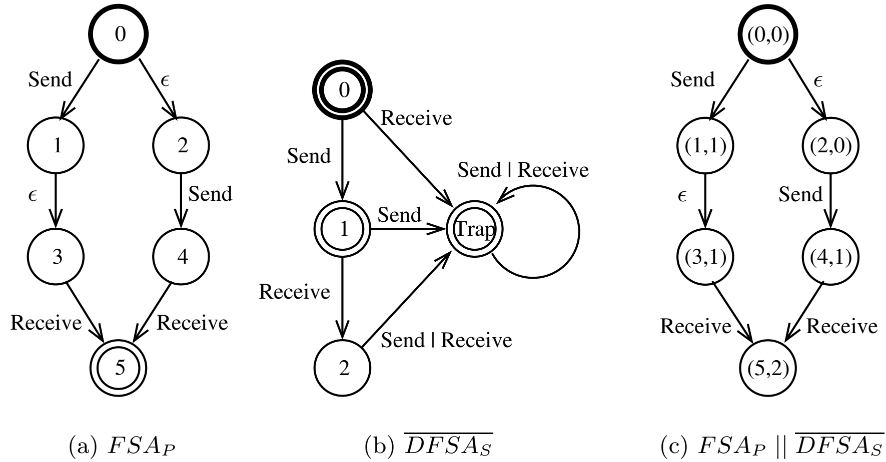


Fig. 3. (a) A simple Service Specification. (b) A simple Protocol Specification. (c) A simple but erroneous Protocol Specification.



(a) $DFSA_S$ (b) Adding the Trap state and Completing the FSA (c) $\overline{DFSA_S}$

Fig. 4. Obtaining $\overline{DFSA_S}$ from $DFSA_S$.



(a) FSA_P (b) $\overline{DFSA_S}$ (c) $FSA_P \mid\mid \overline{DFSA_S}$

Fig. 5. The parallel composition of FSA_P and $\overline{DFSA_S}$.

4 On-the-Fly Language Inclusion Checking with the Sweep-Line Method

Methods for on-the-fly verification of conformance between a system and a specification or property of that system using standard reachability techniques are not new. Valmari [31] describes a procedure for on-the-fly checking of trace equivalence using composition of FSAs. In the area of temporal logic model checking, techniques exist to compose Büchi automata to verify temporal logic propositions on-the-fly [8, 13].

We augment the sweep-line method so that it interprets the OG of a protocol as an FSA and performs the parallel composition with the complemented service language on-the-fly. In essence, the sweep-line algorithm no longer generates the OG of the protocol, but rather it generates the parallel composition of FSA_P with $\overline{DFSA_S}$, *guided* by the exploration of the reachable states of the protocol model. Essentially we are *sweeping the parallel composition*.

It is often the case that when modelling a protocol specification, its occurrence graph is too large and reduction techniques need to be applied. This is not usually the case for the service specification, however, which generally is much less complicated than the protocol specification. Experience has shown us that when service specifications are finite, the occurrence graphs of service specification models tend to be quite small and easily manageable by standard techniques. For the rest of this paper, we assume that this is the case, and assume we are able to obtain $DFSA_S$ and $\overline{DFSA_S}$ for our given service specification.

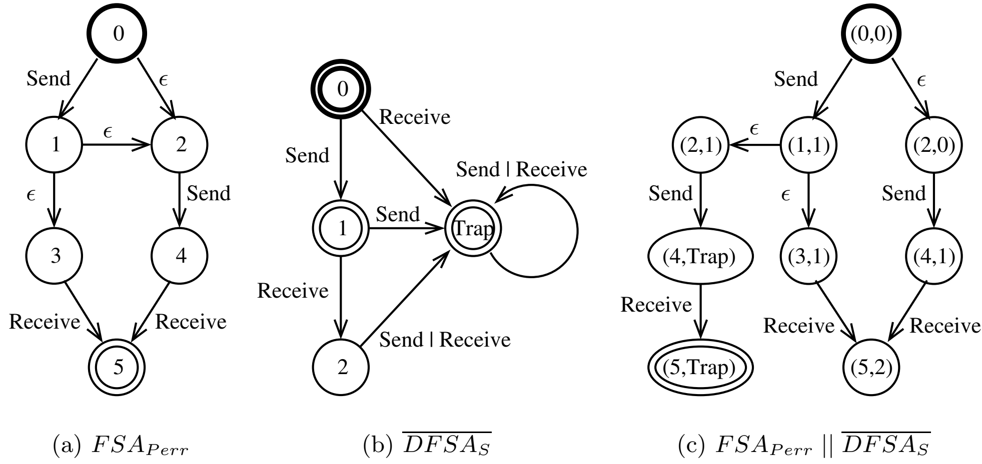


Fig. 6. The parallel composition of FSA_{Perr} and $\overline{DFSA_S}$.

4.1 Sweeping the Parallel Composition

To compute the parallel composition on-the-fly we must interpret the occurrence graph of the protocol specification model, OG_P, as FSA_P on-the-fly. This presents no difficulty given the interpretation provided by Definition 6 with a predicate to determine legitimate halt states. From this point on, we will refer only to the FSA interpretation of the OG of the protocol model.

Because the sweep-line algorithm now operates on pairs of states $(v_P, v_S) \in V_P \times \overline{V_S}$ we define a new progress mapping for the sweep-line method:

Definition 8 (Parallel Composition Progress Mapping).
A progress mapping $\psi_{||}$ from states of $FSA_P \mid\mid \overline{FSA_S}$ to progress values is a function $\psi_{||} : V \rightarrow \mathbb{N}$ where

- *$V = V_P \times \overline{V_S}$ is the set of states of the parallel composition; and*
- *$\mathbb{N}$ is the set of natural numbers.*

Although the progress mapping is from state pairs, it is not obvious to us how the states of $\overline{DFSA_S}$ may contribute to progress. This is because it is the complement of the service. The service has already abstracted as much as possible from states as it is only meant to define the service language. Now taking its complement provides no physical insight into a notion of progress that may be compatible with our feel for progress in the protocol. We therefore do not attempt to derive any measure of progress from the states of $\overline{DFSA_S}$. Thus $\psi_{||}$ is essentially the progress mapping ψ_P for the protocol, that would be used for sweep-line exploration of the OG for properties such as deadlock. The definition of a progress mapping given above can be readily extended to progress vectors as is done in [11].

Because we are calculating the parallel composition on-the-fly, there are two factors that must drive the exploration. Firstly, exploring all reachable states in OG_P, and secondly, exploring all states in the parallel composition.

The second condition is satisfied by the sweep-line method itself. States in V will only be deleted once all successors are known. The only consequence, if any, may be re-exploration of some states of FSA_P, but sweep-line guarantees that truncated exploration will never occur.

The first condition is satisfied because it is the actions of the protocol model that drive the exploration of new states in the parallel composition. Any action enabled in a state $v_P \in V_P$ is mapped either to a service primitive or to ϵ. If mapped to a service primitive, then by

definition an identical action is guaranteed to exist in the corresponding $v_S \in \overline{V_S}$ of $\overline{DFSA_S}$. Or, if mapped to ϵ, the corresponding state in $\overline{DFSA_S}$ remains unchanged.

States of OG_P may be explored many times even when they are not deleted. When calculating the Occurrence graph new states are generated until all reachable states have been explored. In our case, however, we wish to explore all *sequences* of actions. This may require some parts of FSA_P to be explored (and regenerated, if states have been deleted by the sweep-line) more than once. With respect to the parallel composition, this situation may correspond to two different sequences of service primitives leading to the same state in FSA_P but different states in $\overline{DFSA_S}$.

If we are only interested in the fact that there is erroneous behaviour in the protocol, exploration can be terminated as soon as $\overline{DFSA_S}$ enters the 'Trap' state [31]. Similar ideas can be found in temporal logic model checking where verification can be terminated as soon as the proposition is found not to hold [13]. The action of the protocol corresponding to entering the Trap state is an action that violates the service. Once entered, $\overline{DFSA_S}$ can never leave the trap state.

In some sense, detection of erroneous behaviour by detecting $\overline{DFSA_S}$ entering the halt state is slightly more general than detecting erroneous sequences in FSA_P. It is conceivable that an erroneous action of the protocol will not result in acceptance of an erroneous sequence in FSA_P, even though it is clear that an action in the protocol has violated the service. This will happen if FSA_P never reaches a halt state after the erroneous action occurs (recall that the Trap state is a halt state in $\overline{DFSA_S}$.) There are two situations in which this could happen, namely if after the erroneous action, the protocol model reaches a dead marking that is not a halt state, or the protocol enters a livelock composed of non-halt states. However both are unlikely for the following reasons. It is usually the case in protocol verification that (among others) all dead markings of a protocol model are mapped to halt states [3], thus excluding the first possibility. Detecting livelocks is a separate step of the protocol verification process [3] conducted prior to language analysis, thus excluding the second possibility.

4.2 Coping with Regress and Deletion of States

Discovery of regress edges may result in some parts of the parallel composition being explored multiple times, depending on whether the regress edge leads to a state that has not yet been explored or to a state that has been explored and subsequently deleted. We must prove that the conventional parallel composition construction using the full FSA_P is language equivalent to the parallel composition construction created when using the sweep-line method.

Theorem 2. *Let $FSA_{Pf} = (V_f, \Sigma_P, A_f, i_f, F_f)$ be the FSA interpretation of the abstract full occurrence graph of a protocol model and $FSA_{Pu} = (V_u, \Sigma_P, A_u, i_u, F_u)$ be the FSA interpretation of the abstract occurrence graph of the same model, generated using the sweep-line method with an arbitrary progress mapping. Then:*

$$\mathcal{L}(FSA_{Pf} \mid\mid \overline{DFSA_S}) = \mathcal{L}(FSA_{Pu} \mid\mid \overline{DFSA_S})$$

Proof. The first part of this proof involves showing that the full OG of an arbitrary model is language equivalent to an OG obtained when using the sweep-line method. We call such an OG a *sweep-line unrolled* OG, resulting from multiple sweeps and re-exploration due to regress edges. Conceptually, consider that each state in V_u is augmented with an integer to indicate the sweep in which it was explored as was done in [26]. In this way, the states that are explored multiple times are differentiated, hence the term *sweep-line unrolled*.

In [26] Mailund proved that FSA_{Pf} and FSA_{Pu} were strongly bisimilar [31]. To prove that FSA_{Pf} and FSA_{Pu} are language equivalent we first prove *trace equivalence* [31]. In our context, a *trace* is the sequence of service primitives obtained from a finite execution of

FSA_{Pf} or FSA_{Pu} after removal of all ϵ symbols [31]. The set of all traces of an FSA is called its *trace semantics*. It can be shown that if two FSAs are strongly bisimilar, they are also trace equivalent [27]. Trace equivalence is a *congruence* with respect to *hiding* [31] (i.e. application of the mapping $Prim$ from Definition 5) and so when non-service primitives in L are replaced by ϵ, the two FSAs are still trace equivalent.

The final step to prove language equivalence involves showing that the traces ending in a halt state in FSA_{Pf} also end in a halt state in FSA_{Pu}, and vice versa. (The *language* of the FSA is the subset of the traces of the FSA that end in final states.) This result follows directly from the strong bisimilarity of FSA_{Pf} and FSA_{Pu}.

Because FSA_{Pf} and FSA_{Pu} are language equivalent, it follows that the language of the parallel composition is equivalent regardless of whether FSA_{Pf} or FSA_{Pu} is used. From [19] $\mathcal{L}(FSA_{Pf} \mid\mid \overline{DFSA_S}) = \mathcal{L}(FSA_{Pf}) \cap \mathcal{L}(\overline{DFSA_S})$. Because FSA_{Pf} and FSA_{Pu} are language equivalent, this is equal to $\mathcal{L}(FSA_{Pu}) \cap \mathcal{L}(\overline{DFSA_S})$, which is equal to $\mathcal{L}(FSA_{Pu} \mid\mid \overline{DFSA_S})$. So the parallel compositions are language equivalent. □

4.3 Implementation

A prototype sweep-line library incorporating language inclusion checking has been implemented for the tool Design/CPN [9]. In addition to the CPN model of a protocol, the library takes as input a textual representation of $DFSA_S$, from which $\overline{DFSA_S}$ is computed as described in Definition 4. The main algorithm of the library, shown in Fig. 7, explores the parallel composition. (This algorithm can also compute integer bounds, evaluate predicates and return the set of dead markings of the protocol model, although this is not reflected in the simplified algorithms in Fig. 2 or Fig. 7.) Figure 7 assumes that $\overline{DFSA_S}$ has already been calculated. The algorithm is formulated in the context of FSA_P and $DFSA_S$ and the set of states being explored are the composite states $(v_P, v_S) \in V_P \times \overline{V_S}$. Lines 1, 12 and 15 have been modified and lines 6, 16 and 26 are new with respect to the algorithm presented in Fig. 2. The set ACCEPTING, initialised to the empty set on line 6, records the accepting states of the parallel composition on line 16. The algorithm returns true if language inclusion holds, i.e. if ACCEPTING $= \emptyset$ on line 26. Our implementation allows the user to truncate exploration along erroneous paths by not adding successor states to SUCCESSORS on line 14 if $v'_S = Trap$. Given the discussion at the end of Section 4.1, if the user is only interested in an answer to the proposition "Does $\mathcal{L}_P \subseteq \mathcal{L}_S$ hold?" the algorithm could be terminated at line 15 as soon as a successor is generated in which $v'_S = Trap$. Our implementation reports these successors but does not terminate the algorithm.

5 Validation using the Datagram Congestion Control Protocol

The purpose of this section is to validate the algorithm and its implementation for proving language inclusion using the sweep-line method. To do this we use a new protocol being developed for the transport layer of the Internet, called the Datagram Congestion Control Protocol (DCCP) [18]. DCCP has recently been approved as a Proposed Standard by the Internet Engineering Steering Group (IESG). The protocol was designed to overcome the problem of congestion arising in the Internet due to uncontrolled traffic sources (e.g. for delay sensitive applications such as voice) using the User Datagram Protocol. DCCP is connection-oriented to allow negotiation of congestion control mechanisms. An important and new part of DCCP is the way it establishes and releases connections, known as connection management. We are thus interested in verifying DCCP's connection management procedures. These procedures are defined [18] by pseudo code and a finite state machine with nine states: CLOSED, LISTEN, REQUEST, RESPOND, PARTOPEN, OPEN, CLOSEREQ, CLOSING and TIME-WAIT. The

1: $\textsc{Roots} \leftarrow \{(i_P, i_S)\}$
2: $\textsc{Persistent} \leftarrow \emptyset$
3: $\textsc{Unexplored} \leftarrow \emptyset$
4: $\textsc{Explored} \leftarrow \emptyset$
5: $\textsc{Successors} \leftarrow \emptyset$
6: $\textsc{Accepting} \leftarrow \emptyset$
7: **while** $\textsc{Roots} \neq \emptyset$ **do**
8: $\quad\textsc{Unexplored} \leftarrow \textsc{Roots}$
9: $\quad\textsc{Roots} \leftarrow \emptyset$
10: $\quad$**while** $\textsc{Unexplored} \neq \emptyset$ **do**
11: $\quad\quad$(* Generate the successors of a node in $\textsc{Unexplored}$ that has the lowest progress value *)
12: $\quad\quad$Select $s = (v_P, v_S) \in \{s' \in \textsc{Unexplored} \mid \forall s'' \in \textsc{Unexplored}, \psi(s') \leq \psi(s'')\}$
13: $\quad\quad\textsc{Unexplored} \leftarrow \textsc{Unexplored} \setminus \{s\}$
14: $\quad\quad\textsc{Explored} \leftarrow \textsc{Explored} \cup \{s\}$
15: $\quad\quad\textsc{Successors} \leftarrow \{(v'_P, v'_S) \mid (v_P, l, v'_P) \in A_P, (v_S, l, v'_S) \in \overline{A_S}\} \cup \{(v'_P, v_S) | (v_P, l, v'_P) \in A_P, Prim(l) = \epsilon\}$
16: $\quad\quad\textsc{Accepting} \leftarrow \textsc{Accepting} \cup \{(v'_P, v'_S) \in \textsc{Successors} \mid v'_P \in F_P, v'_S \in \overline{F_S}\}$
17: $\quad\quad$**if** $\textsc{Successors} \neq \emptyset$ **then**
18: $\quad\quad\quad\textsc{Roots} \leftarrow \textsc{Roots} \cup \{s' \in \textsc{Successors} \mid \psi(s') < \psi(s) \text{ and } s' \notin \textsc{Persistent}\}$
19: $\quad\quad\quad\textsc{Persistent} \leftarrow \textsc{Persistent} \cup \{s' \in \textsc{Successors} \mid \psi(s') < \psi(s)\}$
20: $\quad\quad\quad\textsc{Unexplored} \leftarrow \textsc{Unexplored} \cup \{s' \in \textsc{Successors} \mid \psi(s') \geq \psi(s) \text{ and } s' \notin \textsc{Explored}\}$
21: $\quad\quad$**end if**
22: $\quad\quad$(* Delete states that have a progress value less than those in $\textsc{Unexplored}$ *)
23: $\quad\quad\textsc{Explored} \leftarrow \textsc{Explored} \setminus \{s \in \textsc{Explored} | \forall s' \in \textsc{Unexplored}, \psi(s) < \psi(s')\}$
24: $\quad$**end while**
25: **end while**
26: **return** $(\textsc{Accepting} = \emptyset)$

Fig. 7. The Generalised Sweep-line Algorithm augmented for Language Inclusion Checking.

procedures are implemented by exchanging packets between peer DCCP entities in end systems. DCCP uses 10 different packet types: Request, Response, Data, DataAck, Ack, CloseReq, Close, Reset, Sync and SyncAck. Each packet includes 48 bit sequence numbers and most packets also include a 48 bit acknowledgement number. For each connection, DCCP entities maintain a set of state variables to keep track of sequence numbers. The important variables for connection management are Greatest Sequence Number Sent (GSS), Greatest Sequence Number Received (GSR), Greatest Acknowledgement Number Received (GAR), and the Initial Sequence Number Sent and Received (ISS and ISR). For a complete description of the protocol, please see [18, 33].

We modelled and analysed the Connection Management (CM) procedures of version 5 of DCCP with CPNs and found a deadlock [32]. Since then, DCCP has been revised 6 times and is now at version 11. We revised our CPN model of DCCP to version 11 and found chatter in the protocol using the OG tool of Design/CPN [33]. However, no attempt has been made as yet to verify DCCP CM against its service.

5.1 DCCP-CPN Model

The DCCP CM CPN model [33] comprises 6 places, 22 substitution transitions, 63 executable transitions and 9 functions. The structure is shown in the hierarchy page in Fig. 8 and the top-level view in the DCCP Overview Page shown in Fig. 9. For a detailed description of the model, the reader is referred to [33].

5.2 DCCP Service Model and Language

DCCP [18] does not define the service of DCCP to its users. Hence our first task was to create a service definition. We did this from our knowledge of the protocol and by following the Open Systems Interconnection Connection-Oriented Transport Service Definition [16]. Firstly

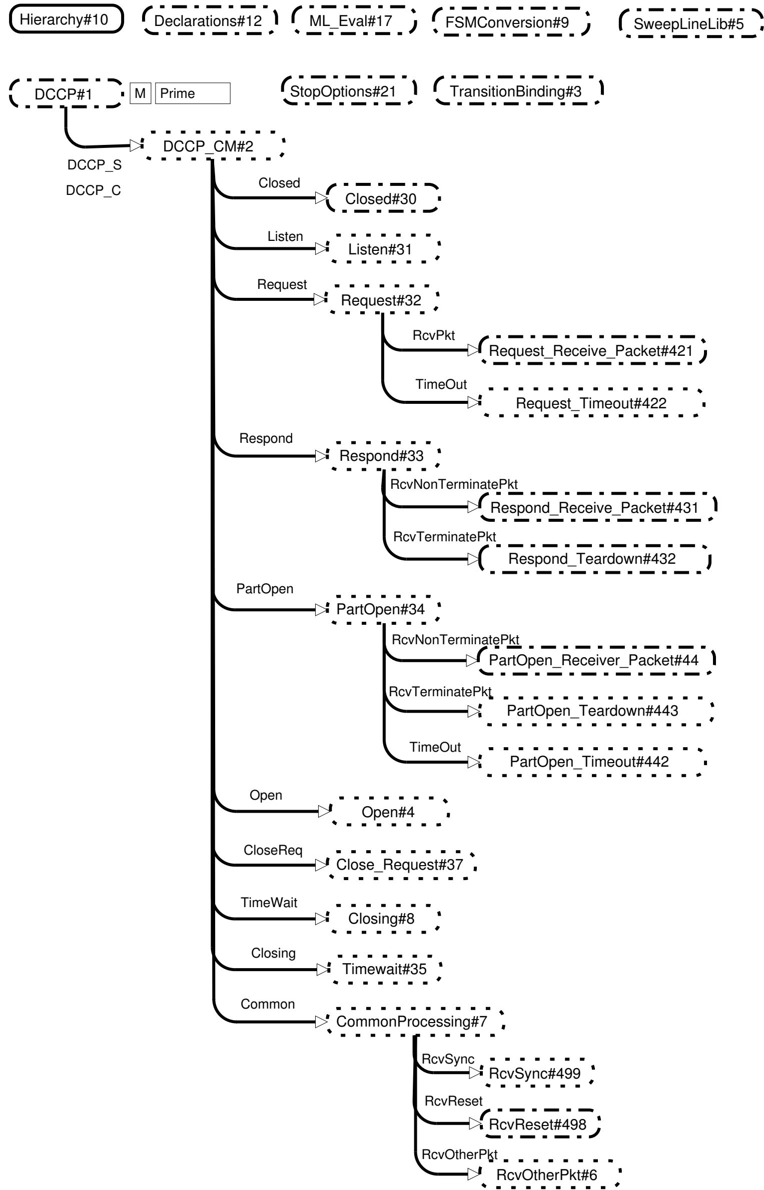


Fig. 8. The DCCP Hierarchy Page.

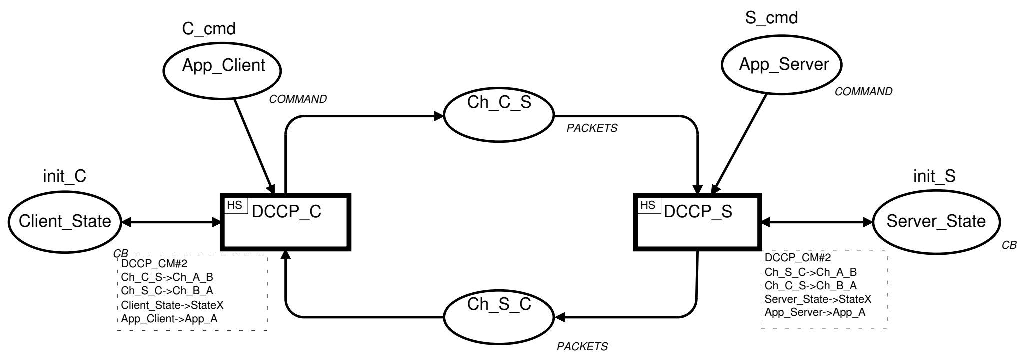


Fig. 9. The DCCP Overview Page.

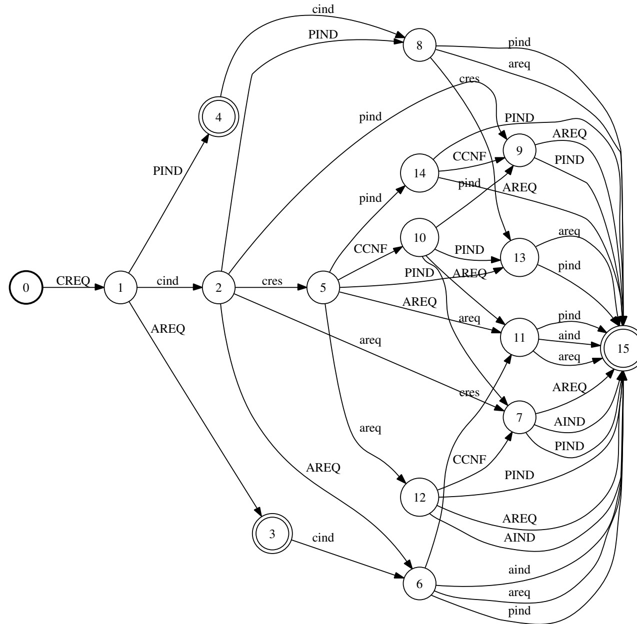


Fig. 10. DCCP's Connection Management Service Language.

we defined the service Primitives. Those shown in Table 1 are only for connection management. We then developed a CPN model of the service specification and generated its OG which can be viewed as a finite state automaton. This automaton was input to FSM tools [10] for FSA minimisation. The minimised FSA for DCCP connection management is shown in Fig. 10 and represents the service language. The upper case abbreviations refer to interactions with the client and lower case abbreviations refer to interactions with the server.

5.3 Experimental Results

We investigated 5 configurations as shown in Table 2. In all configurations the initial markings of the channel places, Ch_C_S and Ch_S_C, are empty and the initial state of each side is CLOSED. Table 2 shows the initial values of the user commands (where '++' represents multiset addition) on the client and server side for scenarios where the connection can be closed by either entity during establishment. All experiments were conducted on a PC Pentium-IV 2.6 GHz with 1 GByte RAM.

Table 3 shows the results obtained by conventional state space generation of the DCCP CM CPN model. Following the methodology in [3], checking the parallel composition of each occurrence graph with the complement of the service shown in Fig. 10 was done using *fsm difference* from the FSM tools [10]. The 4-tuple in the first column represents the values of

Primitive	Abbreviation
DCCP-Connect.request	CREQ
DCCP-Connect.indication	cind
DCCP-Connect.response	cres
DCCP-Connect.confirm	CCNF
DCCP-User_abort.request	AREQ, areq
DCCP-User_abort.indication	AIND, aind
DCCP-Provider_abort.indication	PIND, pind

Table 1. DCCP Service Primitives.

Configuration	Initial Markings	
	C_cmd	S_cmd
A	1‘a_Open	1‘p_Open++1‘a_Close
B	1‘a_Open++1‘a_Close	1‘p_Open
C	1‘a_Open	1‘p_Open++1‘server_a_Close
D	1‘a_Open++1‘a_Close	1‘p_Open++1‘a_Close
E	1‘a_Open++1‘a_Close	1‘p_Open++1‘server_a_Close

Table 2. Configurations of the DCCP CM CPN model.

Config.	Conventional OG_P		
	nodes	time	terminal markings
A-(0,0,x,0)	1,221	00:00:01	3
A-(0,0,x,1)	6,520	00:00:08	3
A-(0,1,x,0)	8,276	00:00:11	3
A-(0,1,x,1)	75,458	00:05:17	3
A-(1,0,x,0)	180,953	00:26:31	3
B-(0,0,x,0)	459	00:00:00	4
B-(0,0,x,1)	1,453	00:00:02	4
B-(0,1,x,0)	2,526	00:00:03	4
B-(0,1,x,1)	11,649	00:00:17	4
B-(1,0,x,0)	84,495	00:05:40	4
C-(0,0,0,0)	1,633	00:00:01	3
C-(0,0,0,1)	3,119	00:00:03	3
C-(0,0,1,0)	45,011	00:02:06	3
C-(0,1,0,0)	12,570	00:00:20	3
C-(0,1,0,1)	33,865	00:01:37	3
D-(0,0,x,0)	4,852	00:00:05	6
D-(0,0,x,1)	93,773	00:07:49	6
D-(0,1,x,0)	42,754	00:01:54	6
E-(0,0,0,0)	7,927	00:00:11	6
E-(0,0,0,1)	50,905	00:02:55	6

Table 3. Conventional OG Generation Results.

the maximum number of retransmissions allowed for Request, DataAck, CloseReq and Close packets respectively. Retransmissions of the CloseReq packet do not occur in Configurations A, B and D, indicated by an ‘x’ in the table. The number of nodes, time taken for OG generation, and the number of terminal markings are shown in columns 2, 3 and 4.

Table 4 shows the results obtained using conventional on-the-fly language inclusion checking and sweep-line language inclusion checking. Conventional on-the-fly language inclusion checking was simulated by providing the sweep-line algorithm with a trivial progress mapping that maps every state to the same progress value, thus preventing deletion of states. This reduces the sweep-line method to conventional state space generation. For conventional on-the-fly language inclusion checking, the number of nodes, the total time taken and the number of terminal markings of the parallel composition are shown. For sweep-line language inclusion checking, the total number of nodes, peak node storage, total time and terminal markings of the parallel composition are shown. The terminal markings in columns 4 and 8 of Table 4 correspond to pairs $(v_P, v_S) \in V_P \times \overline{V_S}$ in which v_P corresponds to a terminal marking of the DCCP CM CPN. The number of unique terminal markings of the DCCP CM CPN, when abstracting from the service complement state, is shown in brackets. This confirms the terminal marking results obtained when using conventional OG generation as shown in Table 3. Interestingly, the additional overhead of calculating non-trivial progress values is greater than the time saved

Config.	Conventional ($FSA_P \parallel \overline{DFSA_S}$)			Sweep-line ($FSA_P \parallel \overline{DFSA_S}$)				%	%
	nodes	time	terminal markings	total nodes	peak node	time	terminal markings	space	time
A-(0,0,x,0)	1,364	00:00:00.85	10 (3)	1,364	514	00:00:00.96	10 (3)	37.68	112.94
A-(0,0,x,1)	7,523	00:00:05.55	10 (3)	7,523	2,492	00:00:06.30	10 (3)	33.13	113.51
A-(0,1,x,0)	9,045	00:00:06.86	10 (3)	9,045	3,098	00:00:07.66	10 (3)	34.25	111.66
A-(0,1,x,1)	83,586	00:01:47.26	10 (3)	83,586	27,631	00:01:56.56	10 (3)	33.06	108.67
A-(1,0,x,0)	194,747	00:07:51.29	10 (3)	194,747	62,016	00:06:58.24	10 (3)	31.84	88.74
A-(1,0,x,1)	-	-	-	2,899,394	720,741	16:14:36.35	10 (3)	-	-
B-(0,0,x,0)	510	00:00:00.29	9 (4)	510	170	00:00:00.32	9 (4)	33.33	110.34
B-(0,0,x,1)	1,594	00:00:00.99	9 (4)	1,594	571	00:00:01.12	9 (4)	35.82	113.13
B-(0,1,x,0)	2,742	00:00:01.80	9 (4)	2,742	903	00:00:02.03	9 (4)	32.93	112.78
B-(0,1,x,1)	12,456	00:00:09.96	9 (4)	12,456	4,118	00:00:11.28	9 (4)	33.06	113.25
B-(1,0,x,0)	88,118	00:02:15.20	11 (4)	88,118	22,028	00:01:47.86	11 (4)	25.00	79.77
B-(1,1,x,0)	906,341	02:15:04.56	11 (4)	906,341	212,352	01:08:53.92	11 (4)	23.43	51.01
C-(0,0,0,0)	1,754	00:00:01.11	10 (3)	1,754	674	00:00:01.25	10 (3)	38.43	112.61
C-(0,0,0,1)	3,314	00:00:02.20	10 (3)	3,314	1,183	00:00:02.45	10 (3)	35.70	111.36
C-(0,0,1,0)	46,454	00:00:43.75	10 (3)	46,454	16,079	00:00:49.27	10 (3)	34.61	112.62
C-(0,0,1,1)	-	-	-	2,211,462	654,651	11:28:09.84	10 (3)	-	-
C-(0,1,0,0)	13,527	00:00:10.38	10 (3)	13,527	4,590	00:00:11.89	10 (3)	33.93	114.55
C-(0,1,0,1)	36,287	00:00:31.88	10 (3)	36,287	11,730	00:00:37.31	10 (3)	32.33	117.03
C-(0,1,1,0)	620,699	00:34:58.91	10 (3)	620,699	195,731	00:38:52.89	10 (3)	31.53	111.15
C-(1,0,0,0)	312,912	00:15:56.09	10 (3)	312,912	100,544	00:14:59.91	10 (3)	32.13	94.12
D-(0,0,x,0)	5,122	00:00:03.50	17 (6)	5,122	1,881	00:00:04.00	17 (6)	36.72	114.29
D-(0,0,x,1)	97,404	00:01:49.80	17 (6)	97,404	30,520	00:02:14.24	17 (6)	31.33	122.26
D-(0,1,x,0)	44,717	00:00:43.35	17 (6)	44,717	15,281	00:00:49.99	17 (6)	34.17	115.32
E-(0,0,0,0)	8,249	00:00:05.75	17 (6)	8,249	2,953	00:00:06.73	17 (6)	35.80	117.04
E-(0,0,0,1)	52,460	00:00:47.08	17 (6)	52,460	17,461	00:01:01.82	17 (6)	33.28	131.31

Table 4. Conventional and Sweep-line results for On-The-Fly Language Inclusion Checking.

through reduced peak state storage, until the total number of states becomes relatively large, i.e. for configurations A-(1,0,x,0) and B-(1,1,x,0).

Language inclusion was found to hold for all configurations analysed by each of the three methods. Moreover, Table 4 shows that in 4 cases the sweep-line completed while the conventional OG generation and conventional on-the-fly language inclusion did not due to state explosion. Thus it is possible to verify the language inclusion property and obtain the dead markings simultaneously.

6 Conclusions and Future Work

Verification of protocols against their service specifications is a difficult task due to the inherent complexity of distributed algorithms. An important property for protocols to preserve is the sequencing of service primitives at their user interfaces, defined in the service specification. To prove this property we need to define the service language: the set of sequences of service primitives that the protocol is meant to obey. In our CPN verification methodology [3] this is done by creating a CPN specification of the service of the protocol, generating the CPN's OG using Design/CPN (or similar) and then using automata reduction tools such as FSM [10] to obtain a deterministic (and minimum) FSA that embodies the service language. We then do the same for the protocol and use FSM to compare the service and protocol languages. This is normally possible for moderately complex protocols for small values of parameters such as retransmission counters. However, for practical Internet protocols, such as the Transmission Control Protocol, which have a number of retransmission counters, it has proven impossible

to analyse them using conventional full state spaces with Design/CPN, when the number of retransmissions of each retransmission counter is only one.

This has stimulated us to consider using the sweep-line method. However, neither Design/CPN nor CPN Tools has the functionality to allow language comparison, let alone language comparison using the sweep-line. This paper makes the first attempt to provide a language comparison facility for Design/CPN. Because we can use FSM for language comparison using conventional reachability analysis, we concentrated on language inclusion using the sweep-line. Firstly we synthesised all the necessary theory for developing an algorithm for checking language inclusion on-the-fly. This entailed demonstrating that language inclusion can be decided by demonstrating that the parallel composition of the protocol FSA with the complement of the deterministic FSA of the service yields an FSA with an empty language. To make this theory accessible we took a tutorial approach and illustrated it with a simple protocol example. We then applied this theory in the context of the sweep-line method and proved that it holds. We showed that language inclusion can be determined by sweeping the parallel composition, rather than just sweeping the state space, and outputting any violation on-the-fly.

Using this theory we have developed a prototype implementation for checking language inclusion on-the-fly with the sweep-line method, and used it to verify the connection management procedures of a new Internet protocol called the Datagram Congestion Control Protocol. We have shown that the protocol does conform to its service and have checked this using FSM, thus validating the prototype. We have also extended this result for DCCP for parameter values that could not be handled previously using the conventional methodology.

Future work may involve combining this algorithm with path-finding [23] in order to record erroneous sequences of service primitives. We would then explore using these sequences to generate error traces in the protocol OG for debugging purposes. We would also like to extend this method for on-the-fly language equivalence checking.

Acknowledgements

The authors would like to acknowledge their colleagues Dr. Thomas Mailund and Dr. Jörn Freiheit for preliminary discussions of the ideas behind this paper and for comments made on early drafts of this paper. We are also grateful to the anonymous reviewers for their constructive comments.

References

1. W.A. Barrett and J.D. Couch. *Compiler Construction: Theory and Practice*. Science Research Associates, 1979.
2. B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verification - Model-Checking Techniques and Tools*. Springer, 2001.
3. J. Billington, G. E. Gallasch, and B. Han. A Coloured Petri Net Approach to Protocol Verification. In *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 210–290. Springer-Verlag, 2004.
4. J. Billington, G.E. Gallasch, L.M. Kristensen, and T. Mailund. Exploiting equivalence reduction and the sweep-line method for detecting terminal states. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 34(1):23–37, January 2004.
5. C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 1999.
6. S. Christensen, K. Jensen, and L.M. Kristensen. *Design/CPN Occurrence Graph Manual*. Department of Computer Science, University of Aarhus, Denmark. On-line version: `http://www.daimi.au.dk/designCPN/`.
7. S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proceedings of TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 450–464. Springer-Verlag, 2001.

8. J-M. Couvreur. On-the-fly Verification of Linear Temporal Logic. In *Proceedings of Formal Methods'99, Toulouse, France*, volume 1708 of *Lecture Notes in Computer Science*, pages 253–271. Springer-Verlag, 1999.
9. Design/CPN Online. `http://www.daimi.au.dk/designCPN/`.
10. FSM Library, AT&T Research Labs. `http://www.research.att.com/sw/tools/fsm/`.
11. G. E. Gallasch, B. Han, and J. Billington. Sweep-line analysis of tcp connection management. Technical report, 2005. (Draft Report).
12. G.E. Gallasch, C. Ouyang, J. Billington, and L.M. Kristensen. Experimenting with Progress Mappings for the Application of the Sweep-Line Analysis fo the Internet Open Trading Protocol. In *Fifth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*. Department of Computer Science, University of Aarhus, 2004. Available via `http://www.daimi.au.dk/CPnets/workshop04/cpn/papers/`.
13. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *Protocol Specification, Testing and Verification, XV*, pages 3–18. Chapman & Hall, UK, 1996.
14. S. Gordon, L.M. Kristensen, and J. Billington. Verification of a Revised WAP Wireless Transaction Protocol. In *Proceedings of 23rd International Conference on Application and Theory of Petri Nets*, volume 2360 of *Lecture Notes in Computer Science*, pages 182–202. Springer-Verlag, 2002.
15. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
16. ITU-T. *Recommendation X.210, Information Technology - Open Systems Interconnection - Basic Reference Model: Conventions for the Definition of OSI Services*. International Telecommunications Union, Nov. 1993.
17. K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts*. Springer-Verlag, 2nd edition, 1997.
18. E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol. draft-ietf-dccp-spec-11, March 2005.
19. D. C. Kozen. *Automata and Computability*. Springer-Verlag, 1997.
20. L.M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.
21. L.M. Kristensen and T. Mailund. A Compositional Sweep-line State Space Exploration Method. In *Proceedings of FORTE'02*, volume 2529 of *Lecture Notes in Computer Science*, pages 327–343. Springer-Verlag, 2002.
22. L.M. Kristensen and T. Mailund. A Generalised Sweep-Line Method for Safety Properties. In *Proceedings of FME'02*, volume 2391 of *Lecture Notes in Computer Science*, pages 549–567. Springer-Verlag, 2002.
23. L.M. Kristensen and T. Mailund. Efficient Path Finding with the Sweep-Line Method using External Storage. In *Proceedings of the International Conference on Formal Engineering Methods (ICFEM'03)*, volume 2885 of *Lecture Notes in Computer Science*, pages 319–337. Springer-Verlag, 2003.
24. O. Lichtenstein and A. Pnueli. Checking That Finite State Concurrent Programs Satisfy Their Linear Specification. In *Proceedings of 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, 1985.
25. T. Mailund. Analysing Infinite-State Systems by Combining Equivalence Reduction and the Sweep-Line Method. In *Proceedings of ICATPN'02*, volume 2360 of *Lecture Notes in Computer Science*, pages 314–334. Springer-Verlag, 2002.
26. T. Mailund. *Sweeping the State Space - A Sweep-Line State Space Exploration Method*. PhD thesis, Department of Computer Science, University of Aarhus, February 2003.
27. R. Milner. *Communication and Concurrency*. Prentice-Hall International Series in Computer Science. Prentice-Hall, 1989.
28. On-The-Fly, LTL Model Checking with SPIN. `http://spinroot.com`.
29. C. Ouyang. *Formal Specification and Verification of the Internet Open Trading Protocol using Coloured Petri Nets*. PhD thesis, Computer Systems Engineering Centre, School of Electrical and Information Engineering, University of South Australia, Adelaide, Australia, June 2004.
30. K. Schmidt. Automated Generation of a Progress Measure for the Sweep-Line Method. In *Proceedings of TACAS'04*, volume 2988 of *Lecture Notes in Computer Science*, pages 192–204. Springer-Verlag, 2004.
31. A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.
32. S. Vanit-Anunchai and J. Billington. Initial Result of a Formal Analysis of DCCP Connection Management. In *Proceedings of INC 2004, Plymouth, UK*, pages 63–70, July 2004.
33. S. Vanit-Anunchai, J. Billington, and T. Kongprakaiwoot. Discovering Chatter and Incompleteness in the Datagram Congestion Control Protocol. In *Proceedings of FORTE'05*, volume 3731 of *Lecture Notes in Computer Science*, pages 143–158. Springer-Verlag, 2005.
34. M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings of 1st Symposium on Logic in Computer Science, Cambridge, USA*, pages 332–344. IEEE Computer Society Press, 1986.

Business Process Redesign at a Mental Healthcare Institute: A Coloured Petri Net Approach

M.H. Jansen-Vullers and H.A. Reijers

Abstract

Business Process Redesign aims to radically improve the performance of business processes. One of the approaches to derive such an improved process design is an evolutionary approach, making use of redesign heuristics (Reijers, 2003). Simulation of the redesigned business process comes into play if one has to decide whether the redesign is better than the previous process design, or if one needs to compare alternative redesigns. Usually, the characteristics of the process are such that a purely analytical performance evaluation is not feasible.

This paper shows the applicability of coloured Petri nets, especially CPN Tools, in such a redesign approach. The starting point of the paper is a case study in a mental healthcare institute, which focussed on improvement of the intake process: reduction of flow time and service time. We show the initial CPN model and an alternative redesigned CPN model for the intake process and evaluate the impact on flow time and service time.

In line with previous research, we conclude that coloured Petri nets are well suited to model and simulate business processes. Applying Monitors in addition to the regular CPN Tools package helped us to carry out the simulations in such a way that we were able to carry out statistical analysis, enabling us to compare the performance of different models. An important drawback of our approach is that modelling resources in a business process is quite laborious and results in complex constructions, which are hard to communicate with people who should be able to evaluate the effect of a particular redesign.

Keywords: Business Process Redesign, Simulation, Healthcare, CPN Tools, Monitors

1 Introduction

Business Process Redesign is frequently applied to optimise business processes. The way in which such a redesign is carried out ranges from clean sheet approaches to small incremental changes. An important aspect to decide whether to implement a particular redesign is the expected benefit of a redesign, measured in the time needed to handle a case, the time requested from (particular) resources, or the costs. The efforts to implement a new business process are big, the effects of a wrong redesign are huge. Simulation of a business process and several alternatives for this process support the decision which process should be implemented or not.

The choice of a tool to perform simulations is dependent on the goals of the simulation. In a redesign project, two stakeholders can be identified: the modeller and the owner of the business process. In practice, we found that simulation models may not cover all aspects of the business process, or can be very detailed if one takes all aspects into account. In the first case, the model doesn't reflect reality and in such a case it is hard to decide what can be concluded from a redesign. In the second case, the model becomes too complex to be completely understood by the process owner.

This paper focuses on the evaluation of coloured Petri Nets, and CPN Tools in particular, for applicability in redesign projects. The approach that we followed is based on so-called redesign heuristics. Based on a given process design, a heuristic can be applied to improve the process. Well-known examples of such redesign heuristics are the decision to put tasks in parallel, and the decision to assign specialists and/or generalists for a particular task. An overview of the approach can be found in (Reijers, 2003). We carried out a case study at a mental healthcare institute, in which we developed a CPN model for the initial situation and a CPN model for the redesigned situation. We carried out simulations for both situations and compared the results. In the paper we evaluate the modelling process, the resulting models and the results of the simulations.

This paper is structured as follows. In Section 2 we present a short overview of Business Process Redesign and the application of simulations when evaluating a redesign. The CPN model of the intake process of non-urgent patients in a mental healthcare institute and the simulation results are described in Section 3. Next, we describe in Section 4 the redesign, we present the changes in the simulation model, the results of the simulations and finally the comparison with the initial situation. The applicability of our approach is evaluated in Section 5, followed by the conclusions in Section 6.

2 Background

In the early nineties, the first reports appeared on more or less systematic approaches to generate radical performance improvement of *entire* business processes (Davenport and Short, 1990; Hammer, 1990; Davenport and Short, 1990). Their major vehicles were the application of information technology on the one hand and the restructuring of business process on the other. This approach was coined with the terms "Business Process Reengineering" (Hammer, 1990) and "Business Process Redesign", to both of which we will refer to as 'BPR'.

The BPR guru's of the first hour propagated the "clean sheet" approach, i.e. a process should be designed from scratch without considering the existing process in too much detail. However, most BPR projects take the existing business process as starting point (Reijers, 2003): Within the setting of a workshop, several parties involved (management consultants, business professionals, and managers) try to think of favorable alternatives to the business process as a whole or parts of it. IT-specialists, change management experts, and other specialists to implement the new layout of the process within the organization then use the resulting process design.

The technical heart of BPR is the sensible application of a number of recurring redesign practices. (Hammer and Champy, 1993) presents several examples, such as "Small tasks in a business process should be combined into larger tasks". An extensive literature survey in this field, extended with actual BPR experiences, has rendered 29 practices that are often applied in the redesign of a business process (Reijers, 2003). This survey will be taken as the basis for exploring the possibilities to apply CPN Tools in a BPR project.

(Brand and Kolk, 1995) distinguish four main dimensions in the effects of redesign measures: *time*, *cost*, *quality*, and *flexibility*. Ideally, a redesign of a business process decreases the time required to handle the case, it decreases the required cost of executing the business process, it improves the quality of the service delivered, and it improves the ability of the business process to react to variation. The appealing property of their model is that, in general, improving upon one dimension may have a weakening effect on another. For example, reconciliation tasks may be added in a business process to improve on the quality of the delivered service, but this may

have a drawback on the timeliness of the service delivery. To signify the difficult trade-offs that sometimes have to be made they refer to their model as the *devil's quadrangle*.

Awareness of the trade-off that underlies a redesign measure is very important in a heuristic redesign of a business process. Sometimes, the effect of a redesign measure may be that the result from some point of view is worse than the existing business process. The application of *several* redesign rules may also result in the partly deactivation of the desired effects of each of the *single* measures.

The analysis of such a trade-off can be performed with simulations. From practice, the use of simulation is advocated to compare the ``to be" alternatives and to understand the ``what" and the ``why" of the current process and possible alternatives (Ardhaldjian and Fahner, 1994). From a scientific point of view, simulation is used to evaluate quantitative criteria such as flow time or costs as input for design decisions (Desel and Erwin, 2000).

For the analysis of the best practices the simulation facility of CPN Tools was used. CPN Tools and Design/CPN, the predecessor of CPN Tools, have been applied in various industrial projects (University of Arhus, 2005a), but we have found only three projects for which a business process has been analysed with simulation. In one project a planning process was modelled with Design/CPN and this process was simulated with the Design/CPN simulator (Mitchell *et al*, 2004). In another project CPN Tools was used to study the bullwhip effect in supply chains. For this project the supply chain was modelled with CPN Tools and with simulation the bullwhip effect, inventory increase in the chain, was demonstrated (Makajic-Nikolic *et al*, 2004). Also a recently submitted paper for this workshop (Netjes *et al*, 2005) focuses on simulation of business processes and aims to evaluate the application of particular redesign heuristics.

3 The intake process in a CPN Model

In this section we describe the case of an intake procedure to process new requests for non-urgent treatment at a mental healthcare institute in the Netherlands. The procedure is slightly simplified from the procedure in actual use at this institute. The CPN model consists of 24 pages in which we model the control flow, the resources in the process, the triggers that are part of the process, and additional steps for simulation purposes. In this section, we describe each of these parts of the model.

3.1 Modelling the control flow

The two highest levels in the hierarchy of the CPN model reflect the global flow of control: patients enter the model and subsequently a notification is made, they are assigned to two intakers and the intake can take place. This is shown in **Figure 1**.

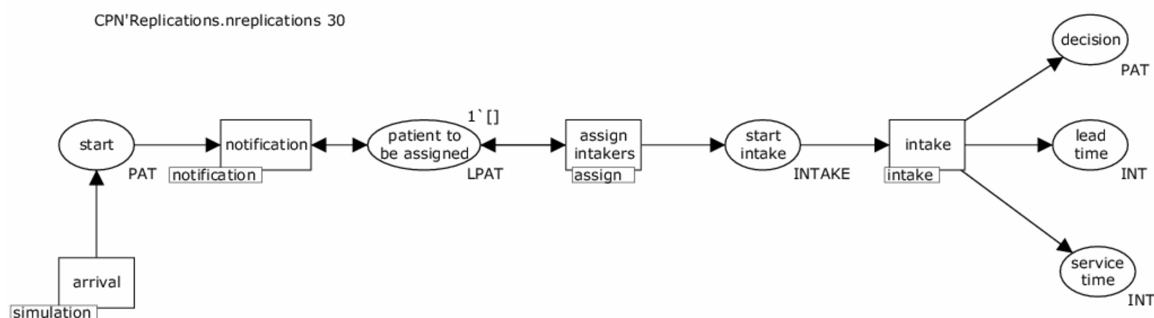


Figure 1 Main process

Subpage 'notification'

The intake process starts with a notice by telephone at the secretarial office of the mental healthcare institute. The secretarial worker inquires after the name and residence of the patient to determine the nursing officer responsible for the part of the region that the patient lives in.

The nursing officer makes a full inquiry into the mental, health, and social state of the patient in question. This information is recorded on a registration form, handed in at the secretarial office, stored in the information system and subsequently printed. For new patients, a patient file is created. The registration form as well as the print from the information system is stored in the patient file. At the secretarial office, two registration cards are produced for respectively the future first and second intaker of the patient.

Subpage 'assign'

Halfway the week, at Wednesday, a staff meeting of the entire medical team (social-medical workers, physicians, and a psychiatrist) takes place to assign all new patients. The assign process first collects all individual patient files of an entire week into a list of patients. When the so-called Wednesday meeting is taking place, the list is emptied in the place 'patient in meeting'.

Each patient will be assigned to a social-medical worker, who will act as the first intaker of the patient. One of the physicians will act as the second intaker. The assignments are recorded on an assignment list, which is handed to the secretarial office. For each new assignment, it is also determined whether the medical file of the patient is required and added to the assignment list.

The secretarial office stores the assignment of each patient of the assignment list in the information system and the actual intakes can take place (output place 'start intake'). For each patient for which the medical file is required, the secretarial office prepares and sends a letter to the family doctor of the patient, requesting for a copy of the medical file.

Subpage 'intake'

The intake process consists of five sub processes: first the intake cards are handed out, then the first and second intake can be done. These can be planned and executed independently, though for the second intaker the medical file needs to be available if previously defined so. If both intakes are done, the data are prepared for the Wednesday meeting in which the treatment of the patient is determined. This process is shown in Figure 2.

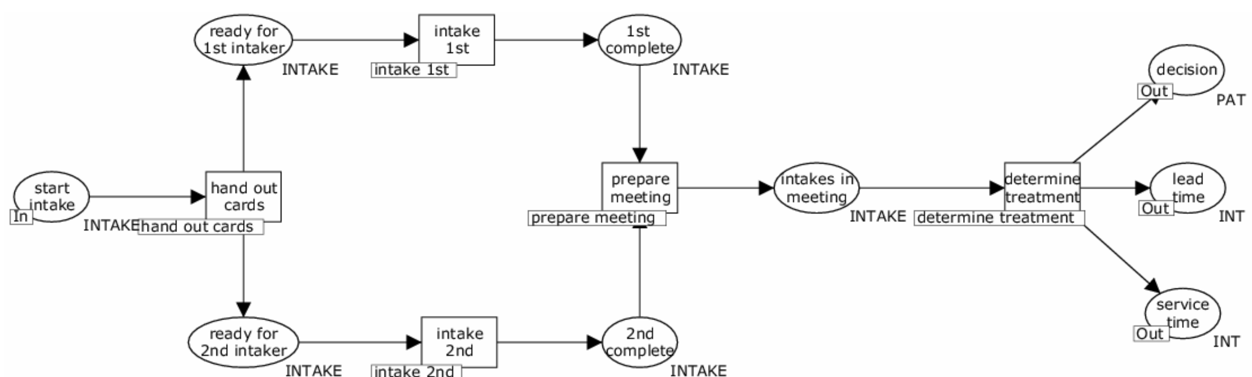


Figure 2 Page 'intake'

The subpage ‘handout cards’ reflects the physical process in which both intakers receive (a set of) cards, of those patients that have been assigned to them during the meeting. This provides them with sufficient information to plan the intake and enables them to work independently.

The intakes for the first and second intaker follow an almost similar pattern: the meeting is planned, then it takes place, and finally it is administratively completed. We first describe the process for the first intaker, and then describe only the differences with respect to the second intaker.

Subpages ‘intake first’ and ‘intake second’

The first intaker plans a meeting with the patient as soon as this is possible. During the first meeting, the patient is examined using a standard checklist, which is filled out. Additional observations are registered in a personal notebook. After a visit, the first intaker puts a copy of these notes and the standard checklist in the patient's file. The second intaker plans the first meeting only after the medical information of the physician – if required – has been received. Physicians use dictaphones to record their observations, which are typed out by the secretarial office and added to the patient file.

To plan the meeting, in the CPN model first the intaker is selected who has been assigned to the patient. If this intaker still has working time available, he can plan the meeting. In the next step, we again select the correct intaker, check whether he has working time available and then the meeting can take place at the scheduled date. In the last step of the intake, the intaker is selected, checked for available working time and finally the intake can be administratively completed. The process for the second intaker differs at two places in the model: when planning the date the patient file needs to be available, and the completion of the file is carried out by a secretarial office worker in stead of the intaker himself.

Subpages ‘prepare meeting’ and ‘determine treatment’

As soon as the meetings of the first and second intaker with the patient have taken place, the secretarial office puts the patient on the list of patients that reach this status. For the staff meeting on Wednesday, they provide the team-leader with a list of these patients. For each of these patients, the first and second intaker together with the team-leader and the attending psychiatrist formulate a treatment plan. This treatment plan formally ends the intake procedure.

The sub process ‘prepare meeting’ synchronizes the information that came from the first and second intaker. When the Wednesday meeting is taking place, the list of intakes is emptied in the place ‘intakes in meeting’. In the meeting, first the correct first and second intaker are selected and checked for available working time. If this is the case, they discuss the proposed treatment for the patient in attendance of the team leader and psychiatrist. The intake is closed.

3.2 Modelling resources

Within the setting of this process, the medical team consists of 16 people: eight social medical workers, four physicians, two team-leaders, and two psychiatrists. Each member of the medical team works full-time and spends about 50 % of his time on the intake of new cases, except for the psychiatrists who spend 10 % of their time on the intake of new cases. (Most of the resources' remaining time is spent on the treatment of patients). The secretarial office consists of eight workers, who work full time. About 50 % of their time is spent on the intake of new cases.

We distinguish five resources in the intake process, which are all modelled as fusion places. The team leader and the psychiatrist are, in view of performance measurements for the healthcare institute, of minor importance, and have been modelled as follows:

```
colset TL = STRING timed; (*name of team leader*)

colset PSY = E timed; (*anonymous team leader*)
```

The secretarial office worker, the social medical worker and the physician do have time constraints, and therefore we keep track of the working time they have available in the model. Because the first (spv) and second (doc) intaker have been assigned to particular cases we also need to keep track of their names. The resulting colours are:

```
colset SPV = product STRING*INT timed; (*name and working time*)

colset DOC = product STRING*INT timed; (*name and working time*)

colset SECR = INT timed; (*working time*)
```

The resources who have working time should, in case of part-time employees, have a notion of a life cycle: they can work for the number of hours assigned to them, and they stop working as soon as they have no more working time available. For this purpose we introduced three 'resting places', see **Figure 3**. In the rest of the model, we check at each transition whether the required resource has working time available.

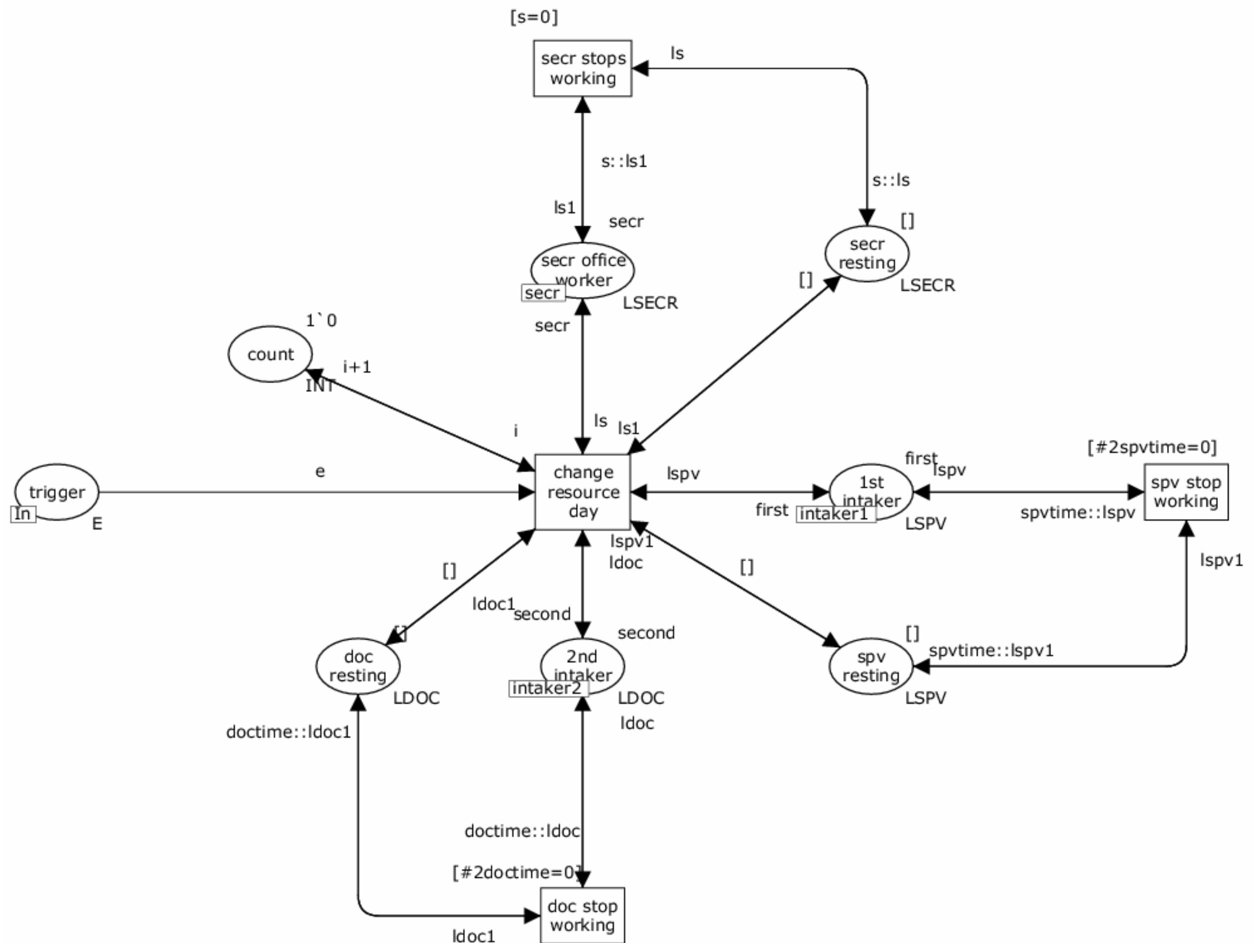


Figure 3 Resource lifecycle

The notion that a resource consumes his working time, introduces the notion of a new working day (resetting all working times) and the requirement that this has been synchronised for all resources. The resetting is carried out by consuming the tokens of the resource places (which are lists) and producing the initial values again (as stored in the .sml files).

The synchronisation is enforced by a global clock (see Figure 4) that produces a trigger to change the resource day. This clock is a transition called 'one more day' that increases the time stamp of the minute counter with 480 minutes. Each time this transition fires the day counter is increased by one and a trigger is produced to change the resource day. The clock stops when all cases in the model have been closed.

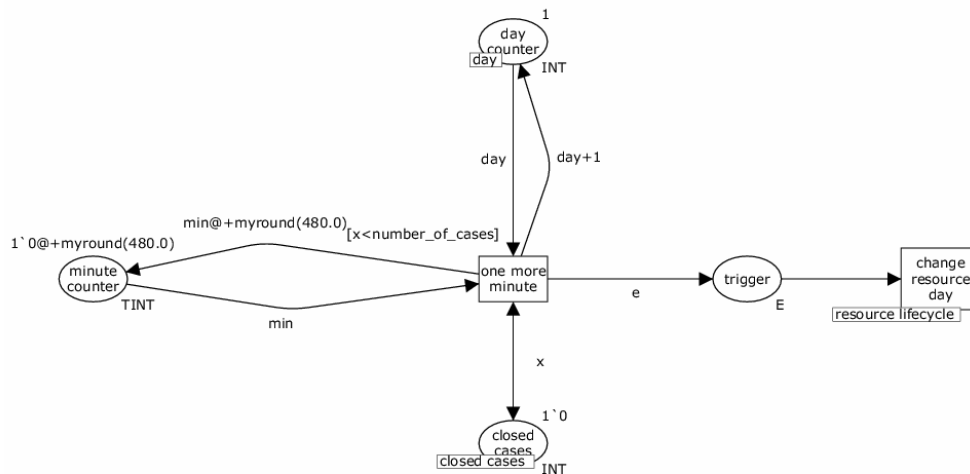


Figure 4 Global clock

3.3 Modelling triggers

In the model we make use of triggers: a work item is worked on only once a resource has taken the initiative. However, also other forms of triggering exist: an external event (for example, the arrival of a message) or reaching a particular time (Aalst and Hee, 2002).

In this model we applied two different kinds of time triggers. The planning of the intake date for the first intaker and for the second intaker are both 'dependent' time triggers, the trigger appears after some time when the previous transition has fired. In

Figure 5 we show the intake by the first intaker. He contacted the patient to set a date for the meeting. This transition (not shown in

Figure 5) produced a token for the fusion place 'date for spv'. The transition 'schedule meeting' uses this as an input token, and produces an output token with delay. This delay is uniformly distributed between 960 and 1920 minutes (i.e., the meeting takes place 2 upto 4 days after the patient has been contacted).

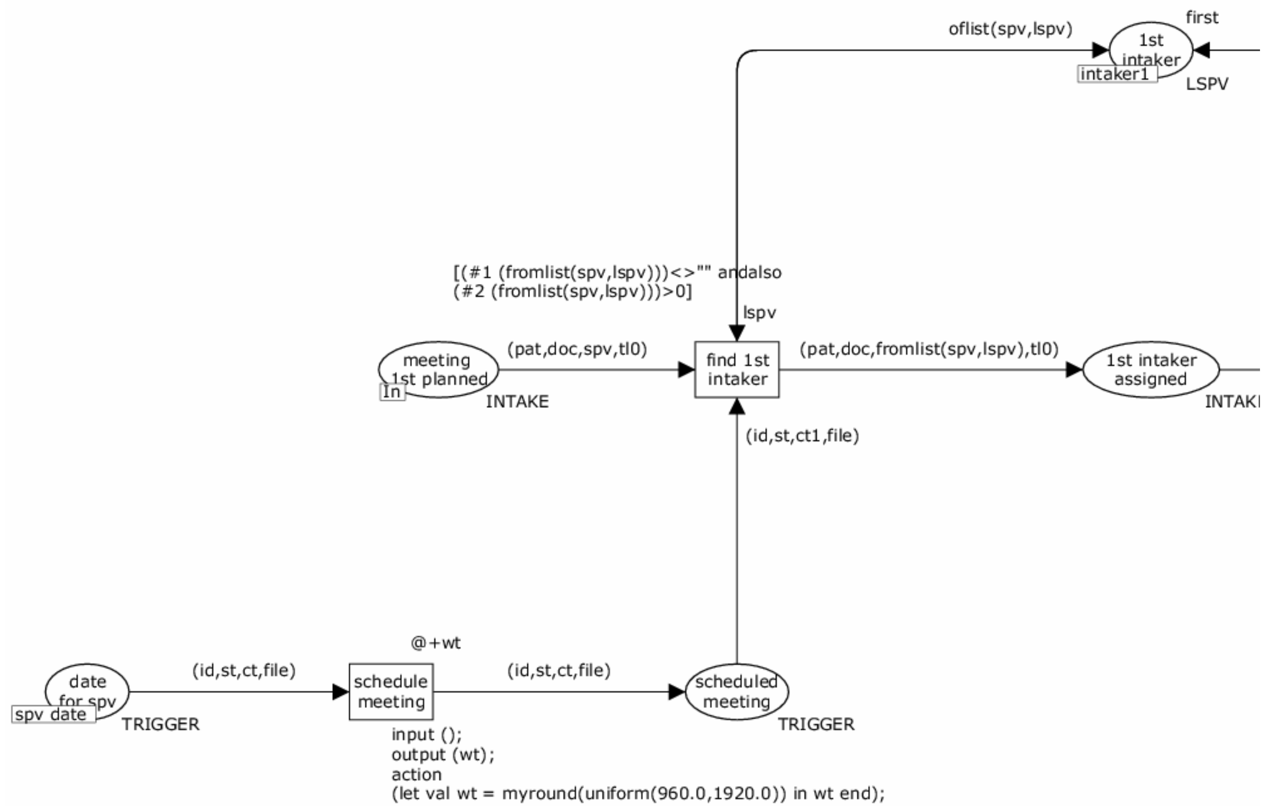


Figure 5 Time trigger for the meeting of the first intaker

Furthermore, we have introduced a time trigger to model the so-called Wednesday meeting. This is a time trigger independent of other transitions and occurs every week on Wednesday morning. Initially, the place 'meeting time' has the value 960 minutes (the first minute of the third day of a working week), and is increased with 2400 minutes (a working week). When all cases have been closed no more meetings are scheduled (see Figure 6).

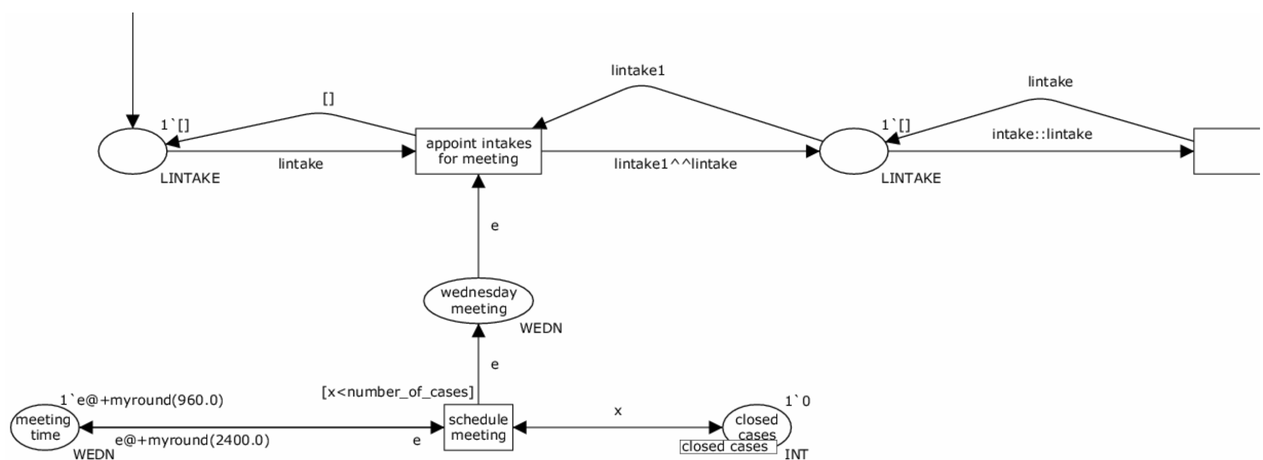


Figure 6 Time trigger for Wednesday meetings

The second kind of trigger in the CPN model is an external event, which may occur in case a medical file is required. The pattern is the same as for the dependent time trigger: the transition

that requests a medical file produces a token that is consumed by a transition called 'receive medical file'. The delay of this transition is 1 upto 3 days.

3.4 Modeling the simulation

The main purpose of the CPN model is to carry out simulations of a business process, to redesign the process and simulate it, and to conclude which of the process variants is more efficient. For each of the steps in the business process (transitions in the CPN model) we measured the duration. These durations are not constant but stochastic. In the CPN model, we applied the following distributions:

- Uniform distribution
- Beta distribution
- Exponential distribution

Furthermore, we modeled the arrival pattern of patient cases, which is Poisson distributed, 20 cases per working day (see Figure 7). The case itself is a product of four parameters: its id, starting time, service time and file type.

```
fun interarrivaltime() = myround(exponential(1.0/24.0));

(*20 cases per working day, this is 20 cases/480 min*)
```

In many places we also used a function called myround, which is implemented as follows.

```
fun myround(r) = round(r*d);
```

Myround multiplies the parameter with a factor d, which can be set in the declarations. We need this factor for precision purposes. The time unit in our model is 1 minute. The distribution functions look like, e.g., uniform(1.0, 3.0) and are applied as time delay for transitions. Therefore we need to convert the outcome of the distribution function to an integer value, with precision problems as a logical result. To avoid longer parameters in many places of the model, we redfined the round function into myround (d*r).

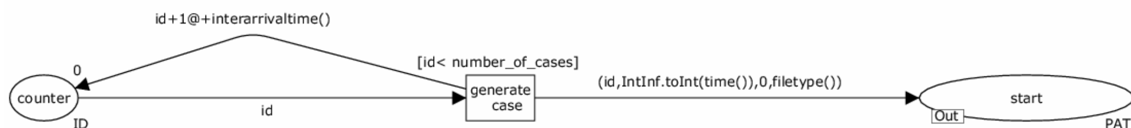


Figure 7 Generating cases

At the output side, we modeled two places: one for calculating service times (third parameter of the case), and one for calculating flow time, which is the difference between the starting time (second parameter) and the clock time. For our statistical analyses we used the monitors-version of CPN Tools. In that case, the service time and flow time are derived from the marking size. The observers are defined as follows:

- For the flow time:

```
fun obsBindElem (determine_treatment'close_intake (1, {i, ct, id,
spv, st, doc, file, tl0})) = IntInf.toInt(time())-(st)
```

– For the service time:

```
fun obsBindElem (determine_treatment'close_intake (1, {i, ct, id,
spv, st, doc, file, tl0})) = ct
```

The simulation results are based on a simulation with 30 sub runs, covering 1000 cases per run (i.e. 50 working days). The time unit in the model was set to 1/100 of a minute. Increasing the number of working days or increasing the time unit with a factor 10 didn't change the results.

3.5 Simulation results

The current performance of the workflow is measured in two ways. As a way of making the external quality of the workflow operational, the average flow time is taken. For the internal efficiency, the average total service time per case is taken. The average flow time is slightly less than 13 working days. In Table 1 we present the detailed results, including the measured average flow time, and the upper- and lower bounds of the 95% reliability intervals.

	flow time		
	LOW	AVG	HIGH
	620201,64	622570,65	624939,65
Minutes	6202,02	6225,71	6249,40
Hours	103,37	103,76	104,16
Days	12,92	12,97	13,02

Table 1 Flow time of the intake process (initial situation)

The total time spent on a new case averages two hours and 55 minutes. This means that the total service time makes up slightly more than 3% of the total flow time. Each day, slightly less than 20 cases arrive. By using Little's law (see, e.g., (van der Aalst en van Hee, 2002), we can deduce that at any time there are on average some 200 new, non-urgent requests for treatment in process. In Table 2 we present the detailed results, including the measured average flow time, and the upper- and lower bounds of the 95% reliability intervals.

	service time		
	LOW	AVG	HIGH
	17418,11	17464,60	17511,08
Minutes	174,18	174,65	175,11
Hours	2,90	2,91	2,92
Days	0,36	0,36	0,36

Table 2 Service time of resources in the intake process (initial situation)

We validated the simulation results with the actual process as it is taken place at the institute. This means, we invited process owners and others from the institute to come to the university and to go through the (digital) models we made. We considered the models at all levels of detail step by step, and we evaluated the results of the simulation. This concludes the description of the initial situation.

4 Redesign

In this section, we give an example of a redesign for the intake process; the approach we followed is based on redesign heuristics. one of the heuristics we found very much applicable is the *case-based work* heuristic. We changed the CPN model of the initial situation according to the redesign in line with this heuristic and comparde the simulation results with those of the initial situation.

4.1 Periodic meetings

In the intake workflow the staff meeting is planned at regular weekly intervals on the Wednesday. During a staff meeting two important things take place:

1. For new cases, the first and second intakers are assigned, and
2. For cases for which both intake interviews have taken place, treatment plans are determined.

From a workflow perspective, periodic restrictions on activities are rather odd. Mental healthcare patients who ‘arrive’ at Wednesday afternoon have to wait one more week until intakers can be assigned. Although these patients are considered to be non-urgent, it is at least not desirable to let them wait longer than necessary. When you are modelling and simulating such process parts, and when being present in the institute, the ‘bad smell’ appears almost automatically. A match with the case based work heuristic is easily made.

Additional analysis of the intake workflow points out that the first activity does not really require a meeting context, provided that the team-leader has sufficient information on the criteria used for new assignments. On the other hand, the second activity is indeed best performed in the context of a meeting. This is because of the limited availability of the psychiatrists, which prohibits more flexible measures.

On basis of the *case-based work* heuristic (CASEB), see (Reijers and Limam Mansar, 2005), we consider as an alternative for the current workflow that the team-leader will carry out new case assignments as soon as they are due; the weekly meeting is strictly used for determining treatment plans. The workflow structure as depicted in Figure 5 then changes in the sense that the time trigger is removed from the subpage "Assign intakers". Because the information is available to the team-leader to base his assignment decision on, we expect that the original duration of the task also decrease from 5 to 2 minutes on average. This time includes the report of the assignment to the secretarial office. Both the social-medical worker and the physician will no longer spend this time on the case.

4.2 Redesign: CPN model and simulation

The CPN model of the initial situation was the starting point for the CPN model of the redesigned situation. This required four changes in the model, all in or in relation to the sub page ‘assign intakers’. First the time trigger for the Wednesday meeting has been removed. Further, the working time of the first and second intaker is not updated anymore, and the guard on the transition ‘assign intakers’ to check available working time of those intakers has been removed. Finally, we had to remove the place of type ‘list of patients’, and to connect the place ‘patient in meeting’ with the previous step. This also included updating the port types and assignments of ports and sockets, as these places appeared at three levels in the model. The redesigned page is displayed in Figure 8. Note the difference with the previous model as displayed in Figure 6.

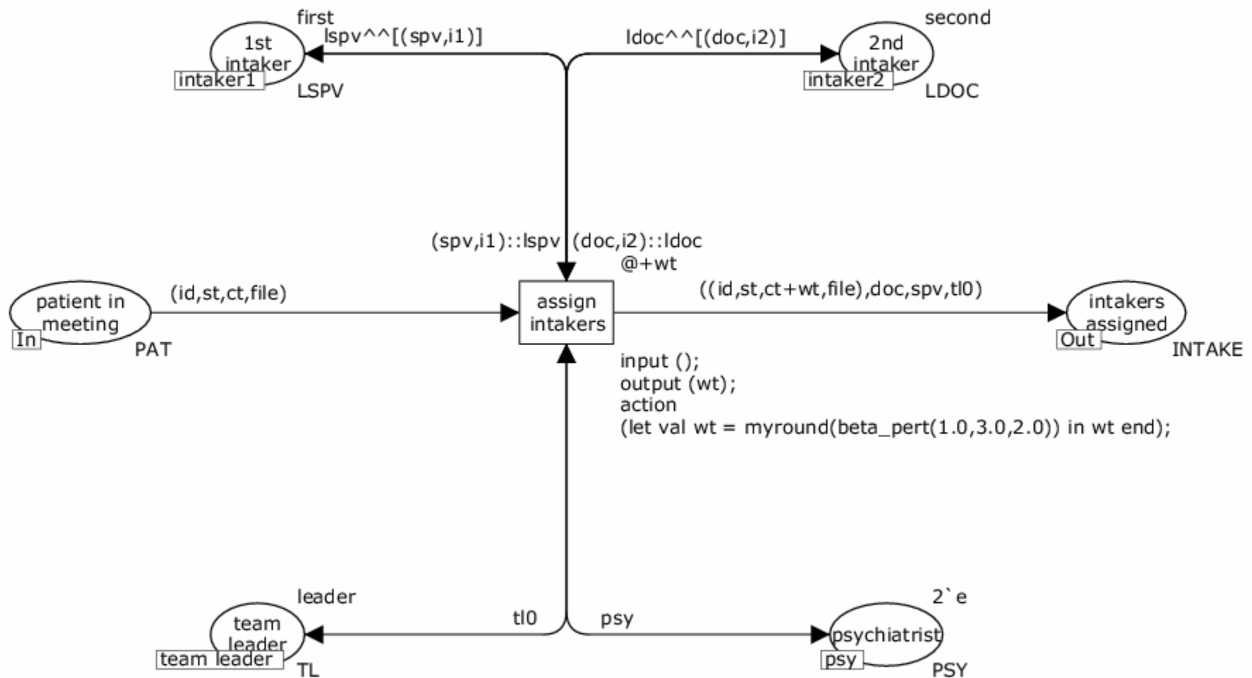


Figure 8 Page 'assign intakers' in the redesigned model

The results of the simulation for the flow time are displayed in Table 3.

	flow time		
	LOW	AVG	HIGH
	432937,52	434554,65	436171,79
Minutes	4329,38	4345,55	4361,72
Hours	72,16	72,43	72,70
Days	9,02	9,05	9,09

Table 3 Flow time of the intake process (redesign)

The flow time of an average case will drop by about 3,8 working days, this is the expected time a new case has to wait before it is assigned (half a working week) and the queuing time for the first intaker, the second intaker or both. The reduction is about 30 %. The reduction of the total service time is 17 minutes, a 10% reduction, as can be calculated from the results in Table 4. Both improvements are significant as the confidence intervals of the designed and the redesigned calculations do not overlap.

	service time		
	LOW	AVG	HIGH
	15639,48	15690,13	15740,79
Minutes	156,39	156,90	157,41
Hours	2,61	2,62	2,62
Days	0,33	0,33	0,33

Table 4 Service time of resources in the intake process (redesign)

5 Discussion

In this section, we discuss the suitability of CPN models for Business Process Redesign purposes based on the case study as described in this paper. We also discuss the suitability of CPN Tools (Kristensen *et al*, 1998), (University of Arhus, 2005b) which we have used to create our CPN models and to perform the simulations.

Our first conclusion is that we managed to model the business process and to carry out all the simulations that we wanted to do. Petri nets, or more specifically CPN Tools, helped us to model the business process in such a way that it resulted in an executable and exact specification of the business process. The monitor's version of CPN Tools helped us to collect the simulation data, and to calculate 95% reliability intervals.

The second conclusion, however, is that the resulting CPN model appears to be too complex, especially if we compare it to the same process modelled in Protos (see Figure 10). Note that this model is not a Petri net itself, though it can be mapped onto a Petri net. The part that we could model quite straightforwardly in CPN Tools is the control flow of the business process. The triggers we needed in the process model are not very nice from an end-user perspective, and also introduce some complexity. The main cause for the complexity in the model comes from the modelling of resources.

The time trigger for the Wednesday meeting demands that all cases before the transition with the trigger pass this transition, thus requiring a list. As a consequence, we needed some additional places to convert lists into single tokens. The model in Figure 6 is easier to understand than the model in Figure 8.

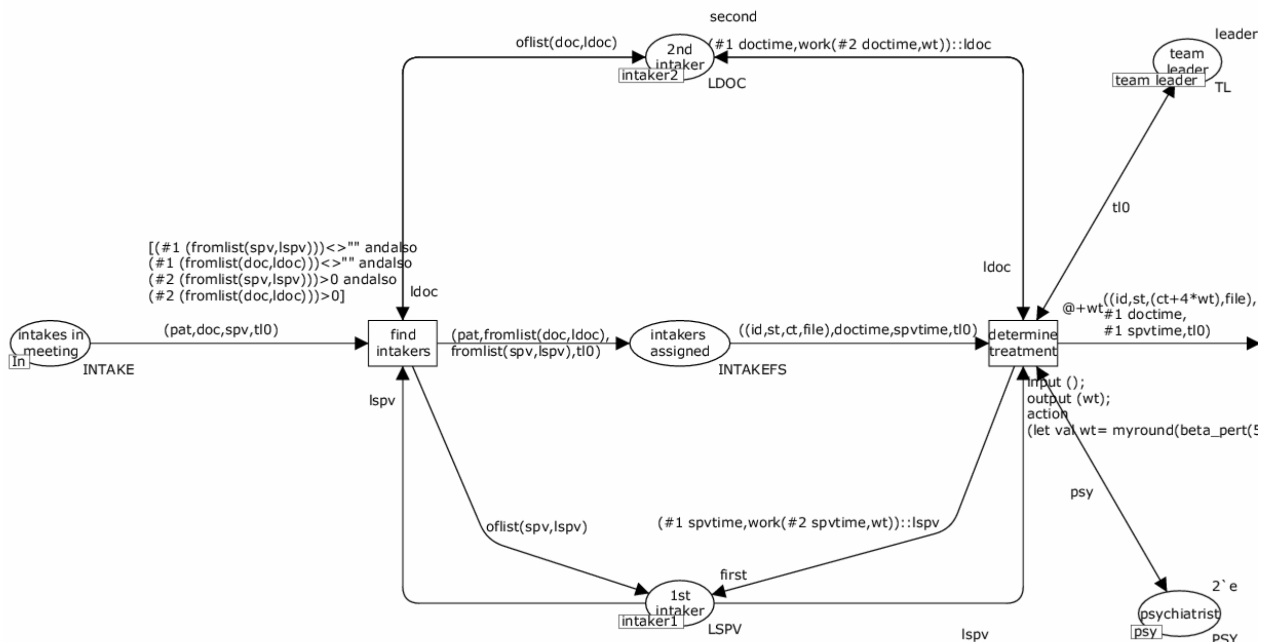


Figure 9 Page 'determine treatment'

The modelling of resources required explicit modelling of a clock, the change of a day, and resetting of the available working time for all resources. To be able to reset the working time of all resources, we used lists. This introduced the necessity to find the correct resources for each particular step in the business process and to check the available working time of that

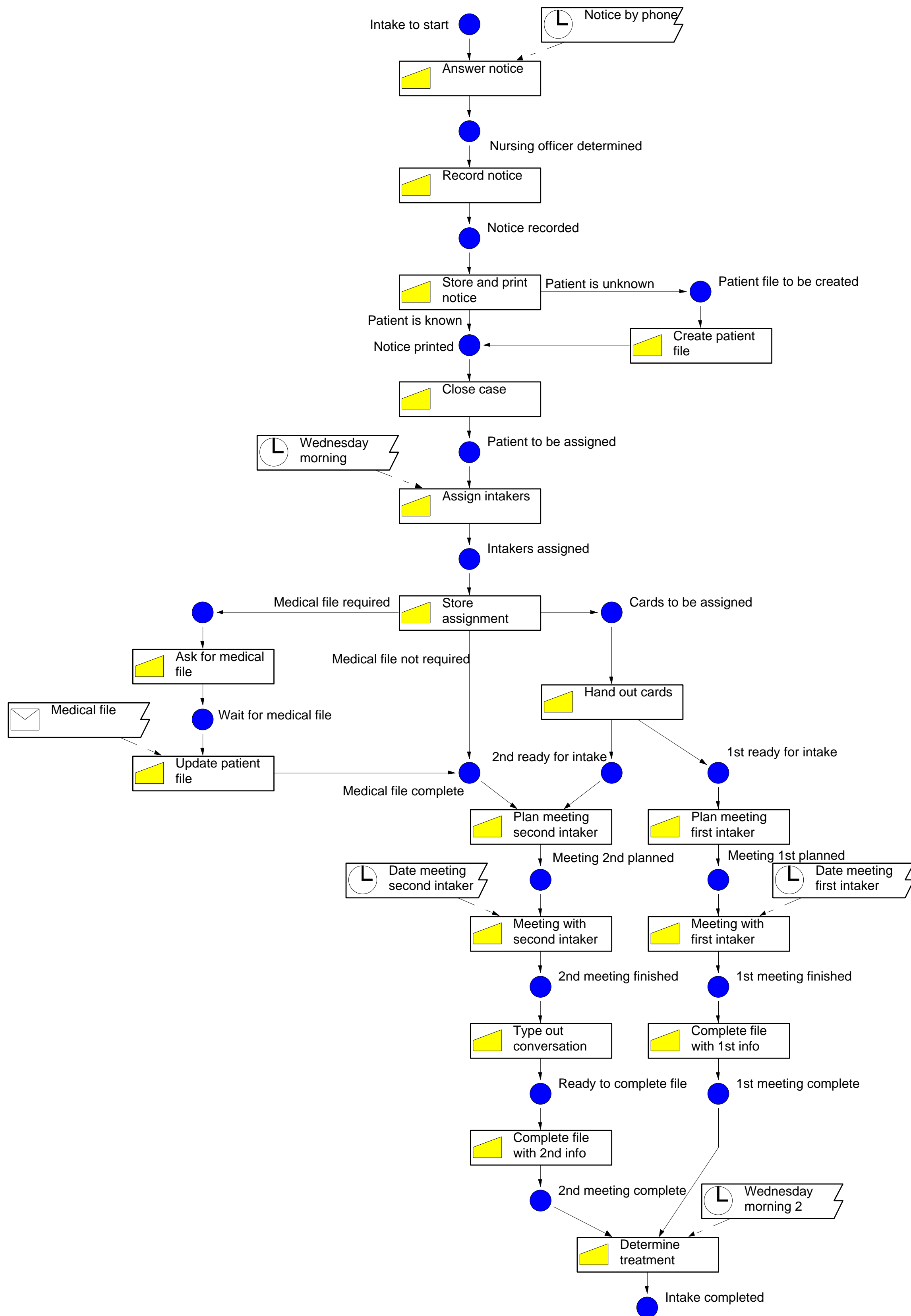


Figure 10 Protos model of the intake workflow

resource at that point in time. This resulted in additional transitions and very long arc inscriptions and guards. An example can be found in the model of Figure 9.

This additional complexity is a drawback for the modeller, because it creates a lot of time-consuming work that is not directly related to the business process that is being modelled, it increases the possibility to make mistakes in the model and it hampers a change of the model. The additional complexity is a severe problem for the end-customer who should be able to validate the models and who should decide which (re) design will serve his purposes best.

A last issue that we want to discuss is the modelling of service time for a particular case. In the model we added a parameter to the colour of a case to calculate the service time manually. In a sequential process this is not a problem, though in processes with a lot of parallel paths this becomes complex and is thus likely to introduce mistakes.

Other tools

An important reason to start modelling business processes in CPN Tools is based on the fact that we are not very happy with other tools we know so far. The Protos models are quite readable for business users, though these are only suited for modelling and cannot be used for simulation purpopses. To overcome this, an interface has been built to export Protos models into ExSpect models to run simulations. Another option, of course, is to build ExSpect models from the start of a project.

ExSpect models are, like CPN Tools models, based on colored Petri nets. As such, these models have the same advantages and disadvantages caused by the CPN language. The expressive power, mathematical foundation and reliability of simulation results is very good. Those functions that are not made available in the tool can be built by the user in GCL (for ExSpect) or ML (for CPN Tools). However, the models become easily quite complex and are difficult to understand by business users. Furthermore, the language does not support you how to develop particular models or constructions (like, e.g., a tool like Arena does). In the view of modelling business processes, and their application in redesign projects, ExSpect has some advantages over CPN Tools. In ExSpect, scripts behind the graphic model hide details, a number of building blocks are already available, and the soundness of models can be evaluated in Woflan. An important advantage of CPN Tools over ExSpect is the simulation time, which is much shorter and as such suitable for interactive workshops with end users.

An industrial simulation tool like Arena focuses on understandability of the resulting models. Important drawbacks of this tool are the expressive power, the handling of parallel processes, the building blocks that hide almost everything and, most important, it is hard to build additonal components. We see here a trade of between flexibility (in CP nets) and modelling support (in Arena). Finally we can refer to simulation engines in workflow tools. We found here, in general, that it is not possibly to configure your parameters, the concept of reliability does not exist and it is not possible to decide yourself what you would like to measure.

Conclusion

Being a modeller, we see the advantages of modelling business processes in CP Nets. However, to provide a tool that appeals to owners of business process, there is a gap which we hope that can be bridged by a tool like CPN Tools. A process model that is much easier to understand is, for example, the Protos model in Figure 10. In this model, you mainly see the control flow of the business process and it is about one page in size. For CPN Tools, we see the benefit of having a number of building blocks.

- Modelling of resources. Such a building block should include modelling of a resource (its ID, its role, working time, etc.) and its functions (reset, check working time, get(id),

store(id), etc.) in such a way that the handling of lists is hidden for the end-user and modeller.

- Modelling a global clock.
- Modelling triggers; this includes time triggers and triggers based on external events.
- Measuring service time of a case.
- Handling of batches. The application of a batch in a business process requires the modelling of a list in the corresponding CPN model. A function "take all tokens" from a place would decrease the complexity of the model.

A consequence of not having such standard building blocks is a more complex graphical model. The levels 1 and 2 of the CPN model don't contain too much information, but the models on level 3/4 certainly do.

As we pointed out, we want to evaluate the applicability of the resulting models for communication purposes. When redesigning a business process, a lot of communication is taking place between the owner of the business process and the modeller. The most important means of communication is the simulation model. At first sight, Petri nets are considered to be complex and difficult to understand by business users, see for example (Sarshar *et al*, 2005). In line with the evaluation above with respect to the lack of standard building blocks, the application of CPN Tools in a BPR approach especially this aspect deserves attention. The two highest levels of the developed model will serve such a purpose, however, the lower levels show too many details related to CPN Tools or Petri nets in general. Further research should reveal whether this would meet business requirements.

6 Conclusions and further research

Simulation of the redesigned business process comes into play if one has to decide whether the redesign is better than the previous process design, or if one needs to compare alternative redesigns. Usually, the characteristics of the process are such that a purely analytical performance evaluation is not feasible.

This paper shows the applicability of coloured Petri nets, especially CPN Tools, in such a redesign approach. In line with previous research, we conclude that coloured Petri nets are well suited to model and simulate business processes. Applying Monitors in addition to the regular CPN Tools package helped us to carry out the simulations in such a way that we were able to carry out statistical analysis, enabling us to compare the performance of different models. An important drawback of our approach is that modelling resources in a business process is quite laborious and results in complex constructions, which are hard to communicate with those people who should be able to evaluate the effect of a particular redesign.

To bridge the gap between current CPN models and the models that could be helpful in a BPR project, the availability of building blocks would be helpful. This enables the possibility to hide those parts of the model that are not directly related to the business process.

In our research, we continue with the modelling of business processes in CP Nets. The simulations of designs and redesigns are used to gather data that could quantify the application of particular redesign rules, such as the case-based work heuristic as mentioned in the case study. Current literature does not provide concrete clues when to apply this well-known rule, and which benefits may be expected. We want to compare differences in service time and flow time, but also differences in costs, which, amongst others, can be expressed in resource

utilisation. In this paper we did not calculate resource utilisation. Further research should reveal whether the monitors can handle the calculation of token availability based on tokens with time stamps or we need to model this in the Petri Net. In combination, we can use the simulation data to evaluate and quantify several redesign heuristics.

Acknowledgement

We would like to thank Kurt Jensen and Lisa Wells from the University of Arhus. We appreciate that the monitors version of CPN Tools has been made available for us. We also thank Mariska Netjes, Boudewijn van Dongen and Eric Verbeek for their support in the modelling process.

Reference List

1. Aalst, W.P.M.v.d. and Hee, K.M.v. (2002). *Workflow Management: Models, Methods, and Systems*. Cambridge: MIT Press.

2. Ardhaldjian, F. and Fahner, M. (1994). Using simulation in the business process reengineering effort. *Industrial Enginering* 26(7): 60-61.

3. Brand, N. and Kolk, H.v.d. (1995). *Workflow Analysis and Design*. Kluwer Bedrijfswetenschappen (in Dutch).

4. Davenport, T.H. and Short, J.E. (1990). The new industrial engineering: Information Technology and Business Process Redesign. *Sloan Management Review* 31(4): 11-27.

5. Desel, J. and Erwin, T. (2000). Modelling, Simulation and Analysis of Business Processes. In: *Business Process Management: models, techniques and empirical studies,* W.P.M.v.d.Aalst (ed), Lecture Notes in Computer Science 1806, pp. 129-141. Springer Verlag, Berlin.

6. Hammer, M. (1990). Reenigineering Work: Don't automate, obliterate. *Harvard Business Review* 68(4): 104-113.

7. Hammer, M. and Champy, J. (1993). *Reengineering the Corporation; a manifesto for Business Revolution*. New York: Harper Business.

8. Kristensen, L.M., Christensen, S. and Jensen, K. (1998). The Practitioner's Guide to Colored Petri Nets. *International Journal on Software Tools for Technology Transfer* 2 98-132.

9. Makajic-Nikolic M., Panic B. and Vujosevic B. (2004). Bullwip effect and supply chain modelling and analysis using CPN Tools. In: *Proceedings of the CPN'04 Workshop*.

10. Mitchell B., Kristensen L.M. and Zhang L. (2004). Formal Specification and State Space Analysis of an Operational Planning Process. In: *Proceedings of the CPN'04 Workshop*.

11. Netjes M., Aalst W.P.M.v.d. and Reijers H.A. (2005). Analysis of resource-constrained processes with Colored Petri Nets. In: *Submitted to CPN Workshop 2005*.

12. Reijers, H.A. (2003). *Design and control of workflow processes: Business Process Management for the service industry*. Berlin: Springer Verlag.

13. Reijers, H.A. and Limam Mansar, S. (2005). Best Practices in Business Process Redesign: An Overview and Qualitative Evaluation of Successful Redesign Heuristics. *Omega: The international Journal of Management Science* 33(4): 283-306.

14. Sarshar K., Dominitzki P. and Loos P. (2005). Empirical Evaluation of EPC and Petri Nets from the End-User Perspective. In: *Proceedings of the BPM'05 Conference*.

15. University of Arhus, Department of Computer Science (2005a) *CPN Tools, Industry Examples* www.daimi.au.dk/CPnets/intro/example_indu.html Arhus, Denmark.

16. University of Arhus, Department of Computer Science (2005b) *CPN Tools, main page* http://wiki.daimi.au.dk/cpntools/cpntools.wiki Arhus, Denmark.

Towards a Pattern Language for Colored Petri Nets

Nataliya Mulyar and Wil M.P. van der Aalst

Department of Technology Management, Eindhoven University of Technology
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands.
{n.mulyar, w.m.p.v.d.Aalst}@tm.tue.nl

Abstract. Experienced Petri net modelers model in terms of patterns, just like object-oriented programmers use the design patterns of Gamma et al. So far there is no any structured collection of patterns for Colored Petri Nets. We have empirically collected 34 patterns in Colored Petri Nets and documented them in the pattern format. The patterns focus on the interplay between data- ow and control- ow, (i.e. the essence of Colored Petri Nets), and have been modeled using CPN Tools. The goal of the patterns is to assist and train inexperienced modelers, and to serve as a domain language for communicating problems and solutions. In this paper, we give a summary of the CPN pattern language and give an overview of the patterns collected. In addition, we examine the clustering of patterns and the different types of relationships between the CPN patterns.

1 Introduction

Process-Aware Information (PAI) systems [16], i.e. systems that are used to support, control, and monitor business processes, are typically driven by models of di erent perspectives, i.e. process, organization, data, etc. In order to e ciently build a feasible model with the help of a PAI system (e.g. WFM software), all dimensions of requirements put on the system from process, data, resources and other perspectives, must be well understood. Developers working in the same domain experience similar di culties while solving the same kind of problems. How to solve a problem? What are the advantages and disadvantages of possible solutions? Which solution to choose and how to realize the selected solution? These are the questions which every developer needs to answer. Since problems to be solved are often non-unique, i.e. they recur in many systems, developers often spend their time solving problems which may already have existing solutions.

A *pattern language* is one of the possible means to help developers to build their models e ciently, while avoiding reinvention of already existing solutions of problems. Pattern languages are based on experience; they express sound solutions for problems frequently recurring in a certain domain in a *pattern*

format. Knowing a problem at hand, a developer can look up a solution for the problem in the pattern catalog, while spending less effort on the development and also ensuring the soundness of a solution.

The work reported in this paper is part of the *Workflow Patterns Initiative* `www.workflowpatterns.com`. In the context of this initiative, we have developed control-flow patterns [6], data patterns [35], and resource patterns [34]. These patterns focus on the different perspectives [24] of PAI systems. In this paper we do not necessarily limit ourselves to workflow or PAI systems. Instead we *focus on the interplay between control flow and data flow*. To do this, we use a specific implementation language: *Colored Petri Nets* (CPNs). In our view, a good understanding of the interplay between control flow and data flow is foundational to PAI systems. Moreover, (colored) Petri nets have shown to be a solid basis for the modeling, analysis, and enactment of workflows [1, 3, 36].[1]

Based on the expert knowledge and an analysis of existing models and literature, we identified 34 patterns that focus on the interplay between control flow and data flow and can be represented in terms of Colored Petri Nets. On the one hand, the patterns we discovered are *implementation patterns*, i.e. they are mainly oriented on model developers who are working with CPN Tools. In particular, the CPN patterns support developers with sound solutions for problems frequently recurring during modeling. Therefore, these patterns are CPN *language-specific*. On the other hand, since CPN is a modeling language, which is often used for the design and modeling of dynamic systems with elements of concurrency, these patterns can be also considered as *design patterns*, which grasp certain problems on the level of model design and offer visualized solutions by means of CPN. Similarly to the 23 design patterns of Gamma [19], the CPN patterns also systematically name, motivate, and explain solutions for generic design problems. However, due to major differences in concepts of object-orientation and Petri Nets, and validity in the CPN context, we will refer to the CPN patterns as implementation patterns.

The remainder of this paper is organized as follows. First, we introduce a set of concepts, central to the CPN patterns and define the scope of the patterns in the context of Petri Nets (Section 2.1). Next, in Section 2.2, we give a brief introduction into the world of patterns, and introduce the pattern format which we selected for the description of the 34 identified patterns. In Section 3, we give an example of a CPN pattern using this format. Then, we introduce the CPN pattern language (Section 4). We not only examine relationships between the discovered patterns to enable easy navigation through the CPN pattern catalog (Section 4.2), but we also classify patterns into clusters in order to simplify the selection of a suitable pattern (Section 4.3). We conclude the paper by discussing related work and future work (Section 5 and Section 6).

---

[1] Although (colored) Petri nets form a good foundation for workflow languages and PAI systems, one could argue that they are too low level as an end-user language [4]. Therefore, we propose Petri nets as a theoretical basis and use higher-level languages such as [5] as the end-user language.

2 Preliminaries

In this section we briefly introduce colored Petri nets and discuss existing patterns languages.

2.1 Colored Petri Nets

Colored Petri Nets (CPNs) [25, 26] extend the classical Petri Nets [15] with colors (to model data), time (to model durations), and hierarchy (to structure large models). Like in classical Petri Nets, CPNs use three basic concepts: transition, place, and token. We will use the terms “event”, “task”, “actor” and “transition” interchangeably, as well as “token” and its mapping on an “object”. We do not refer to the definition of an object from object-oriented programming, but generalize it in such a way that by “token” or “object” we can refer to any of [22]:

- Physical objects, i.e. a chair, a stool, a table, etc;
- Conceptual objects, i.e. policies, insurances, etc;
- Information objects, i.e. anything what can be manipulated by a human or a system as a discrete entity.

Whenever a pattern operates with a specific type of objects, we will specify the type (called color set in CPN) explicitly. For gathering CPN patterns, we concentrate on *discrete dynamic systems*, which are systems with a certain state at any moment of time and a sequence of events which bring a system from one state to another. The examples of discrete dynamic systems are workflow management systems, distributed databases, decision support systems, e-mail systems, payment systems, etc. Discrete systems are made out of actors, which are active components, and objects, which are passive components. Actors consume and produce objects [22]. Actors can be machines, humans, networks of other dynamic systems, etc.

A place is a location where tokens reside. A place can be considered as a temporary or persistent data storage, e.g. either containing a variable or a constant number of tokens at any time.

Note that although the basic rules of classical Petri nets are still valid in the context of CPNs, we do not elaborate on basic control-flow patterns and focus on the extensions of PN by *color* and *time* (in particular the interplay between control flow and data flow). Hence, we abstract from the extension with hierarchy (e.g., the substitution transitions). Figure 1 visualizes the scope of the pattern language presented in this paper.

There are many variants of colored Petri nets, i.e., Petri net models with color and time. Consider for example the different tools: Design/CPN, CPN Tools, ExSpect, ALPHA/Sim, Artifex, GreatSPN, PEP, Renew, etc. The patterns are tool independent. Nevertheless, we need to select a specific language/tool for the examples used to describe the patterns. For this purpose, we selected *CPN Tools* [14]. The language used by CPN Tools is the de facto standard. Moreover, all

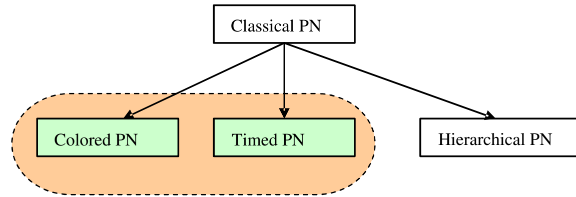


Fig. 1. Scope of CPN patterns: The focus is on color and time while abstracting from hierarchy.

the patterns that we have collected can be downloaded from [28] and executed using CPN Tools. Note that most pattern languages use a specific language to represent examples, for example in [19] both C++ and Smalltalk are used.

2.2 Patterns

Nowadays, there is a generic understanding of what a pattern is, i.e. it is a solution to a problem in a certain context. Note that originally the concept of a pattern was introduced by Christopher Alexander in [9], who wrote:

The pattern is, in short, at the same time a thing, which happens in the world and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process that will generate this thing.

The notion of a pattern language, introduced by Alexander in [8], is similar to the notion of a language as recorded in the Merriam-Webster Dictionary as a "formal system of signs or symbols including rules for the formation and transformation of admissible expressions". If a word is a central entity of a language, then a pattern is a central entity of the pattern language. Similar to the rules describing the use of words in sentences, patterns also have rules associated with them. As such, pattern rules describe relations between the patterns and indicate how one pattern can be combined with other ones. Furthermore, a pattern language can serve as a systematic means of communicating problems and solutions between colleagues working in the same field/domain.

Since the definition of a pattern by Christopher Alexander, different types of patterns in different application domains have been described. This has resulted in a set of pattern languages each of which addresses different aspects of organization, software development, analysis, etc. Due to the differences in the types of problems and the solution means in different application fields and domains, there is an ongoing discussion concerning the suitability of the *pattern format* introduced by Alexander for documenting the patterns.

Since there are multiple views on how to document the patterns and no consensus in the discussions related to selection of a single pattern format has yet been achieved, we took as a basis the pattern format of Gamma [19], and adjuisted it in order to fit our purposes. Every CPN pattern adheres to the following pattern format:

Pattern format

- *Pattern name.* This is an identi er of a pattern which captures the main idea of what the pattern does.
- *Also known as.* This section lines out the alternatively used names for the *Pattern name.*
- *Intent.* This section describes in several sentences the main goal of a pattern, i.e. towards which problem it o ers a solution.
- *Motivation.* This section describes the actual context of the problem addressed and why the underlined problem needs to be solved.
- *Problem description.* This section presents the problem addressed by the pattern. For the sake of clarity, the problem is explained by using a speci c example. The majority of the patterns contain examples which are also illustrated by means of CPN diagrams.
- *Solution.* This section describes possible solutions to the problem. Note that a single problem addressed by the pattern can be solved in several ways, depending on the requirements and/or context in which the pattern is to be applied. Since multiple solutions are possible, we consider every solution separately and for each of the solutions we include an implementation subsection.
- *Implementation of Solution.* This is a part of the *Solution* section, which illustrates how to implement the described solution in CPN Tools. The implementation part shows not only the graphical representation of the pattern with CPN, but also describes how to integrate this solution into the example considered in the *Problem description* section. A solution may have several implementations. The presented implementations may not be the only way to implement a solution correctly. One should select an implementation depending on the context within which the pattern is to be applied. Note that correctness of the solution is not guaranteed if a tool di erent from CPN Tools is used for implementation purposes.
- *Applicability.* This section describes the typical situations in which the pattern can be applied.
- *Consequences.* This section outlines what the possible advantages/disadvantages of using the pattern are. In case if the pattern supplies several solutions, this section elaborates on the di erences between them.
- *Examples.* This section lists several examples demonstrating the use of the pattern in practice.
- *Related Patterns.* This section speci es relations of the pattern to other patterns.

3 Example: AGGREGATE OBJECTS Pattern

Before de ning our pattern language that also relates patterns to one another, we present one of the 34 patterns. Like all the other patterns, it is described using the pattern format described in the previous section.

Pattern: AGGREGATE OBJECTS
Also Known As:

Intent: to allow the manipulation of a set of information objects as a single entity.

Motivation: In many cases, it is natural to represent an information object (e.g., an order, a car, a message) as a single entity, i.e. there is a one-to-one correspondence between objects in a “real system” and tokens in the model. However, sometimes it is necessary to aggregate objects into one token, thus referring to the collection of objects as a single entity.

Problem Description: Figure 2 illustrates the problem addressed by this pattern. In the original model, place object is of type T and transitions `put` and `get` add and remove tokens from this place. Note that each token corresponds to an object.

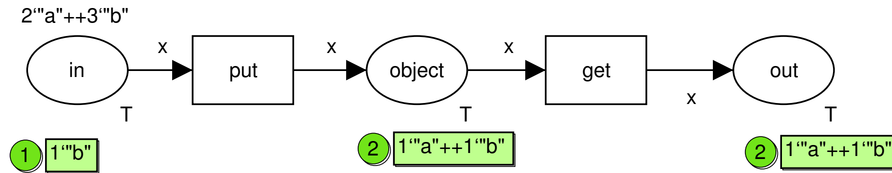


Fig. 2. Example used to explain the various problems.

Suppose that it is necessary to perform an operation from the following list:

- Count the number of objects in place object;
- Select an object from place object with some property relative to the other objects (e.g., the rst, the last, the smallest, the largest, the cheapest, etc.);
- Modify all objects in a single action (e.g., increase the price by 10 percent);
- (Re-) move all objects in one batch (e.g., remove a set of outdated les, items, etc. at once, rather than one by one).

None of these operations is possible in the diagram shown above. Note that it is only possible to inspect one token at a time and this is a non-deterministic choice. Moreover, this choice can be limited by transition guards and arc inscriptions, but it is memoryless and not relative to the other tokens in the place. This makes it very di cult or even impossible to realize the mentioned aspects.

Solution: In order to allow the manipulation of a set of information objects as a single entity, aggregate the objects into a single token of “collection type”. [2]

[2] Note that we assume an interleaving semantics.

Implementation of Solution: The list of instructions below describes how to implement the AGGREGATE OBJECTS pattern (see Figure 3).

- Modify the type `T` of place `objects`, where multiple objects may reside, to the collection type `LT` (e.g., list, set, bag). In this example the collection type list is chosen: `color LT = list T;`.
- Replace arcs between transitions `put` and `get` and place objects by bi-directional arcs with the following inscriptions. An arc which supplies an object to the collection has an inscription `x::l`, which adds an object `x` of type `T` to the list `l`. Return the current list `l` back to transition put. Similar, in order to get an object from the collection use `x::l` and return the changed list. The described behavior represents LIFO (last-in- rst-out) ordering.

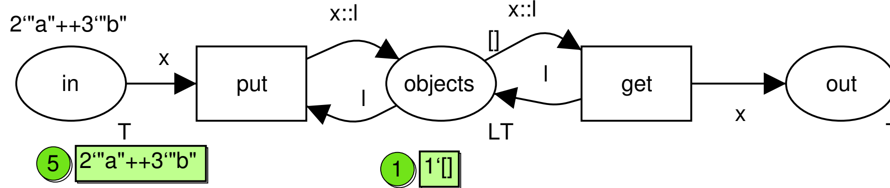


Fig. 3. Model after applying the pattern.

By introducing a collection type, it becomes possible to refer to the collection of objects as a single entity and perform operations on multiple objects contained in the collection at once. Several examples in Figures 4 and 5 show how to implement some operations from the ones mentioned in the Problem description section by extending the net presented in Figure 3.

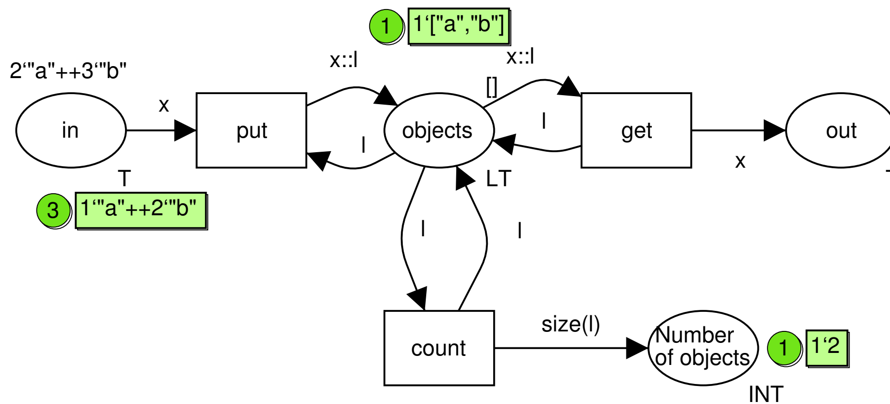


Fig. 4. Example illustrating how place `objects` can now be used.

Figure 4 shows how to calculate the size of the collection, i.e. the number of objects the collection contains. Note that there is always precisely one token in place `objects` representing all objects. Transition `count` takes the current list of objects and sends the size of the list to place `Number of objects`. Note that a function `size(l)` for determining the size of the collection is prede ned and available in the CPN Tools.

It is also possible to select an object from place objects with some property relative to the other objects. For example, the object represented with the rst name can be obtained by the transition `select` as shown in Figure 5.

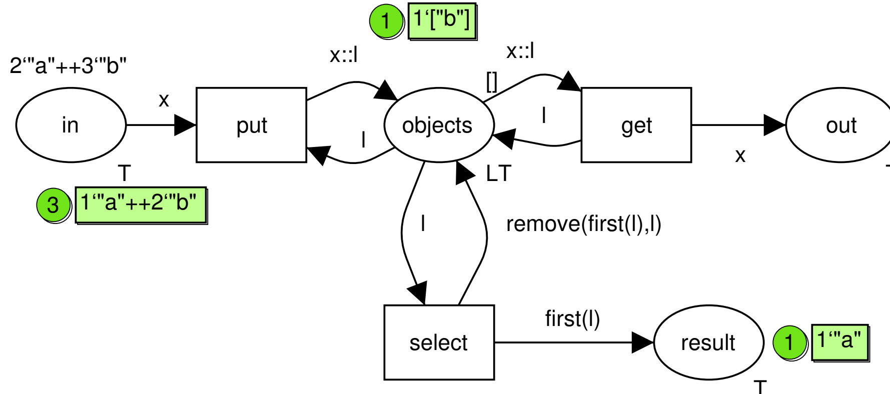


Fig. 5. Another example illustrating how place `objects` can now be used.

Function `first` selects the right object while function remove is used to remove the object, i.e.,

```
fun f(x:T,l:LT) = if l = [] then x else if x< hd(l) then
    f(x,tl(l)) else f(hd(l),tl(l));
fun first(x::l : LT) = f(x,l) |
    first([]) = "null";
fun remove(x,[])=[] |
    remove(x,y::l) = if x=y then l else y::remove(x,l);
```

In a similar way, it is possible to modify all objects in a single action (for instance, increase the price by 10 percent) and to remove all tokens (simply by returning a token with a value `[]`).

Applicability: Apply this pattern to

- Organize multiple objects into a collection.
- Perform an operation on a group of objects or the whole collection at once.

Consequences: In principle, this pattern is not concerned with the order in

which tokens are taken from the collection. The example used in the implementation section uses last-in-first-out ordering (see LIFO QUEUE pattern).[3]

Nevertheless, if the problem of ordering is relevant, one should apply an extension of this pattern by adding the QUEUE pattern, or one of its specializations.

Note that although some of the functions to manipulate the collection of objects are already predefined in CPN Tools, applying special kinds of operations requires the writing of corresponding function(s) from scratch.

Examples:

- The salary administration of a university divided employees into different groups: students, PhD students, and professors. All PhD students got a salary increase of 10%. The salary administration does not need to adjust the salary slips for every PhD student individually, but does it in one-step by increasing the salary of the whole group.
- The documents are collected and organized in one file. This allows the whole file to be taken and sent for processing elsewhere, keeping the documents structured and grouped.

Related Patterns: This pattern is extended by the QUEUE pattern.

4 CPN Pattern Language

In this section, we introduce the CPN pattern language by listing the names and intents of the discovered patterns. Next, we analyze relationships between the CPN patterns, and organize them into a relationship diagram, which allows navigation through the pattern catalog for identifying related patterns. Furthermore, we classify the patterns into categories in order to simplify the process of selecting a pattern from the CPN pattern catalog.

4.1 Overview of CPN Pattern Language

In this section we give an overview of the CPN patterns. In Tables 1 and 2 we present only the names and intents of the patterns; for further details the reader is referred to [27]. Patterns listed may belong to the same classes, and have similarities in their intents, problems and solutions. These relationships between patterns are not covered explicitly in the tables, but are discussed in Section 4.2.

[3] Note that this pattern refers to other patterns like the LIFO QUEUE pattern. These have not yet been discussed. An overview of all 34 patterns in given in Section 4.

Pattern name	**Intent**
ID Matching	to make identical information objects distinguishable
ID Manager	to ensure uniqueness of identi ers used for distinguishing identical objects
Aggregate Objects	to allow manipulation of a set of information objects as a single entity
Queue	to allow manipulation of the queued objects in a strictly speci ed order
FIFO Queue	to allow manipulation of objects from the collection in a strictly speci ed order such that an object which arrived rst is consumed rst
LIFO Queue	to allow manipulation of objects from the collection in a strictly speci ed order, such that the mostly recently added object is retrieved rst
Random Queue	to allow manipulation of objects from the collection such that objects are added to the queue in any order, and an arbitrary object is consumed from it
Priority Queue	to allow manipulation of objects from the collection in the order of the objects' priority
Capacity-bounding	to prevent over-accumulation of objects in a certain place
Inhibitor Arc	to support "zero"-testing of places
Colored Inhibitor Arc	to support "non-containment" property of places
Shared Database	to enable centralized storage of data shared between multiple transitions, supporting different levels of data visibility (i.e. local, group, or global)
Database Management	to specify the interface of accessing data, stored in a shared database for read-only and modi cation purposes
Copy Manager	to make data stored in the shared database available at other locations for local use, maintaining the consistency of data in all places
Lock Manager	to synchronize access to shared data by means of exclusive locks
Bi-lock manager	to synchronize access to shared data for reading and writing purposes by means of shared and exclusive locks
Log Manager	to record the information about actual process execution by means of a data log
BSI Filter	to prevent data non-conforming to a certain property from passing through
BSD Filter	to prevent data non-conforming to a property involving the state of an external data-structure, from passing through
NBSI Filter	to lter out data ful lling a certain property while avoiding accumulation of non-conforming data in the lter input place
NBSD Filter	to lter-out data non-conforming to a property, involving the state of an external data-structure, while avoiding accumulation of non-conforming data in the lter input
Translator	to enable coordinated communication between two actors with originally different data formats
Asynchronous Transfer	to allow transportation of data from one location to another, while avoiding the sender to block
Synchronous Transfer	to allow transportation of data from one location to another, ensuring that an actor, which posted a request, is blocked until it receives the requested information
Rendezvous	allow multiple actors to broadcast and discover data objects concurrently
Asynchronous Router	to enable asynchronous transfer of data from a single source to a dedicated target, providing loose coupling between the source and targets connected to it

Table 1. Summary of CPN patterns

Pattern name	Intent
Asynchronous Aggregator	to provide a holistic view of data, produced by multiple unrelated sources through asynchronous data aggregation
Broadcasting	to allow broadcasting of data from a single source to multiple targets, while avoiding direct dependency between them
Redundancy Manager	to prevent transfer of duplicated data between loosely-coupled actors who communicate asynchronously
Data Distributor	to support parallel data processing by distributing data between several independent actors
Data Merger	to compose a single information object out of several smaller ones when all parts required for composition become available
Deterministic XOR-split	to allow at most one transition out of several possible to execute, based on ful llment of mutually excluding data conditions
Non-deterministic XOR-split	to allow any transition out of several possible, but satisfying the same data condition, to execute
OR	to allow any number of tasks to be selected for execution based on the ful llment of a certain data condition

Table 2. Summary of CPN patterns (Cont.)

4.2 CPN Pattern Relationships

The 34 CPN patterns that we have identi ed, together with the relationships between them, form a pattern language. In order to classify the CPN patterns we examined the nature of relationships between the patterns. We used three types of primary relations: *specialization of a problem*, *use in a solution*, and *extension of an implementation*; and two types of secondary relations: *problem similarity*, and *combination of solutions* to describe the pattern relationships. Some of the relationship types are based on Zimmer's classi cation [39].

The main purpose of this classi cation is to provide a holistic view on the catalog of patterns, providing a means for a user to select a number of patterns and to determine how the patterns can help in solving a given problem. The selected types of relationships can help to trace other patterns related to a chosen pattern, thus allowing the estimation of an overall problem complexity, the tradeo s made, and compare the chosen pattern with other similar patterns, in order to select an optimal solution for a problem in the given context.

Figure 6 shows some of the relationships between the various patterns. The graphical representation and the text depict the type of a relationship. To understand the diagram, we rst need to de ne the di erent types of relationships.

Primary relations

Problem-oriented relation
Pattern A is a *specialization* of the more generic pattern B. Speci c pattern A, which deals with a specialization *of the problem* that generic pattern B addresses, has a similar but more specialized solution than pattern B. Pattern A includes

all the properties of pattern B, but adds further restrictions by adding some specialized characteristics. Note that a specialization often adds more context to the problem thus making it less generic.

Solution-oriented relation
Pattern A *uses* pattern B *in its solution*. When building a solution for a problem addressed by pattern A, one sub-problem is similar to the problem addressed by pattern B. Thus, the *solution of pattern B* is a composite part of the *solution of pattern A*. Whenever pattern A is used, pattern B should also be considered, since it makes a part of A.[4] All instantiations of pattern A use pattern B. Some example relationships: Lock Manager uses ID Matching, Asynchronous Router uses Asynchronous Transfer.

Solution implementation-oriented relation
Pattern A *syntactically extends* pattern B. Pattern A addresses a set of requirements to have more or slightly di erent functionality than the pattern B addresses. However, this is the implementation of B, which is syntactically extended by A, rather than a problem or a solution. For example, the implementation of Non-Blocking State-Independent Filter extends implementation of Blocking State-Independent Filter.

Secondary relations

Problem similarity
Pattern A is *similar* to pattern B. Pattern A addresses a *problem similar* to the one addressed in pattern B. Patterns A and B can be considered as alternatives of each other; therefore, one can compare them and select the one which ts the problem best.

Combinable solutions
Pattern A can be *combined* with pattern B. Neither of the patterns is a part of the other. Combining the solution of pattern B with the solution of pattern A can help to solve a more complex problem than a single pattern solves in isolation. Use this relation to nd out other patterns, which can be used in addition to pattern A. For example, the Shared Database can be combined with Copy Manager; Asynchronous Aggregator can be combined with Aggregate Objects.

Figure 6 shows the ve types of relationships. The listing of the primary relationships is intended to be complete while the secondary relationships depicted only represent typical examples. The solid arrows represent primary relationships while the dashed lines represent secondary relationships. A problem-oriented relation is labelled "is specialization of". A solution-oriented relation is labelled

[4] Since a pattern may have multiple solutions, the relationship may need to refer to a speci c solution. For the sake of clarity we use the mnemonics "s1", "s2" and "s3" as identi ers for referring to the rst, second, and third solution and to make relations between pattern solutions explicit.

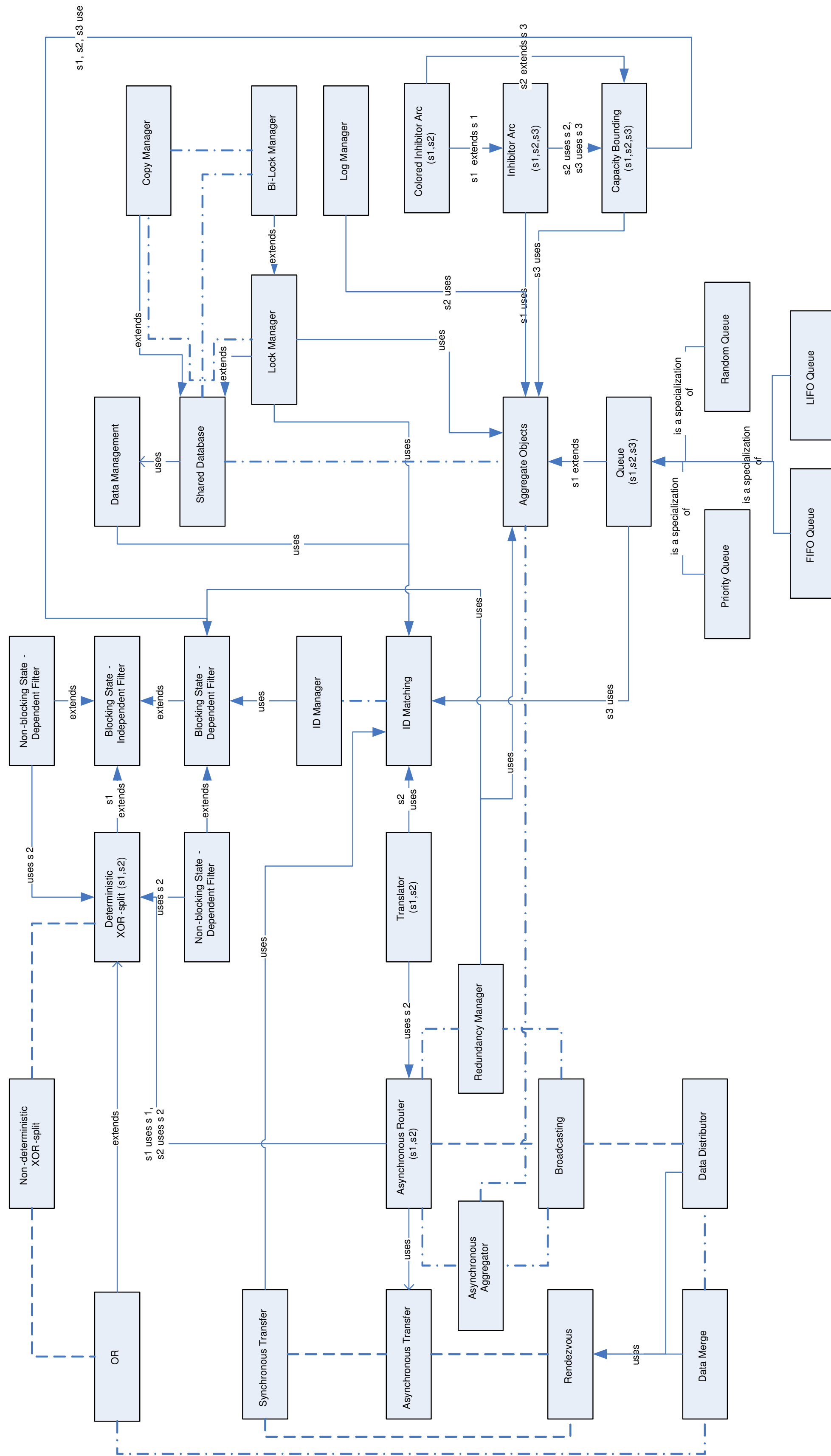


Fig. 6. CPN pattern relationship diagram.

"uses". A solution implementation-oriented relation is labelled "extends". Problem similarity is denoted by dashed lines (without dots) while combinable solutions are denoted by dashed lines with dots. Note that the details regarding the combination of one pattern with another one, or similarities between patterns are not indicated in the relationship diagram, but can be found in the *Consequences* and *Related patterns* sections of a chosen pattern.

In Section 3, we defined the Aggregate Objects pattern. As shown in Figure 6, the Queue pattern extends the first solution of the Aggregate Objects pattern. Moreover, many patterns use the Aggregate Objects pattern (e.g., the Log Manager, Inhibitor Arc, Lock Manager, and Capacity Bounding patterns).

4.3 Classification of CPN Patterns

Although the CPN pattern relationship diagram presented in Figure 6 allows the navigation through the catalog of the CPN patterns, it is not sufficient to classify the patterns precisely and unambiguously.

As was mentioned in the introduction, the CPN patterns aim at solving problems in the domain where data and control-flow perspectives interplay. In this domain, three pattern groups can be distinguished:

– patterns where the data perspective dominates, but which must be considered in the context of the control-flow;
– patterns where the control-flow perspective dominates, but which are data-based;
– patterns where both data perspectives and control-flow perspectives are important and involved.

However, this classification turns out to be not very meaningful and is rather subjective.

In order to provide a more useful means for selecting an appropriate pattern, we adopt the classification presented in [21] to categorize the CPN patterns. This classification is based on the *intent* of each pattern. The intent of every pattern has been analyzed according to a structure where *common components* contain *diagnostic elements*, and in turn diagnostic elements contain *supplementary components*. This structure will be represented using the following format.

Common component
- ◦ Diagnostic component
  - • Supplementary component

Common components define the set of related meanings, by which different patterns can be placed into one group. For instance, patterns addressing the problems of creating new elements or entities, belong to the same group with the common component *create*. Thus, this is the intent of a pattern from the process (functionality) point of view. For example, patterns, whose main intent is to manage or control something, will be combined into the group with a common component *control*.

Diagnostic elements define the contrastive features which distinguish the patterns belonging to the same common component. For instance, patterns belonging to the same common component *control*, i.e. control patterns, can involve different participants or differ by control parameters. For example, patterns, whose main purpose is to control such features as the order, the throughput, the quantity, belong to the same group with a common component *control* and can be distinguished by the diagnostic elements *Order, Throughput, Quantity* respectively.

Supplementary components address additional features for extended definitions of meanings. This components address special circumstances of applying a pattern. This feature could be applied to distinguish the pattern from other patterns belonging to the same common component with the same diagnostic elements; however, multiple patterns may have the same supplementary component.

Using the nested format described above (i.e., common components, diagnostic elements, supplementary components), we are able to classify the 34 CPN patterns:

Control

- ○ Order of information objects (Queue)
  - by predefined scheduling policy (FIFO Queue, LIFO Queue, Random Queue)
  - by objects' priority (Priority Queue)
- ○ Availability/Consistency of information objects
  - by regular replication (Copy Manager)
- ○ Concurrent access to information objects
  - by means of exclusive locks (Lock Manager)
  - by means of shared and exclusive locks (Bi-Lock Manager)
- ○ Throughput of information objects
  - by inspecting content (Blocking State-Independent Filter, Non-blocking State-Independent Filter)
  - by inspecting state (Blocking State-Dependent Filter, Non-Blocking State-dependent Filter, Redundancy Manager)
- ○ Number of objects in place
  - by bounding the place capacity (Capacity-Bounding)

Discern

- ○ Information objects
  - by identities (ID Matching)
  - by visibility (Shared Database)

Choose

- ○ 1 branch deterministically (Deterministic XOR-split)
- ○ 1 branch non-deterministically (Non-deterministic XOR-split)
- ○ 1 or more branches deterministically (OR)

Create

- ○ Information objects
    - by unique generation (ID Manager)
    - by decomposing into parts (Data Distributor)

Assemble

- ○ Information objects
    - by aggregating into a collection (Aggregate Objects)
    - by synchronizing composite parts (Data Merge)
    - by asynchronous merging (Asynchronous Aggregator)

Access

- ○ Information objects
    - by read/write operations (Data Management)

Inspect

- ○ "Non-containment" property of place (Colored Inhibitor Arc)
- ○ "Zero"-property of place (Inhibitor Arc)

Monitor

- ○ Process execution-relevant information
    - by data logs (Log Manager)

Transform

- ○ Information objects
    - by adjusting the data format (Translator)

Transfer

- ○ Information objects
    - Asynchronously

      directly from a source to a target: 1-to-1 (Asynchronous Transfer)

      indirectly from a source to one of several targets: 1-to-1 (Asynchronous Router)

      indirectly from a source to multiple targets: 1-to-N (Broadcasting)
    - Synchronously
        - · between two actors: 1-to-1 (Synchronous Transfer)
    - Concurrently
        - · from N sources to M targets: N-to-M (Rendezvous)

The Aggregate Objects pattern de ned in Section 3 is classi ed under common component "Assemble", diagnostic component "Information objects", and supplementary component "by aggregating into a collection". It is interesting to see how the Queue pattern, although it extends Aggregate Objects pattern, is classi ed completely di erent. This can be explained by the clear di erence in intent.

In this section we brie y introduced the set of 34 patterns. Clearly, we cannot list the patterns in full and instead refer to [28, 27]. Then we showed two ways to compare and relate patterns. We identi ed three primary and two secondary pattern relationships and classi ed the patterns using the classi cation of Hasso and Carlson [21].

5 Related Work

It is impossible to give a complete overview of the di erent types of patterns described in literature. The 23 design patterns by Gamma et al. [19] triggered the development of many more patterns in the object-oriented software community. Some of it successors include: the patterns for knowledge and software reuse by Sutcli e [37], the design patterns in communication software by Linda Rising [33], and the framework patterns by Wolfgang Pree [32].

Aside from the generic patterns, a set of language-speci c pattern languages (UML, Smalltalk, XML, Python, etc.), links to which can be found in the pattern digest library [29], has been discovered and documented.

Furthermore, some work has been done on formalizing the organization, process, analysis, and business-related patterns. Among them are the analysis patterns by Martin Fowler [18], the enterprise architecture patterns by Michael Beedle [12], the framework process patterns by James Carey [13], the patterns for e-business [7] (which focus on Business patterns, Integration patterns, and Application patterns), the business patterns at work [17] (which use UML to model a business system), and the process patterns [10]. Other interesting patterns collections focusing on the process-side of things are the enterprise integration patterns by Hophe and Woolf [23] and the service interaction patterns by Barros et al. [11].

However, the real starting point for this work has been the Work ow Patterns Initiative (cf. `www.workflowpatterns.com`). To capture the functionality of PAIS in term of patterns, control- ow patterns [6], data patterns [35], and resource patterns [34] have been uncovered. We consider the CPN patterns foundational for the further development of this initiative.

In the context of Petri nets some initial attempts to capture patterns have been made. Earlier work by Kurt Jensen [25], Wil van der Aalst [2], and Kees van Hee [22], provides some patterns in an implicit and/or fragmented manner. In [31], Petri nets are used to represent work ow and communication patterns in the context of webservices. One of the few papers, linking CPN to patterns is [30]. However, here CPNs are merely used as an underlying representation of the dynamic object-oriented architecture and the real focus is on patterns found in concurrent software designs. The paper that is probably most related to our work is [20] by Matthias Griess et al. They de ne 3 patterns in terms of classical Petri nets using pattern language similar to ours. For a given example they analyze the use of these patterns.

Our work di ers from these papers in at least two ways. First of all, we use CPNs (rather than classical nets) and focus on the interplay between control ow and data ow. Second, our set of patterns is more mature as is illustrated by the number of patterns, classi cation, and relationships.

6 Conclusion and Future Work

In this paper, we described a pattern language for CPNs. We collected a set of 34 patterns focusing on the interplay between control ow and data ow. Although

expressed in a speci c language, the patterns can be applied to model and design of any kind of dynamic systems with elements of data and concurrency.

The language and the patterns have been developed in an explorative manner. This means that we applied empirical methods to gather information, such as observation, content analysis, and simulation. In order to discover patterns we used application models, publications, tutorials, workshop materials, opinion of experts, and feedback from model developers. We applied the content analysis technique for extracting the patterns from the literature sources, and veri ed the correctness of models, which represent solutions for certain problems, by simulating them in CPN Tools.

We do not claim that the CPN patterns we gathered are complete, since they are the result of explorative work and were not derived in a systematic manner. We have made the implementations of CPN patterns available to the CPN community in the form of a pattern library [28]. We want to encourage members of the CPN community to extend the catalog of patterns by including the ones not covered here. Moreover, these patterns can serve as a language enhancing communication between developers, allowing to communicate problems and solutions unambiguously.

Further research will include the development of a patterns repository and empirical research into the actual use of patterns. We have developed a prototype patterns repository allowing people to navigate patterns based on various relationships. This needs to be improved to be really useful. The empirical research into the actual use could involve the analysis of student projects and/or the analysis of papers describing CPN models. In [20] this approach was used for a single example and a small set of basic Petri net patterns. In [38] a similar approach was applied to work ow projects of Atos Origin using the work ow patterns [6].

Acknowledgments

We would like to thank Kurt Jensen for contributing to the work reported in this paper. His experience in modeling using Colored Petri Nets has been vital for collecting and describing the patterns presented. We also thank Maurice Hendrix and Alex Norta for working on the initial prototype of the patterns repository.

References

1. W.M.P. van der Aalst. The Application of Petri Nets to Work ow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
2. W.M.P. van der Aalst. *Process Modeling, Lecture Notes.* Eindhoven University of Technology, Eindhoven, The Netherlands, 2003.
3. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems.* MIT press, Cambridge, MA, 2002.
4. W.M.P. van der Aalst and A.H.M. ter Hofstede. Work ow Patterns: On the Expressive Power of (Petri-net-based) Work ow Languages. In K. Jensen, editor,

Proceedings of the Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002), volume 560 of *DAIMI*, pages 1–20, Aarhus, Denmark, August 2002. University of Aarhus.
5. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Work ow Language. *Information Systems*, 30(4):245–275, 2005.
6. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Work ow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
7. J. Adams, S. Koushik, G. Vasudeva, and G. Galambos. *Patterns for e-Business. A Strategy for Use.* IBM Press, 2001.
8. C. Alexander. *A Pattern Language: Towns, Building and Construction.* Oxford University Press, 1977.
9. C. Alexander. *Timeless Way of Building.* Oxford University Press, 1979.
10. S.W. Ambler. *Process Patterns.* Cambridge University Press, 1998.
11. A. Barros, M. Dumas, and A.H.M. ter Hofstede. Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection. QUT Technical report, FIT-TR-2005-012, Queensland University of Technology, Brisbane, 2005.
12. M.A. Beedle. *Enterprise Architecture Patterns.* Cambridge University Press, 1998.
13. J. Carey and B. Carlson. *Framework Process Patterns.* Addison Wesley Longman, 2001.
14. CPN Group, University of Aarhus, Denmark. CPN Tools Home Page. http://wiki.daimi.au.dk/cpntools/.
15. J. Desel, W. Reisig, and G. Rozenberg, editors. *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2004.
16. M. Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede. *Process-Aware Information Systems.* Wiley & Sons, 2005.
17. H. Eriksson and M. Penker. *Business Modeling with UML. Business Patterns at Work.* Wiley, John and Sons, 1998.
18. M. Fowler. *Analysis Patterns: Reusable Object Models.* Addison-Wesley, Reading, Massachusetts, 1997.
19. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison Wesley, Reading, MA, USA, 1995.
20. M. Gries, J.W. Janneck, and M. Naedele. Reusing Design Experience for Petri Nets Through Patterns. In *Proceedings of High Performance Computing HPC'99*, pages 453–458, San Diego, CA, USA, 1999.
21. S. Hasso and C.R. Carlson. Linguistics-based Software Design Patterns Classi cation. In *Proceedings of the Thirty-Seventh Annual Hawaii International Conference on System Science (HICSS-37).* IEEE Computer Society Press, 2004.
22. K.M. van Hee. *Information System Engineering: a Formal Approach.* Cambridge University Press, 1994.
23. G. Hohpe and B. Woolf. *Enterprise Integration Patterns.* Addison-Wesley Professional, Reading, MA, 2003.
24. S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation.* International Thomson Computer Press, London, UK, 1996.
25. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1.* EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1997.

26. L.M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.
27. N. Mulyar and W.M.P. van der Aalst. Patterns in Colored Petri Nets. BETA Working Paper Series, WP 139, Eindhoven University of Technology, Eindhoven, 2005.
28. N. Mulyar. CPN Patterns Home Page. http://is.tm.tue.nl/staff/nmulyar.
29. Pattern digest library. http://patterndigest.com/books/otherlang.jsp.
30. R.G. Pettit and H. Gomaa. Modeling Behavioral Patterns of Concurrent Software Architectures Using Petri Nets. In *WICSA '04: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, page 57, Washington, DC, USA, 2004. IEEE Computer Society.
31. S.K. Prasad and J. Balasooriya. Fundamental Capabilities of Web Coordination Bonds: Modeling Petri Nets and Expressing Work ow and Communication Patterns over Web Services. In *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 7*, page 165.2, Washington, DC, USA, 2005. IEEE Computer Society.
32. W. Pree. *Framework patterns*. SIGS Books, 1996.
33. L. Rising. *Design Patterns in Communication Software*. Cambridge University Press, 2000.
34. N. Russell, W.M.P.van der Aalst, A.H.M. ter Hofstede, and D. Edmond. Work ow Resource Patterns: Identi cation, Representation and Tool Support. In O. Pastor and J. Falcao e Cunha, editors, *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*, volume 3520 of *Lecture Notes in Computer Science*, pages 216–232. Springer-Verlag, Berlin, 2005.
35. N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Work ow Data Patterns. QUT Technical report, FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.
36. Software-Ley. *COSA 3.0 User Manual*. Software-Ley GmbH, Pullheim, Germany, 1999.
37. A. Sutcliffe. *Patterns for Knowledge and Software Reuse*. Lawrence Erlbaum Associates Inc., 2002.
38. K. de Vries and O. Ommert. Advanced Work ow Patterns in Practice (1): Experiences Based on Pension Processing (in Dutch). *Business Process Magazine*, 7(6):15–18, 2001.
39. W. Zimmer. Relationships between Design Patterns. In *Pattern languages of program design*, pages 345–364, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

Enhancing the CES Protocol and its Verification

Lin Liu[1,2] and Jonathan Billington[2]

[1] School of Computer and Information Science
[2] Computer Systems Engineering Centre
School of Electrical and Information Engineering
University of South Australia, Mawson Lakes, SA 5095, Australia
{lin.liu | jonathan.billington}@unisa.edu.au

Abstract. The Capability Exchange Signalling (CES) protocol is one of the sub-protocols of ITU (International Telecommunication Union) recommendation H.245, "Control protocol for multimedia communication". Previous work has found that the CES protocol could fail when the sequence numbers used by the protocol wrap. In this paper, we firstly further investigate the CES protocol by using language analysis. The result reveals the same error of the protocol. To solve this problem, we propose a set of changes to the CES protocol. The revised protocol is then modelled with Coloured Petri Nets (CPNs). Based on this model, state space and language analyses are applied to the revised CES protocol for given values of parameters of the system. Verification results suggest that the revised protocol satisfies the desired properties with the error discovered in previous work being eliminated.

Key Words: Protocol Revision and Verification, Coloured Petri Nets, Capability Exchange Signalling Protocol, State Space Analysis, Language Equivalence

1 Introduction

Background At present, H.323 [8] is the key standard for multimedia applications, such as IP (Internet Protocol) telephony and multimedia conferencing, operating over packet-based networks, including the Internet. H.323 is a *framework* standard [11] developed by the International Telecommunication Union (ITU). ITU Recommendation H.245 [9], "Control protocol for multimedia communication", has been integrated into this framework as one of the core recommendations for an H.323 system.

The Capability Exchange Signalling (CES) protocol is a sub-protocol of the H.245 standard. This protocol is used by a multimedia communication party to inform its peer of its transmit and/or receive capabilities. A multimedia party has a particular capability if it is able to encode and transmit (transmit capability) or receive and decode (receive capability) a particular multimedia signal [9], e.g. audio G.722 [6] or video H.261 [7] signal. Once the sending party knows the receive capabilities of its peer, it can then send multimedia signals that can be received and decoded appropriately by its peer. This is a pre-condition of a successful multimedia communication session. Furthermore, if a receiver knows the transmit capabilities of its peer, the receiver may request its peer to transmit information that is within its receive capabilities. Thus when a H.323 session is set up, we require that the CES messages are the first H.245 messages transmitted between communication parties.

From the point of view of protocol engineering, it is desirable that the properties of a protocol can be verified before implementation [2]. For important and widespread protocols, like those of H.323, it is vital that design errors are found and eliminated before implementation. Unfortunately, little work has been done on verifying H.323 protocols. H.323 is a comprehensive and complex standard, the length of the documentation of the core H.323 recommendations only, is close to 800 pages. So we started our work on H.323 by investigating the CES protocol.

According to the protocol verification methodology developed in [3], verification of a protocol comprises verification of general properties of the protocol (such as absence of deadlocks and livelocks) and verification of the protocol against its service specification. The major part of a service specification is the definition of the *service language*, the allowable sequences of user observable

events (known as *service primitives*). General properties are verified by analysing the state space of the CPN model of the protocol. Verification against the service is accomplished by using language analysis, which involves comparing the *protocol language* (i.e. the sequences of service primitives that occur as a result of the protocol's operation), with the service language.

Previous Work In [14], the CES protocol is modelled with Coloured Petri Nets (CPNs) [10, 12] and state space analysis shows that the protocol will work correctly when wrapping of the protocol's sequence numbers does not occur. However, when sequence numbers wrap, a user may be misinformed about the acceptance or otherwise of the capabilities it has just sent. This could lead to the failure of the multimedia session.

Contribution of the Paper In this paper, we firstly analyse the CES protocol further by checking it against the CES service. We find that the CES protocol has implemented some sequences of primitives that are not specified in the CES service language (known as *illegal sequences*). Some of these illegal sequences correspond to the same error that we found previously [14].

To remedy the error, we propose two changes to the CES protocol defined in H.245 [9]. We then verify this revised CES protocol by following the protocol verification methodology presented in [3]. State space analysis of the revised CES protocol, for given values of its parameters, shows that this revised protocol satisfies the general properties that we have specified, and that the erroneous behaviour found in the original CES protocol no longer exists. Furthermore, language analysis shows that the protocol language of the revised CES protocol is included in the service language. We show that the protocol implements an acceptable subset of the sequences specified in the service definition and does not generate any illegal sequences.

Structure of the Paper The rest of the paper is organised as follows. Section 2 introduces the CES protocol and its CPN model created in [14]. The desired properties for the CES protocol are defined in Section 3. Section 4 summarises the state space analysis results for the CES protocol obtained in [14], and then presents the language analysis results. Section 5 proposes two changes to the CES protocol and the CPN model for the revised CES protocol is created. State space analysis and language analysis of the revised protocol are described in Sections 6 and 7 respectively. Finally Section 8 summarises the paper and suggests future work.

2 Review of the CES Protocol

The CES protocol and other sub-protocols of H.245 have been designed to be independent of the underlying transport medium. However, it is stated in H.245 [9] that these protocols are intended to be used over a reliable channel, i.e. a channel that does not lose, reorder, or duplicate messages. With an H.323 system, H.245 (including the CES protocol) is required to be used over a reliable transport medium. Therefore, in this paper, we only investigate the behaviour of the CES protocol operating over a reliable channel.

The CES protocol is specified in H.245 [9] in terms of a set of CES messages that are transmitted between the CES Entities (CESEs); the procedure that governs the transmission of these messages; and the exchange of the CES service primitives between the CES service user and the CES *service provider* (i.e. the CESEs and their underlying medium), as seen from the CESE side instead of the user side.

In [14], we created a CPN model for the CES protocol, which is used in this section to illustrate the protocol. Figures 1 and 2 show the model and its declarations (with changes to the names of some net components). In the following, we firstly introduce the components of the CES protocol (including the CESEs, the CES messages and the CES service primitives) and how they are modelled. Based on this, we describe the operation of the protocol.

2.1 Capability Exchange Signalling Entities

H.245 defines two CESEs: an outgoing CESE which initiates communication; and an incoming CESE which responds. The states of the these two CESEs are modelled by places `outgoingCESE` and

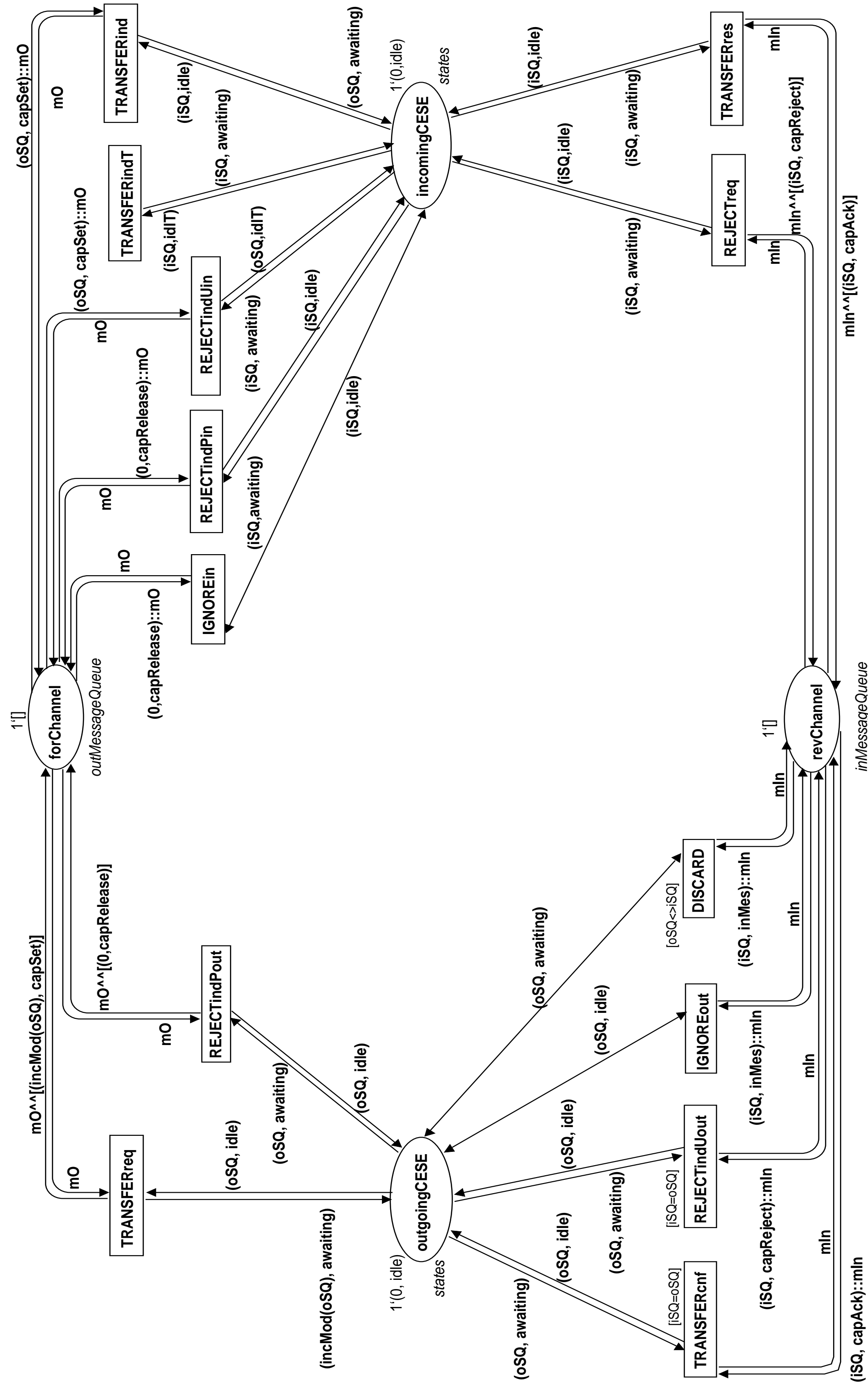


Fig. 1. CPN model of the CES protocol

```
(*---states of CESEs---*)
 color st = with idle | idlT | awaiting ;
(*---sequence numbers/values of state variables---*)
 color sequenceNo = int with 0..255;
(*---states and state variables of CESEs---*)
 color states = product sequenceNo*st;

(*---types of messages sent by outgoing CESE---*)
 color outM = with capSet | capRelease;
(*---messages sent by outgoing CESE---*)
 color outMessages = product sequenceNo*outM;
(*---message queue of forward channel---*)
 color outMessageQueue = list outMessages;

(*---types of messages sent by incoming CESE---*)
 color inM = with capAck | capReject;
(*---messages sent by incoming CESE---*)
 color inMessages = product sequenceNo*inM;
(*---message queue of reverse channel---*)
 color inMessageQueue = list inMessages;

(*---variables---*)
 var mO : outMessageQueue;
 var mIn: inMessageQueue;
 var inMes : inM;
 var oSQ, iSQ : sequenceNo;

(*---function for calculating new value of---*)
(*-sequence number or state variables-*)
 fun incMod (oSQ) = (oSQ+1) mod 256;
```

Fig. 2. The declarations of the CES protocol CPN

`incomingCESE` (Figure 1). Both places are typed by colour set `states` (refer to Figure 2), a product of colour sets `sequenceNo` and `st`. Here `sequenceNo` specifies the range of sequence numbers used in CES messages, as well as the range of the values of state variables of the outgoing CESE (out_SQ) and the incoming CESE (in_SQ). These two state variables are modelled by CPN variables `oSQ` and `iSQ` correspondingly. `st` defines all the possible states of the outgoing and incoming CESEs. It has three possible values, `idle`, `idlT` and `awaiting`. While `idle` and `awaiting` correspond to the two states IDLE and AWAITING RESPONSE of the two CESEs, `idlT` models a temporary state of the incoming CESE, which is not explicitly defined in the CES protocol specification of H.245 (details of this temporary state will be introduced in Section 2.4).

2.2 CES Messages

Four CES messages are defined in H.245, including TerminalCapabilitySet, TerminalCapabilitySetRelease, TerminalCapabilitySetAck, and TerminalCapabilitySetReject. The first two messages are sent by the outgoing CESE to the incoming CESE and the last two are sent by the incoming CESE. Colour set `outM` (Figure 2) defines the types of messages that can be sent by the outgoing CESE, including `capSet` (for TerminalCapabilitySet message) and `capRelease` (TerminalCapabilitySetRelease). Colour set `outMessages` combines `sequenceNo` with `outM`. There is no sequenceNumber field defined for a TerminalCapabilitySetRelease message. However, in order not to make the type of place `forChannel` complicated, we still use a pair (i.e. `(0, capRelease)`) to represent this message, but set the first item of the pair to 0 in all cases. Colour set `outMessageQueue` forms a list of outgoing messages and is used to type place `forChannel` (representing the reliable channel from the outgoing side to the incoming side). Similarly, `inMessageQueue` is a list of messages sent by the incoming CESE and it types place `revChannel` (modelling the reliable channel from the incoming side to the outgoing side).

2.3 CES Service Primitives

There are six CES service primitives that can occur at the interface between the CES service users and the CES service provider. TRANSFER.request and TRANSFER.confirm are primitives that occur at the outgoing side only, and TRANSFER.indication, TRANSFER.response and REJECT.request can occur at the incoming side only. Primitive REJECT.indication can be issued either by the outgoing CESE or the incoming CESE. The usage of these six primitives will become clear when we introduce the operation of the protocol. Service primitives are modelled as transitions (Figure 1). The rules for naming the transitions from the corresponding primitive names are as follows. The general primitive names are preserved, e.g. TRANSFER still appears as `TRANSFER` in the corresponding transition names; the "." in the name is omitted; the type names of primitives are abbreviated to 3 letters (e.g. request becomes `req`); when modelling REJECT.indication, two separate transitions are used to distinguish the two values of the SOURCE parameter of the primitive (indicating the source of rejection), so following the type name, a capital letter `U` (SOURCE = USER), or `P` (SOURCE = PROTOCOL), is appended. In the protocol CPN, to distinguish primitives with the same name but at different ends (e.g. REJECT.indication), the suffix `out` is added to a transition name to show that it is at the outgoing end, and suffix `in` for the incoming end.

2.4 Operation of the CES Protocol

In the following, we use the term *capability set* to describe the set of capabilities of the outgoing CES user that is encapsulated in a TRANSFER.request primitive. Moreover, the term *capability transfer* refers to the process of delivering a capability set to the incoming CES user and the corresponding acknowledgment (if any) back to the outgoing CES user. A capability transfer is *acknowledged* if the corresponding acknowledgment is generated by the incoming CES user and is received by the outgoing CES user. A capability transfer is *aborted* if a CESE has issued a REJECT.indication primitive (SOURCE = PROTOCOL). A capability transfer can be *completed* in three ways: acknowledged; aborted at both sides (in this case no acknowledgment is generated); or aborted at the outgoing side only *and* the acknowledgment has been received and discarded by the outgoing CESE (without informing the outgoing CES user of this invalid acknowledgment). We say that a capability transfer is *outstanding* if it has not been completed.

Referring to Figures 1 and 2, the basic operation of a CES protocol can be described as follows. A capability transfer is started when a TRANSFER.request is issued by the user to the outgoing CESE (i.e. `TRANSFERreq` occurs). Upon receiving the TRANSFER.request primitive, the outgoing CESE increases its state variable, out_SQ, by 1 modulo 256 (modelled by function `incMod(oSQ)`), sends a TerminalCapabilitySet message with sequence number equal to the incremented out_SQ to the incoming side (i.e. `(incMod(oSQ), capSet)` is concatenated to the end of the queue in place `forChannel`). Then a timer is started, which is implicitly modelled by transition `REJECTindPout`, and the outgoing CESE enters its AWAITING RESPONSE state, modelled by changing the marking of `outgoingCESE` from `idle` to `awaiting`.

When receiving the TerminalCapabilitySet message, the incoming CESE copies the sequence number of the message into variable in_SQ, informs the user with a TRANSFER.indication primitive, and changes state from IDLE to AWAITING RESPONSE. This process is modelled by the occurrence of `TRANSFERind`. Then the user can accept the capabilities by issuing a TRANSFER.response primitive (i.e. `TRANSFERres` occurs), or rejects them with a REJECT.request primitive (`REJECTreq` occurs). Accordingly, a TerminalCapabilitySetAck (`capAck`) or a TerminalCapabilitySetReject (`capReject`) message is sent to the peer CESE and the state of the incoming CESE becomes IDLE.

At the outgoing side, if the response message is received before the timer expires, the outgoing CESE notifies the user by submitting a TRANSFER.confirm when receiving a TerminalCapabilitySetAck message (i.e. the occurrence of `TRANSFERcnf`). Otherwise a REJECT.indication with its SOURCE parameter set to value USER is sent to the user when receiving a TerminalCapabilitySetReject message (the occurrence of `REJECTindUout`), indicating that the incoming CES user has rejected the capabilities of the outgoing CES user. After informing the user, the outgoing CESE

returns to IDLE and resets the timer (i.e. transition `REJECTindPout` can not be enabled). If the timer expires before any response is received (i.e. `REJECTindPout` occurs), a TerminalCapabilitySetRelease message is sent to the peer CESE, a REJECT.indication primitive is issued to the user (with its SOURCE parameter set to value PROTOCOL, indicating that the protocol entity has aborted the current capability transfer), and the CESE's state returns to IDLE. When the TerminalCapabilitySetRelease message arrives at the incoming side, if the incoming CESE is in the AWAITING RESPONSE state, it will submit a REJECT.indication (with its SOURCE parameter set to PROTOCOL) to the user to abort the current transfer (i.e. `REJECTindPin` occurs). If it is IDLE, i.e. the release message arrives after the incoming user has issued the response, this message will be ignored (modelled by `IGNOREin`).

Finally, it is possible for an invalid acknowledgment to come back to the outgoing side. Transitions `IGNOREout` and `DISCARD` are used to discard a message received when the outgoing CESE is not in the right state (`IGNOREout`) or when the message received has a different sequence number from that being expected by the outgoing CESE (`DISCARD`).

2.5 Transitions `TRANSFERindT` and `REJECTindUin`

When the incoming CESE is IDLE, it can issue a TRANSFER.indication primitive on receiving a TerminalCapabilitySet message, which is modelled as transition `TRANSFERind`. According to the CES protocol specification, when the incoming CESE is in its AWAITING RESPONSE state and another TerminalCapabilitySet message is received, it will abort the previous capability transfer by issuing a REJECT.indication primitive to its user (with SOURCE=USER, indicating that the outgoing user has decided to abort the previous capability transfer). Immediately following this, a TRANSFER.indication primitive occurs at the incoming CESE, and the CESE returns to its AWAITING RESPONSE state.

If we are only interested in observing the basic properties of this protocol, e.g. whether the capability information can be transfered properly, we can model the event of successively issuing REJECT.indication and TRANSFER.indication as a single CPN transition. However, we are also interested in verifying the CES protocol against its service specification. We need to extract the language of service primitive events from this protocol model (i.e. protocol language) and compare it with the CES service language. It will make language comparison difficult if we model the two primitives as a single CPN transition, because they will appear as one event in the protocol language, while in the service language, they appear as two separate events. Therefore we model the two primitives as two CPN transitions, `REJECTindUin` and `TRANSFERindT`. To do this we introduce a temporary state, `idlT`, for the incoming CESE. Once transition `REJECTindUin` occurs, the CESE will move to `idlT`. Immediately following this, only transition `TRANSFERindT` can occur at the incoming side.

3 Desired Properties for the CES Protocol

In this section, we specify the desired properties for the CES protocol, including desired general properties and a language property. In [14], general properties were analysed by investigating the state space of the CES protocol CPN. However, the properties analysed were not explicitly defined in [14]. Furthermore, verification of the CES protocol against its service was not considered. With the verification of the revised version of the CES protocol, we would like to firstly explicitly define the desired properties that we are going to verify.

As introduced by Holzmann [4], absence of deadlocks, absence of livelocks, and proper termination are three important desired general properties, and a protocol can fail if any of the above properties is not satisfied. A deadlock appears as an undesired dead marking in the state space of a protocol's CPN model. According to H.245, a capability transfer can occur at any time during a multimedia session, so there must not be any dead markings.

Property 1. (Absence of Deadlocks) *The CES protocol does not deadlock.* □

The CES protocol must not be trapped in a set of states where useless or undesired sequences of actions occur repeatedly without ending.

Property 2. (Absence of Livelocks) *There must be no livelocks in the CES protocol.* □

As described in Section 2.4, every capability transfer should finally be completed. That is, it may be acknowledged (i.e. `TRANSFERcnf` or `REJECTindUout` has occurred), aborted at both sides (`REJECTindPout` and `REJECTindPin` have occurred), or aborted at the outgoing side only (`REJECTindPout`, and `IGNOREout` or `DISCARD` have occurred). Furthermore, because the CES protocol uses modulo arithmetic to calculate sequence numbers and values of state variables, out_SQ and in_SQ, from any state, the protocol should be able to go back to the initial state in which both CESEs are IDLE (with the values of out_SQ and in_SQ being zero) and the two channels are empty. Therefore instead of having the property about proper terminal states, we define the *initial state as a home state* for the CES protocol (where a home state refers to a state that can be reached from any other reachable state of the protocol).

Property 3. (Home State) *It must be possible to return to the initial state of the CES protocol, that is the initial marking of the CES protocol CPN must be a home marking.* □

Note that Property 3 implies Properties 1 and 2.

A protocol design should have no dead code, i.e. actions that are specified but never executed.

Property 4. (No Dead Code) *There must be no dead code in the CES protocol, that is there must be no dead transitions in the CES protocol CPN (Figure 1).* □

An important property to be verified for a protocol is that the protocol provides the required service. For the CES protocol, we need to prove that the sequences of the CES service primitives implemented by the CES protocol (i.e. the CES protocol language) is included in the CES service language. This is stated below as Property 5.

Property 5. (Language Property) *The CES protocol language must be included the CES service language.* □

If this property holds for the CES protocol, we say the protocol is a *refinement* of the CES service. Particularly when the protocol language is identical to the service language, we say that the protocol is a *faithful* refinement of the CES service. We call the sequences of primitives that are in the service language but not in the protocol language (if any) *missing sequences*. In this case, we need to show that the subset of the service implemented by the protocol is useful. When there are sequences of primitives in the CES protocol language that are not in the CES service language, i.e. *illegal sequences* as mentioned in Section 1, the CES protocol has undesired behaviour or else the service specification needs modification.

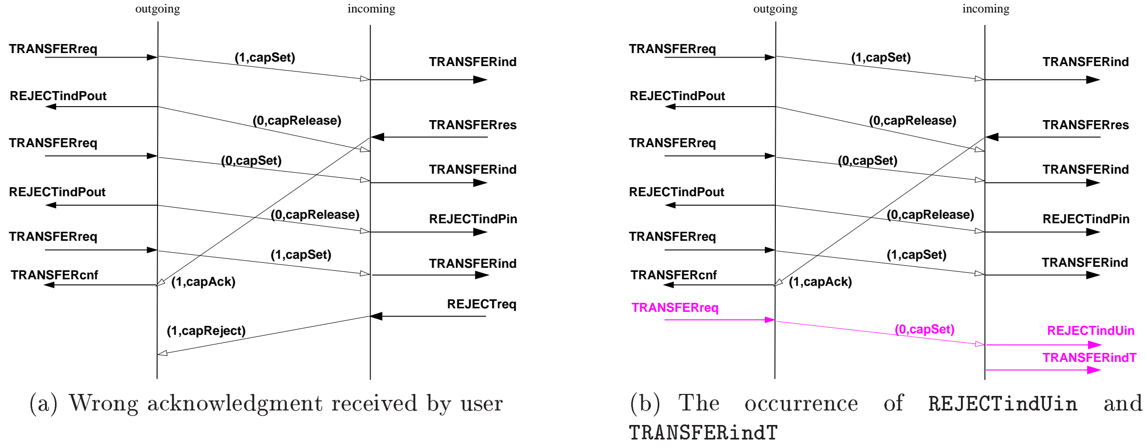
4 Verification of the CES Protocol

In this section, we investigate the five properties defined in the previous section. This includes a review of the verification results for the four general properties [14], and the new language analysis results.

4.1 Previous Work

Because the channels have infinite capacity, the Occurrence Graph (OG) of the CES protocol CPN is infinite. So in [14] state space analysis was applied to a restricted form of the CPN model of the CES protocol (Figure 1). In this restricted model, it is assumed that the channels between the two CESEs have fixed capacity, i.e. the lengths of the queues in places `forChannel` and `revChannel` are limited. Also, to obtain a tractable OG, and to be able to observe the behaviour of the protocol when sequence numbers wrap while not losing the generality of using *different* sequence numbers, sequence numbers are in the range of $\{0, 1\}$, instead of $\{i \mid 0 \leq i \leq 255\}$. State space analysis shows that this model does satisfy Properties 1 to 4.

However, further investigation of the OG of this model revealed that when wrapping of sequence numbers occur, the outgoing CESE can take an old acknowledgment with the same sequence



(a) Wrong acknowledgment received by user

(b) The occurrence of `REJECTindUin` and `TRANSFERindT`

Fig. 3. Illustration of the errors found in [14]

number as that of the expected acknowledgment, and convey the information enclosed in this old acknowledgment to the outgoing CES user. This can be seriously misleading.

As illustrated in Figure 3 (a) (and referring Figure 1), with the restricted model, initially when `TRANSFERreq` occurs, a `capSet` message with sequence number 1 is sent. When the `incomingCESE` receives this message, `TRANSFERind` occurs and the user is informed. Now the timer expires at the outgoing side (`REJECTindPout` occurs), and a `capRelease` message is sent. However, the incoming CES user has accepted the first capability set (`TRANSFERres` occurs) and a `capAck` message with sequence number 1 is sent by the `incomingCESE` before the `capRelease` message is received. The delayed `capRelease` message is then ignored (`IGNOREin` occurs). Because the `outgoingCESE` is `idle`, `TRANSFERreq` now occurs for the second time, and message `(0, capSet)` is sent. However the timer expires again and a `capRelease` message is sent and received by the `incomingCESE` before the incoming CES user has generated a response to the second capability set (`REJECTindPin` occurs). Up to now, the acknowledgment to the first capability set (i.e. `(1, capAck)`) has not arrived at the outgoing side yet, but `TRANSFERreq` occurs for the third time. Because the protocol uses modulo arithmetic (modulo 2 for the restricted model) to generate sequence numbers, another `capSet` message with sequence number 1 is sent, and the `outgoingCESE` is `awaiting` an acknowledgment with sequence number 1 from the incoming side. At this time, the message (1, `capAck`) finally arrives. Then the `outgoingCESE` takes this one as the acknowledgment to the third `capSet` message and informs its user that the third capability set has been accepted by the incoming CES user. However the incoming CES user in fact rejects the third capability set (`REJECTreq` occurs) and a `capReject` massage is sent. However this message is ignored (i.e. `IGNOREout` occurs) by the `outgoingCESE`, because the CESE is `idle`. This shows that the outgoing CES user has been informed of the acceptance of a capability set that has been rejected by the incoming CES user. The consequence of this error is that the outgoing CES user can initiate a multimedia transmission that cannot be properly received and/or interpreted by the incoming CES user.

Because in general it is desirable that a protocol has no dead code, we have specified it as a desirable property for the CES protocol in Section 3. However, as discussed in [14] transitions `TRANSFERindT` and `REJECTindUin` (Figure 1) should be dead when the underlying medium is reliable. The two transitions are only required to recover from an unreliable medium. Further analysis of the OG of the restricted model also reveals that it is the erroneous behaviour described above that leads to the *occurrence* of these two transitions with the CES protocol operating over a reliable medium. This can be illustrated with the scenario shown in Figure 3 (b). The time sequence diagram in Figure 3 (b) is the same as that in Figure 3 (a), except that at the bottom of Figure 3 (b), before `REJECTreq` occurs, `TRANSFERreq` occurs for the fourth time (due to the above error which has changed the state of the `outgoingCESE` to `idle`), and message `(0, capSet)` is

sent. This message is received when the `incomingCESE` is `awaiting`, which enables `REJECTindUin` (refer to Figure 1), thus it occurs. Immediately following this, `TRANSFERindT` occurs.

This result implies that Property 4 is not a desirable property for the CES protocol when the transport medium is reliable, so we revise this property as follows.

Property 4. (No Unexpected Dead Code) *When the transport medium is reliable, there must be no dead code in the CES protocol except for the procedure that occurs when the incoming CESE is in AWAITING RESPONSE and receives a TerminalCapabilitySet message. Correspondingly, with the CES protocol CPN (Figure 1), there must be no dead transitions except for* `REJECTindUin` *and* `TRANSFERindT`*.* □

In the rest of the paper, Property 4 refers to the above **No Unexpected Dead Code** property. The result from [14] has shown that this property does not hold for the CES protocol.

4.2 Language Analysis of the CES Protocol

To complete the verification of the CES protocol, in this section, we check if Property 5 holds, i.e. whether the CES protocol language is equivalent to or included in the CES service language.

The CES Service and Protocol Languages To analyse the language property we follow the methodology of [3]. We firstly generate the service language, which represents all the allowable sequences of service primitives specified in the service definition. We also need to generate the protocol language, which comprises all the sequences of primitives that are implemented by the protocol. Both service and protocol languages are represented by deterministic (and minimal if possible) Finite State Automata (FSA) [1]. Then we compare the two FSAs to see if the protocol language is included in the service language. We may also check if the two languages are equivalent.

In [15] and [17], after improving the CES service definition given in H.245 [9], we have obtained a symbolic representation of the CES service language (represented by a minimal parametric FSA) in terms of the capacity of the channel from the outgoing side to the incoming side. No limitations are put on the capacity of the channel from the incoming side to the outgoing side. In this section, we will use this result directly. Readers interested in more details about how the service language is obtained are referred to [15, 17].

The protocol language of the CES protocol is extracted from the OG of the CPN of the protocol. Because we have modelled each CES service primitive as a CPN transition, the OG of the CPN model of the CES protocol includes all the possible occurrence sequences of the CES service primitives. However, the CPN model also includes transitions that do not model service primitives (called *non-primitive transitions*), but rather internal operations of the protocol, including `IGNOREout`, `DISCARD` and `IGNOREin` in Figure 1. We need to eliminate these non-primitive transitions while preserving service primitive sequences. To do this and further to use FSA comparison, we treat the OG as a FSA, and use language preserving FSA reduction techniques [1] to transform this FSA to its deterministic and minimum form. FSA reduction and language comparison are provided by tools such as the FSM library from AT&T [13].

Choosing a Restricted Model The protocol language of the CES protocol needs to be extracted from the OG of the CES protocol CPN. However, it can be seen from Figure 1 that, if the maximum length of the queue in place `forChannel` (representing the capacity of the channel from the outgoing side to the incoming side) is not limited, an infinite number of interleavings of transition `TRANSFERreq` followed by `REJECTindPout` can occur, resulting in an infinite number of `capSet` and `capRelease` messages being deposited. An infinite number of messages can also be put into the queue in place `revChannel`, if the maximum length of this queue is not limited. So as with the state space analysis in [14], we have to apply language analysis on a restricted model. We would like to find a model which reflects the most common usage scenario of the protocol in practice, then the verification of this restricted model can provide us high confidence that the protocol is correct in general.

Fixing the value of the maximum length of the queue in place forChannel (maxQLf) postpones the enabling of TRANSFERreq or REJECTindP. For example, if the maximum queue length is set to 1, REJECTindPout can not occur immediately after TRANSFERreq, or TRANSFERreq can not occur immediately after REJECTindPout, because the queue is occupied by a capSet or capRelease message. However, this will only *postpone* the enabling of transitions TRANSFERreq and REJECTindPout until capacity becomes available in the queue. Furthermore enabling each of the transitions of IGNOREin, REJECTindPin, REJECTindUin and TRANSFERind at the incoming side only requires one message at the front of the queue in place forChannel and the occurrence of each of the four transitions also only removes one message from the front of the queue. Because the queue in this place is a FIFO queue without message loss, having many messages in this queue and having only one message in this queue will not affect the behaviour of these four transitions. For similar reasons, fixing the maximum length of the queue in place revChannel (maxQLr) will only postpone the enabling of the transitions for which revChannel is an output place, and will not affect the behaviour of the transitions for which revChannel is an input place. Therefore, no extra dead transitions can be caused by fixing the maximum queue lengths of the two places, i.e. Property 4 is preserved. However we cannot guarantee that the other three general properties of the CES protocol are preserved when fixing the values of maxQLf and maxQLr.

On the other hand, in practice, it is seldom that a timeout (i.e. REJECTindPout) can occur before the message has been received by the peer end (i.e. when a capSet message is in forChannel), and it is also rare that a timeout can occur more than once before messages have been delivered to the peer end. So if the restricted model allows a timeout to occur before the capSet message is received by the incoming side, then we will capture the vast majority of the behaviour of the CES protocol. From Figure 1, we see that if the value of maxQLf is 2, REJECTindPout can occur and a capRelease is sent before the capSet message at the front of the queue in forChannel is received by the incoming CESE. However, when a timeout occurs, the outgoing CESE issues a REJECT.indication primitive (SOURCE = PROTOCOL) to inform its user of the failure of the CES service provider in delivering the previously sent capability set. Then it is very likely that the user will issue another TRANSFER.request (which may include the same or an updated capability set) without delay after it receives the REJECT.indication primitive. That is, TRANFERreq can occur immediately after the occurrence of REJECTindPout, with a capSet message put into the queue in place forChannel. Therefore, we need to set the value of maxQLf at least to 3 to allow this capSet message to be sent.

In practice, it is not often that a new acknowledgment is sent before the previous one has been delivered by the reverse channel to the outgoing side. So setting maxQLr to 2 to include this situation will result in a practical model for the CES protocol. However, because we have set maxQLf to 3 to allow two capSet messages to be received by the incoming side within a short period of time, there could be two acknowledgments generated by the incoming CES user within a short time so that they can be in the reverse channel at the same time. To cater for this, we choose maxQLr to be 3.

From Figure 1 we can see that, when setting maxQLf and maxQLr to 3 respectively, there are at most 6 outstanding capability transfers, of which two have the capability sets being transmitted from the outgoing side to the incoming side (i.e. two capSet messages are in place forChannel), one has the capability set being processed by the incoming CES user (i.e. the incomingCESE is in its awaiting state), and three have their acknowledgments being delivered (i.e. three responding messages, capAck or capReject are in place revChannel). Because we are interested in observing the behaviour of the CES protocol when sequence numbers wrap, with the above setting of maxQLf and maxQLr, we need to choose a range of sequence numbers that gives us no more than 5 different sequence numbers. Then when there are 6 outstanding capability transfers, at least two of them are identified by the same sequence number. To avoid state space explosion due the size of the sequence number space, we consider $\{0, 1\}$ as the smallest range that we can choose that fulfils the above condition where ambiguous sequence numbers may occur.

In summary, we choose to analyse the language property of the CES protocol CPN in Figure 1 when fixing maxQLf to 3, maxQLr to 3 and using $\{0, 1\}$ as the range of sequence numbers. We call this model *Model A*.

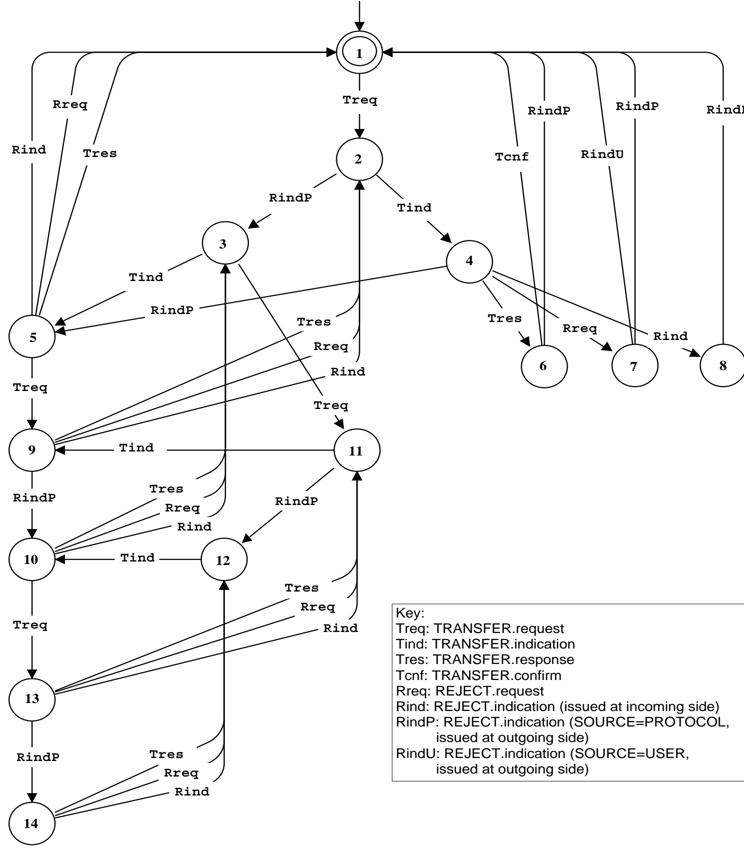


Fig. 4. $\mathcal{A}$, FSA representing the CES service language

Table 1. Comparison of $\mathcal{A}$ and $\mathcal{A}^{A}$

$\mathcal{A}$		$\mathcal{A}^{A}$		$\mathcal{A} - \mathcal{A}^{A}$		$\mathcal{A}^{A} - \mathcal{A}$	
$\mid S \mid$	$\mid \Delta \mid$	$\mid S \mid$	$\mid \Delta \mid$	$\mid S \mid$	$\mid \Delta \mid$	$\mid S \mid$	$\mid \Delta \mid$
14	36	2488	7419	24	63	3210	9579

Language Analysis of Model A The minimal FSA that represents the CES service language for Model A (denoted as $\mathcal{A}$) is shown in Figure 4. We generated the minimal FSA representing the protocol language of Model A (denoted as $\mathcal{A}^{A}$) using FSM and then compared the languages. Table 1 lists the results. In the table, from left to right are the number of nodes ($\mid S \mid$) and arcs ($\mid \Delta \mid$) of four FSAs: $\mathcal{A}$, $\mathcal{A}^{A}$, $(\mathcal{A} - \mathcal{A}^{A})$, and $(\mathcal{A}^{A} - \mathcal{A})$. $(\mathcal{A} - \mathcal{A}^{A})$ is the minimal automaton that represents the language included in $\mathcal{A}$ but not in $\mathcal{A}^{A}$, and $(\mathcal{A}^{A} - \mathcal{A})$ the minimal automaton representing the language included in $\mathcal{A}^{A}$ but not in $\mathcal{A}$. It can be seen that the protocol language of Model A is not included in the CES service language, because $(\mathcal{A}^{A} - \mathcal{A})$ is not empty (it has 3210 nodes and 9579 arcs). So Property 5 does not hold for Model A. As $(\mathcal{A} - \mathcal{A}^{A})$ is not empty, Model A does not implement all the sequences specified in the service language, i.e. there are missing sequences in Model A.

The capability transfer can be wrongly acknowledged if there are still old acknowledgments with the same sequence number in the system. It turns out that illegal sequences of primitives can be generated by the protocol in two ways: when the incoming CES user has generated a positive response (i.e. TRANSFER.response occurs), but the outgoing CES user is informed with a REJECT.indication primitive (with parameter SOURCE =USER); or the capability set is rejected

by the incoming CES user (i.e. REJECT.request occurs), but the outgoing user is informed by a TRANSFER.confirm.

We now use language analysis to show that the protocol language of Model A includes these illegal sequences. To do so we create two FSAs, $\mathcal{A}^{s_1}$ which accepts the bad sequence caused by the first type of mistake (Figure 5), and $\mathcal{A}^{s_2}$, which accepts a sequence illustrating the second type of mistake (Figure 6).

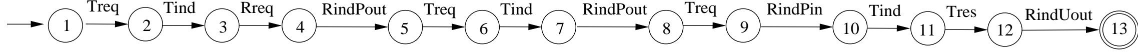


Fig. 5. $\mathcal{A}^{s_1}$

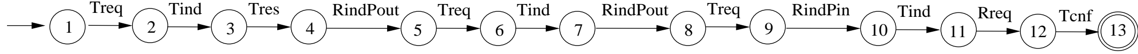


Fig. 6. $\mathcal{A}^{s_2}$

Table 2 summaries the language comparison results for each of the two FSAs ($\mathcal{A}^{s_1}$ and $\mathcal{A}^{s_2}$), with $\mathcal{A}$ and $\mathcal{A}^{A}$. It can be seen that the difference between $\mathcal{A}^{s_1}$ and $\mathcal{A}$ and the difference of $\mathcal{A}^{s_2}$ and $\mathcal{A}$ are both not empty (with 13 nodes and 12 arcs), therefore, the sequences accepted by $\mathcal{A}^{s_1}$ and $\mathcal{A}^{s_2}$ are not in the service language, i.e. they are illegal sequences. However, the difference of $\mathcal{A}^{s_1}$ and $\mathcal{A}^{A}$ and the difference of $\mathcal{A}^{s_2}$ and $\mathcal{A}^{A}$ are both empty. Therefore Model A includes these two illegal sequences.

Table 2. Comparison of $\mathcal{A}^{s_1}$ and $\mathcal{A}^{s_2}$ with $\mathcal{A}$ and $\mathcal{A}^{A}$

$\mathcal{A}^{s_1} - \mathcal{A}$		$\mathcal{A}^{s_1} - \mathcal{A}^{A}$		$\mathcal{A}^{s_2} - \mathcal{A}$		$\mathcal{A}^{s_2} - \mathcal{A}^{A}$	
$\mid S \mid$	$\mid \Delta \mid$	$\mid S \mid$	$\mid \Delta \mid$	$\mid S \mid$	$\mid \Delta \mid$	$\mid S \mid$	$\mid \Delta \mid$
13	12	0	0	13	12	0	0

5 The Revised CES Protocol and its CPN

5.1 Changes to the CES Protocol

In [14] we have shown that the CES protocol could fail when wrapping of sequence numbers is possible. We can design a mechanism to prevent the outgoing CESE from sending a TerminalCapabilitySet message with sequence number i if a capability transfer identified by number i is still outstanding.

Because the CES protocol uses modulo `(maxSeqNo+1)` (`maxSeqNo` stands for maximum sequence number, which is 255) to calculate the sequence number of a TerminalCapabilitySet message, it is not possible for sequence numbers to wrap when there are up to `(maxSeqNo+1)` outstanding capability transfers. Therefore if the outgoing CESE knows the number of outstanding capability transfers, and only sends a TerminalCapabilitySet message when there are `maxSeqNo` or less outstanding capability transfers, the error with the original CES protocol will be removed. So we firstly propose the following change to the CES protocol.

Change 1. *A new state variable, ctrl_CNT, which ranges from zero to (*`maxSeqNo+1`*), is used by the outgoing CESE. A TerminalCapabilitySet message can not be sent if ctrl_CNT >* `maxSeqNo`*. The initial value of ctrl_CNT is set to zero, then once a TerminalCapabilitySet message is sent, its value is increased by one, and when an acknowledgment is received, its value is reduced by one.* □

With the original CES protocol, if a capability transfer is aborted at both the outgoing and the incoming sides (i.e. REJECTindPout and REJECTindPin have occurred), no acknowledgment is generated for this capability transfer. However, to make sure that the CES protocol with Change 1 can work properly, for *every* capability transfer, an acknowledgment must be sent to the outgoing side to have the value of ctrl_CNT decreased, otherwise, the system will deadlock. Therefore, we suggest another change to the CES protocol.

Change 2. *When the incoming CESE is in its AWAITING RESPONSE state and receives a TerminalCapabilitySetRelease message, apart from issuing a REJECT.indication primitive (SOURCE =PROTOCOL), the CESE also sends a TerminalCapabilitySetAbort message that is defined in Table 3 with its sequenceNumber field set to the current value of in_SQ.* □

Table 3. Definition of TerminalCapabilitySetAbort message

Function	Message	Direction	Field
abort	TerminalCapabilitySetAbort	incoming → outgoing	sequenceNumber

Then we define the *Revised CES Protocol* as follows.

Definition 1. *The Revised CES Protocol is the CES protocol defined in H.245 [9] with Changes 1 and 2.* □

5.2 CPN Model for the Revised CES Protocol

Figure 7 shows the CPN model for the revised CES protocol. The declarations of this model are given in Figure 8. Modifications made to the CES protocol CPN of Figure 1 are summarised as follows.

1. The colour set of place outgoingCESE is changed from states to stateOut which is defined as a product of three colour sets, sequenceNo, st and ctrlCNT. Colour set ctrlCNT is defined as a set of integers ranging from 0 to 256 (i.e. the maxSeqNo used by the CES protocol plus 1).
2. Corresponding to the above modification, the inscriptions of the arcs connected to place outgoingCESE are revised by adding the ctrl_CNT information. According to Change 1, upon receiving an acknowledgment (valid or invalid), the value of ctrl_CNT kept by the outgoingCESE is decreased by 1. So the inscriptions of the outgoing arcs from transitions TRANSFERcnf, REJECTindUout, IGNOREout or DISCARD to outgoingCESE, each has a component ct-1, where ct is a variable of type ctrlCNT and represents the current value of ctrl_CNT. Meanwhile, the occurrence of TRANSFERreq increases the value of ctrl_CNT by 1, so the inscription of the arc from this transition to outgoingCESE has a component ct+1.
3. Following Change 1, the guard of transition TRANSFERreq is augmented with an extra condition ct<=255, modelling that when there are already (maxSeqNo+1) outstanding capability transfers, a TerminalCapabilitySet message can not be sent.
4. Because of the change of the colour set of place outgoingCESE, the incomingCESE now does not share the same colour set with the outgoingCESE. The colour set states of the CES protocol CPN is renamed to statesIn to type place incomingCESE in the revised CPN.
5. In accordance with Change 2, colour set inM now includes three types of messages, where capAbort models the newly defined TerminalCapabilitySetAbort message. Therefore place revChannel (typed by inMessageQueue) can hold three types of messages. Moreover, the occurrence of transition REJECTindPin sends a capAbort message with sequence number iSQ (representing the current value of in_SQ), and a guard length(mIn)<maxQLr is added to transition REJECTindPin.

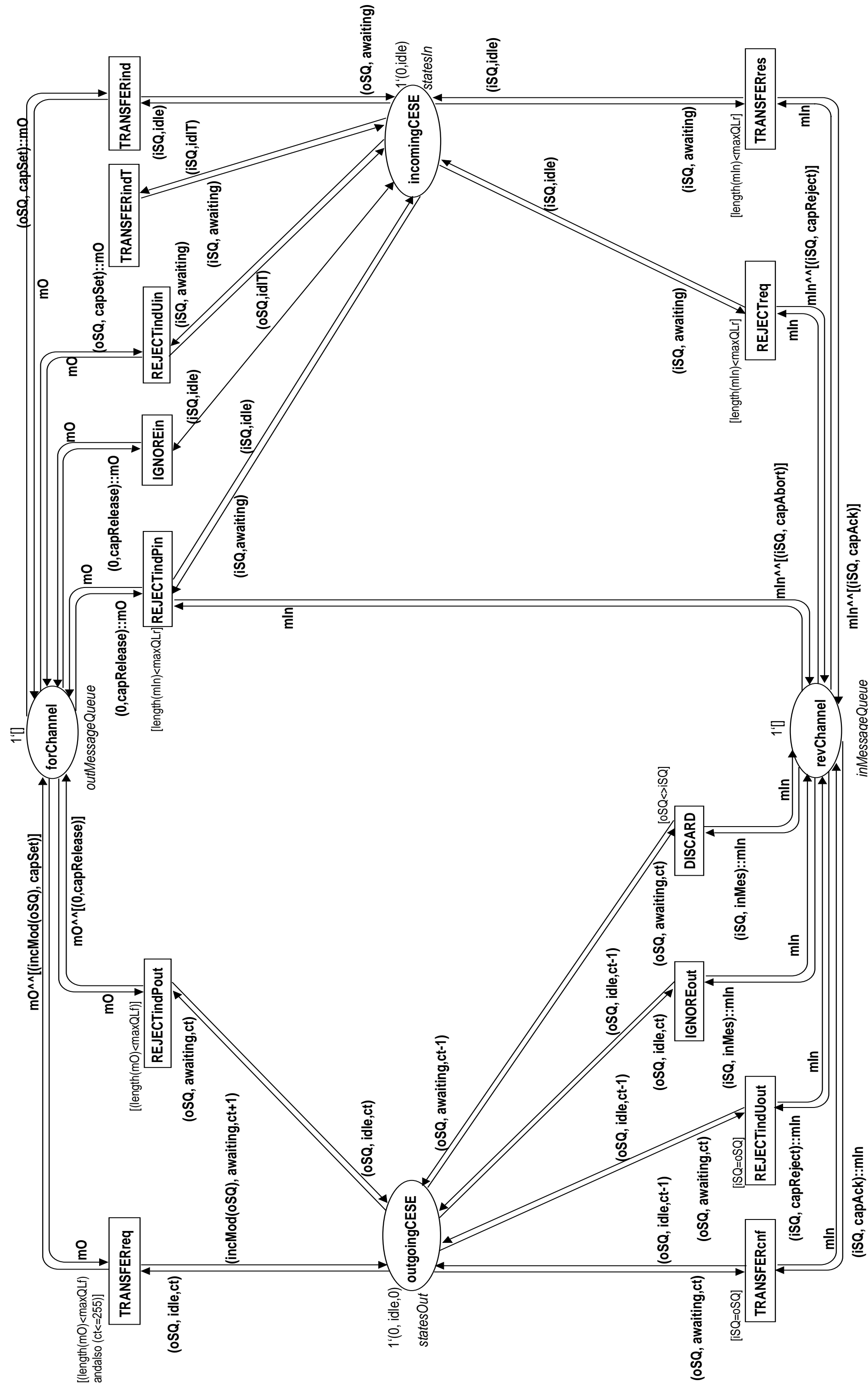


Fig. 7. CPN model of the revised CES protocol

```
(*-maxQLf: maximum queue length of forward channel-*)
(*-maxQLr: maximum queue length of reverse channel-*)
 val maxQLf = 3;
 val maxQLr = 3;

(*---sequence number/values of state variable ---*)
 color sequenceNo = int with 0..255;

(*---ctrl_CNT---*)
 color ctrlCNT = int with 0..256;

(*---states of outgoing CESE---*)
 color st = with idle | idlT | awaiting;
 color statesOut = product sequenceNo*st*ctrlCNT;

(*---states of incoming CESE---*)
 color statesIn = product sequenceNo*st;

(*---types of messages sent by outgoing CESE---*)
 color outM = with capSet | capRelease;
(*---messages sent by outgoing CESE---*)
 color outMessages = product sequenceNo*outM;
(*---message queue of forward channel---*)
 color outMessageQueue = list outMessages;

(*---types of messages sent by incoming CESE---*)
 color inM = with capAck | capReject | capAbort;
(*---messages sent by incoming CESE---*)
 color inMessages = product sequenceNo*inM;
(*---message queue of reverse channel---*)
 color inMessageQueue = list inMessages;

(*---variables---*)
 var mO: outMessageQueue;
 var mIn: inMessageQueue;
 var inMes: inM;
 var oSQ, iSQ: sequenceNo;
 var ct: ctrlCNT;

(*---function for calculating new value of---*)
(*-sequence number or state variables-*)
 fun incMod(oSQ) = (oSQ+1) mod 256;
```

Fig. 8. The declarations of the CPN of the revised CES protocol

In the original CPN model, no restriction was made on the maximum length of the queue in place `forChannel` or `revChannel`, i.e. each queue had infinite capacity. In reality the channel capacity is finite, but we do not know its maximum value. We therefore allow the maximum length of the queue in each of the two places to be *arbitrarily* large. Thus the revised CPN model becomes a *parametric* CPN model. We have defined two parameters, `maxQLf` and `maxQLr` (see top of Figure 8). `maxQLf` represents the maximum queue length of place `forChannel`. Similarly `maxQLr` represents the maximum queue length of place `revChannel`. Both parameters are positive integers. The guards of transitions `TRANSFERreq` and `REJECTindPout` (`length(mO)<maxQLf`), and `REJECTindPin`, `REJECTreq` and `TRANSFERres` (`length(mIn)<maxQLr`) then set the limit on the maximum length of the queue in places `forChannel` and `revChannel` respectively.

6 State Space Analysis of the Revised CES Protocol

Based on the same reason that is discussed in Section 4.2, we choose to analyse the revised CES protocol CPN in Figure 7 when fixing `maxQLf` to 3, `maxQLr` to 3 and using $\{0, 1\}$ as the range of sequence numbers. We call such a model, *Model RA* (Revised Model A).

The state space of Model RA is generated by the OG tool of Design/CPN [5] and part of the state space report is shown in Table 4. By examining the report we prove that Model RA satisfies Properties 1 to 4.

Table 4. Part of the state space report for Model RA

	Occurrence Graph		SCC Graph	
Statistics	Nodes:	110	Nodes:	1
	Arcs:	246	Arcs:	0
	Secs:	0	Secs:	0
Home Properties	Home Markings:	All		
Liveness Properties	Dead Markings:	None		
	Dead Transitions:	`REJECTindUin`,	`TRANSFERindT`	
	Live Transitions:	`DISCARD`,	`IGNOREin`	
		`IGNOREout`,	`REJECTindPin`	
		`REJECTindPout`,	`REJECTindUout`	
		`REJECTreq`,	`TRANSFERcnf`	
		`TRANSFERind`,	`TRANSFERreq`	
		`TRANSFERreq`		

The *Home Properties* stated in the report show that all the markings of Model RA are home markings and this is confirmed by the SCC (Strongly Connected Component) graph of the OG having only one node. So Property 3 (initial state is a home state) is satisfied by Model RA. This implies Properties 1 and 2 also hold, which is confirmed by there being no dead markings and one SCC.

With Model RA, `REJECTindUin` and `TRANSFERindT` are dead transitions (see the Liveness Properties), and all the other transitions are live transitions. This is what we expected. Therefore Property 4 holds for Model RA. Moreover, from the *Best Upper Multi-set Bounds* of place `forChannel` of Model RA (Table 5) we can observe that there are never two successive `capSet` messages in the queue (i.e. transition `REJECTindUin` can never be enabled). Also, place `revChannel` can hold at most two acknowledging messages, although the maximum length of the queue in this place (i.e. `maxQLr`) has been set to 3. This is expected because the range of sequence numbers of Model RA is $\{0, 1\}$, i.e. at most two outstanding capability transfers are allowed, thus it is not possible for more than two acknowledgments to be in the system.

Table 5. Best Upper Multi-set Bounds of places `forChannel` and `revChannel` (Model RA)

`forChannel`	`1‘[]++ 1‘[(0,capSet)]++ 1‘[(0,capSet),(0,capRelease)]++ 1‘[(0,capSet),`
	`(0,capRelease),(1,capSet)] ++ 1‘[(0,capRelease)] ++ 1‘[(0,capRelease),(0,capSet)]`
	`++ 1‘[(0,capRelease),(0,capSet),(0,capRelease)] ++ 1‘[(0,capRelease),(1,capSet)]`
	`++ 1‘[(0,capRelease),(1,capSet),(0,capRelease)] ++ 1‘[(1,capSet)]`
	`++ 1‘[(1,capSet),(0,capRelease)]++ 1‘[(1,capSet),(0,capRelease),(0,capSet)]`

`revChannel`	`1‘[]++ 1‘[(0,capAck)]++ 1‘[(0,capAck),(1,capAck)]++ 1‘[(0,capAck),(1,capReject)]`
	`++ 1‘[(0,capAck),(1,capAbort)]++ 1‘[(0,capReject)]++ 1‘[(0,capReject),(1,capAck)]`
	`++ 1‘[(0,capReject),(1,capReject)] ++ 1‘[(0,capReject),(1,capAbort)]`
	`++ 1‘[(0,capAbort)] ++ 1‘[(0,capAbort),(1,capAck)]++ 1‘[(0,capAbort),`
	`(1,capReject)] ++ 1‘[(0,capAbort),(1,capAbort)] ++ 1‘[(1,capAck)]`
	`++ 1‘[(1,capAck),(0,capAck)] ++ 1‘[(1,capAck),(0,capReject)]`
	`++ 1‘[(1,capAck),(0,capAbort)] ++ 1‘[(1,capReject)] ++ 1‘[(1,capReject),`
	`(0,capAck)]++ 1‘[(1,capReject),(0,capReject)]++ 1‘[(1,capReject),(0,capAbort)]`
	`++ 1‘[(1,capAbort)] ++ 1‘[(1,capAbort),(0,capAck)]`
	`++ 1‘[(1,capAbort),(0,capReject)] ++ 1‘[(1,capAbort),(0,capAbort)]`

The satisfaction of the four general properties by Model RA provides a strong indication of the functional correctness of the system. The selection of parameter values (i.e. `maxQLf`= 3 and `maxQLr`= 3), and the range of sequence numbers (i.e. `maxSeqNo=1` for Model RA) has considered most application scenarios of the revised CES protocol.

To gain more confidence, we also generate the state spaces of the revised CES protocol CPN, for various configurations of `maxQLf`, `maxQLr` and `maxSeqNo`. It has been observed from the state space reports that with each of the configurations, all markings of the CPN are home markings and all transitions except for `REJECTindUin` and `TRANSFERindT` are live transitions. This suggests that with all of these configurations, the revised CES protocol satisfies the four general properties.

Therefore, based on these analysis results we are highly confident that the revised CES protocol will function correctly.

7 Language Analysis of the Revised CES Protocol

7.1 The CES Service and Protocol Languages

As with the language analysis of Model A (Section 4.2), in this section, we will also use the CES service language obtained in [15] and [17] directly. The protocol language of the revised CES protocol is extracted from the OG of the CPN of the protocol in the same way as we obtained the protocol language of Model A.

7.2 Language Analysis of Model RA

The FSA that represents the CES service language for Model RA (denoted as $\mathcal{A}$) is shown as the whole graph in Figure 9, which is the same service language shown in Figure 4. The minimal FSA that represents the protocol language of Model RA (denoted as $\mathcal{A}^{RA}$) is obtained using FSM tools, and is shown in Figure 9 as the subgraph without the 4 grey nodes (i.e. nodes 8, 12, 13 and 14) and the 12 grey arcs associated with the 4 nodes. The same key is used in Figure 9 as that used in Figure 4.

Table 6 shows the number of nodes and arcs of the four minimal FSAs: $\mathcal{A}$, $\mathcal{A}^{RA}$, $(\mathcal{A} - \mathcal{A}^{RA})$, and $(\mathcal{A}^{RA} - \mathcal{A})$. Because $(\mathcal{A}^{RA} - \mathcal{A})$ has no nodes or arcs (empty), the protocol language of Model RA

Table 6. Comparison of $\mathcal{A}$ and $\mathcal{A}^{RA}$

$\mathcal{A}$		$\mathcal{A}^{RA}$		$\mathcal{A} - \mathcal{A}^{RA}$		$\mathcal{A}^{RA} - \mathcal{A}$	
$\lvert S \rvert$	$\lvert \Delta \rvert$	$\lvert S \rvert$	$\lvert \Delta \rvert$	$\lvert S \rvert$	$\lvert \Delta \rvert$	$\lvert S \rvert$	$\lvert \Delta \rvert$
14	36	10	24	24	63	0	0

is included in the service language. Therefore Property 5 holds for Model RA. This means that the system modelled by Model RA does not generate any illegal sequences. Interestingly, $(\mathcal{A} - \mathcal{A}^{RA})$ (Table 6) and $(\mathcal{A} - \mathcal{A}^{A})$ (Table 1) are equivalent FSAs. This implies that Model A and Model RA have the same missing sequences. However, Model RA does not generate any illegal sequences while Model A does.

In the following, we investigate the missing sequences, i.e. the sequences accepted by $(\mathcal{A} - \mathcal{A}^{RA})$. As mentioned in Section 3, by doing so, we show that the service provided by the system modelled by Model RA provides us services that we need, i.e. it is an acceptable implementation of the CES service.

From Figure 9 we know that the sequences that are missing from the protocol language of Model RA are caused by the absence of nodes 8, 12, 13 and 14, and their associated arcs. We see that the partial sequence from node 4 via 8 to node 1 represents that in the CES service, a REJECT.indication primitive (with SOURCE = USER or PROTOCOL) can be issued by the incoming CESE before a REJECT.indication (SOURCE = PROTOCOL) primitive has been issued by the outgoing CESE (see Figure 10 (a)). However, as can be seen from Figures 1 and 7, the CES protocol and its revised version only allow a REJECT.indication primitive (SOURCE=PROTOCOL) to be issued by the incoming CESE (modelled by transition `REJECTindPin`) when a TerminalCapabilitySetRelease (i.e. `capRelease`) message is received (see Figure 10 (b)). This means that a

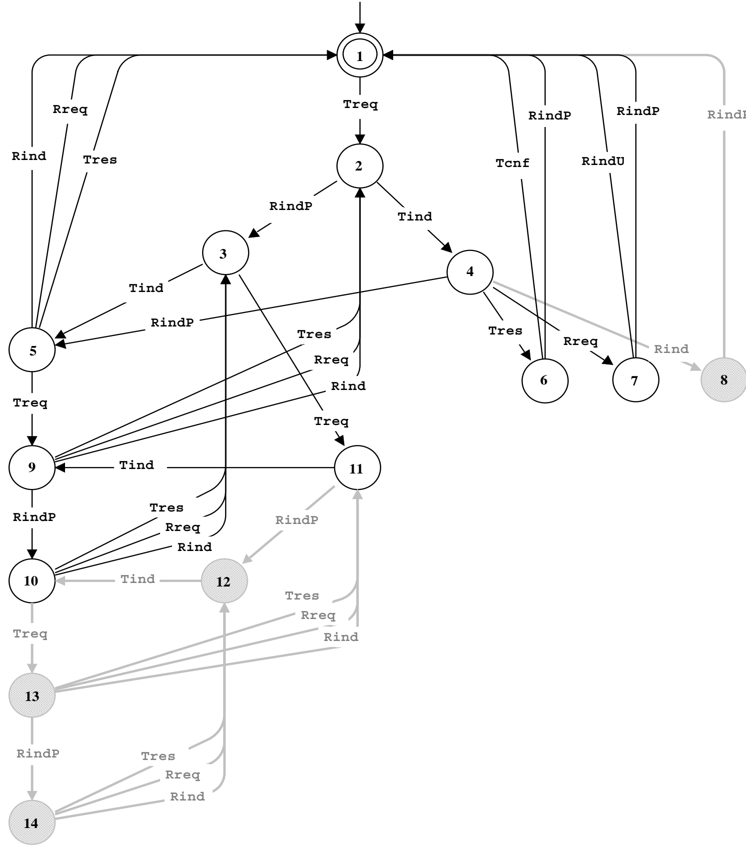
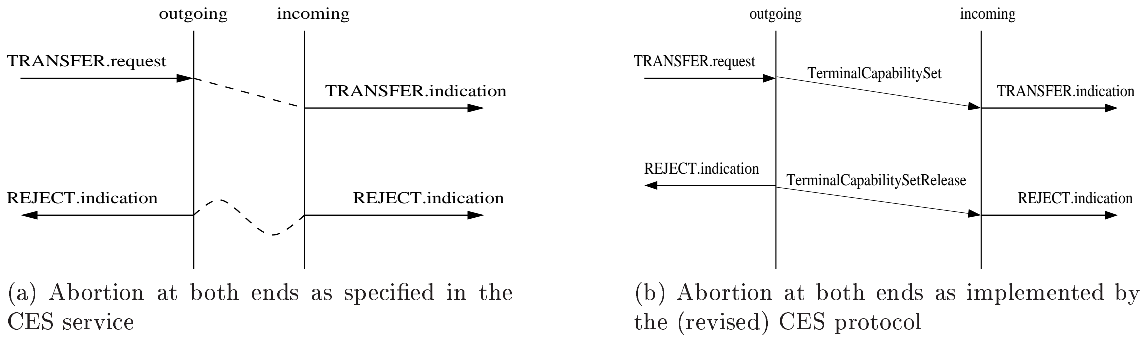


Fig. 9. $\mathcal{A}^{RA}$ (the subgraph without grey nodes and arcs) and $\mathcal{A}$ (the whole graph)



(a) Abortion at both ends as specified in the CES service

(b) Abortion at both ends as implemented by the (revised) CES protocol

Fig. 10. Sequences for abortion at two ends

REJECT.indication primitive must have been issued by the outgoing CESE first. This is acceptable and sufficient behaviour. (It is not important for the incoming CESE to be able to issue a REJECT.indication before the outgoing CESE issues one, but such behaviour is allowed by the service).

The other 3 nodes (12, 13 and 14) are missing due to the limit we have put on `maxQLf` and `maxSeqNo`. The CES service language represented by $\mathcal{A}$ is the service language under the assumption that the CES service provider can transfer at most two transfer requests at any time. With the (revised) CES protocol, this corresponds to the maximum number of `capSet` messages in place

forChannel being two. Because Property 4 holds for Model RA, it not possible for two successive capSet messages to be in place forChannel, otherwise transition REJECTindUin will become enabled. Thus capSet and capRelease messages are interleaved in place forChannel. So when there are two capSet messages in the place, there are four possible patterns of the markings of this place (p_1, p_2, p_3 and p_4), which are:

$$\begin{aligned}
p_1 &: [\texttt{capSet}, \texttt{capRelease}, \texttt{capSet}] \\
p_2 &: [\texttt{capSet}, \texttt{capRelease}, \texttt{capSet}, \texttt{capRelease}] \\
p_3 &: [\texttt{capRelease}, \texttt{capSet}, \texttt{capRelease}, \texttt{capSet}] \\
p_4 &: [\texttt{capRelease}, \texttt{capSet}, \texttt{capRelease}, \texttt{capSet}, \texttt{capRelease}]
\end{aligned}$$

where sequence numbers are omitted from the above representations.

However, Model RA only allows the first pattern, p_1, because maxQLf= 3. Referring to Figure 9, we see that the occurrence of the sequence from node 1, via nodes 2, 3, 11, to 12 in the revised CES protocol would require the marking of place forChannel to have pattern p_2. We confirm that this is the case by changing the model.

We modify the guard of transition REJECTindPout from [length(mO)<maxQLf] to [length(mO) <=maxQLf] (refer to Figure 7), then pattern p_2 can exist. We call this model *Model RA′*. The protocol language of this model is generated, and it turns out to be the subgraph in Figure 9 without nodes 8, 13, and 14 and their associated arcs. Node 12 and the arcs between this node and nodes 10 and 11 are included in the FSA representing the protocol language of this changed model. Thus the fact that node 12 and its associated arcs are missing is due to the capacity constraint and is not a problem of the protocol.

We then further change Model RA′ to *Model RA″* by setting maxQLf= 4 and maxSeqNo=2 to allow patterns p_3 and p_4. Then the protocol language of Model RA″ is generated, and it is the subgraph in Figure 9 without node 8 and its associated arcs. This shows that the absence of nodes 13 and 14 and their associated arcs is caused by the limit to the maximum sequence number and the channel capacity.

To summarise, language analysis shows that Model RA provides all the essential parts of the CES service without generating any illegal sequences. The missing sequences are caused by the capacity and maximum sequence number constraints, and the way the protocol implements rejection.

8 Conclusion

As a continuation of the work presented in [14], in this paper we firstly analyse the language property of the CES protocol. Then we revise the CES protocol by adding to the protocol a mechanism to prevent the outgoing CESE from sending a TerminalCapabilitySet message when the number of outstanding capability transfers is equal to the maximum sequence number plus 1. With this modification, it is no longer possible for the outgoing CESE to confuse an old acknowledgment with the same sequence number as that of the expected acknowledgment, even when sequence numbers wrap. This ensures that the outgoing CES service user is always informed correctly of the acknowledgment corresponding to the capability set it has sent. State space analysis and language analysis (for the values of the parameters considered) have shown that the revised protocol possesses the desired general properties and meets the sequencing requirements of the CES service.

To achieve a thorough verification result for the CES protocol and its revised version, and to explore the approach to verifying parametric systems, we are interested in verifying the CES protocol and its revised version for any values of the parameters. In [15], starting with observing the structural regularities of the state spaces of the CES service CPN when the value of channel capacity is varied, we have found a recursive representation of the (family of) state spaces of the CPN in terms of the channel capacity. In [16,17], we have also shown that a recursive representation also exists for the deterministic FSA which represents the CES service language for the same parameter. In this paper, language analysis of Model RA has provided us with high confidence

that the revised CES protocol in general is a refinement of the CES service. So we anticipate a similar recursive representation of the protocol language of the revised CES protocol for any values of the parameters. Therefore as the next step, we shall investigate the possible structural regularities of the state space of the CPN model of the revised CES protocol. Then, based on this, we hope to find a symbolic representation for the protocol language of the revised CES protocol, which allows us to verify the revised CES protocol symbolically for all values of channel capacity.

Acknowledgements

The authors gratefully acknowledge the constructive comments of the anonymous reviewers. They would also like to acknowledge the generosity of their colleague, Guy Gallasch, for agreeing to present this paper.

References

1. W. A. Barrett and J. D. Couch. *Compiler Construction: Theory and Practice.* Science Research Associates, 2nd edition, 1986.
2. J. Billington. Formal Specification of Protocols: Protocol Engineering. In A. Kent, J. G. Williams, and R. Kent, editors, *Encyclopedia of Microcomputers, Vol. 7*, pages 299–314. Marcel Dekker, Inc., 1991.
3. J. Billington, G. E. Gallasch, and B. Han. A Coloured Petri Net Approach to Protocol Verification. In *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, Lecture Notes in Computer Science, Vol. 3098, pages 210–290. Springer, 2004.
4. G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1990.
5. Design/CPN homepage. http://www.daimi.au.dk/designCPN/.
6. ITU-T. *Recommendation G.722, 7 kHz Audio-Coding within 64 kbit/s*, 1988.
7. ITU-T. *Recommendation H.261,Video Codec for Audiovisual Services at* $p \times 64$ *kbit/s*, March 1993.
8. ITU-T. *Recommendation H.323, Packet-based multimedia communications systems*, July 2003.
9. ITU-T. *Recommendation H.245, Control protocol for multimedia communication*, January 2005.
10. K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer, 2nd edition, 1997.
11. P. E. Jones. Overview of H.323 (power point presentation). http://www.packetizer.com/voip/h323/papers/overview_of_h323.html, June 2004.
12. L. M. Kristensen, S. Christensen, and K. Jensen. The practioner's guide to coloured Petri nets. *International Journal on Software Tools for Technlogy Transfer*, 2:98–132, 1998.
13. AT&T FSM Library^{TM}. http://www.research.att.com/sw/tools/fsm/.
14. L. Liu and J. Billington. Modelling and Analysis of the CES Protocol of H.245. In *Proc. 3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 95–114, Aarhus, Denmark, August 2001.
15. L. Liu and J. Billington. Tackling the Infinite State Space of a Multimedia Control Protocol Service Specification. In *Proc. 23rd Int. Conf. on Application and Theory of Petri Nets*, Lecture Notes in Computer Science, Vol. 2360, pages 273–293. Springer, June 2002.
16. L. Liu and J. Billington. A Proof of the Recursive Formula for the Infinite Service Language of the CES Protocol. Technical report, CSEC-13, Computer Systems Engineering Centre, University of South Australia, January 2003 (revised in June 2004).
17. L. Liu and J. Billington. Obtaining the Service Language for H.245's Multimedia Capability Exchange Signalling Protocol: the Final Step. In *Proc. 10th Int. Multi-Media Modelling Conf.*, pages 323–328, Brisbane, Australia, January 2004.

Synthesis of Active Controller for Resources Allocation Systems

B. ZOUARI and S. ZAIRI

LIP2 Laboratory - University of Tunis Manar-
Faculte des Sciences Tunis- Dep. Informatique
Campus Universitaire - 2092 - Manar II- Tunis(ia)
belhassen.zouari@fst.rnu.tn

Abstract: In this paper, we present a controller synthesis method for resource allocation systems based on coloured Petri nets (CPN). The first contribution of this work is the use of High-level Petri nets as a basis model for the specification of the plant model as well as for the controller generation. Indeed, we present an automatic generation of a controller characterised by a CPN subnet (with a fixed number of places and transitions) representing its behaviour. The second contribution is the complete integration of the implementation of our method, called SACoRAS, in the CPN Tools. SACoRAS is made up of modules (graphical/textual editor, controller generator) that interact with CPN Tools modules (state space module, etc.).

Keywords: Supervisory control, resource allocation systems, coloured Petri nets, controller synthesis, active controller.

1 Introduction

Supervisor synthesis, originated by the work of Ramadge and Wonham [8], is subject to numerous works in supervisory control of discrete-event systems (DES) [2][3][4][6][11]. A synthesised supervisor prevents a given system from some undesirable behaviours as for example those leading to deadlocks. In many cases, deadlocks must be imperatively avoided (recovering from a deadlock can be expensive). In many of these works, Resource Allocation Systems (RAS) have been covered by supervisory control methods [9][1]. RAS represent a particular class of DES suitable for describing numerous plant models as flexible manufacturing systems (FMS), production systems, workflow systems, etc. A RAS may be viewed as a collection of processes using shared resources in a repetitive manner.

Among the most efficient methods used in supervisory control of DES, Petri nets based approaches have been frequently adopted. One may distinguish methods based on structural properties of Petri nets [1] and those based on their dynamic properties as exploring state graphs [2][3][11]. The main limitation with both of these approaches is the use, in most cases [2][3], of classical Petri net models (i.e. ordinary or close models). These are known for their main disadvantage that is the great size of models at the specification stage. Moreover, the growing availability of efficient methods and tools based on High-level Petri net models is making ordinary Petri nets less attractive.

In this paper, we introduce a High-level Petri net approach, called "Active Controller", allowing the automatic synthesis of controllers for RAS. We consider the problem of forbidden states under the hypotheses of existence of uncontrollable transitions [8][2]. Our objective is to develop a supervisory control method and a tool based on the standard coloured Petri net (CPN) model [5]. Our motivation is to make possible the use of the available CPN methods and tools and to exploit the results and benefits of the recent Petri net theory research. We also present an implementation of our method based on CPN Tools [7].

It is worth noting that our work is integrated in the framework of an engineer-oriented application, called SACoRAS (for **S**ynthesis of **A**ctive **Co**ntroller for **R**esources **A**llocation **S**ystems), that may be viewed as a functional layer above CPN Tools. Hence, a user (a non-specialist of Petri nets) starts by introducing a specification through SACoRAS interface (graphical or textual editor) in terms of RAS constructs (process structures, resources description, allocation/restitution primitives) and control constraints (forbidden states and uncontrollable transitions). Then, this specification is automatically translated into CPN to make possible its exploiting by CPN Tools, in particular for the generation of the occurrence graph. From this reachability graph, we apply (through two SACoRAS modules) the active controller method to obtain the CPN modelling the controlled system. The first module determines the *admissible graph* on the basis of both the occurrence graph and the control specifications. The second module generates the controlled model (the initial model connected to the controller) that ensures the initial RAS to behave correctly and not to reach any forbidden state. We also prove the equivalence between the admissible behaviour (the aimed behaviour of the RAS) and the controlled behaviour (that obtained from the generated model). The paper is organised as follows: resource allocation systems are presented in section 2. In section 3, we introduce the synthesis method of the "Active Controller". In section 4, we present the SACoRAS application. Finally, section 5 presents the conclusion of the paper.

2 Resource Allocation Systems (RAS)

In this section, we present RAS used for the specification of the uncontrolled system. A RAS, as defined in [9], consists of a set of parallel processes. Each process is defined as being a succession of states, where each state is characterised by a set of resources necessary to its execution. These resources represent finite buffering capacity, machines, data or any other tool that is used in the execution of the process. Thus, formally, a process is described as being a sequence of vectors representing the resource allocations.

In our work, we consider a RAS as a system made up of a set of resources, that can be of many types and available in any amount of copies, and a set of generic processes, that can be instantiated in a collection of processes. Each generic process has a repetitive and conservative behaviour.

Formally, a RAS is a tuple (R,GP, *GetR*, *PutR*) where:

- R = (RT, Rnb) is the resource specification, such that
  RT is the finite set of resource types,

$Rnb \in Bag(RT)$[1] is a multiset on RT representing the amount of available copies of each resource type.

- GP is the finite set of generic processes.
  A generic process G_i is defined by a finite state machine (FSM) as follows:
  $Gi=<P_{si},F_i,Ci^->$such that:
  - $P_{si}=\{p_{0i}, p_{i1}, p_{i2},\ldots\ldots\ldots, p_{il(s)}\}$ is the finite set of states, where p_{0i} represents the 'idle' place of the generic process. The 'idle' place is the initial place of whole process instance.
  - $F_i : P_{si} \rightarrow P_{si}$ is the flow relation representing the state transitions.
  - C_i is the finite set of identities of process instances.

- *GetR* : $F_i \rightarrow Bag(R)$ is the resource allocation function

$$f \rightarrow \sum_{i=1}^{|R|} \alpha\, i\, ri$$

- *PutR* : $F_i \rightarrow Bag(R)$ is the resource restitution function

$$f \rightarrow \sum_{i=1}^{|R|} \lambda\, i\, ri$$

To each RAS, as previously defined, we can derive a CPN representing whole the RAS behaviour. Intuitively, the underlying CPN is built from a collection of FSMs, representing the different generic processes, a resources place, and connection arcs related to the resource management.

Formally, a marked CPN $<N,M_0>$ representing a RAS, where M_0 is its initial marking and $N = <P, T, C, W^-,W^+>$, is defined as following:

- $P= P_S \cup \{p_R\}$is a set of places where $P_S = \bigcup_{i=1}^{|GP|} P_{si}$ and p_R represents the resources place such that $P_{si} \cap P_{sj} =\emptyset$ and $P_S \cap p_R = \emptyset$.

- T is a set of transitions recursively built as follows:

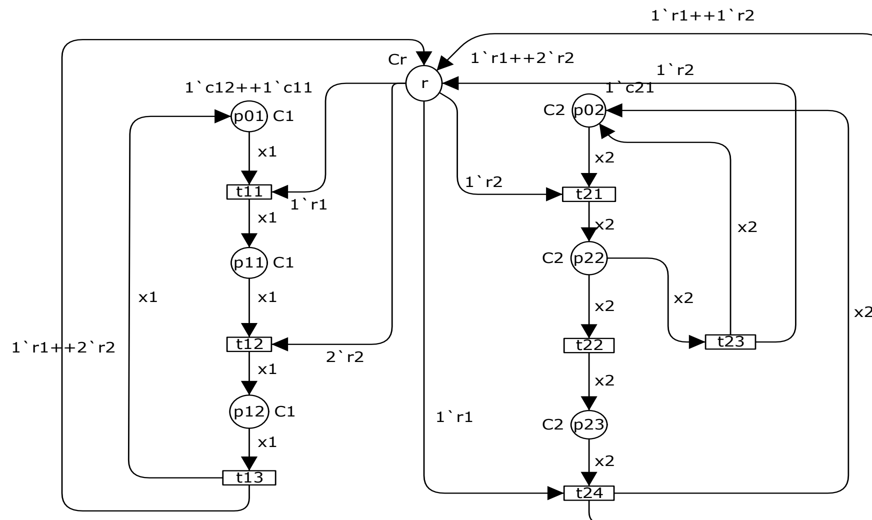
  Initially $T = \emptyset$, then $T=T \cup \{f_{ij}\}$ $i=1,..,|GP|$; $j=1,..,|F_i|$.
  The elements of T may be lexicographically renamed so as we can use the notation $T= \{t_i\} i=1,..,|T|$.
- $C = \bigcup_{i=1}^{|GP|} C_i \bigcup_{i=1}^{|GP|} (C_i \times C_r) \cup C_r$ where C_r is a colour domain of p_R, $C_r = RT$
- $W^-= W_S^- \cup W_R^-$, where $W_S^-: (P_S \times T) \rightarrow \{0, \bigcup_{i=1}^{|GP|} X_i\}$ such that X_i is a variable defined on C_i, and $W_R^- : (p_R \times T) \rightarrow Bag(C_r)$ such that $\forall\ t \in T$ $W_R^-(p_R,t)= GetR(f_{ij})$
- $W^+= W_S^+ \cup W_R^+$, where $W_S^+: (P_S \times T) \rightarrow \{0, \bigcup_{i=1}^{|GP|} X_i\}$ such that X_i is a variable defined on C_i and $W_R^+ : (p_R \times T) \rightarrow Bag(C_r)$.such that $\forall\ t \in T$ $W_R^+(p_R,t)= PutR(f_{ij})$

[1] Bag(R) is the set of multisets over the set R.

- The initial marking of the system is defined as follows: $\forall p \in P_S \backslash (\bigcup_{i=1}^{|GP|} p_{0i})$; $M_0(p) = 0$; $M_0(p_R) = Rnb$; and $\forall p_{0i} \in P_0$; $M_0(p_{0i}) \geq 1$.

2.1 Example

Let us consider an example of RAS made up of two generic processes P_1 and P_2 where P_1 is instantiated by $C_1 = \{c_{11}, c_{12}\}$ and P_2 by $C_2 = \{c_{21}\}$. All these processes share resources represented by $C_r = \{r_1, r_2\}$. The following CPN model describes this system.



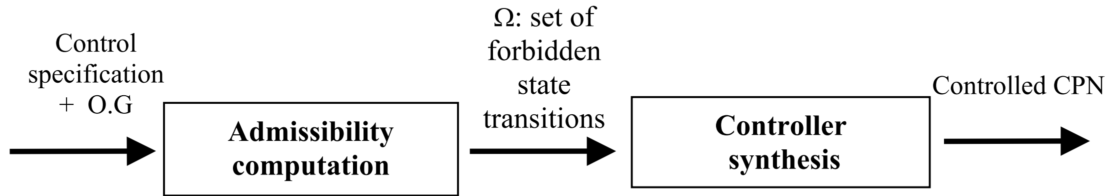
Figure[1]: Uncontrolled system

C_1, C_2 and C_r are the colour classes of the CPN.
The colour domains of places are:
$C(p_{01})=C(p_{11})=C(p_{12})=C_1$, $C(p_{02})=C(p_{22})=C(p_{23})=C_2$, and $C(r)= C_r$.
Here, colour functions are of two types: variables (like X_1 defined on C_1 and X_2 defined on C_2) or constants (like r_1, r_2, or r_1+2r_2).
An initial marking of this net is: $M_0(p_{01})=C_1$; $M_0(p_{02})=C_2$; $M_0(r)=r_1+2r_2$;

3 The "Active Controller" approach

In this section, we present the "Active Controller" approach for RAS. The proposed method is based on the dynamic properties of the CPN representing the initial RAS model by exploring its occurrence graph. To apply the "Active Controller" approach, we suppose that the behaviour of the initial uncontrolled system is modelled using CPN as defined in the previous section. We also assume that control constraints are specified through a set of *forbidden states.*

The proposed approach is based on two steps. The following figure describes the inputs and the outputs of each one. A more detailed description of those steps will be given in the following sections.



Figure[2]: "Active Controller" steps.

3.1 Admissible behaviour

The admissible behaviour is an accessible behaviour respecting the control specification. In this section, we present the different steps allowing its computation. We suppose that we have the reachability graph of the initial CPN and the finite set of specified forbidden states. These states can be either blocking markings or any other undesirable markings.

In a first stage, we need to go all over the reachability graph *G* and determine information about some reachability markings (called *dangerous markings*) from which control has to be applied in order to *avoid* reaching one forbidden state.

A *dangerous marking* (or dangerous state) is defined as a node of G, from which there exists at least one transition that inevitably leads (when fired) to a forbidden state. It is worth to note that a dangerous marking (that is not initially specified as forbidden) can be qualified as inadmissible if all firing sequences from this marking inevitably lead to a forbidden or inadmissible state or if it has an uncontrollable transition which leads to a forbidden or inadmissible state. Moreover, it is important to handle, with every dangerous state, information about transitions that must not be fired because each of them leads to a forbidden marking. Let *d* be a dangerous state, then the set *FT(d)* of forbidden transitions related to *d* is the set of transitions enabled from *d* according to the reachability graph *G* and which inevitably lead to a forbidden or inadmissible state. Given a reachability graph *G* and a specification of forbidden states *FS*, the first output of this stage is the set *DS* of dangerous markings and the associated sets *FT(d)* for all $d \in DS$. *FT* may be viewed as an application from *DS* to the set of subsets of T. Its second output is the graph R_c, which is called *admissible graph*, obtained when all forbidden states (specified and calculated) are removed from *G* and representing the desirable behaviour for the controlled system.

For this purpose, we propose an algorithm that computes R_c and *FT(d)* for any $d \in DS$. The idea is to use a colouring technique on graph G (the meaning of colouring here is independent of CPN's meaning) in order to identify paths that lead to forbidden or inadmissible states. The algorithm begins by colouring the specified forbidden markings (FS) and the input arcs of these markings. Then, we use backward propagation and forward propagation of colouring nodes and arcs according to the following rules:

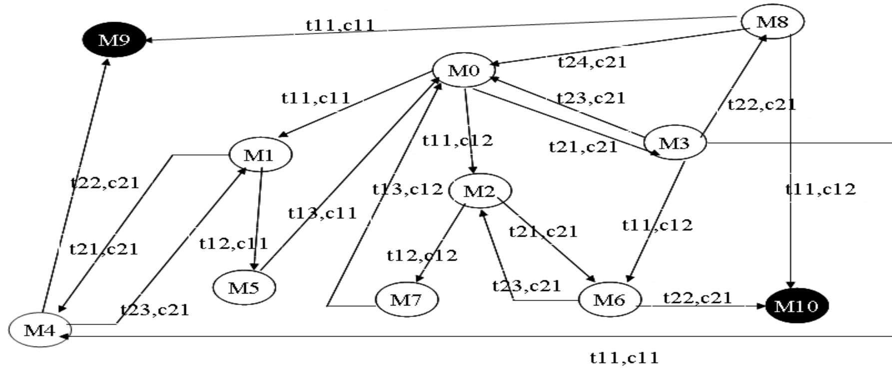
- a node is dangerous if there exists at least one output coloured arc,
- a dangerous node becomes an inadmissible one if all its output arcs are coloured or if it has a forbidden transition, which is uncontrollable,
- all output and input arcs of a forbidden or inadmissible node must be coloured

Algorithm **"Compute Admissibility"** // *Compute R_c and $\Omega = \bigcup_{d \in DS} FT(d)$*

```
Input   G: Reachability graph of the uncontrolled system with G.nodes
           set of nodes and G.arcs set of arcs.
        FS0: Set of initial forbidden states.
        Tu: Set of uncontrollable transitions.
Initially DM=ϕ ; TE=ϕ ; Ω=ϕ ; Rc=G; FIS=FS0;

Repeat
    take a non-coloured node f from FIS ;
    colour the node f ;

  // backward propagation

    For every input arc (x,f) of f do
        colour the arc (x,f) ;
        DM= DM∪{x};
        E=TE∪{(x,(t,c))};

    If t∈Tu then FIS= FIS∪{x}
    If (x,j) is coloured for all j then FIS= FIS∪{x}
    If M0∈FIS then exit   // No solution for the control problem

  // forward propagation

    For every output arc (f,x) of f do
        colour the arc (f,x) ;
    If (x,j) is coloured for all j then FIS= FIS∪{x}

  // eliminate node f from Rc and all its arcs

    Rc.nodes=Rc..nodes\{f};
    Rc..arcs=Rc.arcs\{(f,z) and (z,f) for all z∈Rc..nodes};

Until all nodes of FIS are coloured

    If M0∈ FIS then exit   // No solution for the control problem

    DM= DM\ FIS;
    For all element y of DM do
    For all element (x,(t,c)) of TE do
       If y==x
       then Ω=Ω∪{(x,(t,c))}
Output Rc ,Ω;
```

One can easily verify that the algorithm terminates correctly. As we deal with finite reachability graph, *FS* is finite, and the main loop of the algorithm will necessarily stop.

Let us apply the algorithm on the following graph representing the reachable graph of the example presented previously:

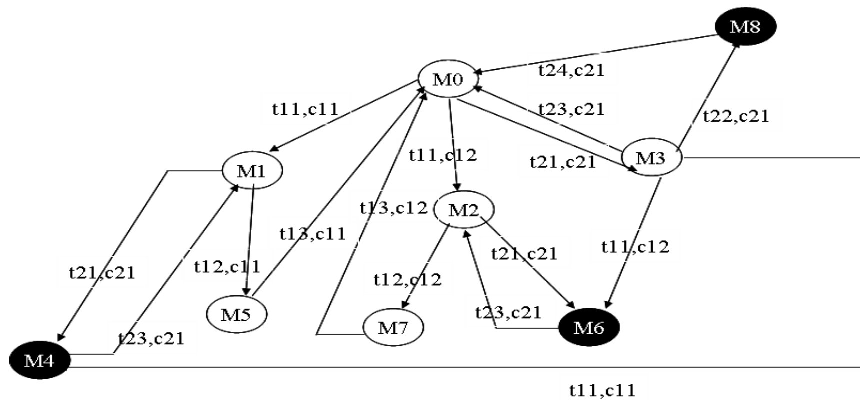


Figure[3]: reachability graph

The graph nodes are described as follow:

	M0	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
p_{01}	$c_{11}+c_{12}$	c_{12}	c_{11}	$c_{11}+c_{12}$	c_{12}	c_{12}	c_{11}	c_{11}	$c_{11}+c_{12}$	c_{12}	c_{11}
p_{11}	0	c_{11}	c_{12}	0	c_{11}	0	c_{12}	0	0	c_{11}	c_{12}
p_{12}	0	0	0	0	0	c_{11}	0	c_{12}	0	0	0
p_{02}	c_{21}	c_{21}	c_{21}	0	0	c_{21}	0	c_{21}	0	0	0
p_{22}	0	0	0	c_{21}	c_{21}	0	c_{21}	0	0	0	0
p_{23}	0	0	0	0	0	0	0	0	c_{21}	c_{21}	c_{21}
r	r_1+2r_2	$2r_2$	$2r_2$	r_1+r_2	r_2	0	r_2	0	r_1+r_2	r_2	r_2

If the specified forbidden states are 'M9' and 'M10' (representing deadlock markings), then the algorithm gives : *DM*={M4,M6,M8}, *FT*(M4)={t_{22}}, *FT*(M6)={t_{22}}, *FT*(M8)={t_{11}} and Ω={((c_{12},c_{11},0,0,c_{21},0,r_2),t_{22}) ; ((c_{11},c_{12},0,0,c_{21},0,r_2),t_{22}); (($c_{11}+c_{12}$,0,0,0,0,c_{21},r_1+r_2),t_{11}) }.
R_c is represented by the following graph:



Figure[4]: Admissibility graph

3.2 Controller construction method

After computing the set of the forbidden state transitions, we describe in this section the generation of the controller CPN and its connection to the initial system.

At this stage, we have two kinds of specification:

- the control specification of supervisory as described in section 3.1. More precisely, the set DS of dangerous markings and the application FT, that determines the associated forbidden transitions,

- the uncontrolled system specification represented by a CPN ($N=<P,T,C,W^+,W^-,M_0>$) and the associated information as defined in section 2.

Our method will result in a new CPN obtained on the basis of the previous specification and which functioning automatically satisfies the control. The key idea is to handle enough information from the controller to detect reaching a dangerous state, and from which, remove appropriate authorizations in order to disable the firing of forbidden transitions. This method relies on the following points:

- we introduce a new place holding information on dangerous states and associated forbidden transitions. Its marking is defined by Ω.
- we handle the current state (marking) of all processes and resources in a special added place. The information is modelled by a composed token (tuple of colours) in accordance with CPN semantics.
- we add a place that manages authorizations to fire forbidden transitions.
- we add two transitions : one is fired when a dangerous marking is detected, and the other is fired when it is quitted. These two supervisor transitions have special high priority over all the other transitions and are fired immediately when enabled.
- we define the necessary additional CPN components (colour functions, synchronization arcs, markings, etc.) to ensure the desired management by the supervisor.

Let us formally define this method. The system under control is a CPN N* obtained from N so that $N^*=<P^*,T^*,C^*,W^{*-},W^{*+},M^*_0>$ where :

$P^*=P\cup\{CM, DM, AT, AS\}$, with

CM representing the Current Marking,

DM the Dangerous Markings,

AT the Authorizations for forbidden Transitions, and

AS the Alert State of supervisor,

$T^*=T\cup\{A_In, A_Out\}$, with

A_In representing entering the alert state,

A_Out quitting the alert state,

A_In and A_Out have the highest priority.

Now, we define the following additional colour classes:

C_{num}={0,1,2,…,MaxInt} is a class representing a set of finite positive integers. Its elements will model the occurrences of some given tokens. We assume MaxInt large enough to be greater than the bound of the maximum occurrences of any token in a reachable marking. As, we deal with bounded CPNs, this property holds.

C_{FT} is a class representing all forbidden transitions.

In the following, and for simplicity reasons, we may denote N* by N.

Let us determine the colour domain of the additional places as well as their initial marking:

- place CM has a complex colour domain which is a Cartesian product of C_{num} performed on the basis of the number of process classes, the number of places per process and the resource class. The role of CM is to handle information about the current marking.

$$C(CM) = \bigotimes_{i=1}^{k} \bigotimes_{1}^{xi} C_{num} \bigotimes_{j=1}^{card(Cr)} C_{num}$$

CM is always mono-marked and its initial marking M_0(CM) is performed on the basis of initial markings of P_S, P_0 and {P_R} places. The token marking CM is a long tuple made up of counters where each one holds the information about the occurrence of tokens in a given place (according to the lexical order) among process places and the occurrences of tokens in the resource place P_R. M_0(CM) may be algorithmically determined.

- place DM has the colour domain : C(DM)=C(CM)×C_{FT}
  Its initial marking contains tokens that represent dangerous markings with the associated forbidden transitions. The initial marking does not change since this place is only read accessed. The number of tokens in DM is given by:

$$\Sigma_{d\in DS} \text{card}(FT(d))$$

- The colour domain and initial marking of place AT are: C(AT)= C_{FT} , and M_0(AT)= C_{FT} ;
  Initially, all forbidden transitions are authorized.

- The colour domain and initial marking of place AS are: C(AS)= C(DM) ; M_0(AS)= 0 ;

Finally, we have to determine the additional arcs of transitions by defining the associated colour functions:

- Input and output arcs of place CM: as its role is to hold the current marking of N, it is associated with every transition of T, an input arc (reading marking) and an output arc (updating marking).

  $\forall t\in T$, $W^-(CM,t)=\langle X_{1,1},\ldots,X_{k,xk}, Y_1,\ldots,Y_u\rangle = \langle X\rangle$

where $X_{i,j}$ is a variable defined on C_{num} allowing reading the marking of place j in generic process i, and Y_u is a variable defined on C_{num} allowing reading the occurrences of colour u in place P_R ;

and $W^+(CM,t) = \langle X'_{1,1}, \ldots, X'_{k,xk}, Y'_1, \ldots, Y'_u \rangle = \langle X' \rangle$
where $X'_{i,j}$ and Y_u are variables defined on C_{num} determined as follows :

$$X'_{i,j} = X_{i,j} - \chi, \text{ with } \chi = W^+(p_{ij},t) - W^-(p_{ij},t),$$

and

$$Y'_u = Y_u - \xi, \text{ where } \xi \text{ is computed as follows:}$$

as $W^-(r,t) = \Sigma_i\, \alpha_i.r_i$ and $W^+(r,t) = \Sigma_i\, \alpha'_i.r_i$,

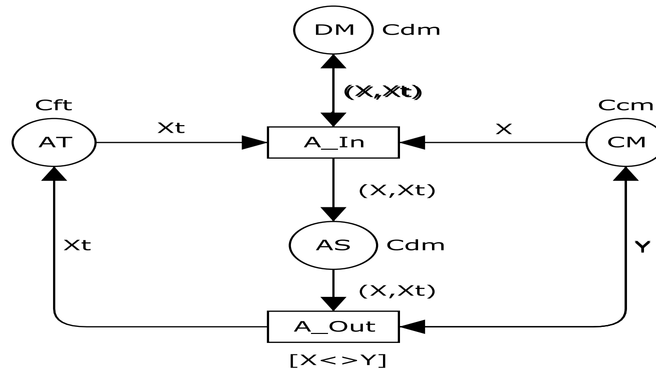
then $\xi = \alpha'_u - \alpha_u$

- To every forbidden transition, are added one input arc and one output arc associated with place AT labelled by the same variable in order to check the presence of firing authorization :
  $\forall t \in C_{FT}$, $W^-(AT,t) = W^+(AT,t) = \langle X_t \rangle$,
  where X_t is defined on C_{FT} and represent the identity of the transition.

- The place DM acts like a database of dangerous markings which is accessed in read-only mode. Then, a double arc (loop) is added to transition A_In with the following functions:
  $W^-(DM,A_In) = W^+(DM,A_In) = \langle X, X_t \rangle$
  where $X \in C(CM)$ (i.e. X is a tuple of variables)

- Transitions A_In and A_out require the additional following arcs:
  $W^-(CM,A_In) = W^+(CM,A_In) = \langle X \rangle$;
  $W^-(AT,A_In) = \langle X_t \rangle$; $W^+(AS,A_In) = \langle X, X_t \rangle$;
  $W^-(AS,A_Out) = \langle X, X_t \rangle$; $W^-(CM,A_Out) = \langle Y \rangle$;
  $W^+(AT,A_Out) = \langle X_t \rangle$;
  Transition A_Out is associated with the predicate: $[X \neq Y]$

The following figure represents the subnet modelling the controller behaviour. It is worth to notice that it is connected to the initial CPN specification through the places AT and CM as it was previously defined.



Figure[5]: Controller subnet

3.3 Example

Let us consider the example of section 2.1. In our example, we have:
$C(CM)= (C_{num})^8$;

a token marking CM is a tuple $<x_1, \ldots, x_8>$ where :

- x_1, x_2, x_3 are values of token counters representing the markings in p_{01}, p_{11}, p_{12} respectively.
- x_4, x_5, x_6 are values of token counters representing the markings in p_{02}, p_{22}, p_{23} respectively.
- x_7, x_8 are values of colour counters representing the occurrences of resources r_1 and r_2 respectively.
  $M_0(CM)=<2,0,0,1,0,0,1,2>$ (current marking place)

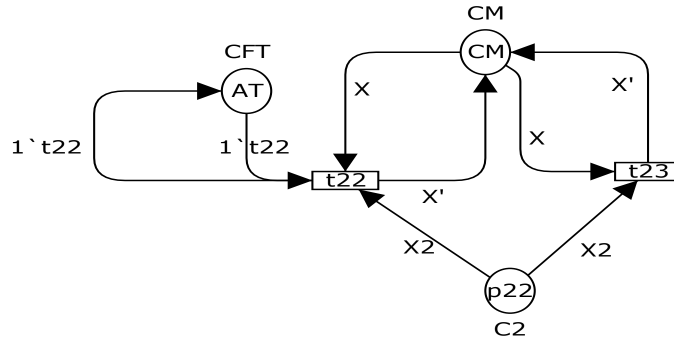
$C(DM)= C(CM)\times C_{FT}$; (dangerous markings place)
$M_0(DM)= <<1, 1, 0, 0, 1, 0, 0,1>, t_{22}> + <<2,0,0,0,0,1,1,1>,t_{11}>$

$M_0(AT)= \{t_{11},t_{22}\}$; (authorizations place)

As example of colour functions which join CM to the transitions of the initial model we have:

$W^-(CM,t_{11}) = <X> = <X_{1,1},X_{1,2},X_{1,3},X_{2,1},X_{2,2},X_{2,3},Y_1,Y_2>$,
$W^+(CM,t_{11}) = <X'> = <X_{1,1}-1,X_{1,2}+1,X_{1,3},X_{2,1},X_{2,2},X_{2,3},Y_1-1,Y_2>$
and,
$W^-(CM,t_{24}) = <X> = <X_{1,1},X_{1,2},X_{1,3},X_{2,1},X_{2,2},X_{2,3},Y_1,Y_2>$,
$W^+(CM,t_{24}) = <X'> = <X_{1,1},X_{1,2},X_{1,3},X_{2,1}+1,X_{2,2},X_{2,3}-1,Y_1,Y_2+1>$

Through the next example, we show how the controller subnet is connected to some transitions of the initial system specification.



Figure[6]: Controller connection

3.4 Equivalence between the Admissible and the Controlled behaviours

In this section, we present and proof the equivalence between the computed admissible behaviour and the controlled behaviour (that obtained from the generated model). This result is presented in the next theorem.

Theorem

The behaviour of the controlled system is equivalent to the computed admissible behaviour represented by R_c.

Proof

In order to verify the equivalence between the two behaviours, first we prove that all information about the current state are stored in CM. Second, we verify that, at any state, we have in AT only the authorisations of the forbidden transition which could be fired at the current state. Finally, we prove that only admissible markings are reachable.

STEP 1

A current state is defined when we have the information about each process place marking and the information about the number of available copies for each resource type. For this reason, we associate a counter with each process place and with each resource type to save current state. This hypothesis justifies the colour domain of CM. But let us prove that the structure of the CM's token permits to represent any accessible state.
The initial marking of the RAS model can be modelled using the specified structure. Indeed, all process instances are in the idle place of the associated generic process and for each resource type the initial marking of resources place indicates the finite number of available copies.
Now, let us prove that all accessible states from the initial marking could be modelled using the structure of the CM's token.
Suppose M' such that $M[t(c)\rangle M'$. M is modelled using specified structure. When t(c) is fired, the token handled in CM will be modified. Indeed, we add (respectively take off) 1 to each counter associated with successors (respectively predecessors) process places. For each counter associated with a used resource type, we add the difference between the number of allocated and restored copies. As a result, we have in CM a new token modelling the new state of the system.

⇒ The token handled in CM place specifies, at any time, the current state of the system.

STEP 2

Now let us prove, using absurd, that any forbidden state is reachable.
Suppose that the system has reached a forbidden state. So, there exists a sequence of events which permits to access this state from an admissible one. We will consider the last controllable transition "t". According to the algorithm "Compute Admissibility", this transition will be determined as forbidden. For this reason it must find authorisation in AT when it will be fired. Or, the place AT is joined to the transition A_In, which will be validated when the system reaches the dangerous state associated with "t". As A_In has a high priority, it will be fired immediately and removes all forbidden transitions authorisations. All removed authorisations will be replaced in AT only when A_Out is fired. This transition will be fired only when the system exits the dangerous state.
According to the last results, "t" will never find its authorisation in AT when the current state of the system corresponds to its associated dangerous state. For this reason a forbidden state cannot be reached.

⇒ All forbidden states will never be reached

STEP 3

Contrarily to forbidden states, we prove that all admissible states can be reached. Since R_c represents the connected admissible graph. Indeed, the occurrence graph is connected, and according to algorithm "Compute Admissibility", if a node is declared as a forbidden state, all its related nodes will be declared forbidden if these nodes are not reached at least from an admissible one. Finally, all forbidden nodes will be removed from R_c, and the resulting graph is a connected one Indeed, any admissible node could be reached from another and all nodes could be reached from the initial node.

Under control there are two types of nodes:

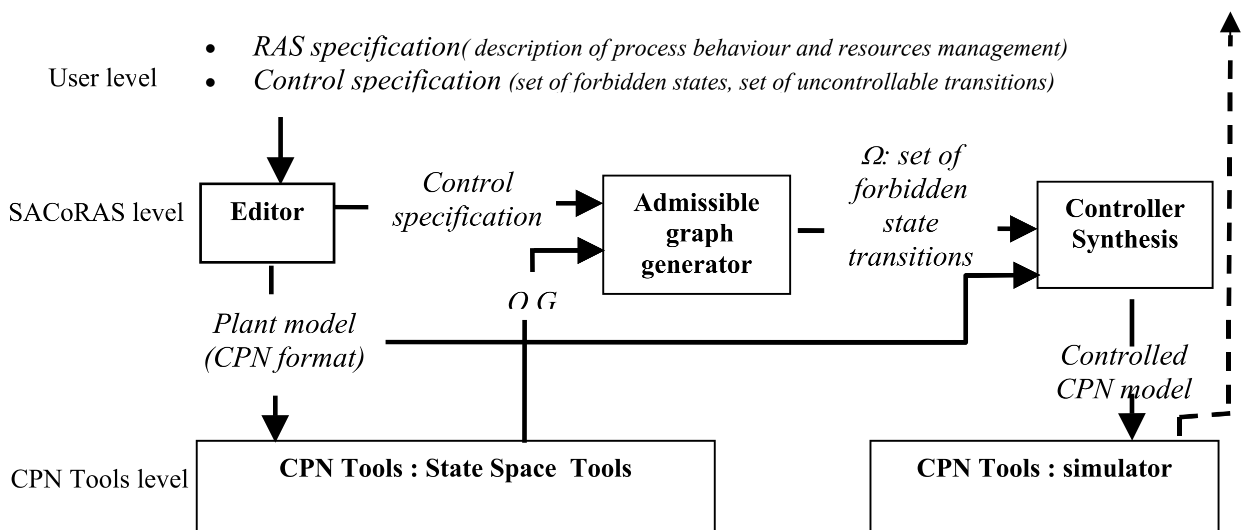
- Non-dangerous states: here all transitions are authorised and consequently all successor nodes could be reached.
- Dangerous states: as we have previously proved, only authorisations permitting to reach forbidden states, will be removed from AT; consequently all admissible successors states could be reached.

$\Rightarrow$ Under control all admissible states are reachable.

According to the previous results, we can affirm that the behaviour of the controlled system is equivalent to the admissible behaviour.

4 SACoRAS application

In this section we introduce SACoRAS application that implements the presented approach. The next figure describes its architecture and presents the integration of our application to CPN Tools. All internal and external data exchange are also presented in the next figure.



Figure[7]: SACoRAS architecture

SACoRAS is made up of four components:

- A graphical/textual interface allowing the edition of RAS model and the specification of control constraints.
- A module allowing the determination of the occurrence graph of the specified model using the state space tools of CPN Tools.
- A module generating the set of forbidden state transitions.
- A module allowing the generation of the Active Controller associated with the initial model if the controller exists.

The different components of our application will be described in this section.

4.1 Design interface for RAS

As previously considered, RAS have numerous characteristics that distinguish themselves from other DES. On the basis of the particular structuring of RAS, we offer the possibility of easily editing plant models. Hence, the user may specify the number of generic processes and the associated instances, resource types and the available copies, and, each process behaviour regarding the use (allocation and restitution) of resources. We indicate that we are not interested, in our application, on flows computation. The proposed interface could be used in two different ways: textually and graphically.

4.1.1 Textual interface

Using this interface, the user can specify his RAS model textually. To do it, he must indicate the number of generic process running in his system. For each generic process, he specifies the number of states. To indicate the different types of resources and the corresponding number of copies handled in each state, the designer uses two different primitives. The first one permits to indicate the resources allocation to execute a specified state. Using the second primitive, the user indicates the number of copies of resources restored when exiting the indicated state. To finish the specification of his RAS model, the user indicates the number and the identity of instances for each generic process and also the initial marking of resources place. If the initial model has uncontrollable transitions, the user indicates it by enumerating their identities.

When the user validates his choice, a textual file describing the CPN representing the specified model will be generated. The interface will manage the colour functions of arcs associate with different state machines representing generic processes. It also manages the specification of different colour classes representing the colour domains of different places. These declarations will be generated from the initial marking of the different model places.

4.1.2 Graphical interface

To specify a RAS model using textual interface, the user must know the syntax of the textual specification. To make the edition easier, we propose a graphical interface allowing the specification of RAS. Using this interface, the user models the initial RAS by adding process places, resources place, transitions and arcs. To

help the designer to edit correct models, the interface ensures some controls. For example, the user can not change the colour functions of an arc belonging to a generic process. These colour functions will be automatically created. On the other hand, the graphical interface offers to the user the possibility to modify the colour functions of arcs joining transitions to resources place which permit to indicate the number and the types of resources allocated or restored when a transition is fired.

To ensure that the designed model has the characteristics of RAS, the interface forbids the connection of a transition to more than one predecessor process place, and to more than one successor process place. Other facilities are available, such as, the user does not have to specify the declarations of colour classes or variables. The colour domains are automatically computed based on the initial marking of places. The declaration of variables is also automatically generated.

Thanks to this restrictions and facilities, to model a RAS is easy and the user can be sure that his model respects characteristics of the RAS class.

The information related to controllability of transitions are also described thanks to SACoRAS graphical interface. Indeed, to specify whether a transition is uncontrollable, the user clicks on the concerned transition and sets the controllability flag to false.

The user can save his model, and a textual file is created describing different components of CPN representing the designed RAS model.

4.1.3 Textual file formats

After specifying his RAS model, using the proposed interface, the user would like to save his system. To ensure this operation, we had to choose between saving binary file or textual file. To facilitate the exchange with another CPN environment, we chose textual file format. In order to allow the integration of SACoRAS application to CPN Tools, a compatible format is used. Hence, it is possible to open a file describing the CPN representing the initial RAS model using CPN Tools. To ensure this goal, we chose to use the textual exchange file format offered by CPN Tools and representing one of the many advantages of this CPN environment.

We have also defined another textual file format allowing to save the description of the occurrence graph. Indeed, after generating the occurrence graph associated with the specified model, we must save its description. To generate the description of this graph, we have defined functions respecting the syntax of standard ML language supported by CPN Tools. These functions permit to save the textual description of different elements madding up the occurrence graph: nodes and arcs. For each node we save the marking of all places of the model at the specified state. For each arc, we save the name of the fired transition and also the instance of colour for the different variables when the transition is fired. We save also for each arc the source and destination node. The definition of these functions was possible using the primitives offered by CPN Tools and permitting to handle different components of the occurrence graph. These primitives represent another advantage of CPN Tools exploited by SACoRAS.

As previously considered, the file generated by the proposed interface can be opened using CPN Tools. This permits to benefit from the State Space Tools

offered by the CPN Tools environment. Also, we generate a textual description of the generated occurrence graph associated with the initial model and which may be used later for the generation of the "Active Controller".

4.2 Occurrence Graph module

To generate the occurrence graph associated with the specified model, SACoRAS uses the State Space Tools of CPN Tools. To ensure this operation, OG module first generates, from the internal data structures handled by SACoRAS, an XML file describing a RAS model and respecting the syntax of the textual exchange file format offered by CPN Tools. It also generates functions that allow to save the description of the occurrence graph. Next, OG module opens a CPN Tools window containing the initial model. When the user exits CPN Tools, OG module will verify whether the textual description of the occurrence graph was saved. If the OG module finds the specified file, it will create and fill up internal structures describing the occurrence graph of the initial model. Conversely, if the file was not found, the OG module will alert user that textual description of the occurrence graph was not saved and a new CPN Tools window containing the initial model will be opened to allow the generation and the save of the occurrence graph. This operation is repeated until the textual description will be defined.

When the execution of OG module is terminated, SACoRAS has an internal description of the occurrence graph associated with the modelled system which can be used later for the generation of the Active Controller.

4.3 Admissible graph generator

This module allows the computation of the set of forbidden state transitions. Admissible graph generator works according two different steps.

4.3.1 Compute accessibility: During this first step, the admissible graph generator module verifies if the occurrence graph of the specified model was generated. If not, it asks the OG module for the generation of the occurrence graph. When the internal description of the occurrence graph is found, the admissible graph generator module can execute its second step.

4.3.2 Compute admissibility: This second step, which is the implementation of the algorithm "Compute Admissibility", allows the computation of the forbidden state transitions based on the specification of the forbidden states that could be described using two modes. Indeed, forbidden states could correspond to deadlock states or to states that are explicitly specified by the user. For example, user could ask to forbid, for the model presented in section 2.1, all markings where $card(M_0(P_{01}))=1$ and $card(M_0(P_{23}))=1$. For the textual specification, we suppose that the user enters a correct format of the control constraints. When the user validates the control specification, the compute admissibility module explores the internal structure describing the occurrence graph searching nodes that verify the control constraints. All

nodes which comply with the control constraints will be considered as forbidden and will be included to FS_0.

When the compute admissibility module terminates the determination of forbidden states two situations can occur:

* FS_0 is empty or M_0 is included to FS_0, in this situation the execution of the admissible graph generator module will be exited. An alert message will indicate to the user that the initial model does not contain any forbidden state if FS_0 is empty or that is not possible to control the initial model when M_0 is a forbidden node.

* FS_0 contains forbidden states, in this situation the compute admissibility module will terminate its execution and will determine a set of forbidden state transitions associated with the determined forbidden states. To look for dangerous states, the compute admissibility module follows the different steps described in the "Compute Admissibility" algorithm. When all dangerous states are computed, we can also find two different situations. The initial state M_0 was determined, by propagation, as an inadmissible state, in this case the generation of the admissible graph generator will be stopped and an alert message will indicate to the user that it is not possible to control the specified model. Otherwise, the active controller synthesis module is caused.

4.4 Active Controller synthesis

The execution of this module allows the generation and the connection of the Active Controller associated with the initial model.

When the active controller module is caused, it has in its internal data structure all necessary information for the generation of the Active Controller related to the initial RAS model. To generate this controller, different steps are followed:

* Two additional colour classes are included to the global declaration structure of the initial model. The first class, called Cnum, is based on the integer class used in CPN Tools. The second class, which represents all forbidden transitions and called Cft, will be declared by enumeration of all the forbidden transition identities.

* In this step, the active controller generation module creates the four additional places and the two transitions making up the controller subnet and then determines the colour domain and the initial marking of all additional places. To ensure this operation, a lexical ordering is defined, based on the identities of processes, places and resource types. Now colour domain and initial marking can be determined.

* The colour domain of the CM place, declared as a colour class called Ccm, will be determined as a product based on Cnum and the number of nodes found in the structure describing the lexical ordering. The initial marking of CM will be computed on the basis of initial marking of all places of the uncontrolled model. Indeed, for each process place, we compute the cardinal of its initial marking and for the resources place; its initial marking will be decomposed following the different resources types. Then, the token representing the initial marking of CM will be generated on the basis of the lexical ordering and the last computed information.

  The class Cdm representing the colour domain of DM will be declared as a product based on Ccm and Cft. The initial tokens representing the initial marking of DM will be generated on the basis of information representing dangerous nodes, and particularly the marking of all places at this state, and information about the associated forbidden transitions. In all tokens, the part representing the dangerous states is generated following the lexical ordering. This hypothesis permits a correct comparison between dangerous and current marking. Indeed, the counter associated with each process place and with each resource type follows the same order. When two tokens are compared, we compare respectively their associated counters (the same resource type or to the same process place).

  For AT, its colour domain is Cft and its initial marking will be the addition of all instances associated with each element included in Cft.

  The colour domain of AS is Cdm, and initially it is empty.

* After having created all additional places and transitions making up the active controller, the active controller generation module will connect the net representing the active controller to the net of the uncontrolled system. To realise this operation, all transitions will be linked to CM markings with an input and an output arc and all forbidden transitions will be connected to AT place with an input and an output arc. The colour functions of these last arcs are constants representing the identity of the forbidden transitions. For all input arcs connecting all transitions to CM their colour functions will be a tuple made up of variables defined on Cnum. To determine the colour functions of output arcs connecting each forbidden transition to CM, the active controller generation module will first determine its successors and its predecessors process places and the number of copies of all resource types handled when the transition is fired. Then, a colour function of these output arcs will be computed based on this information and the input tuple. Indeed, for each process place, the variable Xi associated with the counter of this place will be replaced by the expression $Xi-\mathcal{X}$ where $\mathcal{X}$ represents the difference between the number of the replaced and the removed

tokens from the specified place. Then, for each resource type the corresponding variable Yi will be replaced by the expression Yi- Ξ where Ξ is the difference between the number of the restored and the allocated copies. At the end of this step, the active controller generation module has in its internal structure all information representing the controlled system.

* Seeing that the priority is not handled in CPN Tools, we have implemented the priority of our two controller transitions. To do it we have add places which were joined to the transitions of the controlled model.

*At this final step, the active controller module will generate a file respecting the syntax of CPN Tools files and modelling the initial system under control. A message will indicate to the user the path where the generated file was saved.

In this section we have presented the different modules of the SACoRAS application and we have described the working of each module. When all modules of the SACoRAS application are executed successfully, they permit, if it is possible, the generation of a CPN model representing the controlled system associated with the initial RAS model.

To test our application, we have edited the RAS described in section 2 and we have asked to control deadlock nodes. The following figure provides a piece of report, generated using CPN Tools, and associate with the resulting controlled system. It indicates that the system does not have deadlock markings.

```
Home Properties
------------------------------------------------------------------------
 Home Markings:  All

Liveness Properties
------------------------------------------------------------------------
 Dead Markings:  None
 Dead Transitions Instances: None

 Live Transitions Instances: All

Fairness Properties
------------------------------------------------------------------------
 Top'AIn 1              Fair
 Top'AOut 1             Fair
 Top't11 1              No Fairness
```

Figure[8]: Piece of CPN Tools report.

5 Conclusion

We introduced a supervisory control approach based on coloured Petri nets. The first advantage of the presented method is that it does not define a derived model (neither extension nor abbreviation) from the well-known general CPN one. We

described a controller synthesis method for Resource Allocation Systems. This method is implemented, in the SACoRAS application, within the CPN Tools environment. One of the advantages of the SACoRAS application is its interface that allows the specification of only correct RAS models. The second advantage of this application is the automatic generation of a controller and its integration to CPN Tools.

References

[1] K. Barkaoui, A. Chaoui and B. Zouari, Supervisory Control of Discrete Event Systems based on Structure Theory of Petri Nets, in Proc. of the IEEE International Conference on Systems, Man and Cybernetics, 1997.

[2] Ghaffari, A, N. Rezg and X.L. Xie. Design of a live and maximally permissive Petri net controller using theory of regions. IEEE Trans. on robotics and Automation, 2002.

[3] Giua, A. and F. DiCesare. Blocking and controllability of Petri nets in supervisory control. In: IEEE Trans on Automatic Control, 1991.

[4] M.D. Jeng and F. DiCesare. Synthesis using resource control nets for modeling shared-resource systems. IEEE Trans. on Robotics and Automation, 11(3):317–327, June 1995.

[5] Jensen, K. and G. Rozenberg. High-Level Petri Nets. Theory and application. S. Verlag, 1991.

[6] L. Petrucci. Design and validation of a controller. In Proc. 4th World Multiconference on Systemics, Cybernetics and Informatics (SCI'2000), Orlando, FL, USA, volume VIII, pages 684-688, July 2000.

[7] Ratzer, A.V. and Wells, L. and Lassen, H.M. and Laursen, M. and Qvortrup, J.F. and Stissing, M.S. and Westergaard, M. and Christensen, S. and Jensen, K.. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. Proc. of {ICATPN} 2003. Springer-Verlag.

[8] Ramadge, P.J., W.M. Wonham. The control of discrete event systems. In: Proceedings IEEE, vol.77, n°1, pp. 81-98, 1989.

[9] S. A. Reveliotis, M. A. Lawley, and P. M. Ferreira. Polynomial complexity deadlock avoidance policies for sequential resource allocation systems. IEEE Trans. on Automatic Control, 42:1344-1357, 1997.

[10] W.M.Wonham and P.J. Ramadge. On the supremal controllable sublanguage of a given language. SIAM J. Control and Optimization, vol. 25, No. 3, pp. 637-659, 1987.

[11] B. Zouari, K. Ghédira, “Synthesis of Controllers using Coloured Petri nets and Theory of Regions" - IFAC Workshop on Discrete Event Systems (WODES'04), Reims, September 2004

Modelling a product based workflow system in CPN tools

Irene Vanderfeesten, Wil van der Aalst, Hajo A. Reijers

Technische Universiteit Eindhoven, Department of Technology Management,
PO Box 513, 5600 MB Eindhoven, The Netherlands
{i.t.p.vanderfeesten, w.m.p.v.d.aalst, h.a.reijers}@tm.tue.nl

Abstract. A new approach to workflow process (re)design is the use of Product Based Workflow Design (PBWD) theory. This theory takes the workflow product as the central concept (instead of the workflow process) and provides a product data model of the structure of information processing in the workflow process. In this paper we introduce a prototype for applying PBWD constructed using CPN Tools. The goal of this prototype is to provide a tool for 'experimentation' with the product data model. The CPN model supports the step-by-step calculation and processing of data in the workflow process based on several strategies and provides insight in the way PBWD works. Finally, the prototype is evaluated with on the one hand a state space analysis to assess the correctness of the model and on the other hand a simulation of a sample product data model.

1 Introduction

Workflow processes are administrative business processes in which much information is processed. Although the focus on the business process instead of a focus on a single department has become more and more popular during the past decade, the (re)design of models for these business processes is still more art than science.
In redesigning workflow processes generally two approaches can be distinguished [10]:

- *Evolutionary approaches*: the existing process is taken as a starting point, which is gradually refined or improved by using a set of best practices or rules to transform that process.
- *Revolutionary approaches*: a clean-sheet of paper is taken to design the complete process from scratch.

Product Based Workflow Design (PBWD, see [1,2,3,15,16,17]) is a revolutionary approach. It takes a different view on the workflow process in which the processing of data and the workflow end product are the central concepts, rather than the activities and the workflow process itself. In PBWD, the workflow product is represented by a product data model, i.e. a network structure of the construction of the product (cf. the Bill-of-Material (BOM) in Manufacturing, [11]). In manufacturing processes the BOM is used to structure the operations process [6,7,19,20]. It seems a promising step to adopt this view in administrative business processes too [12]. Based on the product representation the workflow process model can be derived [17].
Although the "PBWD view" on a workflow process may not be the most popular way people think about designing a workflow process, it has several advantages [15]:

- The clean sheet approach that is taken allows for maximal space to establish performance improvements (*radicalism*). Approaches that use the existing process will to some extent copy constructions from the current process, taking over errors or undesirable constructs.
- Moreover, PBWD is *objective.* In the first place because the product specification is taken as the basis for a workflow design, each recognized information element and each production rule can be justified and verified with this specification. As a consequence there are no unnecessary tasks in the resulting workflow. Secondly, the ordering of (tasks with) production rules themselves is completely driven by the performance targets of the design effort.
- The *analytical* approach of PBWD renders detailed deliverables suitable to use for systems development purposes.

- Finally, the *focus on the product* can help to create consensus between different stakeholders. It gives a clear and objective representation of the workflow product that can help in discussing the process. Because the product data model contains less information than a process model it is easier and less complex to design, understand, and maintain.

At this point in time, there is no tool support for PBWD yet. Because it is a time-consuming and error prone method to come from the product data model to a workflow process model, it is useful to develop tools that support this process. However, the first step towards such a tool is a better understanding of and experimenting with the product data model. The goal of this research is the development of a prototype to get more insight in the product data model. This prototype provides a way to qualitatively compare the sequential unfolding of the product tree based on several strategies for local selection of the next step. The prototype is built in CPN Tools [4,8]. Modelling and analysis with Colored Petri Nets is quite common in a manufacturing context [5,13,14]. There are several reasons for using Colored Petri Nets and CPN Tools:

- First, the use of Petri nets provides correctness analysis of the model by means of, for instance, a state space analysis.
- It is very easy to make small adaptations to the models and then compare the outcomes.
- Moreover, the integration of process and data is essential to this view on a workflow process. CPN Tools provides the possibility to integrate them in one model. The data is then captured in the data structures of the model, i.e. in the colours or types, and the process in the Petri net itself.
- Finally, CPN Tools contains a simulation environment, in which one can go step-by-step through the model, following or defining oneself a sequence of firing.

In this paper we describe the CPN prototype for PBWD. The remainder is organised as follows. First we introduce the concept of PBWD and the product data model. Next, the CPN model is explained. In section four we go into further detail of the selection strategies and explain their differences. This section is followed by an evaluation of the prototype based on a state space analysis and a simulation example. Finally, the paper concludes with a discussion and directions for future research.

2 Product Data Model

PBWD theory is based on a product data model. This product data model describes the most elementary parts of a workflow process. It contains the data and information that is processed in the workflow process. It can be compared to a Bill-of-Material from manufacturing [11]. For example, a car is assembled from an engine and a subassembly. This subassembly consists of a chassis and four wheels (see Figure 1).
Similarly, in an administrative context the balance of a company is determined by its assets and liabilities. Moreover, the assets are based on the fixed assets (such as goodwill, (in)tangible assets) and current assets, i.e. debtors, cash at bank and in hand, of the company and the liabilities can be calculated based on the long-term liabilities (like loans and mortgages) and the short-term liabilities (for example creditors).
The data pieces in the product data model are called *data elements*. On these data elements *operations* are defined (e.g. a calculation on numbers, decision making based on certain information). Operations can be executed automatically or manually or through a combination of automatic and manual execution. They denote the relationship between several data elements. Considering the example of the balance of a company as introduced above, three of the data elements are the balance, the assets and the liabilities. An operation defined on these data elements is then the calculation of the balance of the company, i.e. the output of the operation, out of the assets and liabilities, i.e. the input elements of the operation. For a better understanding of these product data structures we refer to [15,16,17].

The goal of the prototype described in this paper is to provide insight in the way the PBWD method works. With the help of this prototype it is possible to simulate the calculation of new information elements based on the elements that are already known.
For implementation of the CPN prototype the product data model needs to be formalized. This formalization is discussed below.

2.1 Formalization of product data model

Taking a mathematical view, a product data model is a set of data elements and 'directed' links or relations between them. The data elements of the product data model are denoted by an identifier (e.g. a). An operation is represented by a tuple $(a, [b, c, d])$. The first element of the tuple (a) is the output data element of the operation, while the second element is a list of input data elements ($[b, c, d]$). Note that an operation can have several input elements, but only one output element.
Next, this tuple is extended with some other information: $(\text{"}opID\text{"}, a, [b, c, d], attributes)$. "$opID$" is a unique identifier of an operation and *attributes* is a tuple containing a value for duration and costs of the operation ($attributes = (duration, cost)$). An operation can be executed when the values for all input information elements are available and when the values of these input elements satisfy certain conditions (if declared).

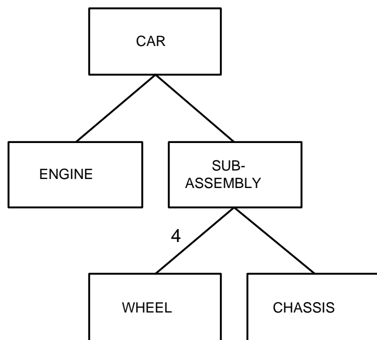


Fig. 1. The bill-of-material of a car

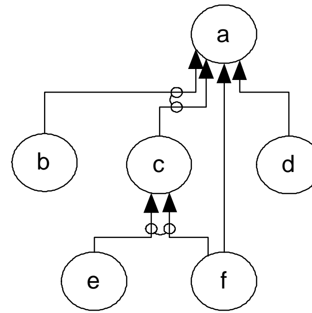


Fig. 2. The product data model of suitability to become a helicopter pilot

This formalization will be clarified with an example (from [15]). In Figure 2, the procedure to determine one's suitability to become a helicopter pilot is denoted in terms of a product data model. All nodes in the figure are data elements that can be used to decide whether an applicant is suitable to become a helicopter pilot or not. The meaning of these data elements (denoted by a, b, c, d, e, and f) is:

a = Suitability to become a helicopter pilot.
b = Psychological fitness.
c = Physical fitness.
d = Latest result of suitability test in the previous two years.
e = Quality of reflexes.
f = Quality of eye-sight.

The suitability to become a helicopter pilot can be determined in either of three ways:

1. Combine the results of the psychological test <u>and</u> physical test. (In Figure 2 this is: e, f combined to c and b, c combined to a)
2. Use the result of a previous suitability test. ($d \rightarrow a$)
3. Make a decision based on the candidate's quality of eye-sight. ($f \rightarrow a$)

These different ways to determine a candidate's suitability may be applicable under different conditions. It can be imagined that if a pilot's eye-sight is bad, then this directly gives a result that the candidate is not suitable. However, in a more common case, the eye-sight quality is one of the many aspects that are incorporated in a physical test, which should be combined with the outcome of the psychological test to determine the suitability result. Also, not for each candidate that applies, a previous test result is available. But if there is one of a recent date, it can be used directly, without knowing the values for the other information elements.

The description of this situation can be formalized in a product data model as follows:

$(\text{“}Op1\text{”}, a, [b, c], (0, 0))$
$(\text{“}Op2\text{”}, a, [d], (0, 0))$
$(\text{“}Op3\text{”}, a, [f], (0, 0))$
$(\text{“}Op4\text{”}, c, [e, f], (0, 0))$

Given this product data model and a set of available data element values, it is possible to determine which operations could be executed and thus which new data element values could be produced. For instance, looking at the example of the helicopter pilot, when the values for data element e (quality of reflexes) and f (quality of eye-sight) are available, the value for element c (physical fitness) can be determined, i.e. the execution of "$Op4$". Initially, the values of elements with no ingoing arcs are available (in our example that is b, d, e, f).
Moreover, some constraints for execution can be added to the operations. Consider for instance operation number three ("$Op3$") from the helicopter pilot example. This operation can only be executed when the value of data element f (quality of eye-sight) is 'bad'. It should not be executed when the value of f is otherwise. The constraints on the execution of an operation are in this prototype added as a function that checks whether an operation satisfies the condition when all of its input elements are available (see appendix A.3).

The formalization of the product data model as presented in this section will be used in the prototype. The CPN-model of this prototype is explained in detail in the next section.

3 CPN prototype

The CPN model we developed consists of two levels: the main level and the sublevel. In this section we first explain the main level and afterwards we elaborate on different variants modelling the sublevel.

3.1 Main level

The main level of the CPN model is depicted in Figure 3. It contains eight places and three transitions. The main stream in the model is indicated by thick lines. Initially, places *available data elements*, *not yet executable operations*, *ready for calculation*, *executable operations*, *total cost* and *total duration* contain a token.
The token in *ready for calculation* does not contain real information. It is only needed to ensure transition *calculate* only fires when it is allowed to, i.e. when the operation that was selected in the previous execution of the main stream has been executed.
However, the tokens in the other places do contain information in order to parameterize the model. The place *available data elements* contains a list of data elements that are available with their corresponding value. The elements of the list change over time. Initially, this place holds the data elements that initially were available and are not obtained through the execution of an operation in this process, i.e. the elements with no ingoing arcs. In many cases these data elements will be the elements that are provided by the customer that starts the case in the workflow process.
The place *not yet executable operations* holds a list of operations that can not be executed yet. Initially, this list contains all operations in the product data model. The format of an operation

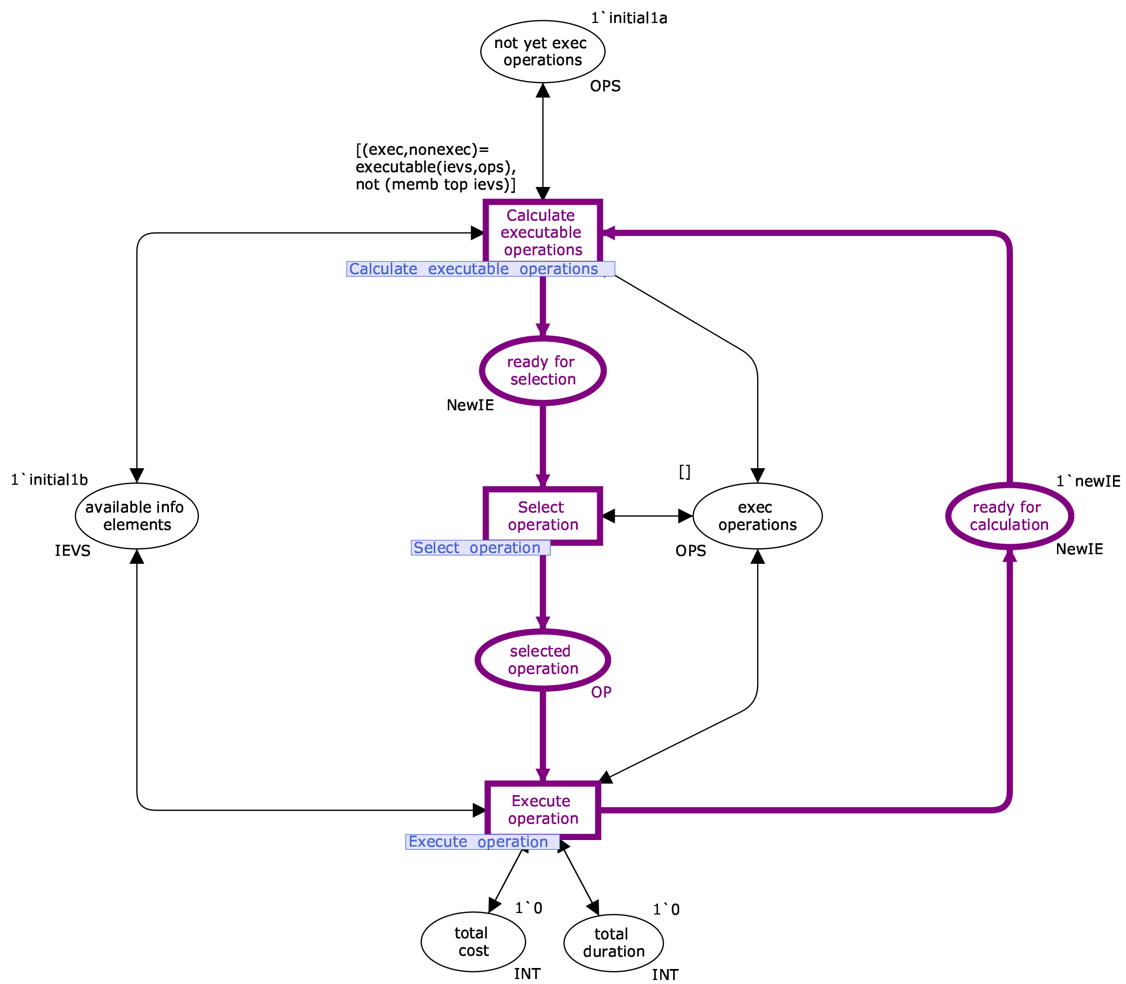


Fig. 3. The main level of the CPN model.

is as defined above in section 2.1. Similarly, *executable operations* contains the list of operations that are executable. Initially, this is an empty list, because the executable operations first have to be calculated.
Finally, *total cost* and *total duration* contain an integer value that denotes the total cost and total duration of the process. For every execution of an operation these values are updated and the cost and duration attributes of that operation are added. Clearly, the initial value of these variables is zero.
The CPN model also shows places that initially do not contain tokens: *ready for selection* and *selected operation*. The first one has a similar meaning of the *ready for calculation* place and makes sure that the *select operation* activity can only fire when there has been a recalculation of the executable operations. Place *selected operation* is used to store operations that are selected. Clearly, this place can not contain a token initially, because the executable operations first have to be calculated before one of them can be selected.

The first step in the execution of this prototype is the firing of transition *calculate executable operations*. When it fires, the operations that are executable, i.e. all of its input elements are in the list of available data elements, are determined from the list of *not yet executable operations*. The executable operations are stored in the place *executable operations*. Next, one of the executable operations is selected and finally this operation is executed, i.e. the output element is added to the list of available data elements and the value for this output element is determined. Then the flow starts again: the executable operations can be calculated based on new information, an operation can be selected, etcetera, until the end product of the process is reached. More specifically this means that the top data element is in the list of available data elements.

In the next sections the subpages for the transitions *calculate executable operations*, *select operation*, and *execute operation* are clarified.

3.2 Calculation of executable operations

As explained before, the *calculate executable operations* transition determines which of the operations from the list of *not yet executable operations* become executable based on the list of available data elements. This calculation turns out to be a bit more involved than one may expect.
As is shown in Figure 4, the *calculate* transition takes the list of available data elements and the list of not yet executable operations as inputs. For each operation in the list it determines to which output place it should be sent.
When the condition of an operation is not satisfied, i.e. if the value of one of the input elements is not satisfying a pre-defined condition, the operation should not become executable and is sent to place *condition not satisfied*. After that, the operation can never become executable again. We assume that conditions remain stable during the process.
If the output element of the operation is already in the list of available data elements then the operation is put in the place *data element already known* and again it stays there. There is no need to determine the value again. When one or more of the required input data elements are not available yet, i.e. a required item is not in the list of available data elements, the operation is put back in *not yet executable operations*.
Finally, the operations which satisfy the predefined condition on the input data element values, of which the output element is not already known, and of which all input elements occur in the list of available data elements, are sent to the *executable operations* place.
Note that *calculate* also recalculates the operations that where put in the place *executable operations* in one of the earlier iterations, because they can become superfluous (e.g. already determined or not satisfying the condition after the calculation of the value of one of the input elements).
Besides that, if the end product is determined, it makes no sense to still perform some operations that are available. Therefore, a guard is added to *calculate executable operations* which only allows this transition to fire when the end product ('top') is not yet in the list of available data elements.

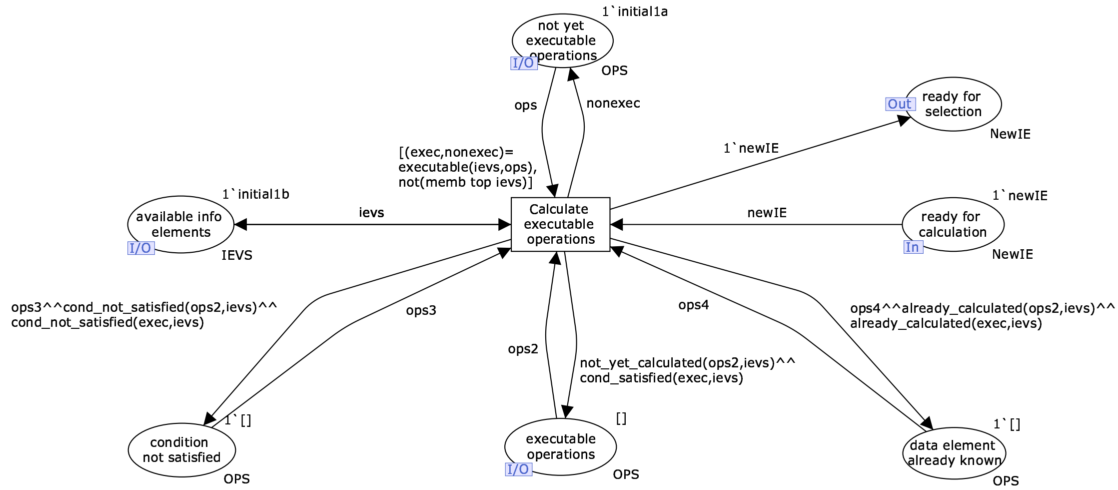


Fig. 4. The calculate executable operations subpage

3.3 Selection of operation

When the executable operations are determined by the *calculate* transition (there is a token available in *ready for selection* and *executable operations* contains a list of operations), one of these operations can be selected for execution. Several strategies can be used to determine which operation from the list of executable operations should be selected. The strategies we considered are: (a) the first operation of the list, (b) the operation with the lowest cost, (c) the operation with the shortest duration, (d) random selection, and (e) selection by a user.
The way in which the next operation to be executed is determined influences the performance of the process of making the workflow end product. We need performance measures to compare the selection strategies in actual situations. To assess the performance of a certain strategy we limit ourselves to total cost and total duration, although one could think of other performance indicators. The selection strategies are further explained in Section 4, and are applied to an example in Section 5.

3.4 Execution of operation

After selecting an operation for execution, this operation will be executed in two steps (see Figure 5). First the execution will be started and the total cost and duration are updated, while the execution of this operation takes some time (the time delay is determined by the attribute *duration*). Next, the execution is ended, either successfully or unsuccessfully. When the operation is performed successfully the output data element of the operation is put in the list of available data elements together with its newly determined value (the simple function *calculation* determines this new value, see Appendix A.3).
When the execution of the operation fails, the operation is put back at the end of the list of *executable operations*, from where it can be selected again some time. The *end execution* transition also puts back a token in *ready for calculation*, so that the loop can start again by calculating the executable operations for the next possible step.

4 Selection strategies

Once the executable operations have been determined, one of them can be selected for execution. The selection in a way influences the costs and duration as explained before. In this section we

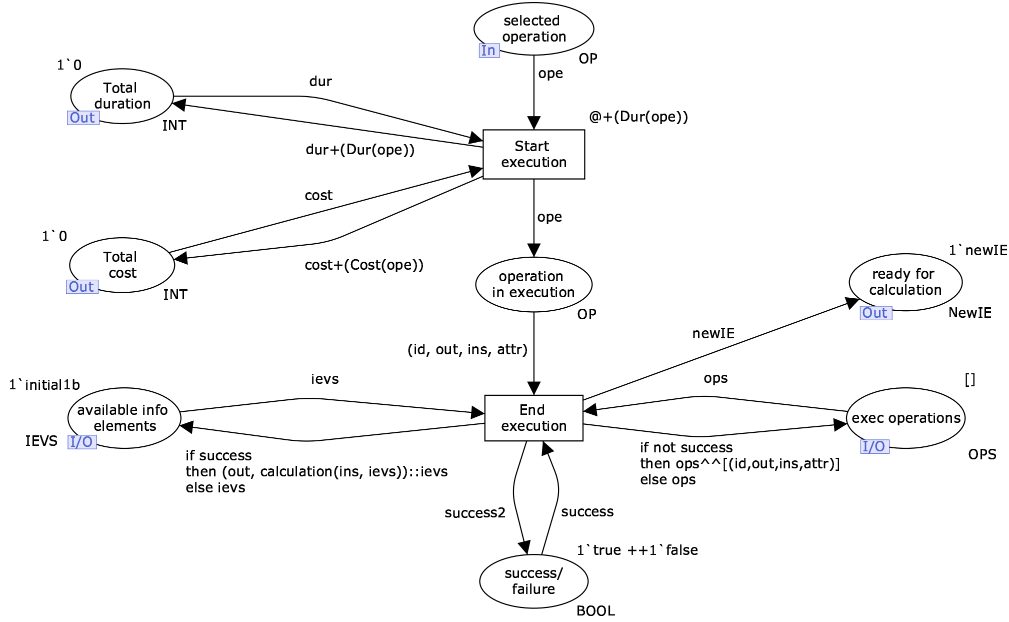


Fig. 5. The execute operation subpage

elaborate on several strategies to select the operation to be executed. We have identified five different strategies that are described separately. We start with a basic strategy: the selection of the first element from the list. Next, for every other strategy it is shown which adaptations to the first model have to be made in order to get a model that supports this strategy.

4.1 Select first element from list

To select the first element from the list with executable operations it is not necessary to write a complex function in CPN Tools[1]. As is shown in the subpage of *select operation* in Figure 6 this element can be specified by explicitly distinguishing the head from the rest of the list: *ope* :: *ops*, where *ope* is declared as an operation (type OP, see declarations in appendix A.1) and *ops* as a list of operations (type OPS). This first element is then taken from the list and passed on to the place *selected operation*, while the tail of the list of operations is put back in the place *executable operations*.
Clearly, the order of elements in the list determines which element is the first element of the list. The list can have an arbitrary order (like a lexicographical order or an order based on one of the attributes). The order of the list can be defined by the user by means of the declaration of the list containing all operations. In this way the selection strategy can be influenced. In our example we use a list order based on the ID of the operation, i.e. "$Op1$" is before "$Op2$", etcetera.

4.2 Select element with lowest cost

The selection of the element from the list with the lowest cost is slightly more complex. The basic structure of the select first element subpage can be maintained but the arc inscriptions have to be changed as shown in Figure 7. The whole list is taken as an input to the transition. Next, the element with the lowest value for a certain attribute is determined by a function called *SelMin*. To this function we add the attribute that we want to have a minimal value, that is *Cost*.

[1] Note that CPN Tools uses the functional programming language Standard ML [18] to manipulate token colors

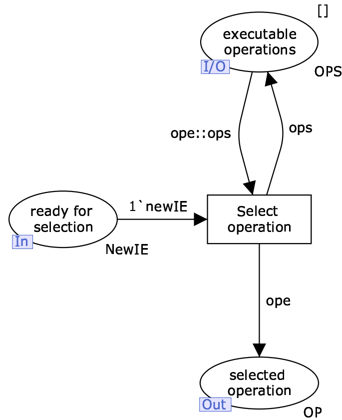


Fig. 6. The select first element from the list subpage

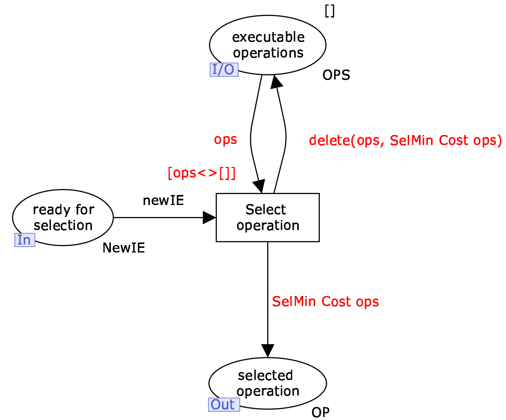


Fig. 7. The select element with lowest cost subpage

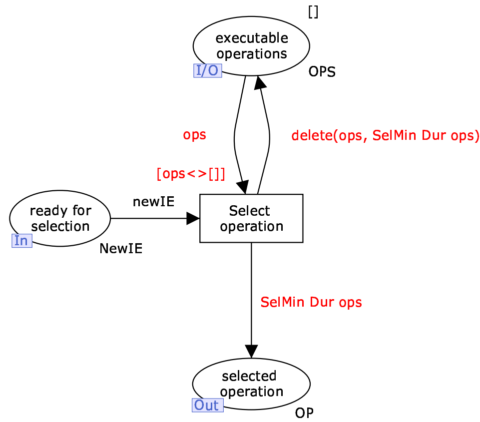


Fig. 8. The select element with minimal duration subpage

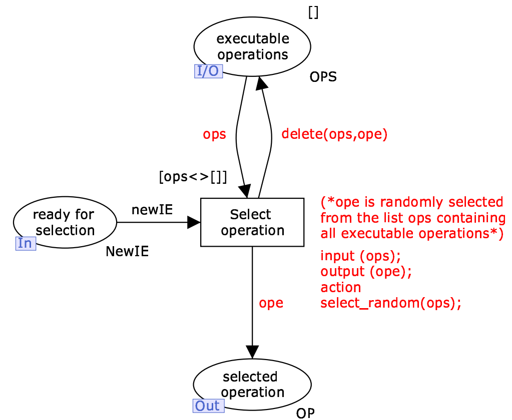


Fig. 9. The random selection subpage

The *SelMin Cost* function is applied to the list of executable operations *ops*. The operation with the lowest cost is sent to *selected operation* and is deleted from the list of executable operations. Of course this only works for a non-empty list in *executable operations*. Therefore a guard is added to the transition.

4.3 Select element with shortest duration

Similar to the selection of the element with the lowest cost we can select the element with the shortest duration (Figure 8). In this case the structure of the model is the same, only the *SelMin* function is applied to the list of executable operations considering the value of attribute *Dur* (Duration) instead of *Cost*.

4.4 Random selection

Another way to select an operation from the list of operations is through random selection. Basically, we can keep the structure of the model as it was in the previous three strategies. To select an

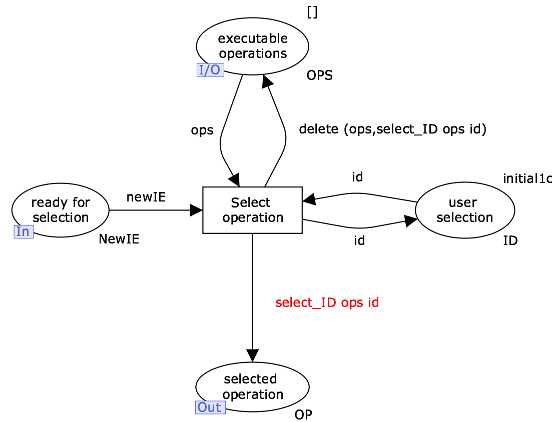


Fig. 10. The selection by user subpage

arbitrary element from the list, we need a function that transforms a real number from a random number generator into an integer as the number of an element from the list.
Because the random function returns a different value every time it is evaluated we can not use the function as we used the *SelMin* function before. Therefore, the random function is put in the action part of the transition. In this way the random number is bound to a variable *ope*. Through this variable the element can be selected and deleted from the list. This function is shown in Figure 9 and is explained in more detail in Appendix A.3.

4.5 User selection

The last strategy for selecting an operation is to let the user decide which operation is selected. In this way, of course, any strategy can be 'imitated' through the actions of a user (e.g. when the user selects the first element of the list every time, the strategy is equal to the select first element of list strategy).
Most of the structure of the model can be reused. An extra input place *user selection* with colour *ID* is added. By changing the current marking of that place a user can specify which element from the list, denoted by its ID, he or she wants to take. This ID is an input to the transition and is used on the outgoing arcs to specify the element from the list as shown in Figure 10.
Moreover, when CPN Tools version 1.4.0 is used it is possible to select a binding for the next step from all possible binding elements (see Figure 11). Initially, all *opID*-s are present in the place user selection, although not all operations may be present in the list of executable operations. Therefore, the user should pay attention to select an operation that really is executable because it is possible to bind an *opID* that does not exist in the list. When an operation is chosen that is not in the list, the 'Empty' exception is raised and the simulation is stopped. However, we feel this is a nicer solution than manually changing a marking. Obviously, this is not a very sophisticated solution from a GUI perspective, but it serves our purpose.

5 Evaluation

In the previous two sections, the prototype for experimenting with a product driven workflow system is explained. In this section the model is assessed. As explained in the introduction, CPN Tools provides several ways to evaluate the correctness of a model. The first one is by means of a state space analysis, which is elaborated on in Section 5.1. In the second part of this section, simulation experiments are conducted to compare several selection strategies with each other. Section 5.2 describes the simulation.

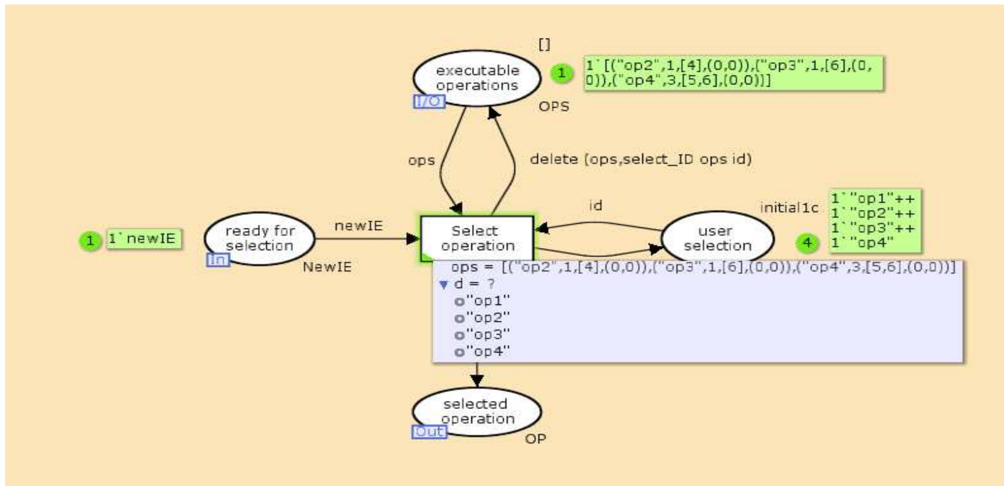


Fig. 11. Selecting a binding element in CPN Tools 1.4.0

5.1 State space analysis

A state space analysis is an analysis of all possible occurrence paths in the model. Based on the occurrence graph, containing all occurrence paths, a lot of information on the properties of the model (and thus on the correctness of the model) can be gained.
To limit the calculation time we have conducted a state space analysis of the model based on a small example product data model. The Helicopter Pilot example only contains six data elements and four operations, as was shown in Section 2.1. We have only reported on the state space for the random selection strategy[2] because it covers all the other strategies. CPN Tools calculated a full state space, containing 157 nodes and 241 arcs, in less than one second. Below, the properties of the net are discussed based on the generated state space report.

Boundedness properties - The first property that is derived from the state space analysis is the upper and lower bound for all places of the model. Upper and lower bounds indicate how many and how few token elements a place can contain, e.g., how many tokens we can have of a particular color on a particular place instance [8].
The integer upper and lower bounds give an integer number of tokens that are at most, respectively at least, present in the place. Multi-set upper and lower bounds in addition give information on the value of the token(s) in the place.
In Table 1 and Table 2, the integer upper and lower bounds and the multi-set upper and lower bounds of the model are given respectively. For the sake of clarity, the operations in the multi-set bounds are represented by an abbreviation: $(\text{“}op1\text{”}, a, [b, c], (0, 0))$ is denoted by “$op1$”.
To give an example of an upper multi-set bound we focus on the place *selected operation*. The upper bound of this place is the multi-set of all operations in our example, i.e. $1`(\text{“}op1\text{”}) + +1`(\text{“}op2\text{”}) + +1`(\text{“}op3\text{”}) + +1`(\text{“}op4\text{”})$. Because of the random selection strategy all operations can be selected from the list, i.e. for every operation there is a path in the occurrence graph in which the operation appears in the place selected operation. However, there can never be more than one operation in the place at the same time.
The tables show that all places have an upper and lower bound. Thus, the net is bounded. This makes sense because the Helicopter Pilot product data model is represented by a finite list of operations, from which the elements are taken one by one.

[2] To avoid problems with calculating the full state space, the ‘random’ function is replaced by a variable that denotes the range of the list of operations. Through this variable a random selection can still be made from the list.

Table 1. Integer Bounds Helicopter Pilot Model

Place	Upper Integer Bound	Lower Integer Bound
Available info elements	1	1
Executable operations	1	1
Not yet executable operations	1	1
Ready for calculation	1	0
Ready for selection	1	0
Selected operation	1	0
Total cost	1	1
Total duration	1	1
Condition not satisfied	1	1
Data element already known	1	1
Operation in execution	1	0
Success	2	2

Table 2. Multi-set Bounds Helicopter Pilot Model

Place	Upper Multi-set Bound	Lower Multi-set Bound
Available info elements	1‘[(1,0),(2,0),(4,0),(5,0),(6,0)]++ 1‘[(1,0),(3,0),(2,0),(4,0),(5,0),(6,0)]++ 1‘[(2,0),(4,0),(5,0),(6,0)]++ 1‘[(3,0),(2,0),(4,0),(5,0),(6,0)]	empty
Executable operations	1‘[]++1‘[(“op1”),(“op2”)]++ 1‘[(“op1”),(“op2”),(“op3”)]++ 1‘[(“op1”),(“op3”)]++ 1‘[(“op1”),(“op3”),(“op2”)]++ 1‘[(“op2”),(“op1”)]++ 1‘[(“op2”),(“op1”),(“op3”)]++ 1‘[(“op2”),(“op3”)]++ 1‘[(“op2”),(“op3”),(“op1”)]++ 1‘[(“op2”),(“op3”),(“op4”)]++ 1‘[(“op2”),(“op4”)]++ 1‘[(“op3”),(“op1”)]++ 1‘[(“op3”),(“op1”),(“op2”)]++ 1‘[(“op3”),(“op2”)]++ 1‘[(“op3”),(“op2”),(“op1”)]++ 1‘[(“op3”),(“op4”)]++ 1‘[(“op2”),(“op4”),(“op3”)]++ 1‘[(“op3”),(“op2”),(“op4”)]++ 1‘[(“op3”),(“op4”),(“op2”)]++ 1‘[(“op4”),(“op2”),(“op3”)]++ 1‘[(“op4”),(“op3”),(“op2”)]++ 1‘[(“op4”),(“op2”)]++ 1‘[(“op4”),(“op3”)]	empty
Not yet executable operations	1‘[]++1‘[(“op1”)]++ 1‘[(“op1”),(“op2”),(“op3”),(“op4”)]	empty
Ready for calculation	1‘newIE	empty
Ready for selection	1‘newIE	empty
Selected operation	1‘(“op1”)++1‘(“op2”)++ 1‘(“op3”)++1‘(“op4”)	empty
Total cost	1‘0	1‘0
Total duration	1‘0	1‘0
Condition not satisfied	1‘[]	1‘[]
Data element already known	1‘[]	1‘[]
Operation in execution	1‘(“op1”)++1‘(“op2”)++ 1‘(“op3”)++1‘(“op4”)	empty
Success	2‘false++2‘true	empty

Home properties - The existence of home markings in the net is the next property of colored Petri nets that we discuss. A home marking is a marking to which it is always possible to return to [8].
In our prototype we consider the fulfilment of the product tree. This is a step-by-step development of the end product, through the 'assembly' of subproducts. In this case, it is not desirable to return to a previous state, i.e. undo an assembly. Thus, the net should have no home marking(s) and the state space analysis shows that is indeed the case.

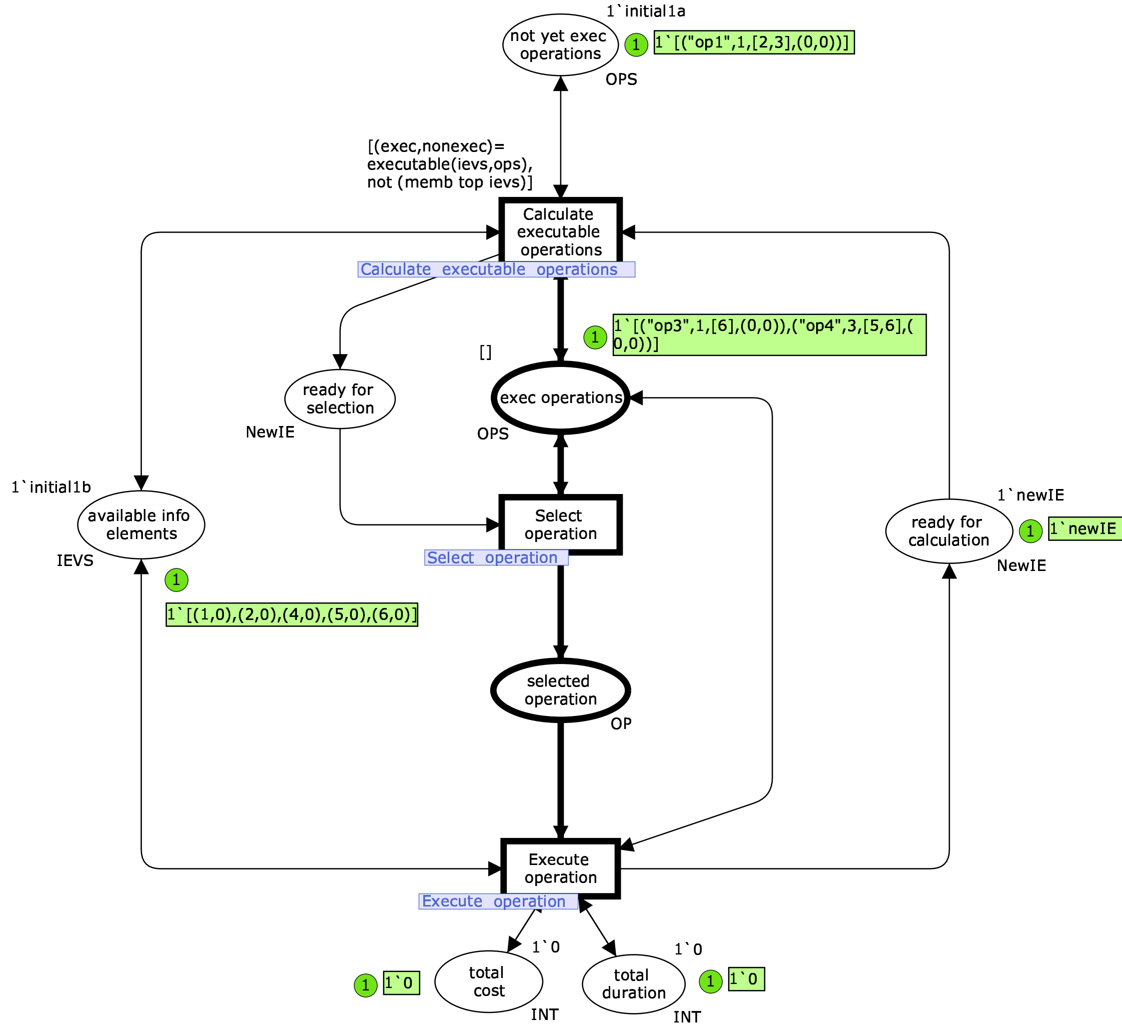


Fig. 12. State '9' from the state space is a dead marking.

Liveness properties - Liveness denotes that a set of binding elements remains active, i.e. every transition can become enabled by firing an arbitrary number of transitions [8]. In contrast, a dead marking is a state of the net in which no transition is enabled.
Our model contains thirty dead markings according to the state space report. These dead markings are caused by the guard for transition *calculate executable operations*. The guard *not (memb top ievs)* enforces that when the end product is produced, nothing else is allowed to happen. In this

case, the tokens in places *not yet executable operations* and *executable operations* can not be removed and form a dead marking.
The state space reports the states of the state space that are dead markings. For instance, state '9' from the state space is a dead marking. By using the 'SStoSim' tool in CPNTools we can visualize this state as is shown in Figure 12. This state indeed is a valid dead marking. The end product (data element 1) is determined by executing the operation with ID *"op2"* starting from the initial marking. Nothing else can happen in the net and there are still operations left in places *not yet executable operations* ("*op*1") and *executable operations* ("*op*3", "*op*4").
Furthermore, the net contains neither dead transition instances, i.e. a transition that never becomes enabled, nor live transition instances (i.e. a transition that will always get enabled again after firing).

Fairness properties - Finally, we look at the fairness of this net. Fairness indicates how often different binding elements occur [8]. The state space analysis report shows that all transitions in the model are impartial. An impartial transition will fire infinitely often given an arbitrary infinite occurrence sequence. This means that any infinite occurrence sequence in our model will contain all transitions infinitely often.
This behavior can be explained by the sequential order of the transitions and the possibility of failure of the execution of an operation. When the execution of an operation has failed, the loop of calculation, selection, and execution has to be started over again. An operation can fail infinitely often (although this is not likely). Therefore infinite occurrence sequences exist in the occurrence graph. Since all transitions have to occur in a predefined sequential order before the operation can be executed again all transitions in the model will occur infinitely often in an arbitrary infinite occurrence sequence.

5.2 Comparison of strategies

The correctness of the model has been shown using state space analysis. Now, we will illustrate the correct working and results of our model based on a real-life example. The example is taken from [15,16] and describes the process of a request for social benefits within a Dutch social security administration office ('GAK'). This agency implements the social security legislation in the Netherlands. On a daily basis, it handles large amounts of requests for unemployment benefits and occupational disability allowances. Social security laws as well as contracts with employer organizations impose restrictions on the way these requests are handled. To decide on unemployment benefits much information is processed. Figure 13 shows the product data model for this case. The exact meaning of all data elements can be found in the original study [15].
The product data structure of the unemployment benefits case consists of 42 data elements and 33 operations in total. Initially, 18 data elements are available. In [15], the costs for each operation are given. We have copied them and multiplied them by ten to avoid rounding problems in the CPN simulation. However, the duration of operations was not indicated in this case study. Therefore, we have added realistic values for the duration of every operation (see the declaration of the list of operations in Appendix A.2).

To be able to compare the different strategies of selecting executable operations in this case study, we have executed several simulation runs for each strategy. A run corresponds to one complete unfolding of the product tree until the end product is produced and thus gives us one sample of the total cost and total duration of the process. The data on cost and duration are stored for each sample. Per strategy we collected 60 samples ($n = 60$) and constructed 90%-confidence intervals, according to the formula for a t-distribution[3] [9].
In Figure 14 and 15 the confidence intervals are represented. These figures show how much the value for total cost and total duration may vary given a certain selection strategy. Because most

[3] We assume that the population has a normal distribution.

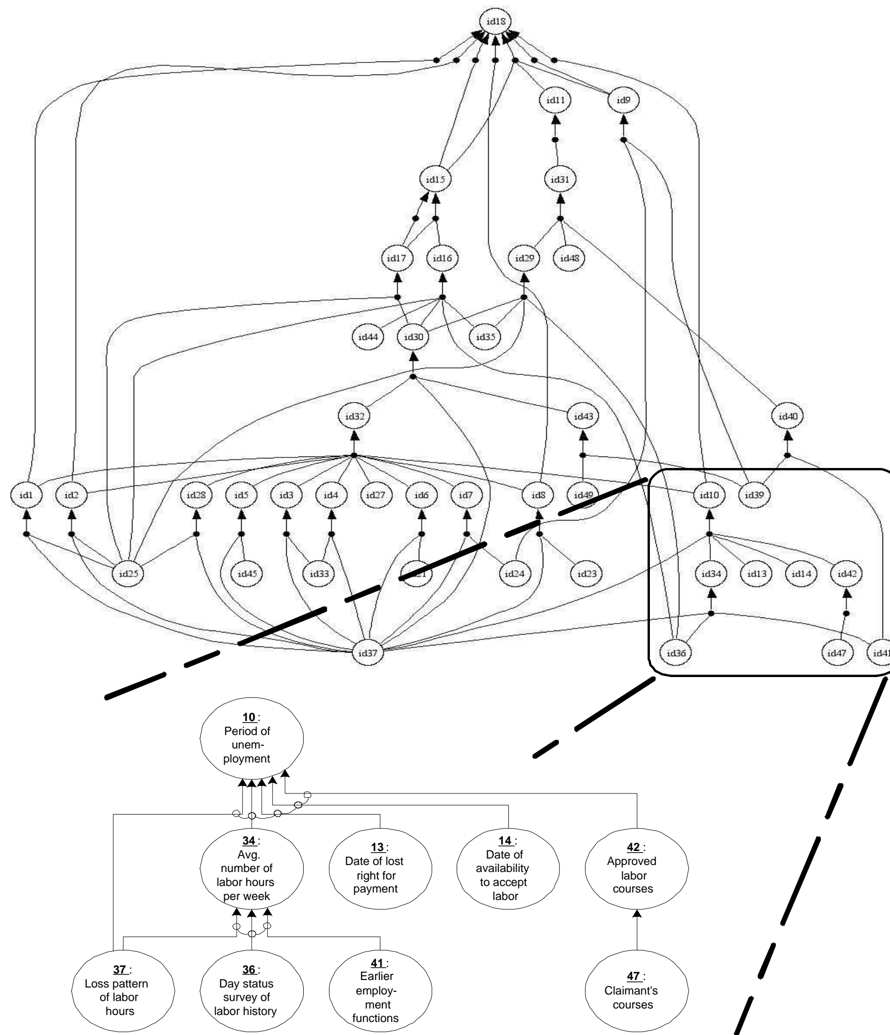


Fig. 13. Product data model for the social benefits case

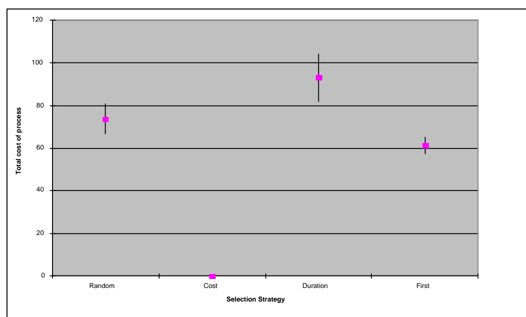


Fig. 14. Confidence intervals for total cost

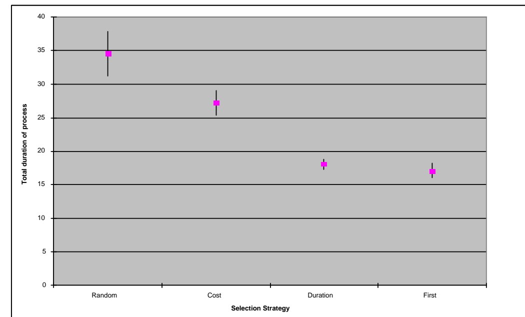


Fig. 15. Confidence intervals for total duration

of the confidence intervals do not overlap it is possible to draw reliable conclusions from the constructed confidence intervals. Some examples of such quantitative conclusions and some qualitative conclusions as well are:

- The total costs are, for example, minimal in case of a 'minimal cost' strategy, followed by a 'first element from list' strategy and a 'random selection' strategy. The 'minimal duration' strategy gives the highest total costs with high variation, i.e. a broad confidence interval, compared to the other strategies.
- For total duration under different strategies the confidence intervals are less explicit: The confidence intervals of the strategies of 'minimal duration' and 'first element from list' selection do overlap. The figures clearly show that a 'minimal cost' strategy leads to a shorter total duration than a random strategy does. Next, the selection of 'minimal duration' and 'first element from list' lead to a shorter total duration than in case of a 'minimal cost' strategy.
- The total cost confidence interval under a 'minimal cost' selection is only one point (see Figure 14). This can be explained by the nature of the example we are considering. The example contains many operations that can be executed automatically and which therefore have cost 0. It is possible to get to the end product by only selecting operations with zero costs. Therefore, by using a 'minimal cost' selection strategy only operations with no costs are selected and we can get to total costs of zero in each sample. This leads to a confidence interval of only one point.
- Confidence intervals for selection of the 'first element from the list' are small. This makes sense because in this case every sample starts with (almost) the same sequence of execution. The only factor that can influence this order is the failure of an execution of an operation which leads to the 'skip' of one operation execution and an extra selection in the end.

The figures show there is no optimal strategy under both optimization criteria. Which strategy is best therefore depends on the importance of each of the criteria. If costs are important, then the strategy of selecting operations with minimal costs is best in our example. If flow times are important, then a minimal duration or first element from list strategy would be best. However, this simulation example gives a clear idea about the information that can be retrieved using our prototype.

6 Conclusion

In this paper we have presented a prototype, built in CPN Tools, for modelling a product based workflow system. This prototype shows how a workflow process can be derived from a product data model (cf. Bill-of-Material). The core of the prototype is formed by the selection strategy. We have shown several strategies to select the next step that has to be performed in the process, from all possible next steps. Note that these strategies only focus on local optimization of the process. They only consider the operations with, for instance, the lowest cost or shortest duration that are executable at that point in time. In the end the selection of the local optimal operation can lead to a suboptimal strategy in terms of overall costs or duration. The prototype provides a way to qualitatively, and to some extend quantitatively, evaluate these different strategies as is shown by a real-life example.
The prototype we developed is a generic tool to play with product data models of different cases. To do so only the initial marking has to be changed by providing different declarations for *top*, *initial1a*, *initial1b*, *initial1c* and the function *check*.

The main goal of this research was to provide more insight in PBWD and the product data model. During the process of the prototype development we have made some design choices that gave us a better understanding of the concepts and issues in this area. The first choice we made is to represent the operations of the product data model by a list of operations instead of a number of separate tokens. Although the latter would perhaps be the most intuitive way, this raised some problems implementing the selection strategies. In a CPN model it is not possible to check all

tokens present in a place and then select the one satisfying a certain condition. The only solution to this problem was to put the operations in a list. This issue made us think about operations, their representation and their relations.
Another choice we made is the strict sequential execution of operations. This point can also be seen as a limitation to the prototype. It would be more realistic if another operation could already be calculated, selected and started while the previous operation is still in execution, especially when execution delays are long. To implement this, extra constructs are needed to make the model work. For instance, it should be possible to break off an already started execution to avoid duplicate determinations of one data element. Moreover, one has to make sure *calculate executable operations* is fired every time a new data element is added to the list of *available data elements*. For the sake of understandability and readability we have not incorporated this possibility in the prototype, because it adds much more complexity to the model. However, this issue made us think of how to deal with parallel execution and other concepts related to the execution of a product data model.

Furthermore, the user interface of the prototype could be improved by the automatic reading of a file containing declarations and other information, and a nicer way for user selection. Nevertheless, by this simple prototype we think we have reached our goal for a better understanding of the product data model and PBWD theory.

Acknowledgements

This research is supported by the Technology Foundation STW, applied science division of NWO and the technology programme of the Dutch Ministry of Economic Affairs.

References

1. W.M.P. van der Aalst. Designing workflows based on product structures. In: K. Li, S. Olariu, Y. Pan, I. Stojmenovic (eds.), Proceedings of the ninth IASTED International Conference on Parallel and Distributed Computing Systems, IASTED/Acta press, Anaheim, pp. 337-342, 1997.
2. W.M.P. van der Aalst. On the automatic generation of workflow processes based on product structures. Computers in Industry, 39, pp. 97-111, 1999.
3. W.M.P. van der Aalst, H.A. Reijers, S.Limam. Product-based workflow design. In: W. Shen, Z. Lin, J.P. Barthes, M. Kamel (eds.). Proceedings of the sixth international conference on CSCW in design. Ottawa: Research Press, pp. 397-402, 2001.
4. CPN Tools home page and manual at http://wiki.daimi.au.dk/cpntools/cpntools.wiki. University of Aarhus, Denmark, 2005.
5. A.A. Desrochers, R.Y. Al-Jaar. Applications of Petri Nets in Manufacturing Systems: Modeling, Control, and Performance Analysis. IEEE Press, 1995.
6. F. Erens, A. MacKay, R. Sulonen. Product modelling using multiple levels of abstraction - instances and types. Computers in Industry, 24 (1), pp. 17-28, 1994.
7. H.M.H. Hegge. Intelligent product family descriptions for business applications. PhD Thesis, Eindhoven University of Technology, Eindhoven, 1995.
8. K. Jensen. Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use, Volume 1, 2 and 3. Springer-Verlag, 1992.
9. D.C. Montgomery, G.C. Runger. Applied Statistics and Probability for Engineers. John Wiley & sons, 2nd edition, 1999.
10. M. Netjes, I.T.P. Vanderfeesten, H.A. Reijers. "Intelligent" software tools for workflow process redesign: a research agenda. Proceedings of the first international workshop on Business Process Design (BPD'05), Nancy, September 2005 (*accepted*).
11. A. Orlicky. Structuring the Bill of Materials for MRP. Production and Inventory Management, december, 1972, pp. 19-42.
12. E.A.H. Platier. A Logistical View on Business Processes: Concepts for Business Process Redesign and Workflow Management. PhD thesis, Eindhoven University of Technology, Eindhoven, 1996. (in Dutch)
13. J.M. Proth, X. Xie. Petri Nets, A Tool for Design and Management of Manufacturing Systems. Wiley, 1996.

14. L. Recalde, M. Silva, J. Ezpeleta, E. Teruel. Petri Nets in Manufacturing Systems: An Examples-Driven Tour. In: J. Desel, W. Reisig, and G. Rozenberg (eds.), ACPN 2003, LNCS 3098, pp. 742-788. Springer-Verlag, Berlin, 2004.
15. H.A. Reijers. Design and Control of Workflow Processes: Business Process Management for the Service Industry. Lecture Notes in Computer Science 2617. Springer-Verlag, Berlin, 2003.
16. H.A. Reijers, S. Limam, W.M.P. van der Aalst. Product-Based Workflow Design. Journal of Management Information Systems, 20(1): pp. 229-262, 2003.
17. H.A. Reijers, I.T.P. Vanderfeesten. Cohesion and Coupling Metrics for Workflow Process Design. Proceedings of the 2nd International Conference on Business Process Management, pp. 290-305, Potsdam, 2004.
18. J.D. Ullman. Elements of ML Programming. Prentice-Hall, 1993.
19. E.A. van Veen. Modelling Product Structures by Generic Bills-of-Material. PhD Thesis, Eindhoven University of Technology, 1990.
20. E.A. van Veen, J.C. Wortmann. Generative bill of material processing systems. Production Planning and Control 3 (3), pp. 314-326.
21. M. Zhou, K. Venkatesh. Modeling, Simulation, and Control of Flexible Manufacturing Systems: A Petri Net Approach. World Scientific, Londen, 1999.

A Declarations

In this appendix the color, variable and function declarations of the CPN model are listed in ML language [18]. It contains values that belong to the 'GAK'-case, which we used for simulation.

A.1 color declarations

```
color IE = INT;
color ID = STRING;
color NewIE = with newIE;
color VAL = INT;
color IEV = product IE*VAL;
color IES = list IE;
color IEVS = list IEV;
color DURATION = INT;
color COST = INT;
color ATTR = product DURATION*COST;
color OP = product ID*IE*IES*ATTR timed;
color OPS = list OP;
```

A.2 variable declarations

```
var out: IE;
var ins : IES;
var attr:ATTR;
var cost: COST;
var ops,ops2,ops3,ops4: OPS;
var success,success2: BOOL;
var exec, nonexec:OPS;
var id:ID;
var dur: DURATION;
var ope:OP;
var ievs:IEVS;
var number:INT;

(* 'top' is the end product of the workflow process*)
val top=18;
(*value initial1a contains all operations from the product data model, including
their ID, output element, input elements, and duration and cost attributes*)
val initial1a =[("op1", 1,[25,37],(1,0)),("op2", 2,[25,37],(1,0)),
               ("op3", 3,[33,37],(1,0)),("op4", 4,[33,37],(1,0)),
               ("op5", 5,[37,45],(2,0)),("op6", 6,[21,37],(1,0)),
               ("op7", 7,[24,37],(1,0)),("op8", 8,[23,37],(1,0)),
               ("op9", 9,[24,39],(5,0)),("op10", 10,[13,14,34,37,42],(5,0)),
               ("op11", 11,[31],(1,6)),("op12", 15,[16],(1,0)),
               ("op13", 15,[17],(1,0)),("op14", 15,[16,17],(2,0)),
               ("op15", 16,[25,30,35,36,44],(8,56)),
               ("op16", 17,[25,30],(1,0)),("op17", 18,[1],(0,0)),
```

```
              ("op18", 18,[2],(0,0)),("op19", 18,[8],(0,0)),
              ("op20", 18,[9],(0,0)),("op21", 18,[10],(0,0)),
              ("op22", 18,[11],(0,0)),("op23", 18,[15],(0,0)),
              ("op24", 18,[9,11,15],(0,0)),("op25", 28,[25,37],(1,0)),
              ("op26", 29,[25,30,35,36],(9,0)),
              ("op27", 30,[32,37,43],(4,0)),("op28", 31,[29,40,48],(7,0)),
              ("op29", 32,[1,2,3,4,5,6,7,8,10,27,28],(20,0)),
              ("op30", 34,[36,37,41],(1,42)),("op31", 40,[39,41],(0,3)),
              ("op32", 42,[47],(1,3)),("op33", 43,[39,49],(3,6))]
(*value initial1b contains all initially available data elements with their
corresponding value*)
val initial1b =[(13,0),(14,1),(21,1),(23,0),(24,3),(25,2),(27,0),(33,2),(35,1),
              (36,3),(37,1),(39,2),(41,0),(44,3),(45,2),(47,1),(48,0),(49,2)]
(*value initial1c contains a token with the operation ID for every operation
from the product data model*)
val initial1c = 1`"op1" ++ 1`"op2" ++ 1`"op3" ++ 1`"op4" ++ 1`"op5" ++ 1`"op6"
               ++ 1`"op7" ++ 1`"op8" ++ 1`"op9" ++ 1`"op10" ++ 1`"op11"
               ++ 1`"op12" ++ 1`"op13" ++ 1`"op14" ++ 1`"op15" ++ 1`"op16"
               ++ 1`"op17" ++ 1`"op18" ++ 1`"op19" ++ 1`"op20" ++ 1`"op21"
               ++ 1`"op22" ++ 1`"op23" ++ 1`"op24" ++ 1`"op25" ++ 1`"op26"
               ++ 1`"op27" ++ 1`"op28" ++ 1`"op29" ++ 1`"op30" ++ 1`"op31"
               ++ 1`"op32" ++ 1`"op33";
```

A.3 function declarations

```
(*memb checks whether x is an element of the list l*)
fun memb x l = List.exists (fn (y,_)=>(y=x)) l;

(*membs checks whether all elements of the first list are elements of the
second list*)
fun membs l1 l2 = List.all (fn x => memb x l2) l1;

(*divides the list ops into those operations that are ready for execution and
those that are not; it returns a pair of lists*)
fun executable (ievs:IEVS, ops:OPS) =
List.partition (fn (_,_,ies',_) => (membs ies' ievs)) ops;

(*returns a list containing the elements of list ops of which the output element
and its value already are a member of list ievs*)
fun already_calculated(ops,ievs) =
List.filter (fn (_,ie,_,_) =>  memb ie ievs) ops;

(*returns a list containing the elements of list ops of which the output element
and its value is not a member of list ievs*)
fun not_yet_calculated(ops,ievs) =
List.filter(fn (_,ie,_,_) => not(memb ie ievs)) ops;

(*returns the value of data element ie*)
fun get_value (ie:IE, (ie2,v)::ievs) =
if ie=ie2 then v else get_value(ie,ievs)
| get_value(ie:IE, []) = 0;

(*checks for operation with ID id, whether condition for execution is satisfied*)
```

```
fun check(id:ID, ievs:IEVS) =
case id of
"op12" => ((get_value(16,ievs))>8)
|"op13" => ((get_value(17,ievs))<23)
|"op17" => false
|"op19" => false
|"op20" => ((get_value(9,ievs))<0)
|"op22" => ((get_value(11,ievs))<0)
|"op23" => ((get_value(15,ievs))<0)
| _ => true;

(*returns a list containing those elements of ops that don't satisfy the
condition for execution*)
fun cond_not_satisfied(ops,ievs) =
List.filter (fn (id,_,_,_)=> not(check(id,ievs))) ops;

(*returns a list containing those elements of ops that satisfy the condition
of execution*)
fun cond_satisfied(ops,ievs) =
List.filter (fn (id,_,_,_) => (check(id,ievs))) ops;

(*calculates the value for the newly produced data element by adding the values
of the input elements when the input set contains two or more elements and by
changing the sign in front of the value when the input set contains only a
single element*)
fun calculation(one::two::ins, ievs:IEVS) =
get_value(one, ievs) + get_value(two,ievs) + calculation(ins, ievs)
| calculation([one], ievs) = ~(get_value(one,ievs))
| calculation([], ievs) = 0;

(*deletes element op1 from list ops*)
fun delete(ops, op1) = List.filter (fn ope => (ope<>op1)) ops

(* returns the cost of an operation *)
fun Cost((id,out,ins,(dur,cost)):OP) = cost;

(* returns the duration of an operation *)
fun Dur((id,out,ins,(dur,cost)):OP) = dur;

(* returns the operation ope for which Val(ope) is minimal,fails if it is
called with an empty list *)
fun SelMin Val (op1::op2::tail) = if Val(op1) < Val(op2)
           then (SelMin Val (op1::tail)) else (SelMin Val (op2::tail))
| SelMin Val (op1::[]) = op1
| SelMin Val ([]) = raise Match;

(* Selects an arbitrary element from the list l *)
fun select_random(l)  = List.nth (l,discrete(0,(List.length l)-1))

(* returns ID of an operation *)
fun ID((id,out,ins,(dur,cost)):OP)= id;

(* returns element with ID id from list ops *)
fun select_ID ops id = hd(List.filter (fn ope => (ID(ope) = id)) ops);
```

Building and Deploying Visualizations of Coloured Petri Net Models Using BRITNeY Animation and CPN Tools

Michael Westergaard and Kristian Bisgaard Lassen

Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark,
Email: {mw,k.b.lassen}@daimi.au.dk

Abstract. The contribution of this paper is a tutorial in the use of BRITNeY animation tool together with CPN Tools to make different views on Coloured Petri Nets. Examples of such views are message-sequence charts, gantt-charts, or SceneBeans animations showing the state of the model. In this paper we will describe how to generate message-sequence charts from executions of Coloured Petri Nets and how to create SceneBeans animations.

Keywords: Model-driven prototyping; visualization; Coloured Petri nets; CPN Tools.

1 Introduction

Formal models have proved their usefulness in modeling and understanding of complex systems [2,12,14,18], e.g., for verification of existing behavior or requirements engineering of needed behavior. However, when using a formal model such as Colored Petri nets (CP-nets or CPN) [8,11], it is only people familiar with the formalism who are truly able to understand the model of the system. A domain-user may understand the formalism used but is not capable, as the expert, to fully understand the model. Therefore formal models of systems are prone to be erroneous if they can not be fully understood and validated by a user with domain knowledge.

In this paper we present the tool BRITNeY[1] animation [23] which introduce an animation layer for CPNs. By using BRITNeY animation to animate CP-nets of systems the above problems of formal methods, being hard to understand, is alleviated. BRITNeY animation provides a uniform way to implement, integrate, and deploy visualizations of CPN models and has a pluggable architecture which make it possible to write customized plug-ins to animate the model but it also have more than a dozen predefined plug-ins for message sequence charts, SceneBeans [16, 19], plot different kinds of graphs and more. BRITNeY animation has already been used successfully to animate a network protocol [13] and

[1] Originally an abbreviation for Basic Real-time Interactive Tools for Net-based animation.

to animate the work flow of a bank work process for the purpose of requirements engineering [9].

We will describe methods to develop an animation on top of a CPN model by using an example of two runners. We describe the methods at the level where it is possible to replicate our example and use the methods in other projects.

The paper is structured as follows: In Sect. 2 we give a short overview of the architecture of the BRITNeY animation package. In Sect. 3 we briefly describe the example used throughout the paper, and go on to describe how to visualize the execution of the model using message sequence charts and a graphical animation. In Sect. 4 we mention related work and outline some of the planned new features of BRITNeY animation.

2 Architectural Overview

A well-known design pattern from object oriented software, is the model-view-controller (MVC) design pattern [5]. In the MVC design pattern, three participants collaborate to provide the implementation of an application, namely a model, a view, and a controller, see Fig. 1. The model models the state of the system, the view is a (graphical) representation of the current state of the model, and the controller implements the behavior of the system. The view may initiate actions in the controller.

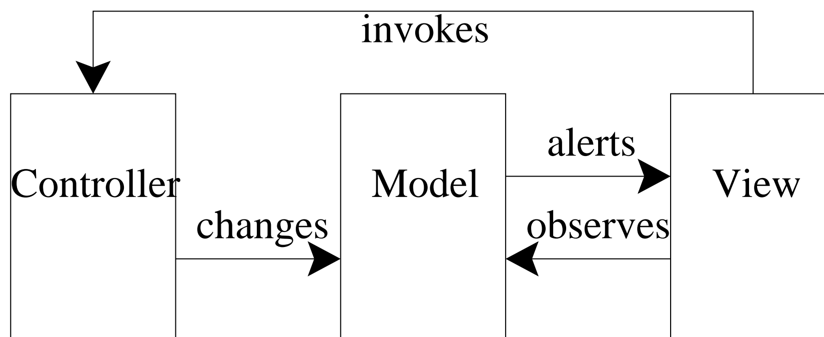


Fig. 1. Architectural overview of the model-view-controller design pattern.

The idea behind the BRITNeY animation package is to use a CPN model to model the state and behavior of the system (the model and controller), and use BRITNeY animation to model the view of the system. For more information on the architecture of BRITNeY animation, please refer to [22, Chap. 4].

3 Building Visualizations

In this section we will describe how to create two different graphical views of a CPN model. In Sect. 3.1, we give a brief description of how to work with the animation tools and the CPN model that will be visualized. In Sect. 3.2 we will describe the model we intend to create views of. In Sect. 3.3, we will show how to generate message sequence charts from the execution and finally in Sect. 3.4, we will describe how to create a domain specific animation. We will assume the reader knows CPN Tools, but we assume no prior knowledge of BRITNeY animation.

3.1 Working with BRITNeY

The animation tool can be obtained from the home-page [23] either as a downloadable version or as a Java Webstart [7] application.

In order to design animations that are easy to deploy, a certain design pattern must be used when adding animation to your model. The main idea of the design pattern is to add an init-transition, which is the only transition enabled in the initial step. All initialization of the animation can then be done in the action part of this transition. The reason we need to do this is that we will need to set up the animation for each simulation, but for efficiency, declarations are only evaluated once in CPN Tools. An overview of the points we have to go through is in Listing 1.1. In rest of this section, we will describe and exemplify each of these points in more detail.

Listing 1.1. The steps required to create an animation from an existing CPN-model.

1. Add declarations of structures for each animation plug-in you will need
2. Add an init-transition
3. Add code needed to set up the animation in an action part on the init-transition
4. Tie the execution of the model to the animation using action parts
5. (Optionally) deploy your animation on the web

3.2 Model

In this paper we will use a simple example to show the different aspects of animating a CP-net. In the example, two runners are competing to win a race. Initially the runners are at the start of the race, and they can then continue to the end of the race. During the race, the runners pass a drink stand. When the first runner passes the finish-line, he is declared the winner of the race, and a flag is lowered to celebrate. When the other runner crosses the finish-line, he is declared the loser of the race.

The CPN model in Fig. 2 captures the behavior of the example. In the CP-net, the two runners are modeled by tokens runner(1) and runner(2), both initially

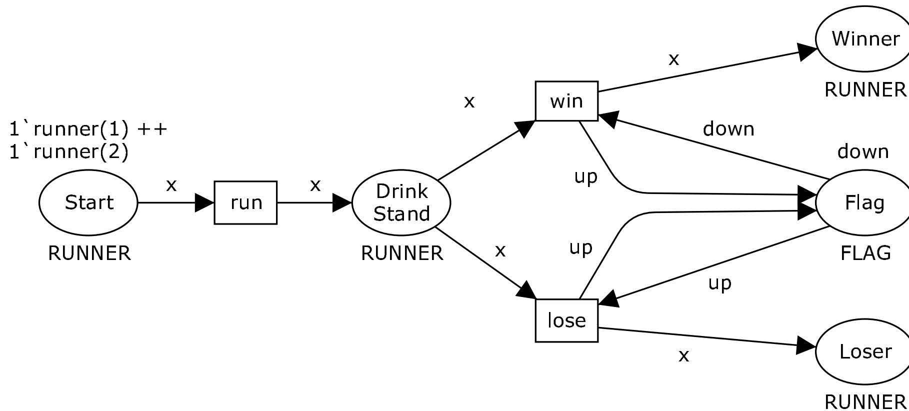


Fig. 2. CPN model of runner example.

placed on the Start-place. And the Flag is up. The run-transition moves the token representing a runner from the Start-place to the Drink Stand-place. When the flag is up, a runner-token on the Drink Stand-place is moved to the Winner-place and the flag is moved down by the win-transition. When the flag is down, the lose transition moved runner-tokens from the Drink Stand-place to the Loser-place.

3.3 Message-Sequence Charts

Message sequence charts (MSC) are well-known to protocol engineers, and it is therefore a good idea to be able to present the execution of a CPN model as an MSC. Using BRITNeY animation, it is very easy to create MSCs from CP-net executions. In this section, we will describe the process of adding an animation view to your model using MSCs as example. An example of an MSC that is generated from the model can be seen in Fig. 3.

The main points of this section are presented in the listings, which can later be followed in order to add animation to another CPN-model.

BRITNeY animation has a pluggable structure, meaning it can support different animation plug-ins. Out of the box, BRITNeY animation supports more than 10 different plug-ins. In order to be able to use an animation plug-in in our model, we must set up a connection to it using a declaration (as we would normally do in CPN Tools to declare a color-set or a CPN-variable). An example of a declaration of a connection to an animation plug-in can be seen in Listing 1.2. Here we declare a new connection called msc to an MSC animation plug-in. We initialize the animation plug-in with a user-friendly name, which will displayed to the user, namely Runners. All declarations of connections to animation objects are of this kind. We can interchange MSC with the name of the desired animation plug-in, we can use any identifier instead of msc, and we can give any string as the user-friendly name instead of Runners. The operations supported by the MSC animation plug-in can be seen in Listing 1.3. The plug-in allows

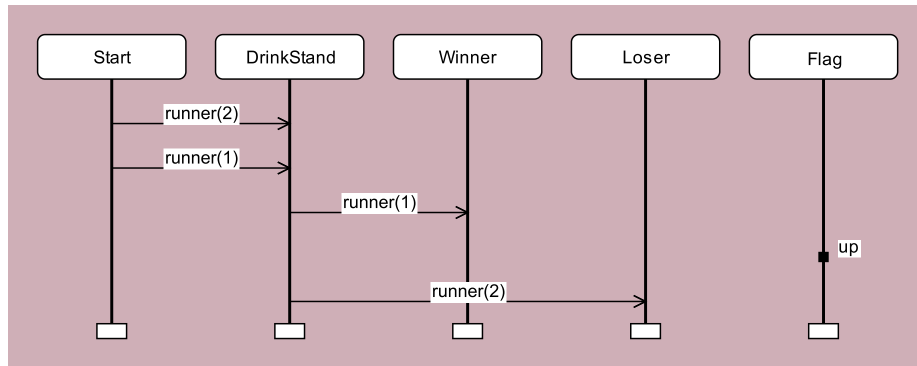


Fig. 3. Example of an MSC generated by the model in Fig. 2

us to create processes, create events between 2 processes and to create internal events inside a single process. A more detailed description of the MSC animation plug-in interface along with documentation of all animation plug-ins can be seen on the BRITNeY animation home-page [23].

Listing 1.2. The declaration of a structure that can be used to communicate with the animation plug-in for drawing MSCs.

```
structure msc = MSC(val name = "Runners");
```

Listing 1.3. The interface of the MSC animation plug-in.

```
void addProcess(String name)
void addEvent(String from, String to, String name)
void addInternalEvent(String process, String name)
```

The next step is to create an init-transition, a new transition that is always executed before all other transitions, while allowing the rest of the execution of the model to proceed as before adding the init-transition. One way to do this is outlined in Listing 1.4. Concrete examples of the application of this procedure can be seen by comparing the net in Fig. 2 with the nets in Figs. 4 and 6 where init-transitions have been added at the top left.

Listing 1.4. Adding an init-transition.

1. Create a transition named init
2. Create a place named Not inited of type UNIT with initial marking ()
3. Create a place named Inited of type UNIT with no initial marking

4. Create an arc from the Not inited-place to the init-transition with inscription () and an arc from the init-transition to the Inited-place
5. (If you have a hierarchical model) add copies of the Not inited-place on all pages, and fuse all together in a single fusion-set
6. Create a double-arc[2] with inscription () between the Inited-place and all transitions enabled in the inital step (except for the init-transition)

When we have created an init-transition, we need to add our initialization code in the action part of the init-transition. In our case, we will create 5 processes in the MSC, one for each of the places. The action part of the init-transition can be seen in Listing 1.5. Here we simply make 5 calls to the MSC animation plug-in, each call creating a new process.

Listing 1.5. The action part used to set up the animation.

```
1 INPUT ();
2 OUTPUT ();
3 ACTION
4 msc.addProcess("Start");
5 msc.addProcess("DrinkStand");
6 msc.addProcess("Winner");
7 msc.addProcess("Loser");
8 msc.addProcess("Flag")
```

Now, we have set up the entire animation and only need to update it during the animation. This is currently done by adding action parts to the model. In this example, we will create an action part for each of the transitions, which adds events to the MSC. For example, the action part for the run-transition can be seen in Listing 1.6. Whenever the run-transition occurs, an event is added from the Start process to the DrinkStand process, with a label describing which runner moved from the start of the race to the drink stand. Similar action parts are added to the other transitions.

Listing 1.6. The action part of the run-transition which updates the animation.

```
1 INPUT x;
2 OUTPUT ();
3 ACTION
4 msc.addEvent("Start", "DrinkStand", RUNNER.mkstr x)
```

The net we obtain after adding an init-transition, set-up code and action parts to all transitions can be seen in Fig. 4. The changes created to the original model from Fig. 2 are highlighted.

[2] CPN Tools uses interleaving semantics for simulation. In tools using partial order semantics, we would use a test-arc rather than a double-arc to preserve the semantics of the model.

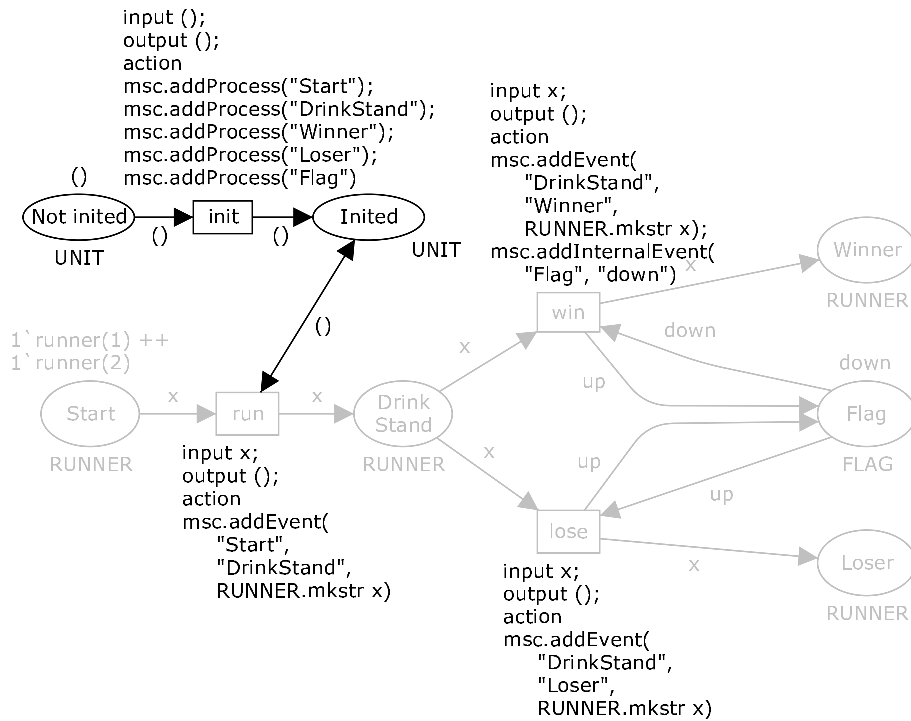


Fig. 4. CPN model of runner example which is able to generate an MSC of the execution.

We can now simulate the model in CPN Tools, and will notice that the animation is updated. We can also quit CPN Tools, and just simulate the model in BRITNeY animation. We will no longer be able to see how the model is updated, but we are still able to see that the animation is updated. If we want to see how the model is updated, the easiest way to deploy the animation is to simply require that users have CPN Tools and BRITNeY animation installed, and distribute the model saved by CPN Tools.

If we want to distribute a prototype to several users, who have little or no knowledge of CP-nets, we have an easier option, however. We can combine BRITNeY animation's ability to dump a simulator for a net with BRITNeY animation's powerful scripting engine and Java Webstart to deploy a visualization of a CP-net that can be started by clicking on a single link on a web-page. The steps we have to go through are outlined in Listing 1.7. We basically have to dump a simulator for our net, create a script to load a simulator and create a webstart loader for our simulator.

Listing 1.7. How to deploy a visualization of a CPN model.

1. Save a simulator for your net
2. Create a script to load and start your simulator
3. Download the offline version of BRITNeY animation
4. Modify `webstart/britney.jnlp` to suit your needs
5. Copy the files to a web-server

Saving a simulator for your net is easy:

Listing 1.8. Saving a simulator for a CP-net.

1. Load your net in CPN Tools
2. Ensure you are at the initial state
3. In BRITNeY animation find the simulator console corresponding to your net
4. Right click on the background of the simulator console
5. Select "Save simulator as" from the marking menu
6. Select a suitable name and location for the simulator

Now we need to create a script to load our simulator. BRITNeY allows us to create scripts using the full power of Java (but a more relaxed syntax if we desire so). We shall not go into detail about the scripting facilities, but just mention that we have access to an object, SimulatorService, which takes care of starting CPN simulators. A simple script to start a simulator can be seen in Listing 1.9. The first two lines of the script downloads and loads a simulator from the location `http://www.daimi.au.dk/~mw/local/tincpn/simulator.x86-win32`. You will obviously need to change the location of the simulator. The resulting simulator is very low-level, and only used to construct a more high-level simulator. Line 3 of the script obtains this high-level simulator. One feature of the high-level simulator is the ability to simulate nets. In line 4 of the script, we start a new simulation of 10 steps in 1 step increments. We pause for 1000 ms after each simulation step and we show a progress-bar at the bottom of the tool. You will probably need to adjust these parameters, at least the total number of steps and the amount of time to pause. Save the script as something reasonable at the same location you saved your simulator.

Listing 1.9. A simple script to download and start a simulator.

```
1  s = SimulatorService.getNewSimulator(
2      new java.net.URL("http://www.daimi.au.dk/~mw/local/tincpn/sim.x86-win32"));
3  hs = s.getHighLevelSimulator();
4  hs.startSimulation(10, 1, 1000, true);
```

The offline version of BRITNeY animation can be obtained from the homepage [23], and when you unzip the archive, you will find three jar-files and a number of directories. You should edit the file `webstart/britney.jnlp`. You need to make the changes:

Listing 1.10. Changes to make to `webstart/britney.jnlp`.

1. Change `http://www.daimi.au.dk/~mw/local/tincpn/` to location you intend to deploy your visualization everywhere in the file
2. Uncomment the line containing “¡property name=”tincpn.script.load”. . . .”
3. Change the location of the configuration script to the location you intend to store your configuration script
4. (Optionally, for experienced users only) remove any plug-ins, you do not need

Finally you need to copy the files to your web-server. The files, you will need are:

Listing 1.11. The files you need to make available to deploy an animation.

1. All files and directories from the offline version of BRITNeY animation
2. Your edited copy of `britney.jnlp` (should live at the top with `britney.jar`)
3. Your configuration script
4. Your simulator

Now you can simply point your users to the location of `britney.jnlp` on your web-server. A nice way to do that is to create a web-page with some information about the visualization and some use-scenarios and link to `britney.jnlp` from that page.

3.4 SceneBeans Animation

Fig. 5. Animation of runner example.

This section describes how to build a SceneBeans animation. We will reuse our example and show two runners go from start to finish crossing the drink stand on the way. Fig. 5 shows a snapshot of the animation.

We describe the process by incrementally changing our CP-net from Fig. 2 and a SceneBeans XML description to achieve our goal. We will use the procedure from Listing 1.1, which we used to create an MSC view of the model.

The first thing that needs to be done is to set up a connection to the SceneBeans animation plug-in in BRITNeY. This is done using the declaration as in Listing 1.12.

Listing 1.12. The declaration of a structure that can be used to communicate with the animation plug-in for SceneBeans.

```
1 structure runners = SceneBeans(val name = "Runners")
```

Listing 1.13 shows the interface accessible through the structure.

Listing 1.13. The interface of the SceneBeans animation plug-in.

```
1 String  getNextEvent()
2 boolean hasMoreEvents()
3 void    invokeCommand(String name)
4 String  peekNextEvent()
5 void    setAnimation(String filename)
```

We then need to create an initialization transition as described in Listing 1.4.

SceneBeans descriptions are written in an XML file. To specify the location of this XML file use the expression as in Listing 1.14 on the initialization transition, as in Fig. 6.

Listing 1.14. Setting the location of the SceneBeans specification.

```
1 runners.setAnimation "path-to-XML-file"
```

In Listing 1.14 the path-to-XML-file can e.g. be c:/animation.xml. When the simulator has evaluated the expression, BRITNeY animation shows the animation as it will look before any behavior is executed.

We tie the CP-net to the animation by specifying in code segments what should happen when a transition occurs. In Fig. 6 we have annotated our example net with the needed code segments.

The two functions which are called in the code segments have similar definitions. In the following we will only look at the runToDrinkStand function and how it is hooked up to the animation. The function is defined as:

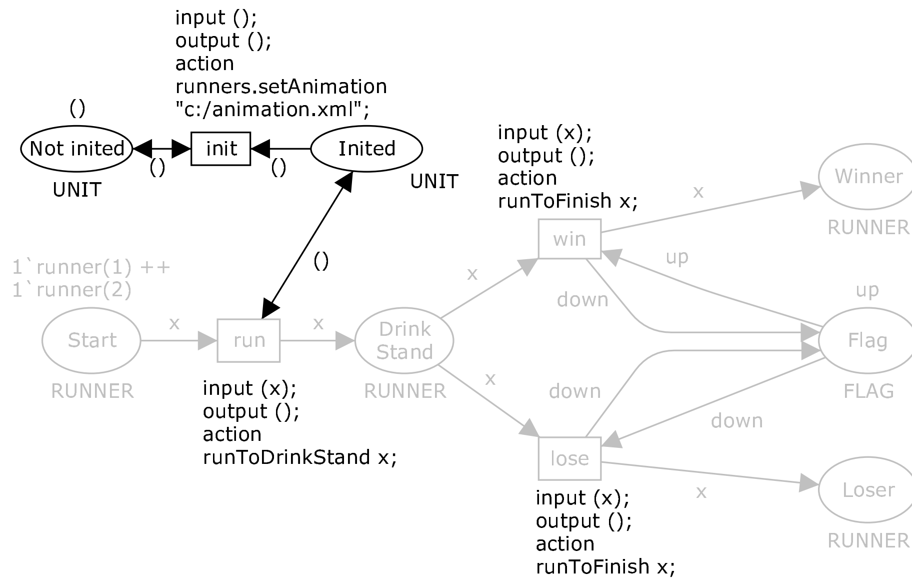


Fig. 6. Runner CP-net annotated for the SceneBeans animation.

```
1  fun runToDrinkStand x =
2   let val _ = if x = r(1)
3                then runners.invokeCommand "runToDrinkStand-r(1)"
4                else runners.invokeCommand "runToDrinkStand-r(2)"
5       val _ = runners.waitForEvent "command-executed"
6   in ()
7   done
```

Two things are worth to notice in this function. Firstly, we use the function invokeCommand to execute a predefined command in SceneBeans; this is executed asynchronously. Secondly, we use the function waitForEvent to wait for a named event; in our case the event command-executed. The reason for why not just use invokeCommand is that it is executed asynchronously, so a situation could occur where two behaviors are executed concurrently. This might be the wanted behavior in some cases but not in our example.

We have now finished in adapting our CP-net to the SceneBeans animation. The final thing that needs to be done is to specify the commands in SceneBeans which we called from our CP-net. The XML code for this animation can be seen in Listing 1.15. For the purpose of this tutorial it is not important to read and understand the code. It is included for the reader to see the connection between the CP-net and the SceneBeans specification.

Listing 1.15. SceneBeans Specification of the Runners

```
<animation height="150" width="400">
  <forall var="i" values="1 2">
    <behaviour algorithm="move" event="command-executed"
               id="runToDrinkStand-r(${i})">
      <param name="from" value="10"/>
      <param name="to" value="150 + 30 * ${i}"/>
      <param name="duration" value="1"/>
    </behaviour>
    <event event="command-executed" object="runToDrinkStand-r(${i})">
      <announce event="command-executed"/>
    </event>
    <behaviour algorithm="move" event="command-executed"
               id="runToFinish-r(${i})">
      <param name="from" value="150 + 30 * ${i}"/>
      <param name="to" value="340 + 10 * ${i}"/>
      <param name="duration" value="1"/>
    </behaviour>
    <event event="command-executed" object="runToFinish-r(${i})">
      <announce event="command-executed"/>
    </event>
    <command name="runToDrinkStand-r(${i})">
      <reset behaviour="runToDrinkStand-r(${i})"/>
      <start behaviour="runToDrinkStand-r(${i})"/>
    </command>
    <command name="runToFinish-r(${i})">
      <reset behaviour="runToFinish-r(${i})"/>
      <start behaviour="runToFinish-r(${i})"/>
    </command>
  </forall>
  <draw>
    <forall var="i" values="1 2">
      <transform type="translate">
        <param name="translation" value="(0,50)"/>
        <animate behaviour="runToDrinkStand-r(${i})" param="x"/>
        <animate behaviour="runToFinish-r(${i})" param="x"/>
        <transform type="scale">
          <param name="x" value="0.4"/>
          <param name="y" value=".4"/>
          <primitive type="sprite">
            <param name="src"
                   value="file:///c:/Runners/runner-${i}.gif"/>
          </primitive>
        </transform>
```

```
44        </transform>
45      </forall>
46      <transform type="translate">
47        <param name="translation" value="(150,10)"/>
48        <transform type="scale">
49          <param name="x" value="0.4"/>
50          <param name="y" value=".4"/>
51          <transform type="scale">
52            <param name="y" value="0.9"/>
53            <primitive type="sprite">
54              <param name="src"
55                     value="file:///c:/Runners/drink-stand.jpg"/>
56            </primitive>
57          </transform>
58        </transform>
59      </transform>
60      <transform type="translate">
61        <param name="translation" value="(10,50)"/>
62          <transform type="scale">
63          <param name="x" value="0.4"/>
64          <param name="y" value=".4"/>
65          <primitive type="sprite">
66            <param name="src"
67                   value="file:///c:/Runners/start.JPG"/>
68          </primitive>
69        </transform>
70      </transform>
71      <transform type="translate">
72        <param name="translation" value="(350,50)"/>
73        <transform type="scale">
74          <param name="x" value="0.4"/>
75          <param name="y" value=".4"/>
76          <primitive type="sprite">
77            <param name="src"
78                   value="file:///c:/Runners/stop.JPG"/>
79          </primitive>
80        </transform>
81      </transform>
82    </draw>
83  </animation>
```

Lines 3-8 and 12-17 in the XML file describe the behaviors that exist in the animation; i.e. both runners can run from start to the drink stand and from the drink stand to finish. Lines 9-11 and 18-20 describes that events will be generated after each behavior is executed; e.g. after runner one goes from start to the drink stand an event is generated saying that the behavior has completed

to signal to the CP-net it can proceed on simulating. Lines 21-28 specify the commands which can be executed on our SceneBeans; i.e. the commands we call from the CP-net, e.g., runToDrinkStand-r(1). Lines 30-82 describes the figures that is in the animation; i.e. the two runners, start and finish signs and the drink stand. Also, this part describes how the figures are related to the defined behaviors; i.e. runner number one have the two behaviors runToDrinkStand-r(1) and runToFinish-r(1).

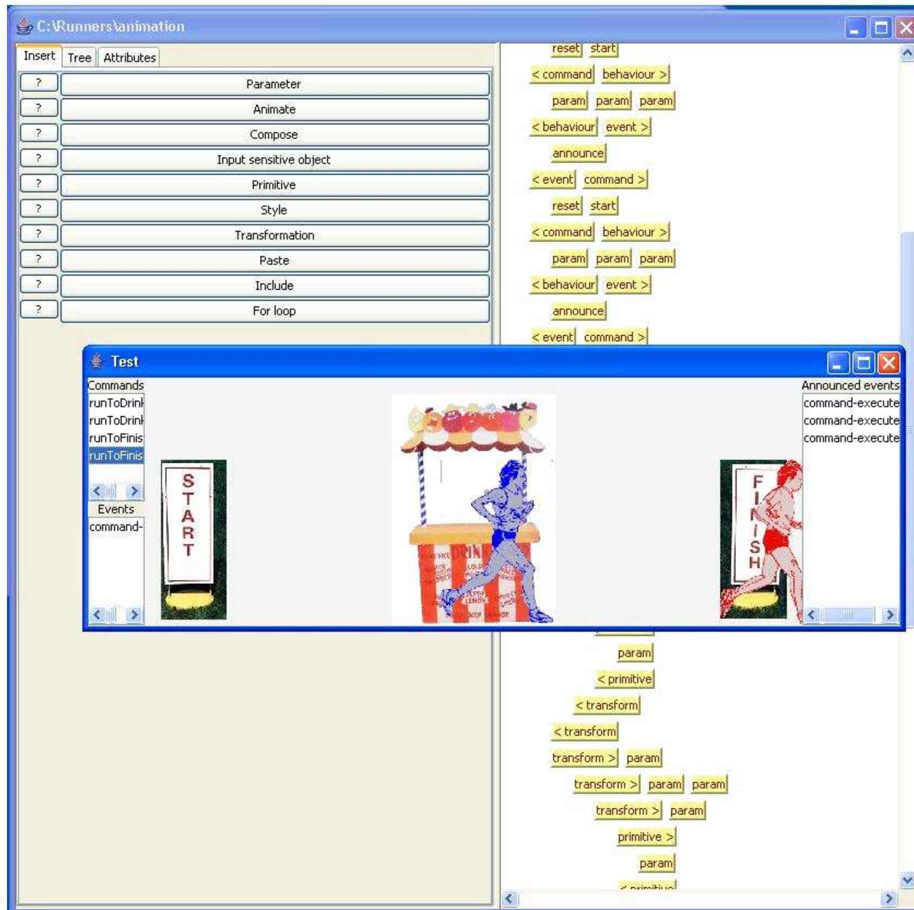


Fig. 7. SceneBeans XML editor.

The actual SceneBeans animation can be built in SClub, which is an extended version of BRITNeY animation, which provides, among other things, an XML editor for specifying SceneBeans animations. This editor interprets the XML file while it is being constructed so it is possible to preview the animation and test

commands as you go along. Figure 7 shows the runners animation being built and tested. In the figure you can see that the editor is aware of which constructs to suggest depending on the context of position of the cursor. Also, you can see the defined commands and events. These can be executed by double-clicking on them.

This is all that is needed to make a simple animation using the SceneBeans plug-in. Obviously, this is a very small example but the techniques used to make this animation are well suited for large scale animations.

4 Related Work and Future Improvements

BRITNeY animation supports adding animations to CPN models by annotating transitions with function calls, which are executed whenever the transition occurs. In the following, we outline how a number of other modeling tools allow users to use visualization.

ExSpect [20], a tool for modeling based on CP-nets, allows the user to view the state by associating widgets with the state of the model, and asynchronously interact with the model, also using simple widgets. In this way, it is easy to create simple user interfaces that support displaying information, but support for creating more elaborate animations is not easily available.

Mimic/CPN [17] makes it possible to animate models within Design/CPN [1, 3], which is another tool for modeling using CP-nets. CPN models are animated by Mimic/CPN by using function calls that are executed whenever a transition of the CP-net occurs. The animations are drawn using an application that resembles traditional drawing programs. Input from the user is possible by showing a modal dialog, where the simulation of the model is stopped while the user is expected to input information. It is also possible to make click-able regions, and the model can then query if one of these has been clicked.

LTSA [15], a tool for modeling using timed labeled transition systems, allows users to animate models using the SceneBeans library [16, 19]. In LTSA animations are tied to the models by associating each animation activity with a clock; resetting a clock corresponds to starting an animation sequence. The animation sequence or a user with his mouse can then send events which correspond to the progress of the timer.

Another approach, taken by e.g. the Comms/CPN [4] library for Design/CPN and CPN Tools, is to provide a TCP/IP abstraction, allowing the user to code the user interface in any language and use RPC to communicate with it. This approach resembles creating real programs quite a lot, but the user has to go through the hassle of implementing RPC himself, making this approach difficult to use in practice.

PNVis [10] is an add-on for the Petri Net Kernel [21], a highly modular tool for editing Petri nets. PNVis associates tokens with 3D objects and certain places with locations in a 3D world. Moving tokens corresponds to moving the associated object in the 3D world. PNVis is suitable for modeling physical systems, but not really useful for creating prototypes of software.

The Play-Engine [6] supports the developer in implementing a prototype by inputting scenarios (play-in) via an application-specific GUI, and then executes the resulting program (play-out). This makes the model implicit as the model is created indirectly via the input scenarios. In a sense, we create a prototype via direct manipulation, but as the model of the system is created indirectly via the input scenarios it may be difficult to use the model for analysis and as basis for implementation of the final system. The reason is that an implicitly created model is difficult to interpret as it is automatically generated.

We have mentioned a number of libraries, all of which support animation in different ways. Using some libraries, animation is integrated with the modeling formalism, such as the use of timers in LTSA or the ability to view or change the marking of places in ExSpect. Some libraries are easy to extend, such as animations in LTSA, as the SceneBeans library allows users to easily extend it with new animation primitives. Also, animations created using Comms/CPN can easily be extended, as the "animation" is just a custom (Java) application. Some libraries make it easy to design animations, such as ExSpect and Mimic/CPN, which both provide a graphical user interface to design animations.

The approach of the current version of BRITNeY animation resembles a combination of Mimic/CPN and Comms/CPN, as the animation is driven by function calls associated with transitions to an external application. The main feature offered by BRITNeY from a user point of view is thus compatibility with CPN Tools (rather than the discontinued Design/CPN) and platform-independence. From a developer point of view, BRITNeY provides good foundations for allowing closer integration with the model by allowing parts of the animation to inspect and modify tokens on fusion places of the CPN model, much like how widgets are associated with places in ExSpect. This is an important part of future work.

An important new feature of BRITNeY animation is that it is possible to deploy animations in a way that allows even non-technical users to download and experiment with the animation. Another part of the future work is to make this process even easier by adding a wizard to take care of all the details.

BRITNeY animation is currently useful enough to use in real projects, and has, as mentioned in the introduction, already been used in two projects as part of the development.

References

1. Design/CPN. Online `www.daimi.au.dk/designCPN`.
2. C. Bossen and J.B. Jørgensen. Context-descriptive prototypes and their application to medicine administration. In *DIS '04: Proc. of the 2004 conference on Designing interactive systems*, pages 297–306, Boston, MA, USA, 2004. ACM Press.
3. S. Christensen, J.B. Jørgensen, and L.M. Kristensen. Design/CPN—A Computer Tool for Coloured Petri Nets. In *Proc. of TACAS'97*, volume 1217 of *LNCS*, pages 209–223. Springer-Verlag, 1997.
4. G. Gallasch and L.M. Kristensen. A Communication Infrastructure for External Communication with Design/CPN. In *Proc. of 3rd Workshop on Practical Use of*

Coloured Petri Nets and the CPN Tools, volume PB-554 of *DAIMI*, pages 79–93. Department of Computer Science, University of Aarhus, 2001.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
6. D. Harel and R. Marelly. *Come, Let's Play.* Springer-Verlag, 2003.
7. Java Network Launching Protocol and API. `http://jcp.org/en/jsr/detail?id=56`.
8. K. Jensen. *Coloured Petri Nets—Basic Concepts, Analysis Methods and Practical Use. Volume 1: Basic Concepts.* Springer-Verlag, 1992.
9. J.B. Jørgensen and K.B. Lassen. Aligning Work Processes and the Adviser Portal Bank System. In *REBNITA05*, 2005.
10. E. Kindler and C. Páles. 3D-Visualization of Petri Net Models: Concept and Realization. In *Proc. of ICATPN 2004*, volume 3099 of *LNCS*, pages 464–473. Springer-Verlag, 2003.
11. L.M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.
12. L.M. Kristensen and K. Jensen. Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad-hoc Networks. In *Integration of Software Specification Techniques for Application in Engineering*, volume 3147 of *LNCS*, pages 248–269. Springer-Verlag, 2004.
13. L.M. Kristensen, M. Westergaard, and P.C. Nørgaard. Model-based Prototyping of an Interoperability Protocol for Mobile Ad-hoc Networks. Accepted for Fifth International Conference on Integrated Formal Methods, 2005.
14. L. Lorentsen, A-P Tuovinen, and J. Xu. Modelling Features and Feature Interactions of Nokia Mobile Phones Using Coloured Petri Nets. In *Proc. of ICATPN 2002*, volume 2360 of *LNCS*, pages 294–313, 2002.
15. J. Magee and J. Kramer. *Concurrency – State Models and Java Programs.* John Wiley & Sons, 1999.
16. J. Magee, N. Pryce, D. Giannakopoulou, and J. Kramer. Graphical Animation of Behavior Models. In *Proc. of 22nd International Conference on Software Engineering*, pages 499–508. ACM Press, 2000.
17. J.L. Rasmussen and M. Singh. *Mimic/CPN. A Graphical Simulation Utility for Design/CPN. User's Manual.* `www.daimi.au.dk/designCPN`.
18. J.L. Rasmussen and M. Singh. Designing a Security System by Means of Coloured Petri Nets. In *Proc. ICATPN 1996*, volume 1091 of *LNCS*, pages 400–419. Springer-Verlag, 1996.
19. SceneBeans. Online `www-dse.doc.ic.ac.uk/Software/SceneBeans`.
20. The ExSpect tool. `www.exspect.com`.
21. M. Weber and E. Kindler. The Petri Net Kernel. In *Petri Net Technologies for Modeling Communication Based Systems*, volume 2472 of *LNCS*, pages 109–123. Springer-Verlag, 2003.
22. M. Westergaard. Building Verifiable Software Prototypes using Coloured Petri Nets. Progress report, Department of Computer Science, University of Aarhus.
23. M. Westergaard. TIN-CPN, BRITNeY animation, and SClub website. Online `wiki.daimi.au.dk/tincpn`.

An Abstract Model of Routing in Mobile Ad Hoc Networks

Cong Yuan[1], Jonathan Billington[1], and Jörn Freiheit[2]

[1] Computer Systems Engineering Centre
University of South Australia
Mawson Lakes Campus, SA 5095, AUSTRALIA
Email: yuacy003@students.unisa.edu.au, jonathan.billington@unisa.edu.au
[2] Programming Logics Group, Max-Planck Institute
Saarland, 66123, GERMANY
Email: freiheit@mpi-inf.mpg.de

Abstract. Modelling Mobile Ad Hoc Networks (MANETs) is a challenge because the topology of such networks changes dynamically and unpredictably. We create a highly abstract Coloured Petri Net model of routing in a MANET based on the Destination-Sequenced Distance-Vector (DSDV) routing protocol. Our experiments show that this model can simulate the required dynamic changes of network topology and reveal that incorrect routing information can be created and propagated in the MANET.

Keywords: MANETs, DSDV routing protocol, Modelling and Simulation, Coloured Petri Nets.

1 Introduction

A Mobile Ad Hoc Network (MANET) [1] is a collection of wireless mobile nodes, such as laptops, mobile phones and Personal Digital Assistants (PDAs), which establish a temporary network without the help of any pre-existing infrastructure or centralized administration. Packet forwarding, routing, and other network operations are carried out by the individual nodes themselves [18]. In addition, nodes freely join in and move out, which results in the network topology constantly changing, so that conventional routing protocols do not work well for MANETs. Most current routing protocols designed for MANETs are still under development [2]. So far their definitions are not mature enough for Internet standards and they have mainly been evaluated by simulation and live testing. Unfortunately, simulation and testing are not sufficient to verify that there are no subtle errors or design flaws left in a protocol [28]. To achieve this goal we need to formally verify its operation. Formal verification firstly requires the creation of a formal model of the system.

Concerning formal verification of routing protocols for MANETs, Bhargavan et al. [4] and Obradovic [22] verify the Routing Information Protocol (RIP) [19] and the Ad-hoc On-Demand Distance Vector (AODV) Routing Protocol [23]. They analysed AODV and identified a flaw that could lead to a loop. This was done using SPIN [11] and the HOL Theorem Proving System [10]. In order to realise loop free behaviour, a modification was suggested and verified. Their approach verified the general case, but required a significant amount of user interaction. Wibling et al. [28] consider an automatic verification strategy, and use two model checking tools, SPIN and UPPAAL [17], to verify both the data and control aspects of the Lightweight Underlay Network Ad hoc Routing (LUNAR) protocol. However, they just studied a limited set of topologies.

The purpose of this paper is to illustrate the dynamic operations of a MANET using Coloured Petri Nets (CPNs) [13, 14]. However, it is not easy to create a CPN model of a MANET because the topology of such a network changes dynamically and unpredictably. Only a few attempts have been made to model a highly dynamic system topologies using CPNs. Findlow and Billington [7] used High-Level Petri Nets [5] to model dynamic dining

philosophers, where philosophers can come and go unpredictably. The topology of the system is circular, but arbitrarily expanding and contracting as philosophers come and go and can take different positions in the circle. In MANETs there is no regular structure but an arbitrarily changing network topology.

Xiong et al. [29] present a timed CPN model for AODV. They considered that it was too difficult to model the highly dynamic topology of MANETs with CPNs directly and proposed a topology approximation (TA). They assume that every node has the same transmission range and thus that the neighbourhood relation is symmetric. They also assume that each node in the MANET has the same number of neighbours, a rather unrealistic assumption. Further, in simulation experiments, the number of neighbours needs to be supplied by the analyst.

Kristensen and Jensen [15] model and analyse the Edge Router Discovery Protocol (ERDP), a protocol for connecting gateways in MANETs to edge routers in fixed networks. Thus their CPN model does not involve the dynamically changing topology of MANETs. Kristensen et al [16] use hierarchical CPNs to model a fixed number (in their case 4) of MANETs which communicate with each other via an IPv6 network. Nodes can join and leave a particular MANET and can move from one MANET to another. The topology of a MANET is described explicitly by storing pairs of nodes, where each pair represents a one directional link. The model includes simple forwarding of user packets directly from the source node to the destination node, irrespective of which MANET they belong to. The model thus abstracts from the mechanism by which these packets are forwarded through different nodes of the various MANETs to their destination, which we believe is a key part of MANET design. Thus no attempt is made to model a routing protocol or how the packets are routed in the network.

In this paper, we consider the Destination-Sequenced Distance-Vector (DSDV) routing protocol [24, 25], because DSDV has relatively low complexity compared with many other ad hoc routing protocols. DSDV is a representative *proactive protocol*. The primary characteristic of such a routing approach is that each node tries to maintain a route to every other node in the network at all times. It has the advantage that there is no delay to begin a session, and tends to perform well in networks where there are a significant number of data sessions within the network [2].

The main aim of this paper is to provide the first CPN model of the basic functions of the DSDV routing protocol as a fisrt step towards its formal specification and verification. We introduce the basic features of DSDV and then provide an abstract CPN model of the DSDV routing mechanism. We perform some simulation experiments to demonstrate the way the model captures the dynamic changes in network topology. In doing so, we uncover some interesting errors and suggest modifications to the procedures to eliminate these errors.

There are several contributions of this paper. Firstly, we demonstrate that CPNs can model the dynamically changing network topologies associated with MANETs in an elegant and simple way, without using the assumptions made in [29] and without requiring the relatively complex 4-level hierarchical structure used in [16]. Secondly, we provide the first CPN model of the DSDV routing procedures and discover two errors in these procedures. This provides the first analysis of the key component of DSDV, the use of sequence numbers to discard old information. Thirdly we discuss modifications to the DSDV procedures for updating routing tables that we believe will remove these errors. These modifications are implemented in a revised CPN model and our simulations have shown that they have been effective in eliminating these errors in the scenarios run so far.

The rest of the paper is organised as follows. Section 2 gives an introduction to DSDV and explains the basic operations of the protocol. In Section 3, we describe a highly abstract

CPN model of the DSDV protocol. Simulation experiments are conducted in Section 4, and discussed in Section 5. Finally, conclusions and future work are presented in Section 6.

2 Destination-Sequenced Distance-Vector Routing Protocol

2.1 Background

The destination-sequenced distance-vector routing protocol is derived from distance-vector routing algorithms [20]. Such algorithms are often referred as the Distributed Bellman-Ford (DBF) algorithm since they are based on a shortest path computation algorithm presented by Bellman [3]. The first description of the distributed algorithm was given by Ford and Fulkerson [8]. This algorithm was the original Arpanet routing algorithm, and was also used in the Internet under the name RIP and in early versions of DECnet and Novell's Internet Packet eXchange (IPX). AppleTalk and Cisco routers use improved distance vector protocols [26]. The distance vector algorithm can cause the formation of both short-lived and long-lived loops [6]. The main cause of the formation of routing loops is that nodes choose their next hops in a completely distributed fashion based on information that may be stale and therefore incorrect. The modifications [9, 12, 21] designed to eliminate the looping problem are not feasible in MANETs because of the rapidly changing topology of such a network [25].

RIP [19] is a simple and practical distance vector protocol. It is easy to understand and modify. However, like other distance-vector algorithms, RIP also suffers from very slow convergence (the *counting to infinity* problem [26]). Despite this problem, without the ability to handle rapid topological changes, the usefulness of RIP in MANET is limited [25]. Furthermore, the techniques designed in [19] to solve *counting to infinity* are not useful within the wireless environment. For these reasons, in 1994 C. E. Perkins and P. Bhagwat [24] presented the destination-sequenced distance-vector routing protocol for ad hoc networks. In 2001, a more comprehensive protocol specification was given by Perkins [25], and a recent description by Royer can be found in [2]. This protocol preserves the simplicity of RIP and avoids the looping problem by using sequence numbers. The sequence number is attached to each route entry in the routing tables stored in nodes, so they can quickly distinguish stale routes from new ones, and thus avoid formation of routing loops. A brief introduction to the basic routing algorithm of DSDV is given in the next subsection mainly based on [25]. The purpose is not to give a complete description of DSDV, but to provide sufficient information to understand the CPN model in the next section. In this model, we relax the assumption that links are bidirectional [25], because asymmetries of transmission ranges are prevalent in a wireless environment.

2.2 Protocol Overview

In DSDV, every mobile node maintains a route to every other node (i.e. destination) in the network. Thus its routing table comprises a list of route entries. A route entry corresponding to a destination contains the following attributes:

- **Destination**: IP address of the destination;
- **Nexthop**: IP address of next node along the route to the destination;
- **Metric**: the number of nodes (hops) required to reach the destination;
- **SeqNr**: last recorded *sequence number* for the destination;
- **Installtime**: the time when this route entry is received.

The purpose of sequence numbers is to track changes in topology. Each node keeps its own sequence number, which is increased whenever important changes are made to its routing table. When a route entry to a destination is established, it is stamped with the current sequence number of this destination. As the topology of the network changes, more recent route entries have higher sequence numbers, so that nodes can distinguish between current and stale route entries by comparing the sequence numbers of these entries. In order to keep routing tables consistent in a dynamically changing topology, each node periodically transmits updates using a *full dump* packet, and transmits an *incremental* update immediately after a significant change to its routing table. These updates comprise a list of triples of the form: $(\mathbf{Destination}, \mathbf{Metric}, \mathbf{SeqNr})$, derived directly from the routing table of a node. A full dump comprises a list of triples where each triple corresponds to an entry in the routing table. An incremental update only carries triples that have changed since the last full dump. Each node may receive routing information sent by another node. It utilizes this information to recompute its routing table entries. For every triple received, the node firstly checks its own routing table to find whether or not an entry to the same destination exists. If such an entry does not exist, the node adds this route entry received to its table after: incrementing the metric of this entry by one hop; adding the sender as the next hop; and including the current sequence number of this destination. If such an entry exists, the node will choose the entry with the higher sequence number. If two entries have the same sequence number, the entry with the shorter metric will be chosen. These two situations can guarantee loop-free paths to each destination in DSDV (A proof of correctness of this point is given in [25]). Nodes in a MANET may cause *broken links* as they move from place to place. A broken link may be inferred by a node if no message has been received for a while (e.g. one time interval of the periodic broadcast) from a former neighbour. The metric of a broken link is represented by ∞. When a link to the next hop is broken, any route through this next hop is immediately assigned an ∞ metric, and its sequence number is increased by 1 [25].

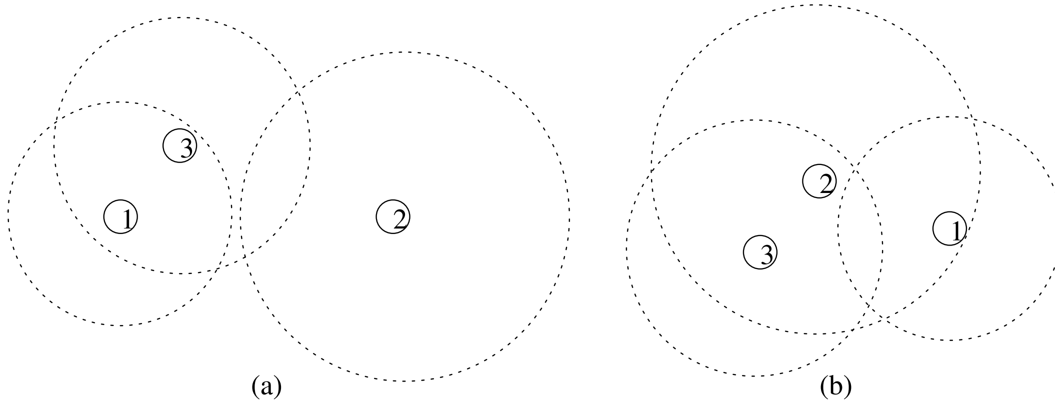


Fig. 1. The neighbourhood relation between the nodes in a MANET

In the context of mobile ad hoc networking, each node can communicate directly with any other node within its transmission range. If the node wants to communicate with nodes which reside beyond this range, intermediate nodes between them are used to relay the messages. We consider that node A is a neighbour of node B if node A is within node B's transmission range. In a realistic environment, mobile hosts have different software/hardware configurations and different radio interfaces [18], which can lead to variability in their transmission ranges. Consider the example of a MANET comprising 3 nodes (node 1, node 2 and node 3) shown in Fig. 1. In this figure, the full circles represent nodes, and the dotted circles represent transmission ranges of the nodes. The number positioned in each of the small circles represents

the node's identity (i.e. IP address). Fig. 1 shows 2 snapshots of the network as time increases and the nodes move in the network. As shown in Fig. 1(a), node 1 and node 3 are neighbours of each other. Node 2 is not a neighbour of node 1 or node 3, because it is not within the transmission range of either node. Similarly, node 2 has no neighbours because no other nodes are in its range. Within a MANET, each node can move at will, changing its neighbours arbitrarily. Thus in Fig. 1(b), node 1 is now a neighbour of node 2, node 2 and node 3 are neighbours of each other, but node 2 is not a neighbour of node 1. Therefore in a highly dynamic MANET the neighbourhood relation may not be symmetric. In previous work, the general assumption has been made that every node has identical capability and responsibility, so that a MANET is fully symmetric [4, 25, 29]. However, a key objective of our research is to precisely mimic the dynamic nature of a MANET, and thus we do not make this assumption.

In [25], many parameters which control the behaviour of DSDV, such as the frequency of broadcast, the frequency of full dumps versus incremental updates and the percentage change in the routing metric which can trigger an incremental update are not given. Moreover, when and how each node updates its sequence number in DSDV are not described explicitly. There is also not any description of the initial state in the DSDV process. Therefore, in order to illustrate the behaviour of DSDV completely and unambiguously, we need to make some assumptions. According to the example presented in [25], there is a route entry to itself in the routing table of each node. For our modelling and analysis, we assume that each node initially only has the route entry to itself in its routing table with its sequence number set to 0. The node increases its sequence number by 2 whenever there is a significant change to its routing table. Then this node broadcasts the triples associated with the changed route entries immediately. This assumption is indeed safe for DSDV because it satisfies the need of DSDV: more recent route entries should have higher sequence numbers in the routing table. Meanwhile, it does not influence the routing algorithm of DSDV in [25]. To simplify the problem, we ignore the install time of each route entry.

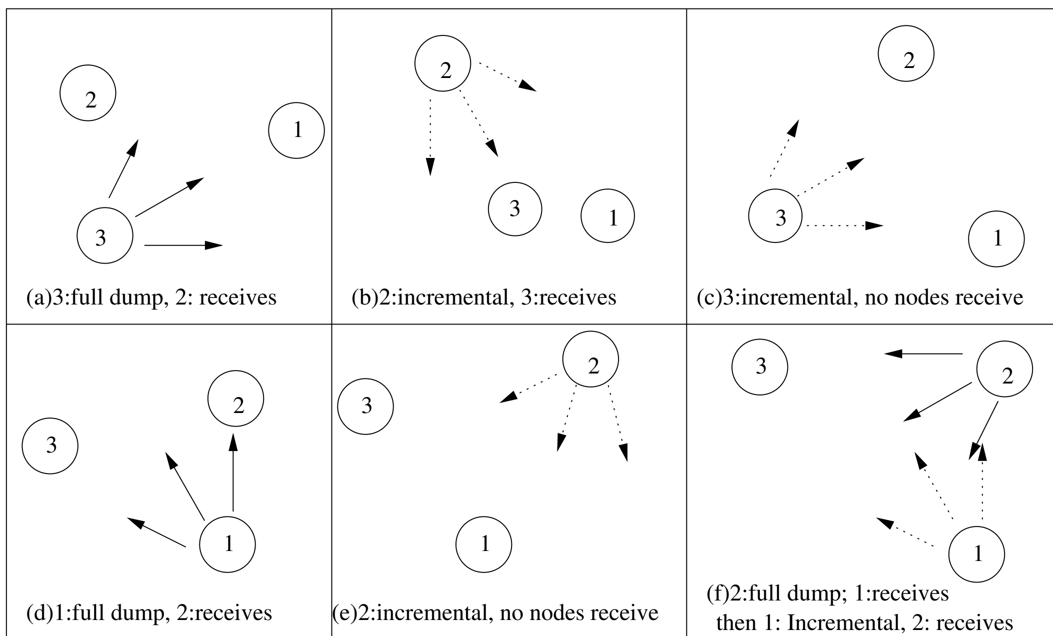


Fig. 2. An example of DSDV in operation (1) route advertisements

An example of DSDV in operation with unidirectional links is depicted in Fig. 2. In this figure, the solid arrows indicate that the sender broadcasts a full dump to its neighbours, and the dashed arrows indicate that the sender broadcasts an incremental update. In order

Table 1. Initial routing tables of nodes 1, 2 and 3

Dest.	Next	Met.	SeqNr
1	1	0	(1, 0)

Dest.	Next	Met.	SeqNr
2	2	0	(2, 0)

Dest.	Next	Met.	SeqNr
3	3	0	(3, 0)

Table 2. Routing tables of nodes 1, 2 and 3 in **Fig. 2(a)**

Dest.	Next	Met.	SeqNr
1	1	0	(1, 0)

Dest.	Next	Met.	SeqNr
2	2	0	(2, 2)
3	3	1	(3, 0)

Dest.	Next	Met.	SeqNr
3	3	0	(3, 0)

Table 3. Routing tables of nodes 1, 2 and 3 in **Fig. 2(b)** and **(c)**

Dest.	Next	Met.	SeqNr
1	1	0	(1, 0)

Dest.	Next	Met.	SeqNr
2	2	0	(2, 2)
3	3	1	(3, 0)

Dest.	Next	Met.	SeqNr
2	2	1	(2, 2)
3	3	0	(3, 2)

Table 4. Routing tables of nodes 1, 2 and 3 in **Fig. 2(d)** and **(e)**

Dest.	Next	Met.	SeqNr
1	1	0	(1, 0)

Dest.	Next	Met.	SeqNr
1	1	1	(1, 0)
2	2	0	(2, 4)
3	3	1	(3, 0)

Dest.	Next	Met.	SeqNr
2	2	1	(2, 2)
3	3	0	(3, 2)

Table 5. Routing tables of nodes 1, 2 and 3 in **Fig. 2(f)**

Dest.	Next	Met.	SeqNr
1	1	0	(1, 2)
2	2	1	(2, 4)
3	2	2	(3, 0)

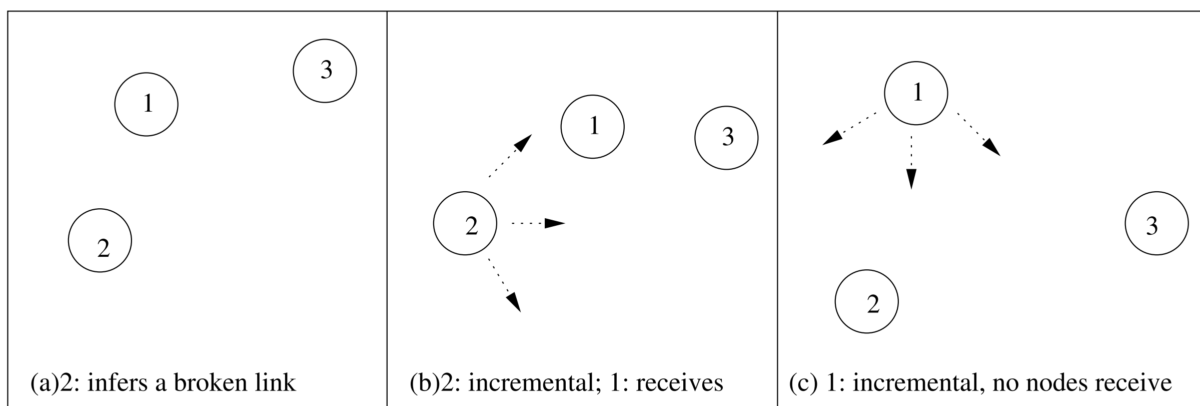
Dest.	Next	Met.	SeqNr
1	1	1	(1, 2)
2	2	0	(2, 4)
3	3	1	(3, 0)

Dest.	Next	Met.	SeqNr
2	2	1	(2, 2)
3	3	0	(3, 2)

Table 6. Routing tables of nodes 1 and 2 corresponding to **Fig. 3**

Dest.	Next	Met.	SeqNr
1	1	0	(1, 4)
2	2	1	(2, 6)
3	2	∞	(3, 1)

Dest.	Next	Met.	SeqNr
1	1	1	(1, 2)
2	2	0	(2, 6)
3	3	∞	(3, 1)



(a)2: infers a broken link
(b)2: incremental; 1: receives
(c) 1: incremental, no nodes receive

Fig. 3. An example of DSDV in operation (2) link breakage

to keep the figure succinct, the transmission ranges of the nodes are not given in Fig. 2. If a node receives the routing information sent by another node that means the receiver is a neighbour of the sender, i.e. it is in the range of the sender.

Fig. 2 shows possible route advertisements of 3 nodes running DSDV in a MANET from the initial state as time increases. These advertisements include a full dump of each node and consequential incremental updates. The initial routing tables of 3 nodes are shown in Table 1. (To save space, 1 is used for node 1, 2 for node 2 and 3 for node 3.) In the table, (1, 1, 0, (1, 0)) means that node 1 has a routing table comprising one route entry to itself, in which the destination and the next hop both are node 1, the number of hops is 0 and its initial sequence number is set to 0, represented as (1, 0).

We assume that node 3 is the first to broadcast its full dump (3, 0, (3, 0)) within the MANET. Only node 2 receives this information as indicated in Fig. 2(a). Node 2 updates its routing table based on this information. It adds the new entry for node 3 to its routing table, after increasing the metric of this entry by 1 and adding node 3 as the next hop of this entry. After that, it increases its own sequence number to 2. The updated routing tables of the nodes are given in Table 2. As shown in Fig. 2(b), node 2 immediately broadcasts an incremental update, ((2, 0, (2, 2)), (3, 1, (3, 0))). Node 3 receives this information and updates its routing table in a similar way, adding a new route entry for node 2 to its routing table, and increases its sequence number to 2, as shown in Table 3. In Fig. 2(c), node 3 broadcasts an incremental update, ((2, 1, (2, 2)), (3, 0, (3, 2))), but no other nodes receive it, because node 2 is out of the range of node 3. Now, node 1 originates its full dump broadcasting (1, 0, (1, 0)) which node 2 receives, as shown in Fig. 2(d). Node 2 updates its routing table as shown in Table 4. In Fig. 2(e), node 2 broadcasts an incremental update ((1, 1, (1, 0)), (2, 0, (2, 4))), but no others receive it. Now in Fig. 2(f), node 2 originates its full dump and transmits ((1, 1, (1, 0)), (2, 0, (2, 4)), (3, 1, (3, 0))), and node 1 receives this information. Node 1 keeps the entry to itself unchanged because of the identical sequence number 0. It adds the entries for node 2 and node 3 to its routing table respectively, and it increases its sequence number by 2. Then node 1 broadcasts an incremental update ((1, 1, (1, 2)), (2, 1, (2, 4)), (3, 2, (3, 0))) while node 2 receives it (to save space, also shown in Fig. 2(f)). Node 2 only updates the route entry for node 1 as (1, 1, (1, 2)), and keeps the other entries unchanged because there is no new information received, so it does not broadcast any incremental update. The modified routing tables are shown in Table 5.

An example that illustrates how a node deals with a broken link is given in Fig. 3. Assume the routing tables of nodes 1 and 2 are the same as those in Table 5. Consider node 2 finds that it has not received a broadcast from node 3 for a while (e.g. a periodic interval), as shown in Fig. 3(a). It infers the link between them is broken. Hence, node 2 assigns the metric of this link to ∞ and increases the sequence number by 1. Then it increases its own sequence number by 2 and immediately broadcasts an incremeantal update ((2, 0, (2, 6)), (3, ∞, (3, **1**))), while node 1 receives it in Fig. 3(b) (note: at this moment, the routing table of node 1 is the same as that in Table 5, while the routing table of node 2 is shown in Table 6). Node 1 updates the route entry for node 2 to the received sequence number 6. For the route entry for node 3, node 1 selects the one received since it has the higher sequence number. Because this is a broken link, node 1 increases its own sequence number by 2. The modified routing table of node 1 is shown in Table 6. Then node 1 immediately broadcasts an incremental update ((1, 0, (1, 4)), (2, 1, (2, 6)), (3, ∞, (3, **1**))) and no other nodes receive it, as shown in Fig. 3(c). So routing tables of node 1 and node 2 are the same as those in Table 6.

3 Abstract CPN Model of a MANET based on DSDV

3.1 Design Rationale

The intent of our CPN model is to show that CPNs can be used for the modelling of routing protocols in a MANET environment where arbitrary changes of network topology are possible. We therefore start by modelling the basic operation of the routing protocol: nodes discover other nodes by receiving broadcast messages and update their routing tables accordingly; and nodes discover that previously established links are no longer valid, and mark them as broken in their routing tables.

We do not model the routing messages explicitly. Instead we just consider events where the information from the message is received and processed. We assume that this cannot occur simultaneously in different nodes, i.e. no two nodes receive and process the broadcast at exactly the same time. Instead, these events are interleaved in the different nodes. Further, because of arbitrary movements of the nodes, there is no synchronisation between different nodes for a broadcast. Hence the broadcast by node A may be received by: no nodes (corresponding to A being an isolated node); one node (all other nodes are out of range); or any number of nodes. The interpretation is that all nodes that have updated their routing tables are in range at that time. Conversely, all nodes that have not updated their routing tables are not in range at that time. Moreover, a lost message has the same effect as being out of range. With this understanding, we model the updating of routing tables in the MANET non-deterministically. Thus updating a routing table is considered an arbitrary event.

Further, we model broken links in a similar way. Because a functional model abstracts from time, we consider that a broken link can be interleaved with any other event and is thus modelled by an arbitrary event. Hence it is possible for a node to receive a full dump from a neighbour and then to declare the link down as the next event. This could correspond to the node leaving the MANET and other nodes becoming dispersed so that they are all out of range.

We believe this captures the asymmetry between nodes and their arbitrary movement realistically at a high-level of abstraction. Because the functional model abstracts from probabilities, it includes many situations that would be considered rare events.

3.2 CPN Model

A CPN diagram of the MANET routing protocol, based on DSDV, is given in Fig. 4. The purpose of the CPN is to model how nodes update their routing tables and deal with broken links. Thus we model the nodes in the MANET by just their identity and their routing table. The nodes are stored in the only place in the CPN, Nodes. This place is typed by the colour set MNode as shown in line 13 in Fig. 5. MNode is a product of its identity, NodeId, which corresponds to its address, and its routing table, RT. The routing table is represented as a list of route entries, one for each destination.When ignoring the install time, the route entry is a 4-tuple comprising the destination, nexthop, metric and its sequence number.The destination and nexthop are node addresses, represented by the NodeId. We represent the NodeId as a positive integer up to the maximum number of nodes in the MANET (see line 4 in Fig. 5). The metric field is a little more interesting. It is normally the hopcount, the number of nodes that need to be traversed to reach the destination. This can be represented as a non-negative integer. However, when a link with a neighbour is considered to be down, the metric that is used is ∞. Thus the metric is a union of the hopcount and ∞ (represented as *infinity*) as seen in line 7 in Fig. 5. The sequence number can also be represented as a non-negative integer. In Fig. 5, SeqNr is a product set, including the identity of the destination originating the

sequence number, and the value of this sequence number. This corresponds to its description in DSDV.

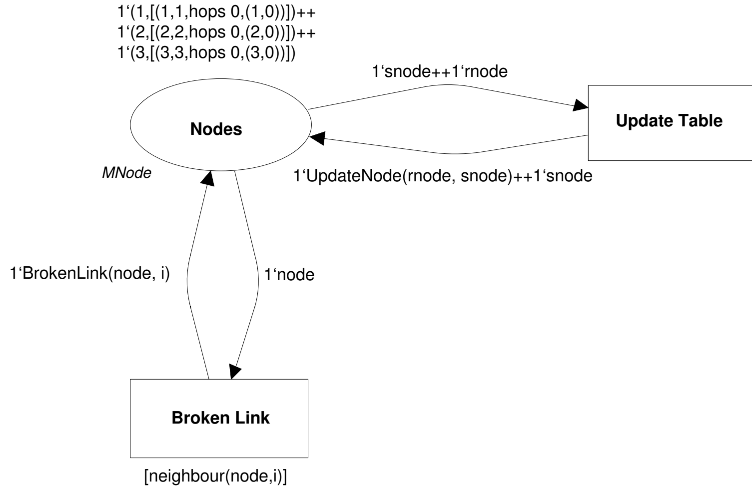


Fig. 4. CPN Model of a MANET

We consider a configuration of 3 nodes in the MANET for our initial experiments (see line 1 of Fig. 5). Nodes is thus initially marked by 3 nodes (1, 2 and 3), where each routing table just has a single entry to itself. This corresponds to **Table 1**.

In line 2 of Fig. 5, *UpdateSeq* is an ML reference variable [27]. It is used as a flag to indicate if a sequence number is to be updated by the functions described later in this section.

```
------------------------------------------------------------------
1  val MaxNodes = 3;
2  val UpdateSeqNr = ref false;
3
4  color NodeId = int with 1..MaxNodes;
5  color Destination = NodeId;
6  color Nexthop = NodeId;
7  color Metric = union hops:Hopcount + infinity declare of_hops;
8  color Hopcount = int;
9  color Number = int;
10 color SeqNr = product Destination * Number;
11 color RTEntry = product Destination * Nexthop * Metric * SeqNr;
12 color RT = list RTEntry;
13 color MNode = product NodeId * RT;
14
15 var i: NodeId;
16 var snode, rnode, node: MNode;
------------------------------------------------------------------
```

Fig. 5. Global declarations of the CPN model

Transition Update Table models the process of a node updating its routing table based on the information it receives from another node. The arc inscription from place Nodes to this transition has two variables, *snode* and *rnode*, which represent two arbitrary nodes in the MANET. In our model, *rnode* receives the routing information broadcast by *snode*. The arc inscription from Update Table to the place Nodes includes a function, UpdateNode(rnode, snode). This function returns the updated routing table of *rnode*, while *snode*'s routing table is maintained unchanged when Update Table occurs. This is where we abstract from the

DSDV protocol mechanisms. We interpret the occurrence of Update Table to mean that any node, *rnode*, can update its table, based on the information from another node, *snode*, at any time. Thus if Update Table occurs, *rnode* must have been within the transmission range of *snode*. If Update Table does not occur, then either the update was not sent by *snode* or when *snode* broadcasts the update *rnode* was out of its transmission range. Update Table conceptually implements both full dump and incremental updates, as the information that is in the incremental update is the only information that has any affect when updating the routing table. So although clearly the full information is available, because only the incremental information is used, the incremental update is modelled. This is the case because we do not model the messages explicitly. So at our level of abstraction the two mechanisms are not distinguishable. However, when a certain scenario is created by executing transitions, then it can be interpreted as incremental or full dump as is appropriate.

Transition Broken Link models a node detecting that it has not received an update from another node within the expected time, and updating its routing table accordingly. This can occur for any node and for any one of its neighbours. On the occurrence of Broken Link, a node (bound to the variable *node*) and one of its neighbours (i) are chosen arbitrarily. The guard neighbour(node,i) ensures that the destination in the route entry of *node* that corresponds to i does have a metric (hopcount) of 1 (which indicates that it is a neighbour). The function BrokenLink(node,i) updates *node*'s routing table accordingly.

After Broken Link occurs, DSDV requires that an incremental update is broadcast. This broadcast may or may not be received by the nodes in the MANET. If no node receives the broadcast, then conceptually this is modelled by Update Table not occurring with *snode* bound to the node that has just updated its table for the broken link. On the other hand if a node does receive the incremental broadcast, then this is modelled by Update Table occurring with that node binding to variable *rnode*, and the node that is broadcasting, binding to *snode*. The effect of the function UpdateNode is the same whether or not it is a full dump or an incremental update. Thus transition Update Table represents the behaviour of updating a node's routing table, irrespective of whether it receives a full dump or an incremental update.

3.3 Functions for Update Table

All functions needed for updating the routing table of a node are described in this subsection (see Listing 1.1 and Listing 1.2). The main function is UpdateNode(rnode, snode) shown in lines 1-9 of Listing 1.1. By this function, a node, *rnode*, updates its routing table (including its own sequence number) based on the routing information sent by another node, *snode*. This function contains a local declaration, *Let/in/end*. Using it, one value declaration binds a value returned by $UpdateSeqNr$, a ref-variable with initial value $false$ (see line 3), and the other one binds a value returned by the function, UpdateRT() (see lines 1-34 in Listing 1.2), to an identifier variable *updated*. In line 6, the value of $UpdateSeqNr$ is checked. If it is $true$, the sequence number of *rnode* is increased by 2 using function UpdateOwnRT() (see lines 17-20 in Listing 1.1). Then the updated routing table with the node's ID (obtained from function GetNodeId(), see lines 11-12) is returned (see line 7). Otherwise, *rnode*'s identity and the value contained in *updated* (in lines 4-5) is returned directly (see line 8). The function UpdateNode(rnode, snode) involves the whole procedure of updating the routing table of *rnode*, using two steps as follows.

1. **First Step:** the *rnode* updates its routing table based on the routing information broadcast by *snode*. This step is realised by function UpdateRT().
2. **Second Step:** the *rnode* determines whether or not to increase its own sequence number. This step is done by checking the value of $UpdateSeqNr$.

Listing 1.1. Function UpdateNode

```
1  (*--rnode updates its routing table based on information broadcast by snode--*)
2  fun  UpdateNode(rnode,snode)=
3  let  val _ = (UpdateSeqNr:=false)
4       val updated =  UpdateRT(GetNodeId(rnode),GetRTNode(rnode),
5                                GetNodeId(snode),GetRTNode(snode))
6  in   if(!UpdateSeqNr)
7       then (GetNodeId(rnode),UpdateOwnRT(updated))
8       else (GetNodeId(rnode),updated)
9  end;
10
11 (*--to get a node's identity--*)
12 fun GetNodeId(n,rt)= n;
13
14 (*--to get a node's routing table--*)
15 fun GetRTNode(n,rt)= rt;
16
17 (*--to update a node's own sequence number--*)
18 fun UpdateOwnRT(n,rte::rt)= if OwnRTEntry(n,rte)
19                             then (IncreaseSeqNr(rte))::rt
20                             else rte::UpdateOwnRT(n,rt);
21
22 (*--to find the route entry to the node itself--*)
23 fun OwnRTEntry(n,(des1,next1,metr1,seqnr1)) = (n = des1);
24
25 (*--to increase the value of sequence number by 2--*)
26 fun IncreaseSeqNr(des1,next1,metr1,(des2,num1))= (des1,next1,metr1,(des2,num1+2));
```

Now, we focus on the operations in the first step. Functions designed for this step are shown in Listing 1.2. In lines 36-41, function AddRouteEntry() realises adding new route entries. In line 44, the function hopnumbers() returns an integer value that is of type Hopcount from the union type Metric (see line 7 in Fig. 5). In line 47, function add1() increments the metric that is of type Hopcount by 1. As shown in lines 1-34, the main function UpdateRT() involves the whole procedure of updating the routing table of a node based on the information sent by another node. The node compares each route entry in this information with the corresponding one in its routing table. To give some insight into the operation of this function we illustrate the procedure for updating a route entry in Fig. 6. This is then applied recursively for each route entry (see lines 5-34 of Listing 1.2). For convenience, in our model, all route entries are listed in ascending order of the destination's node identifier (address), to simply the procedure. This has the added benefit reducing state space explosion that would otherwise occur due to arbitrary ordering of the list.

As shown in Fig. 6, the receiver is represented as $(mnid, [(md, mn, mm, (md1, mseq)) :: mrt])$, in which $mnid$ represents the node identity and $(md, mn, mm, (md1, mseq))$ represents the route entry (also represented as $entry1$), currently listed in the first position in its routing table. The rest of the route entries are represented by mrt. Similarly, the sender is represented as $(n, [(nd, nn, nm, (nd1, nseq)) :: rt])$. Its identity is n, and the route entry listed in the first position in its routing table is $(nd, nn, nm, (nd1, nseq))$ (also represented as $entry2$). The metrics of $entry1$ and $entry2$ are represented as mm and nm respectively. In Fig. 6, $mm{:}int$ or $nm{:}int$ means that the metric is of type Hopcount, i.e. an integer. $mm{:}infinity$

Listing 1.2. Function UpdateRT

```
1  (*--a node updates its route entries based on another node --*)
2  fun UpdateRT(mnid,[],n,[])= []
3  | UpdateRT(mnid,[],n,rte::rt)=(UpdateSeqNr:=true; AddRouteEntry(mnid,n,rte::rt))
4  | UpdateRT(mnid,mrte::mrt,n,[])=  mrte::mrt
5  | UpdateRT(mnid,(md,mn,mm,(md1,mseq))::mrt,n,(nd,nn,nm,(nd1,nseq))::rt)=
6  if      md=nd
7  then
8    if     (mseq>nseq)
9    then   (md,mn,mm,(md1,mseq))::UpdateRT(mnid,mrt,n,rt)
10   else
11     if   (mseq=nseq)
12     then case (of_hops'Metric(mm)) of
13   (true) =>if   (hopnumbers(mm) <= hopnumbers(nm))
14             then (md,mn,mm,(md1,mseq))::UpdateRT(mnid,mrt,n,rt)
15             else (UpdateSeqNr:=true;(nd,n,add1(nm),(nd1,nseq))::UpdateRT(mnid,mrt,n,rt))
16  |(false)=>(md,mn,mm,(md1,mseq))::UpdateRT(mnid,mrt,n,rt)
17     else case ((of_hops'Metric(mm)),(of_hops'Metric(nm))) of
18  ((true),(true))=>if  (mm = add1(nm))
19                   then (nd,n,add1(nm),(nd1,nseq))::UpdateRT(mnid,mrt,n,rt)
20                   else (UpdateSeqNr:=true;(nd,n,add1(nm),(nd1,nseq))
21                        ::UpdateRT(mnid,mrt,n,rt))
22  |((true),(false)) => (UpdateSeqNr:=true;(nd,n,nm,(nd1,nseq))::UpdateRT(mnid,mrt,n,rt))
23  |((false),(true)) => (UpdateSeqNr:=true;(nd,n,add1(nm),(nd1,nseq))
24                        ::UpdateRT(mnid,mrt,n,rt))
25  |((false),(false))=> (nd,n,nm,(nd1,nseq))::UpdateRT(mnid,mrt,n,rt)
26 else
27   if      md<nd
28   then   (md,mn,mm,(md1,mseq))::UpdateRT(mnid,mrt,n,(nd,nn,nm,(nd1,nseq))::rt)
29   else
30     if   of_hops'Metric(nm)
31     then (UpdateSeqNr:=true;(nd,n,add1(nm),(nd1,nseq))
32          ::UpdateRT(mnid,(md,mn,mm,(md1,mseq))::mrt,n,rt))
33     else (UpdateSeqNr:=true;(nd,n,nm,(nd1,nseq))
34          ::UpdateRT(mnid,(md,mn,mm,(md1,mseq))::mrt,n,rt));
35
36 (*--a node with identity m adds new route entries based on another node--*)
37 fun AddRouteEntry(m,n,[])=[]
38 | AddRouteEntry(m,n,(des1,nexthop1,metric1,seqnr1)::rtm)=
39 if  of_hops'Metric(metric1)
40 then (des1,n,add1(metric1),seqnr1)::AddRouteEntry(m,n,rtm)
41 else (des1,n,metric1,seqnr1)::AddRouteEntry(m,n,rtm);
42
43 (*--to get the value of the metric--*)
44 fun hopnumbers(hops metr) = metr;
45
46 (*--to increase the metric by 1--*)
47 fun add1(hops metr)= let val M = metr+1
48                      in  hops M
49                      end;
```

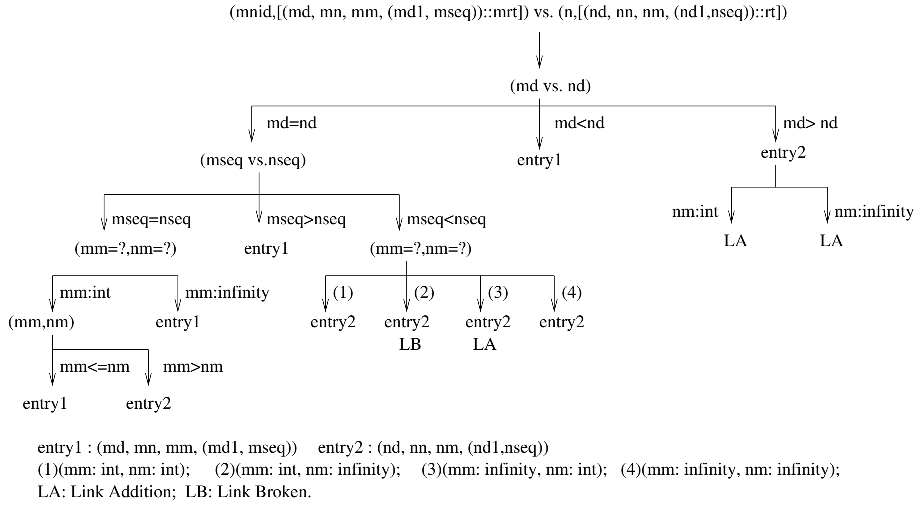


Fig. 6. Procedure for updating a route entry

or *nm*:*infinity* means that the metric is ∞ (represented as *infinity* in Fig. 6), namely the corresponding entry is a broken link. The purpose of *UpdateSeqNr* is to track changes to the routing table of the receiver during updating. The value of *UpdateSeqNr* is set to *true*, whenever an important change happens, such as a link addition (LA), a link breakage (LB) or metric change, as shown in Fig. 6. In Fig. 6, if *entry*1 occurs under an arrow, it means the receiver does not change its route entry. Otherwise, the route entry sent by the sender, *entry*2, is used to update the receiver's route entry. In this case, if the metric of *entry*2 is of type Hopcount, the receiver increases this metric by 1 by function add1(), and includes the sender's node identifier as the next hop of the entry. If the metric is ∞, the receiver just adds this entry without any change. Now, a comparison between two route entries, $(md, mn, mm, (md1, mseq))$ and $(nd, nn, nm, (nd1, nseq))$, is described as follows.

1. If two entries are to the same destination, $md = nd$ (see lines 6-25 in Listing 1.2). The receiver then compares their sequence numbers, $mseq$ and $nseq$. If $mseq > nseq$, the receiver will choose *entry*1. If $mseq = nseq$, that means the two metrics are of the same type. Thus both entries are available or both are broken. In the first case (corresponding to lines 13-15 in Listing 1.2), the receiver will select the route entry with the shorter metric. If *entry*2 is chosen, the value of *UpdateSeqNr* is set to *true* because the metric has changed. If they have the same metric, the receiver arbitrarily chooses one. In our model, *entry*1 is chosen for convenience. In the second case, the receiver can arbitrarily choose, and *entry*1 is chosen for convenience. Otherwise, in the case $mseq < nseq$, the receiver always selects *entry*2 instead of *entry*1. While according to the metrics of two entries, there also are four possible conditions (corresponding to lines 22-26 in Listing 1.2): in (1), *entry*2 is of type Hopcount, so the receiver deals with this available entry. If the metric changes, the value of *UpdateSeqNr* is set to *true*. In (2), *entry*1 is an available entry but *entry*2 is a broken one. So it is a LB. In (3), *entry*1 is a broken entry but *entry*2 is an available one. So it is a LA. In (4), *entry*2 is a broken link, so the receiver adds it unchanged.
2. If two entries are to different destinations, and $md < nd$ (see lines 27-28 in Listing 1.2). That means there is no route entry to destination md in the routing information broadcast by the sender. So the receiver keeps *entry*1 unchanged.
3. Otherwise, if $md > nd$ (see lines 29-34 in Listing 1.2), which means that the route entry to destination nd (i.e. *entry*2) is a new route entry for the receiver. Both conditions can be taken as LAs, even if *entry*2 in the second condition is a broken link.

The procedure described above is used for the next route entries in both routing tables. This procedure is repeated until at least all route entries in one of the routing tables have been processed. In this case, if some route entries listed in the receiver's routing table have not been compared, the receiver will keep them unchanged (see line 4 in Listing 1.2). In line 3 in Listing 1.2, some route entries listed in the sender's routing table are not in the receiver's routing table. In this case, the receiver will add them (i.e. LAs) using function AddRouteEntry() (see lines 36-41 in Listing 1.2). If all route entries of both routing tables have been processed, the computation stops and an empty list is returned (see line 2 in Listing 1.2).

3.4 Functions for Broken Link

The functions associated with the transition Broken Link are given in Listing 1.3. Lines 1-6 define the function neighbour(node,i), the guard of transition Broken Link in Fig. 4. It ensures that a node, *node*, detects that the link to one neighbour i is down. Function check() (see lines 14-19 in Listing 1.3) describes the metric of each broken link as ∞ (represented as $infinity$ in functions), and increases the sequence number of such a link by 1 when its next hop is i. Then, the updated routing table of this node is returned, and this value is bound to an identifier *Broken* in line 10 of the main function BrokenLink(node, i) (see lines 8-12 of Listing 1.3). In line 11, the node increases its sequence number by function UpdateOwnRT(), and then a node with its IP address and updated routing table is returned.

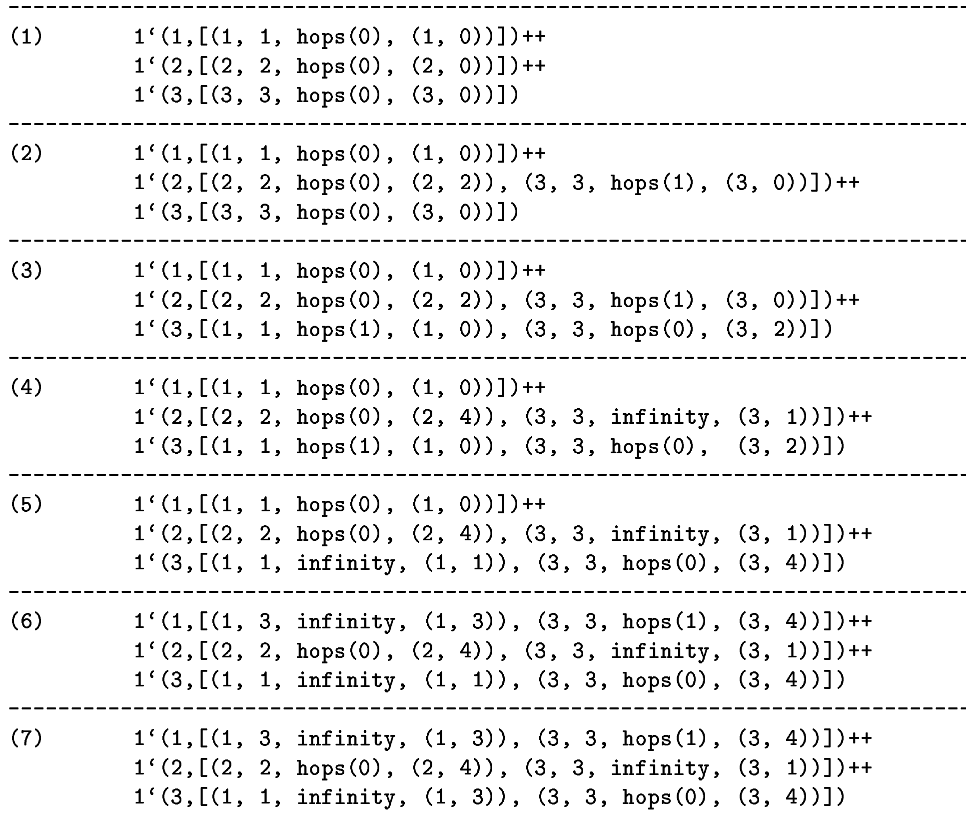
Listing 1.3. Function BrokenLink

```
1  (*--to ensure a neighbour with IP address i --*)
2  fun neighbour((n,[]),i)= false
3  | neighbour((n,(des1,next1,metr1,(des2,num1))::rt),i)=
4  if (i=des1) andalso (of_hops'Metric(metr1)) andalso (hopnumbers(metr1)=1)
5  then true
6  else neighbour((n,rt),i);
7
8  (*-- a node deals with broken links and updates its sequence number--*)
9  fun BrokenLink(node,i)=
10 let val Broken = check(i,GetNodeId(node),GetRTNode(node))
11 in (GetNodeId(node),UpdateOwnRT(GetNodeId(node),Broken))
12 end;
13
14 (*--to deal with broken links--*)
15 fun check(m,n,[])=[]
16 | check(m,n,(des1,next1,metr1,(des2,num1))::rt)=
17 if (m=next1)
18 then (des1,next1,infinity,(des2,num1+1))::check(m,n,rt)
19 else (des1,next1,metr1,(des2,num1))::check(m,n,rt);
```

4 Simulation of the CPN Model

In order to provide some insight into the operation of our CPN model, we consider the following simulation of its behaviour. The number of nodes in the MANET is governed by the initial marking of the CPN model in Fig. 4 (i.e. the marking of place Nodes), and can

be extended easily. Here, we just consider the operation of the CPN with 3 nodes. Suppose their addresses are represented as 1, 2 and 3 respectively. As shown in Fig. 4, there are two transitions, Update Table and Broken Link, in the CPN model. In order to explain the simulation of the CPN explicitly, we describe the occurrence of transition Update Table: $(UpdateTable, < rnode = a, snode = b >)$, in which $rnode$ and $snode$ are variables in the arc expression of the input arc to this transition, and a and b are values bound to $rnode$ and $snode$ respectively. As described in section 3.2, $rnode$ represents a node that can receive and update its routing table based on the routing information sent by another node, $snode$. For example, $(UpdateTable, < rnode = 1, snode = 3 >)$ means that when transition Update Table occurs, node 1 is bound to $rnode$ and node 3 is bound to $snode$. Thus, node 1 updates its routing table based on the routing information it received from node 3. Analogously, the occurrence of transition Broken Link is depicted as: $(BrokenLink, < node = c, i = d >)$, in which $node$ is the variable in the arc expression of the input arc to this transition, and c is a value assigned to $node$. Here, $node$ represents a node which detects and deals with the broken links to a neighbour i, which is ensured by the guard neighbour(node,i). For example, $(BrokenLink, < node = 1, i = 2 >)$ means that when transition Broken Link occurs, node 1 is bound to $node$ and 2 is bound to i. So node 1 detects that it has not received an update from a former neighbour, node 2, within the expected time, so it makes all route entries through node 2 as broken links. Now, the steps that occur in a execution of the CPN model are depicted as follows, and the markings reached are given in Fig. 7.



```
----------------------------------------------------------------------------
(1)     1`(1,[(1, 1, hops(0), (1, 0))])++
        1`(2,[(2, 2, hops(0), (2, 0))])++
        1`(3,[(3, 3, hops(0), (3, 0))])
----------------------------------------------------------------------------
(2)     1`(1,[(1, 1, hops(0), (1, 0))])++
        1`(2,[(2, 2, hops(0), (2, 2)), (3, 3, hops(1), (3, 0))])++
        1`(3,[(3, 3, hops(0), (3, 0))])
----------------------------------------------------------------------------
(3)     1`(1,[(1, 1, hops(0), (1, 0))])++
        1`(2,[(2, 2, hops(0), (2, 2)), (3, 3, hops(1), (3, 0))])++
        1`(3,[(1, 1, hops(1), (1, 0)), (3, 3, hops(0), (3, 2))])
----------------------------------------------------------------------------
(4)     1`(1,[(1, 1, hops(0), (1, 0))])++
        1`(2,[(2, 2, hops(0), (2, 4)), (3, 3, infinity, (3, 1))])++
        1`(3,[(1, 1, hops(1), (1, 0)), (3, 3, hops(0),  (3, 2))])
----------------------------------------------------------------------------
(5)     1`(1,[(1, 1, hops(0), (1, 0))])++
        1`(2,[(2, 2, hops(0), (2, 4)), (3, 3, infinity, (3, 1))])++
        1`(3,[(1, 1, infinity, (1, 1)), (3, 3, hops(0), (3, 4))])
----------------------------------------------------------------------------
(6)     1`(1,[(1, 3, infinity, (1, 3)), (3, 3, hops(1), (3, 4))])++
        1`(2,[(2, 2, hops(0), (2, 4)), (3, 3, infinity, (3, 1))])++
        1`(3,[(1, 1, infinity, (1, 1)), (3, 3, hops(0), (3, 4))])
----------------------------------------------------------------------------
(7)     1`(1,[(1, 3, infinity, (1, 3)), (3, 3, hops(1), (3, 4))])++
        1`(2,[(2, 2, hops(0), (2, 4)), (3, 3, infinity, (3, 1))])++
        1`(3,[(1, 1, infinity, (1, 3)), (3, 3, hops(0), (3, 4))])
----------------------------------------------------------------------------
```

Fig. 7. Markings of the CPN during the simulation

The initial marking is given in Fig. 7(1). For example, $(1, [(1, 1, hops(0), (1, 0))])$ is a pair comprising the identity and routing table of node 1 (see line 13 in Fig. 5). In its routing table,

there is only one route entry $(1, 1, hops(0), (1, 0))$. According to line 11 in Fig. 5, we know the destination and the next hop both are node 1, the metric is 0, which is of type Hopcount (see line 7 in Fig. 5), and $(1, 0)$ means the sequence number is originated by node 1 and its value is 0.

$(UpdateTable, < rnode = 2, snode = 3 >)$ occurs, which means node 2 receives the routing information sent by node 3, $[(3, hops(0), (3, 0))]$ (see section 2). Node 2 adds this route entry to its routing table, after increasing the metric by 1 and adding node 3 as the next hop of this entry. Because it is a link addition (see Fig. 6), node 2 increments its sequence number by 2. The marking reached is presented in Fig. 7(2).

After that, $(UpdateTable, < rnode = 3, snode = 1 >)$ occurs, which means node 3 receives the routing information, $[(1, 1, hops(0), (1, 0))]$ sent by node 1. Node 3 adds a new entry to node 1 into its routing table and increases its own sequence number by 2. The marking reached is shown in Fig. 7(3).

Then $(BrokenLink, < node = 2, i = 3 >)$ occurs. That means node 2 does not receive any information from a neighbour, node 3, for a while, so it infers the link to node 3 is broken. Node 2 assigns $infinity$ as the metric of this route entry and increases the sequence number of this entry by 1. Because it is a link breakage, node 2 increases its own sequence number by 2. The marking after this step is given in Fig. 7(4).

Next, $(BrokenLink, < node = 3, i = 1 >)$ occurs. Similarly, node 3 deals with the broken link to node 1 and increases its own sequence number as well. The marking is shown in Fig. 7(5).

$(UpdateTable, < rnode = 1, snode = 3 >)$ occurs. Node 1 receives the routing information, $[(1, 1, infinity, (1, 1)), (3, 3, hops(0), (3, 4))]$ sent by node 3. For two route entries to the same destination, a node always selects the one with the higher sequence number. In this case, there are two route entries to node 1, one containing the routing information sent by node 3, and the other kept in the routing table of node 1. Node 1 selects the former one because it has the higher sequence number, 1. Because a broken link, $(1, 1, infinity, (1, 1))$, replaces the available one, $(1, 1, hops(0), (1, 0))$, it can be taken as link removal. Node 1 adds the route entry to node 3 into its routing table, i.e. a link addition. Then node 1 increases its own sequence number from 1 to 3. The marking reached is shown in Fig. 7(6).

$(UpdateTable, < rnode = 3, snode = 1 >)$ now occurs. Node 3 receives the routing information, $[(1, 3, infinity, (1, 3)), (3, 3, hops(1), (3, 4))]$ sent by node 1. For two route entries to node 1, node 3 selects the one received, because it has higher sequence number 3. Node 3 keeps the route entry to itself in its routing table unchanged, because the two entries to node 3 have the same sequence number 4. The marking reached is shown in Fig. 7(7).

5 Discussion

The execution sequence described above shows that the CPN model can simulate the updating of routing tables in an environment where the topology of the MANET can change dramatically. In this section we illustrate how this execution sequence can be related to events in the MANET, and also analyse the trace by considering the validity of the routing table entries shown in the markings of Fig. 7.

A scenario in the MANET that relates to the execution sequence in the previous section is shown in Fig. 8. The dashed circles include the node that is broadcasting and which nodes receive the broadcast. The solid arrows indicate that the sender broadcasts a *full dump*, and the dashed arrows indicate that the sender broadcasts an *incremental* update. We assume that nodes broadcast their *full dump* in the order: node 3, node 1 and node 2. Initially, the nodes just have their own entries as given by the marking of Fig. 7(1).

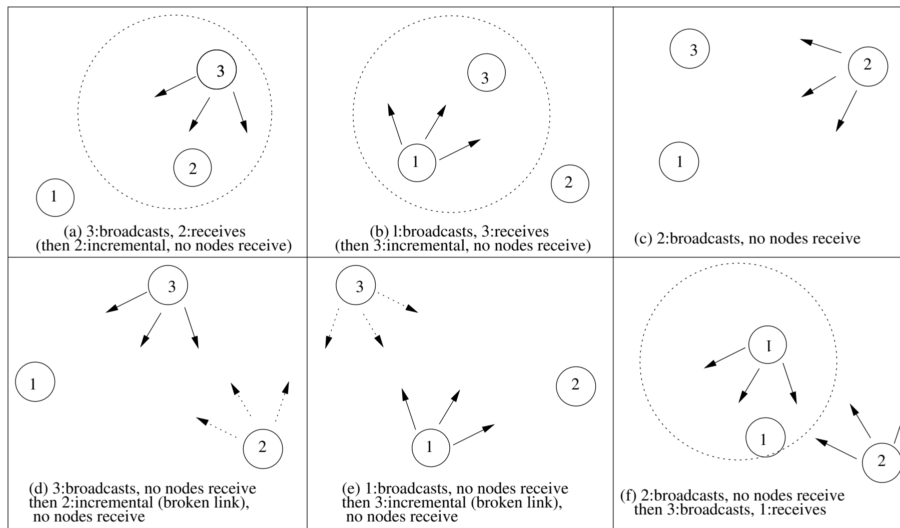


Fig. 8. An illustration of the simulation in section 4

As shown in Fig. 8**(a)**, node 3 broadcasts and node 2 receives and updates its routing table. The updated routing tables are in Fig. 7(2). Node 2 then broadcasts an incremental update, but unfortunately no nodes receive it as they are out of range. This is not shown in Fig. 8. Sometime later (see Fig. 8**(b)**), node 1 broadcasts and node 3 receives and updates its routing table. The updated routing tables of the nodes now correspond to Fig. 7(3). Node 3 now broadcasts an incremental update but it is not received (not shown in the Fig.). Later, as shown in Fig. 8**(c)**, node 2 broadcasts but unfortunately, the other nodes are still too far away to receive it. Similarly, in Fig. 8**(d)**, node 3 broadcasts, but no other nodes receive it. Node 2 has not received any information from node 3 for too long, so it declares this link broken, and immediately broadcasts this route change but no nodes receive it. The updated routing tables at this time are shown in Fig. 7(4). In Fig. 8**(e)**, node 1 broadcasts to no effect. Node 3 judges that the link to node 1 is broken, and immediately broadcasts this broken link in the MANET, but no others receive it. The updated routing tables of the nodes at this stage are shown in Fig. 7(5). Now node 2 broadcasts again, but to no avail (Fig. 8**(f)**). Then node 3 broadcasts and node 1 receives and updates its routing table. The updated routing tables of the nodes now correspond to the marking in Fig. 7(6).

We find that node 1 has not communicated with the others for a while since Fig. 8**(b)**, so its sequence number is not updated. Then in Fig. 8**(e)** node 3 considers that its link to node 1 is broken, so it increases the sequence number of this link to 1. Therefore, there are two route entries to node 1, one in routing table of node 1 being $(1, 1, hops(0), (1, 0))$, and the other in routing table of node 3, $(1, 1, infinity, (1, 1))$. So node 3 has a higher sequence number for node 1 than that of node 1 itself. Thus node 1 updates its own routing entry with an incorrect routing entry from node 3.

The fact that a higher sequence number for a node can occur in a node other than the node itself leads to incorrect routing entries being created in DSDV. We classify these errors caused by this fact into two categories as follows.

1. **A node updates the route entry to itself based on information from another node.** Intuitively, the metric of the route entry to a node itself is 0 and the next hop is the node itself. However, in Fig. 7(6), we find a route entry of $(1, 3, infinity, (1, 3))$ in the routing table of node 1. That means the link from node 1 to itself is broken and should be via another node, node 3. This is obviously wrong.

Listing 1.4. Modified function UpdateRT

```
1 (*--a node updates its route entries based on another node --*)
2 fun UpdateRT(mnid,[],n,[])= []
3 |   UpdateRT(mnid,[],n,rte::rt)=(UpdateSeqNr:=true; AddRouteEntry(mnid,n,rte::rt))
4 |   UpdateRT(mnid,mrte::mrt,n,[])=  mrte::mrt
5 |   UpdateRT(mnid,(md,mn,mm,(md1,mseq))::mrt,n,(nd,nn,nm,(nd1,nseq))::rt)=
6 if  md=nd
7 then
8   if md=mnid
9   then (md,mn,mm,(md1,mseq))::UpdateRT(mnid,mrt,n,rt)
10  else
11     if mseq>nseq
12     then
13        if (md=n) andalso (not(of_hops'Metric(mm))) andalso ((of_hops'Metric(nm))
14        then (UpdateSeqNr:=true;(nd,n,add1(nm),(nd1,nseq))::UpdateRT(mnid,mrt,n,rt))
15        else (md,mn,mm,(md1,mseq))::UpdateRT(mnid,mrt,n,rt)
16  else
17    if (mseq=nseq)
18    ...
```

2. **A node cannot update the broken link to another node even on receiving a broadcast from this node.** Now consider that node 3 receives a broadcast from node 1. Node 3 should update the broken link to node 1 based on the route entry sent by node 1, increase its sequence number, and immediately broadcast this routing change. Unfortunately, node 3 does not update this broken link at all because of the value of the sequence number. The route entry sent by node 1 has the sequence number $(1, 0)$, whereas the broken link to node 1 contained in the routing table of node 3 has a higher sequence number $(1, 1)$.

According to DSDV, more recent route entries should have higher sequence numbers in the routing table. Hence, nodes can distinguish between current and obsolete route entries by comparing the values of their sequence numbers. For this mechanism to be correct, each node should keep its own sequence number as the most current one. Unfortunately, DSDV does not guarantee this point in the general environment of a MANET.

In order to avoid these errors, we modify DSDV by changing the way routing tables are updated. A correction is made to function UpdateRT() (described in subsection 3.3), and thus there is no need to alter the structure of the CPN model given in Fig. 4. The modified function is presented in Listing 1.4 corresponding to lines 1-11 in Listing 1.2.

1. To avoid the first kind of error: a node keeps the route entry to itself unchanged when updating its routing table, as shown in lines 8-9 in Listing 1.4. It only updates its sequence number attached in this route entry when needed in function UpdateNode() (see line 7 of Listing 1.1).
2. To avoid the second kind of error: a node receives routing information from another node, if it already has a route entry to this sender, it will update this route entry regardless of the value of the sequence number, because the route information it just receives is more current. This is implemented in lines 13-15 of Listing 1.4: the receiver receives an available route entry to the sender and updates the broken one to this sender with the new one, even if the new one has a lower sequence number.

The rest of the function is identical to that in Listing 1.2, so it is not included in Listing 1.4. After simulating the modified CPN model, we find that both kinds of errors are eliminated. For example, given the routing tables of the nodes in Fig. 7(5), if $(UpdateTable, < rnode = 1, snode = 3 >)$ occurs (see section 4), node 1 updates its routing table according to the information broadcast by node 3. It keeps the route entry to itself unchanged and adds the route entry to node 3 as a new one, and updates its sequence number, so its updated routing table is: $[(1, 1, hops(0), (1, 2)), (3, 3, hops(1), (3, 4))]$. Whereas, if $(UpdateTable, < rnode = 3, snode = 1 >)$ occurs, node 3 updates its routing table according to the information broadcast by node 1. Regardless of the value of the sequence number, it updates the broken link to node 1, and updates its sequence number, so its updated routing table is: $[(1, 1, hops(1), (1, 0)), (3, 3, hops(1), (3, 4))]$.

6 Conclusions and Future Work

This paper demonstrates the feasibility of using CPNs to faithfully model routing protocols of MANETs given their dynamically changing network topologies. We present the first abstract CPN model for a MANET based on DSDV. Although the model looks deceptively simple, it not only allows for arbitrary changes in topology but also relaxes the assumption that nodes have the same transmission ranges. This allows us to model MANETs in which the nodes are heterogeneous (e.g. a combination of PDAs, notebooks and mobile phones). Our results show that the CPN model captures the highly dynamic topology of such a network, something that has been considered a difficult problem by others [29]. Further, although the model is abstract, it has sufficient detail for us to find errors in the DSDV procedures using simulation.

Two categories of errors have been found. The first is that it is possible for a node to wrongly update its own route entry, replacing its metric of a hopcount of zero (which must always be the case) with *infinity* and directing packets destined for itself to another node. This is a serious error. The second error is that it is possible for a node with a broken link entry for another node to not re-establish the link with that node, even though it has received a broadcast from that node. This is due to incorrect handling of sequence numbers. This is the first time these errors have been discovered in DSDV as far as we are aware. We also suggest modifications to the routing table updating procedures to eliminate these errors. Our simulations have confirmed their effectiveness.

In this paper, we have not attempted to analyse our model using state spaces because the state space is infinite due to the sequence number being unbounded. This faithfully reflects the DSDV specification. However, unbounded sequence numbers are impractical, so we plan to investigate state space analysis of this model when the sequence number space is limited. Further, we would like to enhance the model in two ways by including: a) the install time parameter; and b) nodes powering down and rejoining the MANET. More generally, it may be possible to use the model as a platform to study proactive routing protocols and distance-vector routing algorithms [20] used in MANETs.

Acknowledgements

The authors would like to acknowledge the assistance of their colleagues, Guy Gallasch, Somsak Vanit-Anunchai and Lin Liu, for their assistance with some of the detailed technical parts of the model. We also gratefully acknowledge the constructive comments of the reviewers on this paper. Cong Yuan is supported by an Australian Government International Postgraduate Scholarship.

References

1. S. Basagni, M. Conti, S. Giordano, and I. Stojmenovic. Mobile Ad Hoc Networking. IEEE Press, New York, 2004.
2. E. Belding-Royer. Routing Approaches in Mobile Ad Hoc Networks. In S. Basagni, M. Conti, S. Giordano, and I. Stojmenovic, editors, *Mobile Ad Hoc Networking*. IEEE Press, New York, 2004.
3. R. E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
4. K. Bhargavan, D. Obradovic, and C. A. Gunter. Formal Verification of Standards for Distance Vector Routing Protocols. *Journal of the ACM (JACM)*, 49(4):538–576, July 2002.
5. J. Billington. Many-sorted High-Level Nets. In *the Third International Workshop on Petri Nets and Performance Models*, pages 11–13, Kyoto, Japan, December 1989. IEEE CS Press, Washington, D. C., USA, 1989.
6. C. Cheng, R. Riley, S.P.R Kumar, and J.J Garcia-Luna-Aceves. A Loop-Free Bellman-Ford Routing Protocol without Bouncing Effect. In *ACM SIGCOMM 1989*, pages 224–237, September 1989.
7. G. Findlow and J. Billington. High-Level Nets for Dynamic Dining Philosophers Systems. In M.Z. Kwiatkowska, M.W. Shields, and R.M. Thomas, editors, *Semantics for Concurrency*, pages 185–222. Springer-Verlag, 1990.
8. L. R. Jr. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.
9. J.J Garcia-Luna-Aceves. A Unified Approach to Loop-Free Routing Using Distance Vectors or Link States. In *ACM SIGCOMM 1989*, pages 212–223, September 1989.
10. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
11. G. J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.
12. J. M. Jaffe and F. Moss. A Responsive Distributed Routing Algorithm for Computer Networks. In *IEEE Transaction on Communications COM-30(7)*, pages 1758–1762. July 1982.
13. K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1–3. Springer-Verlag, 1997.
14. L. M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.
15. L. M. Kristensen and K. Jensen. Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad Hoc Networking. In *INT 2004, LNCS 3147*, pages 248–269. Springer-Verlag, 2004.
16. L. M. Kristensen, J. B. Jørgensen, and K. Jensen. Application of Coloured Petri Nets in System Development. In *Lectures on Concurrency and Petri nets - Advanced in Petri Nets.Proc.of 4th Advanced Course on Petri Nets.Vol.3098 of Lecture Notes in Computer Science*, pages 626–685. Springer-Verlag, 2004.
17. K. G. Larsen, P. Pettersson, and Y. Wang. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, October 1997.
18. Jennifer J. N. Liu and I. Chlamtac. Mobile Ad Hoc Networking with a View of 4G Wireless: Imperatives and Challenges. In S. Basagni, M. Conti, S. Giordano, and I. Stojmenovic, editors, *Mobile Ad Hoc Networking*, pages 1–45. IEEE Press, New York, 2004.
19. G. Malkin. RIP Version 2- Carrying Additional Information. RFC 1723(draft standard). Technical report, Internet Engineering Task Force, November 1994.
20. G. S. Malkin and M. E. Steenstrup. Distance-Vector Routing. In *Routing in Communications Networks*, pages 83–98. Prentice-Hall, 1995.
21. P. M. Merlin and A. Segall. A Failsafe Distributed Routing Protocol. In *IEEE Transactions on Communications COM-27(9)*, pages 1280–1287. September 1979.
22. D. Obradovic. *Formal Analysis of Routing Protocols*. PhD thesis, University of Pennsylvania, 2002.
23. C. E. Perkins, E. Belding-Royer, and S. Das. *Ad-hoc On-Demand Distance Vector (AODV) Routing*. IETF Internet draft, draft-ietf-manet-aodv-09.txt, 2001.
24. C. E. Perkins and P. Bhagwat. Highly Dynamic Destination Sequenced Distance Vector (DSDV) for Mobile Computers. In *ACM SIGCOMM 1994 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, August 1994.
25. C. E. Perkins and P. Bhagwat. *Ad Hoc Networking, Chapter 3: DSDV Routing over a Multihop Wireless Network of Mobile Computers*. Addison-Wesley, 2001.
26. A. S. Tanenbaum. *Computer Networks*. Prentice-Hall, Upper Saddle River, NJ, 4th edition, 2003.
27. J. D. Ullman. *Elements of ML Programming*. Prentice-Hall, 1998.
28. O. Wibling, J. Parrow, and A. Pears. Automatized Verification of Ad Hoc Routing Protocols. In *FORTE 2004, LNCS 3235*, pages 343–358. Springer-Verlag, 2004.
29. C. Xiong, T. Murata, and J. Tsai. Modeling and Simulation of Routing Protocol for Mobile Ad Hoc Wireless Networks Using Colored Petri Nets. In *Proc. Workshop on Formal Methods Applied to Defence Systems in Formal Methods in Software Engineering and Defence Systems, Conferences in Research and Practice in Information Technology*, volume 12, pages 145–153, 2002.

Modeling Work Distribution Mechanisms Using Colored Petri Nets

M. Pesic and W.M.P. van der Aalst

Department of Technology Management, Eindhoven University of Technology, P.O.Box 513, NL-5600 MB, Eindhoven, The Netherlands.
m.pesic@tm.tue.nl, w.m.p.v.d.aalst@tm.tue.nl

Abstract. Work ow management systems support business processes and are driven by their models. These models cover different perspectives including the control- ow, resource, and data perspectives. This paper focuses on the *resource perspective*, i.e., the way the system distributes work based on the structure of the organization and capabilities/quali cations of people. Contemporary work ow management systems offer a wide variety of mechanisms to support the resource perspective. Because the resource perspective is essential for the applicability of such systems, it is important to better understand the mechanisms and their interactions. Our goal is not to evaluate and compare *what* different systems do, but to understand *how* they do it. We use *Colored Petri Nets* (CPNs) to model work distribution mechanisms. First, we provide a basic model that can be seen as the "greatest common denominator" of existing work ow management systems. This model is then extended for three speci c systems (Staffware, FileNet, and FLOWer). Moreover, we show how more advanced work distribution mechanisms, referred to as *resource patterns*, can be modelled and analyzed.

Key words: Work distribution, work ow management, business process management, resource patterns, colored Petri nets.

1 Introduction

Work ow management systems are process-aware information systems [5, 20], which are used in companies as a means for the computerized structuring and driving of complex business processes. Work ow management systems implement business process models, and use them for driving the ow of work by allocating the right employees to the right tasks at the right times. The system *manages the work of employees*. It will determine which tasks an employee has to execute and when, which documents will be used, which information will be available during work, etc. Typically, a work ow management system o ers several mechanisms to distribute work. Nevertheless, we believe that existing systems are too limited in this respect. The goal of this paper is not to propose advanced work distribution mechanisms. Instead, we focus on the analysis of functionality in existing systems. The goal is not to evaluate these systems, but to understand *how* they o er speci c functionality. Since work distribution de nes the quality of work, it is important to consider research from the eld of social sciences, e.g., social-technical design [13, 17, 21, 55]. We believe that only by combining both *technical* and *social* approaches, one can truly grasp certain phenomena. A *deeper understanding* of particular aspects of work distribution is essential for developing a new breed of more user-centric systems.

The work reported in this paper can be seen as an extension of the *workflow patterns initiative* [6] (cf. www.work owpatterns.com). Within the context of this initiative 43 resource patterns [51, 49] have been de ned. Using a patterns approach, work distribution is evaluated from the perspective of the end-user as a dynamic property of work ow management systems. The work reported in this paper adds to a better understanding of these mechanisms by providing explicit process models for these patterns, i.e., the descriptive models are augmented with executable models. Most work reported in literature (cf. Section 4) uses static models to describe work distribution. Consider for example the meta modeling approaches presented in [8, 40–42, 48]. These approaches use

static models (e.g., UML class diagrams) to discuss work distribution concepts. This paper takes a truly dynamic model – a *Colored Petri Net* model – as a starting point, thus clearly differentiating our contribution from existing work reported in literature.

Colored Petri Nets (CPNs) [31, 34] are a natural extension of the classical Petri net [46].There are several reasons for selecting CPNs as the language for modeling work distribution in the context of workflow management. First of all, CPNs have formal semantics and allow for different types of analysis, e.g., state-space analysis and invariants [32]. Second, CPNs are executable and allow for rapid prototyping, gaming, and simulation. Third, CPNs are graphical and their notation is similar to existing workflow languages. Finally, the CPN language is supported by CPN Tools[1] – a graphical environment to model, enact and analyze CPNs. In the remainder, we assume that the reader is familiar with the CPN language and refer to [31, 34] for more details.

In this paper, we provide a basic CPN model that can be seen as the "greatest common denominator" of existing workflow management systems. The model will incorporate concepts of a task, case, user, work item, role and group. This model should be seen as a *starting point* towards a more *comprehensive reference model for work distribution*. The basic CPN model is extended and specialized for three specific systems: Staffware [54], FileNet [24], and FLOWer [43]. These three models are used to investigate differences between and similarities among different work distribution mechanisms in order to gain a deeper understanding of these mechanisms. In addition, advanced resource patterns that are not supported by these three systems are modeled by extending the basic CPN model.

The remainder of this paper is organized as follows. Section 2 presents the basic CPN model which should be considered a the "greatest common denominator" of existing workflow management systems. Section 3 extends this model in two directions: (1) Section 3.1 specializes the model for three different systems (i.e., Staffware, FileNet, and FLOWer), and (2) Section 3.2 extends the basic model for selected resource patterns. An overview of related work is given in Section 4. Section 5 discusses our findings and, finally, Section 6 concludes the paper.

2 Basic Model

Different workflow management systems tend to use different work distribution concepts and completely different terminologies. This makes it difficult to compare these systems. Therefore, we will not start by developing CPN models for different systems and see how these can be unified, but, instead, start with modelling the "greatest common denominator" of existing systems. This model can assist in comparing systems and unifying concepts and terminology. We will use the term *Basic Model* to refer to this "greatest common denominator" and represent it in terms of a CPN model.

The Basic Model represents a workflow management system where the business *process* is defined as a set of *tasks*. Before the process can be initiated and executed, it has to be instantiated. One (executable) instance of a process is referred to as a *case*. Each case traverses the process. If a task is enabled for a specific case, a *work item*, i.e., a concrete piece of work, is created. There is a set of *users* that can execute work items. The users are embedded in the organizational structure on the basis of their *roles*, and the *groups* they belong to. Group is an organizational unit (e.g., sales, purchasing, production, etc.), while role represents a capability of the user (e.g., manager, software developer, accountant, etc.). These concepts are mapped onto CPN types as shown in Table 1.

During the work distribution work items change state. The change of state depends on previous and determines the next actions of users and the distribution mechanism. A model of a life cycle of a work item shows how a work item changes states during the work distribution. For more detailed models about life cycle models we refer the reader to literature, e.g., [5, 18, 20, 30, 37, 41]. We develop and use the life cycle models as an aid to describe work distribution mechanisms. The Basic Model uses a simple model of the life cycle of work items and it covers only the general, rather simplified, behavior of workflow management systems (e.g., errors and aborts are not considered).

[1] CPN Tools can be downloaded from wiki.daimi.au.dk/cpntools/.

Figure 1 shows the life cycle of a work item of the Basic Model. After the *new* work item has arrived, it is automatically also *enabled* and then taken into distribution (i.e., state *initiated*). Next, the work item is *o ered* to the user(s). Once a user *selects* the work item, it is *assigned* to him/her, and (s)he can *start* executing it. After the *execution*, the work item is considered to be *completed*, and the user can begin working on the next work item.

Table 1. Basic Work ow Concepts

color Task = string;
color Case = int;
color WI = product Case * Task;
color User = string;
color Role = string;
color Group = string;

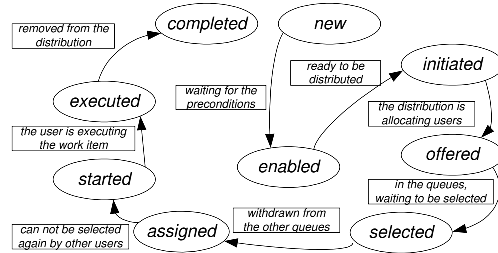


Fig. 1. Basic Model - Work Item Life Cycle

For the simulation (execution) of the work distribution in the model it is necessary to initiate the model by assigning values for four *input elements*[2], as shown in Table 2. Initial values of input elements describe a real-world situation that the model should execute.

Table 2. Input For The Basic Model

name	color	description
new work items	color WI = product Case * Task;	work items that have arrived and are ready to be distributed to users;
system users	color Users = list User;	a set of available users;
task maps	color TMap = product Task * Role * Group;	decision about which work items can be executed by which users is made based on the authorizations given in the process de nition, for every task;
user maps	color UMap = product User * Roles * Groups;	the organizational structure is used to map users to the authorization of tasks;

As a model of an abstract work ow management system, we have developed the Basic Model on the basis of prede ned assumptions: (1) we abstract from the process perspective (i.e., splits, joins, creation of work items), (2) we only consider the "normal" behavior (i.e., work items are completed successfully; errors and aborts are not included), and (3) we abstract from the user interface.

The Basic Model is organized into two sub-systems: the Work Distribution and the Work Lists module. The CPN language allows for the decomposition of complex nets into sub-pages, which are also referred to as sub-systems, sub-processes or modules. By using such modules we obtain a layered hierarchical description. Figure 2 shows the modular structure of the Basic Model. The two sub-modules communicate by exchanging messages via six places. These messages contain information about a user and a work item. Every message place is of the type (i.e., the CPN color set) "user work item" (*color UWI = product User * WI*), which is a combination of a user and a work item. Table 3 shows the description of the semantics of di erent messages that can be exchanged in the model.

[2] Initial marking for the CPN model.

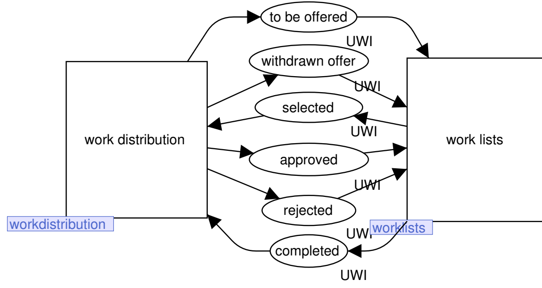


Fig. 2. Basic Model - Main

Table 3. Messages Between Modules

Place	Message
to be offered	A work item is offered to the user.
withdrawn offer	Withdraw the offered work item from the user.
selected	The user requests to select the work item.
approved	Allow the user to select the work item.
rejected	Do not allow the user to select the work item.
completed	The user has completed executing the work item

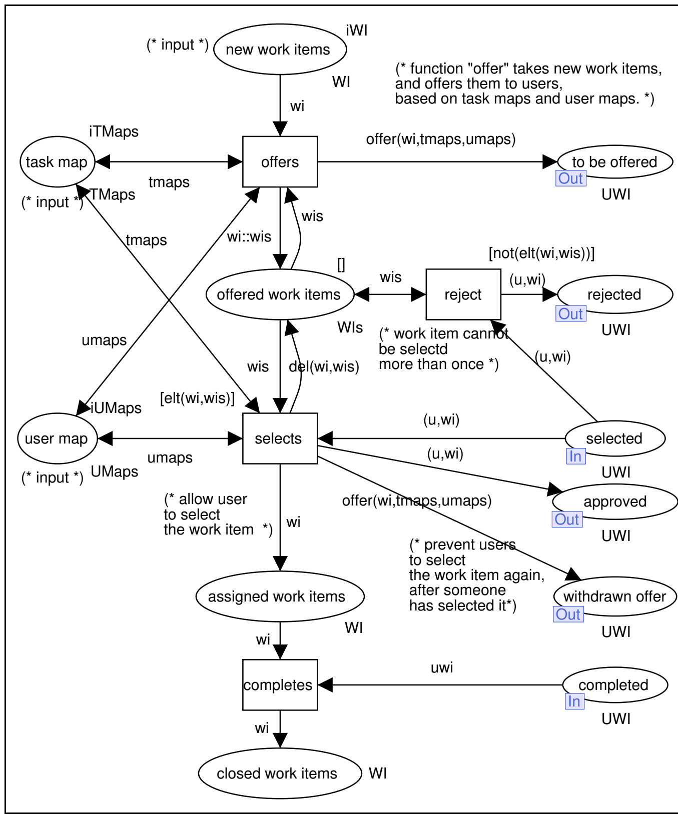
Work Distribution. The Work Distribution module manages the distribution of work items by managing the process of work execution and making sure that work items are executed correctly. It allocates (identi es) users to whom the *new work items* should be o ered, based on authorization (*TMap*) and organization (*UMap*) data. Three (out of four) input elements are placed in this module: *new work items*, *user maps* and *task maps*. The variables used in this module are shown in Table 4.

Table 4. Basic Model - Variables in Work Distribution Module

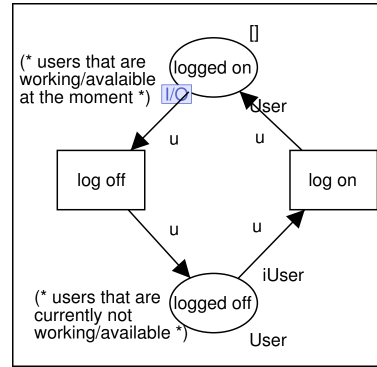
```
var tmaps: TMaps;
var umaps: UMaps;
var wi: WI;
var wis:WIs; ( color WIs = list WI; )
var uwi: UWI;
```

Figure 3(a) shows the Work Distribution module. The allocation function *o er* appears on the arc between the transition *o ers* and the place *to be o ered*. This function contains allocation rules (allocation algorithm) of the speci c distribution mechanism. Work items that are o ered to users are stored in the place *o ered work items*. After receiving a request from the user to select the work item, the decision is made whether to allow the user to select the item (and thus to execute it), or to reject this request. This decision is made based on the assumption that at one moment, only one user can work on the work item. If the work item has already been selected (i.e., it is not in the place *o ered work items*), then the model rejects this request. If nobody has selected the work item yet, the approval is sent to the user and the work item is moved to the place *assigned work items*. A work item that is moved to the place *assigned work items* cannot be selected again.

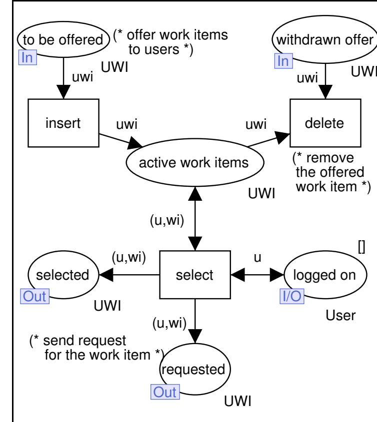
Work Lists. Figure 3(b) shows the Work Lists module. This module receives messages from the Work Distribution module about which work items are to be o ered to which users. The Work Lists module further manages events associated with the activities of users. It is decomposed into three units, which correspond to three basic actions users can make: *log on and o* (cf. Figure 3(c)) in the system, *select work* (cf. Figure 3(d)), *start work* (cf. Figure 3(e)), and *stop work* (cf. Figure 3(f)). Once the work item has been *o ered* to users, they can *select* it. When a user selects the work item, the *request* is sent to the Work Distribution module. If the request is *rejected*, the action is *aborted*. If the Work Distribution Module *approves* the request, the user can start working on the work item. Once the user has started working, the work item is considered to be *in progress*. Next, the user can stop working, and the work item is *completed*. In order to perform any of these actions, it is necessary that the user is *logged on* in the system.



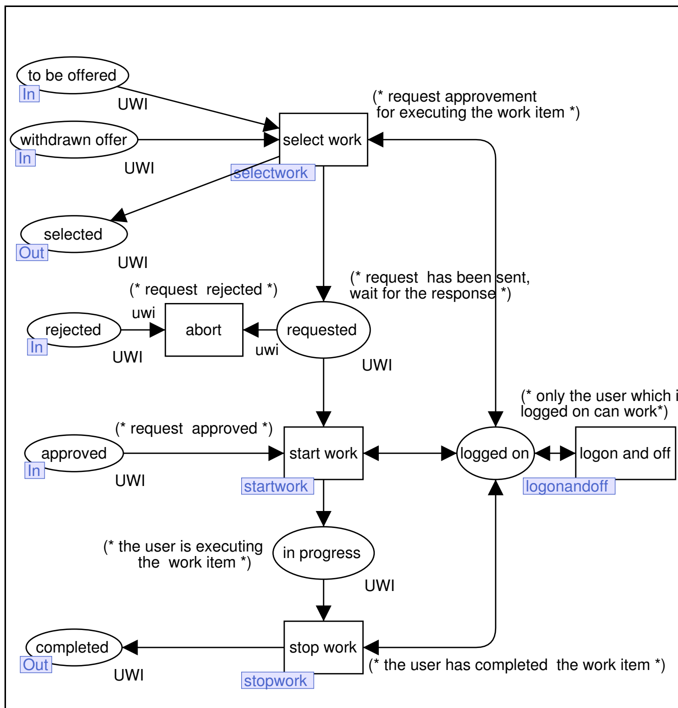
(a) Work Distribution



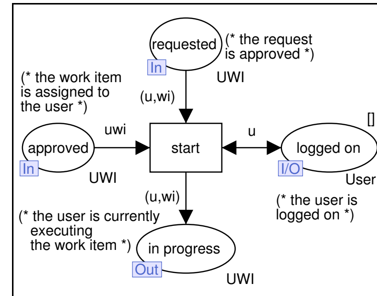
(c) Log On and Off



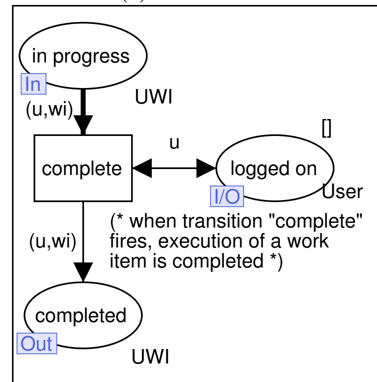
(d) Select Work



(b) Work Lists



(e) Start Work



(f) Stop Work

Fig. 3. Basic Model

3 Work Distribution Models

The Basic Model presented in previous section (Section 2) is used as a reference for different extensions and specializations of work distribution. In this section, we first extend and specialize the Basic Model to accommodate the capabilities of Staffware, FileNet and FLOWer (Section 3.1). In Section 3.2 we select four of the more advanced resource patterns reported in [49, 51]. These four patterns are not supported by Staffware, FileNet and FLOWer, but we will show that it is easy to extend the Basic Model to adequately address the patterns.

3.1 Workflow Management Systems

We have modelled the work distribution mechanisms of three commercial workflow management systems: Staffware, FileNet and FLOWer. FileNet and Staffware are examples of two widely used traditional workflow management systems. FLOWer is based on the case-handling paradigm, which can be characterized as "the more flexible approach" [3, 9]. Each of the models we have developed will be described shortly in the remainder of this section. For a more detailed description of the models we refer the reader to a technical report [44].

Staffware The Basic Model is upgraded to represent the work distribution of Staffware. The way of modelling the *organizational structure* and *resource allocation algorithm* are changed, while the concept of *work queues* and the possibility of the user to *forward and suspend* a work item are added to the model.

Organizational Structure. Simple organizational structure can be created in Staffware using the notions of *groups* and *roles*. The notion of group is defined as in the Basic Model, i.e., one group can contain several users, and one user can be a member of several groups. However, specific for Staffware is that a role can be defined for only one user. This feature does not require any changes in the model itself. It changes the way the initial value for the *user maps* should be defined – one role should be assigned to only one user.

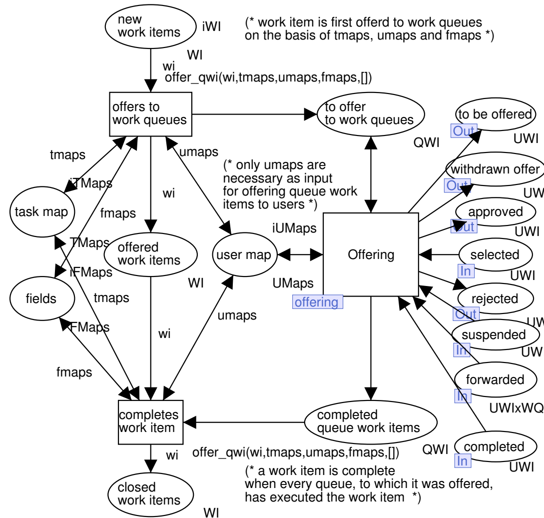
Work Queues. Groups are used in Staffware to model a set of users that share common rights. The work item can be allocated to the whole group, instead of listing the names of users that can execute it. Staffware introduces a *work queue* for every group. The work queue is accessible to all members of the group. Single users are also considered to be groups that contain only one member. Thus, one work queue is also created for every user and this personal queue is only accessible by a single user. From the perspective of the user, (s)he has access to the personal work queue and to work queues of all the groups (s)he is a member of. Table 5 shows which color sets are added to the model to represent work queues in Staffware. While the Basic Model (Section 2) offers the work item directly to *users*, Staffware offers items in two levels. First, the work item is offered to *work queues* (color set WQ). We refer to this kind of work items as to *queue work item* (color set QWI). Second, after a queue work item is offered to a group (work queue) it is offered to each of its members and only one member will execute the queue work item once. We refer to a queue work item that is offered to a member as to *user work item* (color set UWI).

Table 5. Staffware - "Work Queue" Color Sets

```
color WQ = string;
color QWI = product WI * WQ;
color UWI = product User *QWI;
```

Figures 4 and 5 show that we create two levels in the Work Distribution module to support the two-level distribution of Staffware:

1. In the module itself a new work item is o ered to work queues (as a queue work item). The new work item is completed when each of its queue work items is executed. Thus, if a new work item is o ered to multiple work queues, it is executed multiple times.
2. In the sub-module *O ering to Users* every queue work item is o ered to the queue members (user work item). A queue work item is completed when one of the members executes the user work item.



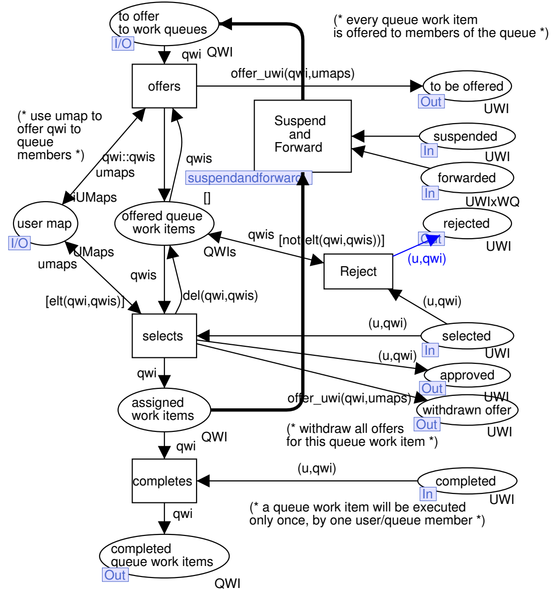


Fig. 4. Staffware - Work Distribution

Fig. 5. Staffware - Offering

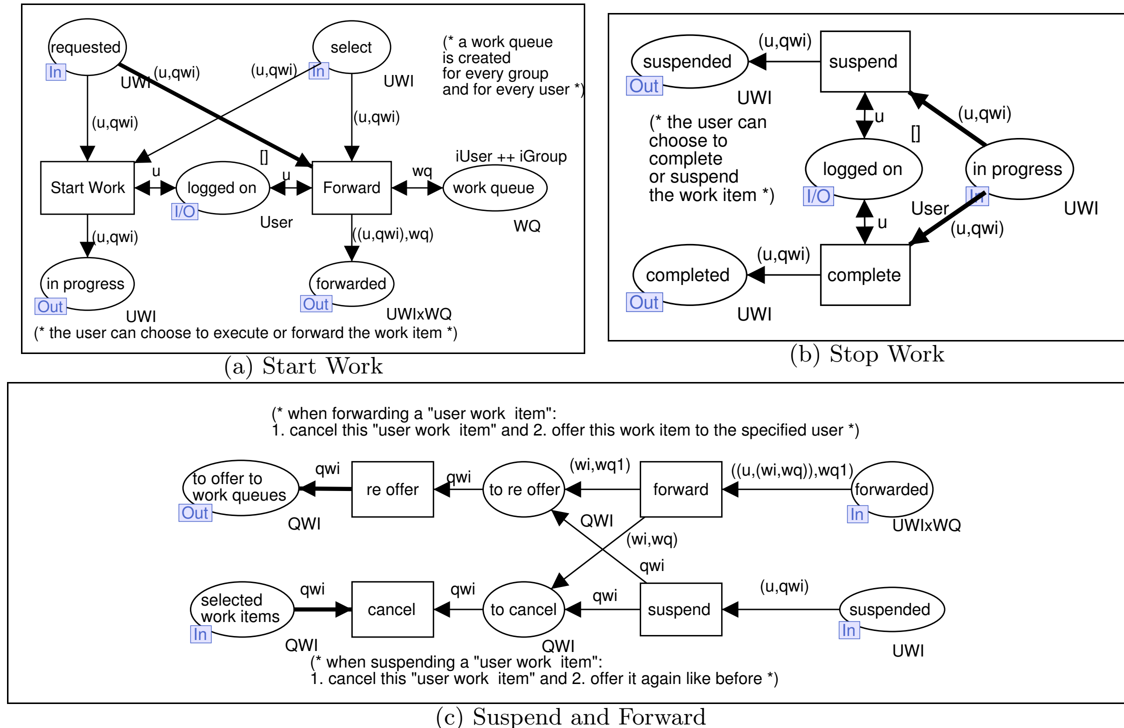
Resource Allocation. We have changed the allocation function *o er* to represent the allocation algorithm in Sta ware. Just like the Basic Model, Sta ware searches for possible users based on *roles* and *groups*. In addition to this, in Sta ware users can be allocated by their *user-names* and *data elds* in the process. In *user maps* we use the elds reserved for groups when we want to specify the allocation for user-names. We do this by assuming that every user-name refers to a group with only one member – the speci ed user. The second addition in Sta ware refers to the fact that resource allocation can be also done at "run-time" on the basis of *data elds* in *task maps* (cf. Table 6). This kind of allocation is referred to as a dynamic work allocation: the allocation is executed based on the current value of the eld during the process execution. Sta ware assumes that the value of the assigned data eld is a group name, a role name or a user name.

If the allocation refers to group names the work item is allocated to *group work queues*. In an allocation that refers to user names or roles the work item is allocated to *personal work queues*.

Table 6. Staffware - Dynamic Work Allocation

```
color Field = string;
color Fields = list Field;
color FValue = string;
color FMap = product Field*FValue;
color FMaps = list FMap;
color TMap = product Task * Users * Roles * Groups * Fields;
```

Forward and Suspend. Sta ware allows for *forward* and *suspend*, i.e., a user can put a work item on hold (suspend) or forward it to another user. Forwarding and suspending of work items adds two messages that are exchanged between Work Distribution and Work Lists modules in Sta ware model. Figures 4 and 5 show two new places – *forward* and *suspend*. These two new actions are triggered in the *Work List* module by the user. Figure 6(a) shows that in the module Start Work the user can choose to *select* or *forward* (to another work queue) the work item. Figure 6(b) shows that in the module Stop Work the user can choose to *complete* or *suspend* the work item. The Work Distribution module handles forwarding and suspending in the O ering to Users sub-module. Figure 6(c) shows how: (1) in case of forwarding the work item is automatically *cancelled* for the current work queue and *o ered* to the new work queue, and (2) in case of suspending the work item is *cancelled* for the current work queue and *re-o ered* as a new work item.



(a) Start Work

(b) Stop Work

(c) Suspend and Forward

Fig. 6. Staffware - Forward and Suspend

FileNet Like Sta ware, FileNet is a widely used traditional process-oriented work ow management system. In this section we will describe the FileNet CPN model that we develop using the Basic Model as a starting reference model.

Organization. The organizational model in FileNet does not allow for modelling *roles*. Table 7 shows which color sets are added to the CPN model to represent the two types of organizational groups:

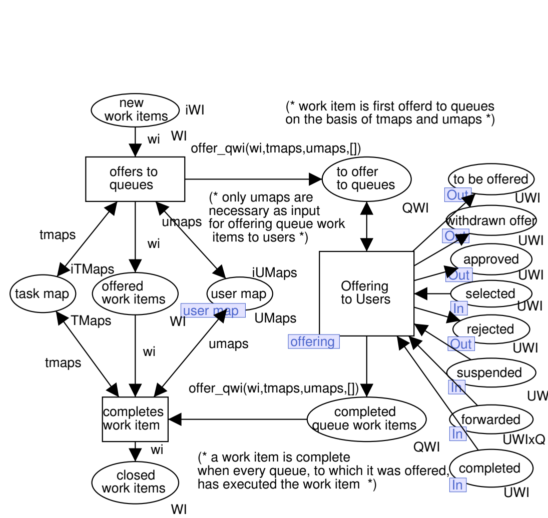
1. Administrators de ne *work queues* (color set *WQ*) and assign their members in the FileNet system. A work queue is valid for every process (work ow) de nition.
2. Process modelers can de ne *workflow groups* (color set *WG*) in every process model. Thus, a work ow group is valid only in the process (work ow) model in which it is de ned. While executing a task of a process de nition, users have the possibility to change the structure of work ow groups that are de ned in that process. Work ow groups represent teams in FileNet.

Queues. *Work queues* and *personal queues* are two types of pools from which users can select and execute work items. A work queue can have a number of members while a personal queue has only one member. When a work item is o ered to a queue one of the queue members can select and execute the work item. Table 7 shows which color sets are added to the FileNet model to represent queues. FileNet distributes work in two levels using queues. First, the work item is o ered to queues as a *queue work item* (color set *QWI*). Second, the queue work item is o ered to the members of the queue as a *user work item* (color set *UWI*).

Table 7. FileNet - "Queue" Color Sets

color Q = string;
color WQ = Q; (color WQs = list WQ;)
color WG = Q; (color WGs = list WG;)
color UMap = product User * WGs* WQs;
color QWI = product WI * Q;
color UWI = product User * QWI;

Figures 7 and 8 show that the model of the two-level work distribution in FileNet is similar to the Sta ware model. For more detailed description of this kind of distribution we refer the reader to the Sta ware description in Section 3.1.



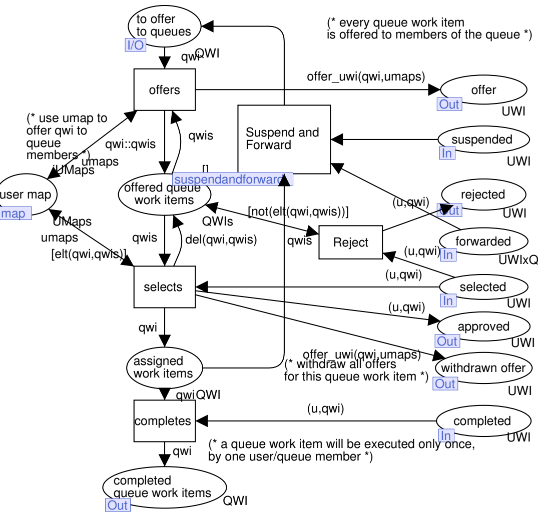


Fig. 7. FileNet - Work Distribution

Fig. 8. FileNet - Offering

Resource Allocation. FileNet allocates work using a work queue or a list of participants. Users and work ow groups can be entries of a list of participants. In the FileNet model task maps are de ned as a combination of a task, a list of work groups, and a work queue (*color TMap = product Task * WGs * WQ;*). It is necessary to highlight that, when de ning the input value for a task map, only work queue or a list of work ow groups should be initiated.

If the task is allocated to a work queue FileNet o ers the work item to the work queue. If the task is allocated to a list of participants then it is o ered to personal queues of all users that are listed as participants or are members in work ow groups that are listed. Allocation via participants is introduced to support team work in FileNet, via so-called "process voting" [44].

Forward and Suspend. Users can forward and suspend work items while working with FileNet. In the model of FileNet we use the same adjustments as in the Sta ware model to implement forwarding and suspension: modules *Start Work* and *Stop Work* are changed and sub-module *Suspend and Forward* is added in the Work Distribution module. For detailed description we refer the reader to Sta ware description and Figure 6 in this section.

FLOWer FLOWer is a case handling system. Case handling systems di er in their perspective from traditional process-oriented work ow management systems because they focus on the case, instead of the process [3, 9]. The user is o ered the whole case by o ering all available work items from the case and s(he) does not have to follow the prede ned order of tasks in the process de nition. When modelling FLOWer, we upgraded the Basic Model in such a way that (1) it supports *case-handling distribution* (instead of the process-oriented one), (2) it enables the complex *authorization* and *distribution* speci cations that FLOWer has, and (3) it enables users to *execute*, *open*, *skip* and *redo* work items.

Authorization and Distribution Rights. When designing the process it is necessary to de ne process-speci c roles and to assign each role authorization rights for tasks in the process. The authorization rights determine what users *can do*. Information about the authorization is stored in task maps (i.e., *color TMap = product Task * Role * CaseType*). Distribution rights de ne what users *should do*. These rights are used to model the organizational structure and to assign authorization rights from the process de nitions to users. *Function pro les* and *work pro les* de ne distribution rights. *Function pro le* is a set of authorization roles from di erent process de nitions. *Work pro les* assign function pro le(s) to users and they can be used to structure organization into groups, departments or units.

Case Handling. Table 8 shows which color sets are used to model FLOWer as a case-handling system. Every process de nition in FLOWer is referred to as a *case type*. One *case* represents an instance of a case type and is identi ed by the case identi cation (color set *CaseID*). Figures 9 and 10 show that FLOWer distributes work in two levels:

1. The case is distributed to users (color set *UCase*). Only one user can select and open the case at one moment. Figure 9 shows that in the FLOWer Work Distribution module a *case* becomes the object of distribution instead of a *work item*.
2. The selected case is opened for the user in the *Case Distribution* sub-module. Work items from the case are o ered to the user, based on the authorization and distribution rules. The user can execute, open, skip and redo work items from the selected case. The *Case Distribution* sub-module (cf. Figure 12) is described in the remainder of this section.

Table 8. FLOWer - Basic Color Sets

color CaseType = string;
color Tasks = list Task;
color Process = product CaseType * Tasks;
color CaseID = INT;
color Case = product CaseID* CaseType;
color WI = product Case * Task;
color UCase = product User * Case;

Open, Execute, Skip and Redo. Although in a case type tasks in the process de nition have the execution order that is suggested to the user, (s)he is not obliged to follow it. When working with an open case in FLOWer, users can: (1) *Execute* the work item which is next in the process de nition; (2) *Open* for execution a work item that is still not ready for execution according to

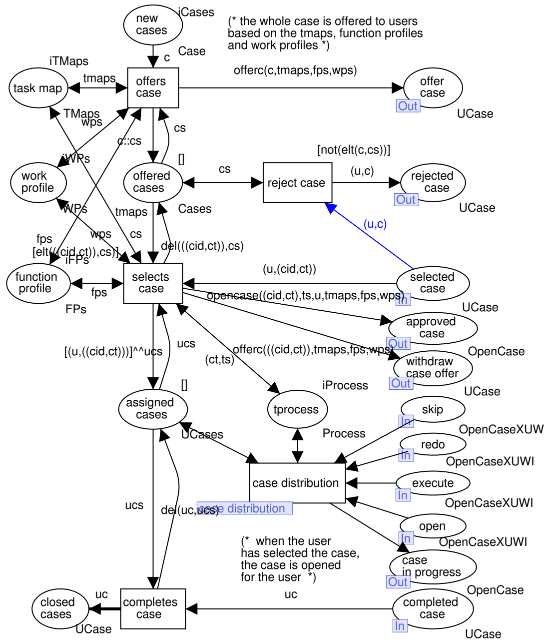


Fig. 9. FLOWer - Work Distribution

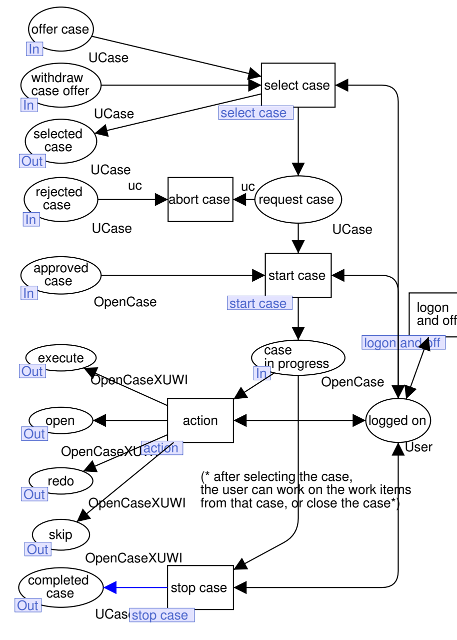


Fig. 10. FLOWer - Work Lists

the process de nition; (3) *Skip* a work item by choosing not to execute the work item which is next according to the process de nition, or (4) *Redo* a work item by executing again a work item which has already been executed. Figures 9 and 10 show that four new places are added to the model to represent these four actions. In order to implement these possibilities in the FLOWer model it is necessary store the information about the *case state*, i.e., about the work items that are (1) *waiting* to be enabled, (2) *active* (i.e. they are enabled and can be executed), (3) *nished* (executed), and (4) *skipped*. Thus, an open case (*color OpenCase = product UCase*CaseState;*) stores information about the case state (*color CaseState = product WIs*WIs*WIs*WIs*) in four lists of work items (waiting, active, nished and skipped).

Figure 11 shows the sub-module *Action* (in the FLOWer Work List module) where we model how user performs the actions to execute, open, skip and redo work items. In FLOWer users can choose work items on their own discretion but (due to the complexity of the model) we model this selection as a random function. When the user wants to:

1. **open** an item s(he) selects a work item from the list of *waiting items*;
2. **execute** an item s(he) selects a work item from the list of *active items*;
3. **skip** an item s(he) selects a work item from the lists of *waiting and active items*;
4. **redo** an item s(he) selects a work item from the lists of *nished and skipped items*.

Each of the four actions the user performs changes the state of the open case. For example, opening a work item transfers it to the state active (and, therefore, it is transferred to the list of active items). Figure 12 shows that the Case Distribution module responses in di erent ways (functions *execute_item*, *open_item*, *skip_item*, and *redo_item*) when each of the four actions is performed. When an action is performed over a work item, the state of the work item changes, as shown in Table 9. The four actions are listed in the column "action". The column "work item becomes" shows how the action changes state of the work item. It often happens that an action performed on a selected work item also a ects other items and this is described in the column "side e ects".

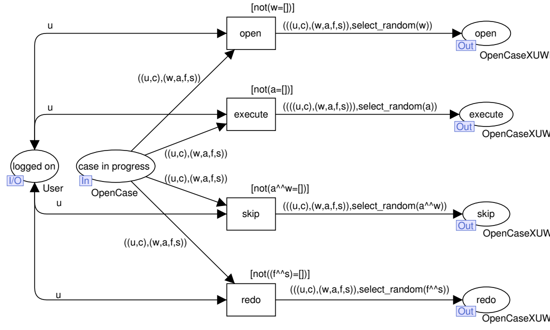


Fig. 11. FLOWer - Action

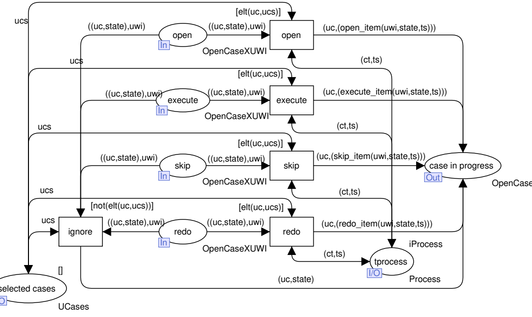


Fig. 12. FLOWer - Case Distribution

Table 9. FLOWer - The Four Actions

action	work item becomes	side effects
open	*active*	Items from *waiting* that preceded become *skipped*.
execute	*nished*	The direct successors in *waiting* become *active*.
skip	*skipped*	Items from *waiting* that preceded become *skipped*. The direct successors in *waiting* become *active*.
redo	*active*	Subsequent items from (*skipped* & *nished*) become *waiting*.

3.2 Resource Patterns

Instead of extending the Basic Model for more systems, we also looked at a more systematic way of work distribution. As indicated, similar concepts are often named and presented di erently in di erent work ow management systems. Therefore, it is interesting to de ne these concepts in a system-independent manner. We have used 43 documented *resource patterns* [49, 51]. These patterns can be used as representative examples for analyzing, evaluating and comparing di erent work ow management systems with respect to work distribution. Resource patterns are grouped into a number of categories: *creation patterns*, *push patterns*, *pull patterns*, *detour patterns*, *auto-start patterns*, *visibility patterns*, and *multiple resource patterns*. Each of these patterns can be modeled in terms of a CPN model. We cannot elaborate on each of the patterns, but we will discuss four to illustrate our work. None of the systems supports *Pattern 16: Round Robin*, *Pattern 17: Shortest Queue*, *Pattern 38: Piled Execution*, and *Pattern 39: Chained Execution*. Patterns 16 and 17 are push patterns, i.e., they push work to a speci c user. As auto-start patterns, patterns 38 and 39 enable the automatic start of the execution of the next work item once the previous has been completed.

Round Robin and Shortest Queue. Round Robin and Shortest Queue push the work item to one user of all users that qualify. Round Robin allocates work on a cyclic basis and Shortest Queue to the user with the shortest queue. This implies that each user has a counter to: (1) count the sequence of allocations in Round Robin and (2) count the number of pending work items in Shortest Queue. As Figures 13 and 14 show, these two patterns are implemented in a similar way in the Work Distribution Module. The required changes to the Basic Model are minimal. A counter is introduced for each user (token in place *available*) and functions *round_robin* and *shortest_queue* are used to select one user from the set of possible users based on these counters. Similarly, most of the other patterns can be realized quite easily. The model for Shortest Queue has an additional connection (two *arcs*) that updates the counter when a work item is completed to remove it from the queue.

Piled and Chained Execution. Piled and Chained Execution are auto-start patterns, i.e., when the user completes execution of current work item the next work item starts automatically. When

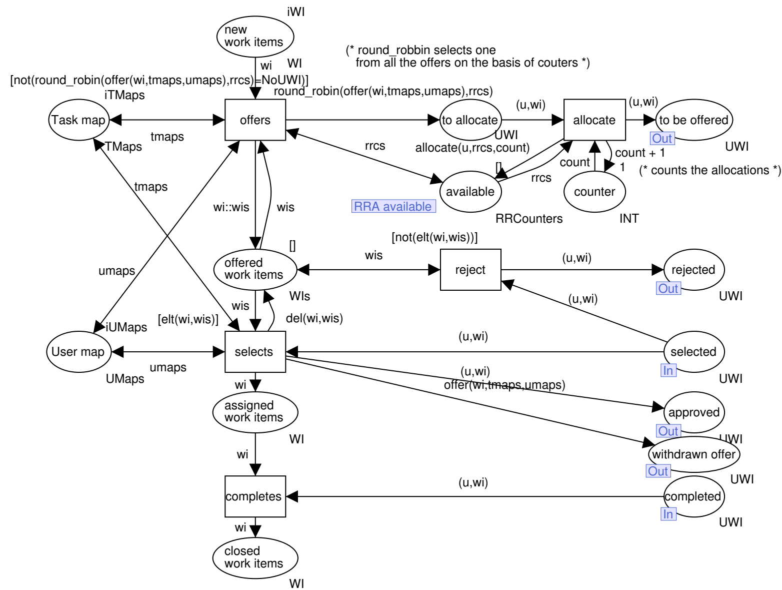


Fig. 13. Push Patterns - Round Robin

working in Chained Execution, the next work item will be for the same *case* as the completed one (the user works on di erent tasks for one case). Similarly, if the user works in Piled Execution the next work item will be for the same *task* as the completed one (the user works on one task for di erent cases). Figures 15 and 16 show how Piled and Chained Execution are implemented similarly in the Stop Work sub-module. Users can choose to work in the normal mode or in the auto-start mode (which is represented by the token in place *special mode*). The function *select* is implemented to search for the next work item for the same: (1) *task* in Piled Execution and (2) *case* in Chained Execution.

4 Related Work

Since the early nineties work ow technology has matured [26] and several textbooks have been published, e.g., [5, 20, 30, 37, 41]. During this period many languages for modelling work ows have been proposed, i.e., languages ranging from generic Petri-net-based languages to tailor-made domain-speci c languages. The Work ow Management Coalition (WfMC) has tried to standardize work- ow languages since 1994 but failed to do so [25]. XPDL, the language proposed by the WfMC, has semantic problems [2] and is rarely used. In a way BPEL [11] succeeded in doing what the WfMC was aiming at. However, both BPEL and XPDL focus on the control- ow rather than the resource perspective.

Despite the central role that resources play in work ow management systems, there is a surprisingly small body of research into resource and organizational modelling in the work ow context [1, 35]. In early work, Bussler and Jablonski [15] identi ed a number of shortcomings of work- ow management systems when modelling organizational and policy issues. In subsequent work [30], they presented one of the rst broad attempts to model the various perspectives of work ow management systems in an integrated manner including detailed consideration of the organizational/resource view.

One line of research into resource modelling and enactment in a work ow context has focused on the characterization of resource managers that can manage organizational resources and enforce resource policies. In [19], the design of a resource manager is presented for a work ow management system. This work includes a high level resource model together with proposals for resource de nition, query and policy languages. Similarly, in [36], an abstract resource model is presented in the

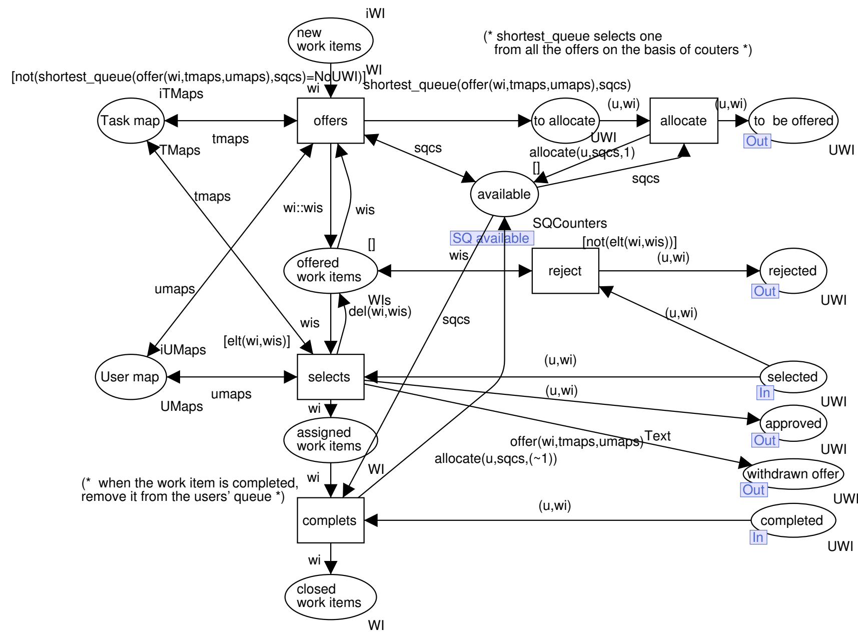


Fig. 14. Push Patterns - Shortest Queue

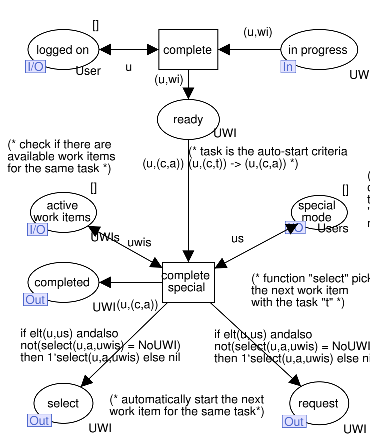


Fig. 15. Piled Execution - Stop Work

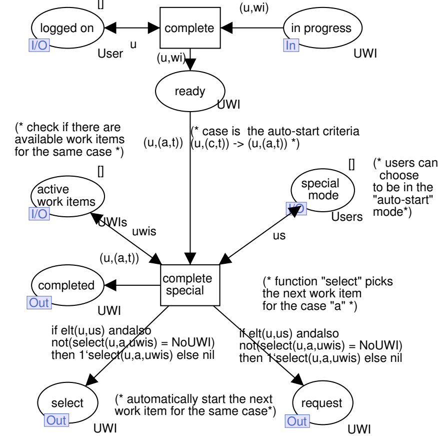


Fig. 16. Chained Execution - Stop Work

context of a work ow management system although the focus is more on the e cient management of resources in a work ow context than the speci c ways in which work is allocated to them. In [29], a proposal is presented for handling resource policies in a work ow context. Three types of policy – quali cation, requirement and substitution – are described together with a means for e ciently implementing them when allocating resources to activities.

Another area of investigation has been into ensuring that only appropriate users are selected to execute a given work item. The RBAC (Role-Based Access Control) model [23] presents an approach for doing this. RBAC models are e ective but they tend to focus on security considerations and neglect other organizational aspects such as resource availability.

Several researchers have developed meta-models, i.e., object models describing the relation between work ow concepts, which include work allocation aspects, cf. [8, 40–42, 48]. However,

these meta-models tend to focus on the structural description of resource properties and typically do not describe the dynamics aspects of work distribution.

Flexibility has been a research topic in workflow literature since the late nineties [4, 7, 9, 10, 16, 22, 28, 33, 45, 47, 56]. Flexibility triggers all kinds of interesting research questions, e.g., if a process changes how this should influence the running cases? [7]. Examples of qualitative analysis of flexibility of workflow management system can be found in [13] and [27]. One way of allowing for more flexibility is to use the case handling concept as defined in [3, 9]. FLOWer [12, 43] can be seen as a reference implementation of the case handling concept. Therefore, its resource perspective was modeled in this paper. Besides FLOWer there are few other case handling tools: E.C.H.O. (Electronic Case-Handling for Offices), a predecessor of FLOWer, the Staffware Case Handler [53] and the COSA Activity Manager [52], both based on the generic solution of BPi [14], Vectus [38, 39], and the open-source system con:cern (http://con-cern.org/).

The work reported in this paper can be seen as an extension of the *workflow patterns initiative* (cf. www.workflowpatterns.com). Besides a variety of control-flow [6] and data [50] patterns, 43 resource patterns [49, 51] have been defined. This paper complements the resource patterns [49, 51] by providing executable models for work distribution mechanisms.

5 Discussion

Workflow management systems should provide flexible work distribution mechanisms for users. This will increase the work satisfaction of users and improve their ability to deal with unpredictable situations at work. Therefore, work distribution is investigated as the functionality provided for the user – workflow management systems are tested in laboratories [49, 51] or observed (in empirical research) in companies [13]. This kind of research observes systems *externally* and provides insights into *what* systems do. Analysis of the systems form an *internal* perspective can explain *how* systems provide for different work distribution mechanisms. Due to the complexity of workflow management systems as software products, internal analysis starts with developing a model of the system. Unlike statical models (e.g., UML models), dynamical models (e.g., CPN models) provide for interactive investigation of work distribution as a dynamic feature. CPN models can be used for the investigation of both *what* systems do and *how* they do it.

Workflow management systems often provide for different features or use different naming for the same features. Investigation of work distribution requires analysis, evaluation and comparison of models of several systems. In order for models of different systems to be comparable, it is necessary to start with developing a common framework – a *reference model*. We have developed the Basic Model as a reference model for work distribution mechanisms in workflow management system. The models of Staffware, FileNet, FLOWer and resource patterns are comparable because all models are developed as upgrades of a reference model (the Basic Model).

The model of a workflow system is structured into two modules (sub-models). The Work Distribution module represents the core of the system which is often called the "workflow engine". The Work Lists module represents the so-called "work list handler" of a workflow system and it serves as an interface between the workflow engine and users. The interface between the two modules (i.e., the messages that are exchanged between them) should contain as little information as possible about the way work items are managed in modules. The Work Lists module should abstract form the way the work items are created, allocated and offered in the Work Distribution module. The reverse also holds: how work items are actually processed by users is implemented in the Work Lists module. Once a proper interface is defined, it is easy to implement various ways of work distribution by adding/removing simple features in either one of the modules. For example, push patterns (Round Robin and Shortest Queue) are implemented in the Work Distribution module and auto-start resource patterns (Chained and Piled Execution) in the Work Lists module.

The flexibility of a work distribution mechanism determines what users can do with work items. In the Basic Module the user follows a fixed predefined path by only executing work items. Users of Staffware and FileNet models have the freedom to forward and suspend work. In FLOWer, as the most flexible system, users have four possibilities: execute, open, skip and redo work. Our models

show that a more complex model work distribution adds messages between the Work Distribution and Work Lists modules. These new messages correspond to new actions (operations) that users can do.

Both the system-based and the patterns-based CPN models showed that one of the core elements of work distribution is the "allocation algorithm". This algorithm includes the "rules" for work distribution. It is implemented in the Work Distribution module as the function *o er*, which allocates work based on (1) new work items, (2) process de nition, and the (3) organizational model. This function should be analyzed further in order to discover an advanced allocation algorithm, which should be more con gurable and less system-dependent.

Every system has its own method of modelling organizational structure. Sta ware models groups and roles. In FileNet the organizational model includes groups of users and teams, but does not model roles. FLOWer groups users based on a hierarchy of roles, function pro les and work pro les. Thus, each of the system o ers a unique prede ned type of the organizational structure. Since every allocation mechanism uses elements of the organizational model, limitations of the organizational model can have a negative impact on the work distribution in the system. For example, because in Sta ware one role can be assigned to only one user, it is not be possible to o er a work item to a set of "call center operator"-s.

Each of the three models of work ow systems distributes work using two hierarchy levels. Sta ware and FileNet use two levels of work distribution: queue work items are rst distributed to work queues, and then work items are distributed within each of the work queues. The FLOWer model starts with the case distribution and then distributes work items of the whole case. Although all three systems distribute work at two levels, they have unique *distribution algorithms* (the set of allocation rules implemented in the function *o er*) and *objects of distribution* (work items, queue work items, cases).

Models of resource patterns [49, 51] show that push patterns (Round Robin and Shortest Queue) can be implemented "on top of" the pull mechanism, as a lter. Once the pull mechanism determines the set of allocated users, the "push" allocation function extracts only one user from this set. Auto-start patterns turned out to be remarkable straightforward to model, triggering the question why this is not supported by systems like Sta ware and FileNet (FLOWer supports the Chained Execution in a limited form).

6 Conclusions

This paper focused on the *resource perspective*, i.e., the way work ow management systems distribute work based on the structure of the organization and capabilities/quali cations of people. To understand work distribution, we used the CPN language and CPN Tools to model and analyze di erent work distribution mechanisms. To serve as a reference model, we provided a model that can be seen as the "greatest common denominator" of existing work ow management systems. This model was upgraded for models of three work ow management systems – Sta ware, FileNet, and FLOWer. Although the reference model already captures many of the *resource patterns*, we also modelled four more advanced patterns by extending the reference model. In contrast to existing research that mainly uses static models (e.g., UML class diagrams), we focused on the dynamics of work distribution. Our experiences revealed that it is relatively easy to model and analyze the work ow systems and resource patterns using CPN Tools. This suggests that CPN language and the basic CPN model are a good basis for future research. We plan to test completely new ways of work distribution using the approach presented in this paper. The goal is to design and implement distribution mechanisms that overcome the limitations of existing systems. An important ingredient will be to use insights from socio-technical design [13, 17, 21, 55] as mentioned in the introduction.

References

1. W.M.P. van der Aalst. Don't go with the ow: Web services composition standards exposed. *IEEE Intelligent Systems*, 18(1):72–76, 2003.

2. W.M.P. van der Aalst. Business Process Management Demysti ed: A Tutorial on Models, Systems and Standards for Work ow Management. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 1–65. Springer-Verlag, Berlin, 2004.
3. W.M.P. van der Aalst and P.J.S. Berens. Beyond Work ow Management: Product-Driven Case Handling. In S. Ellis, T. Rodden, and I. Zigurs, editors, *International ACM SIGGROUP Conference on Supporting Group Work (GROUP 2001)*, pages 42–51. ACM Press, New York, 2001.
4. W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors. *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2000.
5. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2002.
6. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Work ow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
7. W.M.P. van der Aalst and S. Jablonski. Dealing with Work ow Change: Identi cation of Issues and Solutions. *International Journal of Computer Systems, Science, and Engineering*, 15(5):267–276, 2000.
8. W.M.P. van der Aalst and A. Kumar. Team-Enabled Work ow Management Systems. *Data and Knowledge Engineering*, 38(3):335–363, 2001.
9. W.M.P. van der Aalst, M. Weske, and D. Grünbauer. Case Handling: A New Paradigm for Business Process Support. *Data and Knowledge Engineering*, 53(2):129–162, 2005.
10. A. Agostini and G. De Michelis. Improving Flexibility of Work ow Management Systems. In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 218–234. Springer-Verlag, Berlin, 2000.
11. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1. Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation, 2003.
12. Pallas Athena. *Case Handling with FLOWer: Beyond workflow*. Pallas Athena BV, Apeldoorn, The Netherlands, 2002.
13. J. Bowers, G. Button, and W. Sharrock. Work ow From Within and Without: Technology and Cooperative Work on the Print Industry Shop oor. In *The Fourth European Conference on Computer-Supported Cooperative Work (ECSCW 95)*, pages 51–66, Stockholm, September 1995. Kluwer Academic Publishers, Dordrecht, The Netherlands.
14. BPi. *Activity Manager: Standard Program - Standard Forms (Version 1.2)*. Work ow Management Solutions, Oosterbeek, The Netherlands, 2002.
15. C. Bussler and S. Jablonski. Policy Resolution for Work ow Management Systems. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, page 831. IEEE Computer Society, 1995.
16. F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Work ow Evolution. In *Proceedings of ER '96*, pages 438–455, Cottubus, Germany, Oct 1996.
17. L. U. de Sitter, J. F. den Hertog, and B. Dankbaar. From complex organiations with simple jobs to simple organizations wiht complex jobs. *Human Relations*, 510(5):497–534, 1997.
18. B. van Dongen, A.K. Alves de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. The ProM framework: A New Era in Process Mining Tool Support. In G. Ciardo and P. Darondeau, editors, *Application and Theory of Petri Nets 2005*, Lecture Notes in Computer Science, pages 444–454. Springer-Verlag, Berlin, 2005.
19. W. Du and M.C. Shan. Enterprise Work ow Resource Management. In *Ninth International Workshop on Research Issues on Data Engineering: Information Technology for Virtual Enterprises (RIDE-VE'99)*, pages 108–115, Sydney, Australia, 1999. IEEE Computer Society Press.
20. M. Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede. *Process-Aware Information Systems*. Wiley & Sons, 2005.
21. F.M. van Eijnatten and A.H. van der Zwaan. The Dutch IOR approach to organisation design. An alternative to business process re-engineering? *Human Relations*, 51(3):289–318, 1998.
22. C.A. Ellis and K. Keddara. A Work ow Change Is a Work ow. In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 201–217. Springer-Verlag, Berlin, 2000.
23. D.F. Ferraiolo, R. Sandhu, S. Gavrila, D.R. Kuhn, and R. Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

24. FileNET. *FileNet Business Process Manager 3.0*. FileNET Corporation, Costa Mesa, CA, USA, June 2004.
25. L. Fischer, editor. *Workflow Handbook 2003, Workflow Management Coalition*. Future Strategies, Lighthouse Point, Florida, 2003.
26. D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Work ow Management: From Process Modeling to Work ow Automation Infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.
27. R.E. Grinter. Work ow Systems: Occasions for Success and Failure. *Computer Supported Cooperative Work*, 9(2):189–214, 2000.
28. T. Herrmann, M. Hoffmann, K.U. Loser, and K. Moysich. Semistructured models are surprisingly useful for user-centered design. In G. De Michelis, A. Giboin, L. Karsenty, and R. Dieng, editors, *Designing Cooperative Systems (Coop 2000)*, pages 159–174. IOS Press, Amsterdam, 2000.
29. Y.N. Huang and M.C. Shan. Policies in a Resource Manager of Work ow Systems: Modeling, Enforcement and Management. Technical Report HP Tech. Report, HPL-98-156, Palo Alto, CA, USA, 1999. Accessed at http://www.hpl.hp.com/techreports/98/HPL-98-156.pdf on 20 March 2005.
30. S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996.
31. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1*. EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1997.
32. K. Jensen and G. Rozenberg, editors. *High-level Petri Nets: Theory and Application*. Springer-Verlag, Berlin, 1991.
33. M. Klein, C. Dellarocas, and A. Bernstein, editors. *Adaptive Workflow Systems*, volume 9 of *Special issue of the journal of Computer Supported Cooperative Work*, 2000.
34. L.M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.
35. A. Kumar, W.M.P. van der Aalst, and H.M.W. Verbeek. Dynamic Work Distribution in Work ow Management Systems: How to Balance Quality and Performance? *Journal of Management Information Systems*, 18(3):157–193, 2002.
36. B.S. Lerner, A.G. Ninan, L.J. Osterweil, and R.M. Podorozhny. Modeling and Managing Resource Utilization in Process, Work ow, and Activity Coordination. Technical Report UM-CS-2000-058, Department of Computer Science, University of Massachusetts, August 2000. Accessed at http://laser.cs.umass.edu/publications/?category=PROC on 20 March 2005.
37. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999.
38. London Bridge Group. *Vectus Application Developer's Guide*. London Bridge Group, Wellesbourne, Warwick, UK, 2001.
39. London Bridge Group. *Vectus Technical Architecture*. London Bridge Group, Wellesbourne, Warwick, UK, 2001.
40. M. Zur Muehlen. Evaluation of Work ow management Systems Using Meta Models. In *Proceedings of the 32nd Hawaii International Conference on System Sciences - HICSS'99*, pages 1–11, 1999.
41. M. Zur Muehlen. *Workflow-based Process Controlling: Foundation, Design and Application of workflow-driven Process Information Systems*. Logos, Berlin, 2004.
42. M. zur Muhlen. Organizational Management in Work ow Applications Issues and Perspectives. *Information Technology and Management*, 5(3–4):271–291, July-October 2004.
43. Pallas Athena. *Flower User Manual*. Pallas Athena BV, Apeldoorn, The Netherlands, 2002.
44. M. Pesic and W.M.P. van der Aalst. Modeling Work Distribution Mechanisms using Colored Petri Nets. BETA Working Paper Series, WP 146, Eindhoven University of Technology, Eindhoven, 2005.
45. M. Reichert and P. Dadam. ADEPT ex: Supporting Dynamic Changes of Work ow without Loosing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.
46. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.
47. S. Rinderle, M. Reichert, and P. Dadam. Correctness Criteria For Dynamic Changes in Work ow Systems: A Survey. *Data and Knowledge Engineering*, 50(1):9–34, 2004.
48. M. Rosemann and M. Zur Muehlen. Evaluation of Work ow Management Systems - a Meta Model Approach. *Australian Journal of Information Systems*, 6(1):103–116, 1998.
49. N. Russell, W.M.P.van der Aalst, A.H.M. ter Hofstede, and D. Edmond. Work ow Resource Patterns: Identi cation, Representation and Tool Support. In O. Pastor and J. Falcao e Cunha, editors, *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05)* , volume 3520 of *Lecture Notes in Computer Science*, pages 216–232. Springer-Verlag, Berlin, 2005.

50. N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Work ow Data Patterns. QUT Technical report, FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.
51. N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Work ow Resource Patterns. BETA Working Paper Series, WP 127, Eindhoven University of Technology, Eindhoven, 2004.
52. Software-Ley. *COSA Activity Manager*. Software-Ley GmbH, Pullheim, Germany, 2002.
53. Staffware. *Staffware Case Handler – White Paper*. Staffware PLC, Berkshire, UK, 2000.
54. Staffware. *Using the Staffware Process Client*. Staffware, plc, Berkshire, United Kingdom, May 2002.
55. F. M. van Eijnatten. *The Paradigm that Changed the Work Place*. Van Gorcum, Assen, The Netherlands, 1993.
56. M. Weske. Formal Foundation and Conceptual Design of Dynamic Adaptations in a Work ow Management System. In R. Sprague, editor, *Proceedings of the Thirty-Fourth Annual Hawaii International Conference on System Science (HICSS-34)*. IEEE Computer Society Press, Los Alamitos, California, 2001.

Process Mining: Using CPN Tools to Create Test Logs for Mining Algorithms

A.K. Alves de Medeiros and C.W. Gunther

Department of Technology Management, Eindhoven University of Technology
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands.
{a.k.medeiros,c.w.gunther}@tm.tue.nl

Abstract. Process mining aims at automatically generating process models from event logs. The main idea is to use the discovered models as an *objective start point* to deploy systems that support the execution of business processes (for instance, workflow management systems) or as a *feedback mechanism* to check if the prescribed models fit the executed ones. When developing an algorithm to do process mining, one needs some logs to test it. Using real-life logs seems to be the natural choice. However, the real-life event logs usually contain imperfections that can hinder the tuning of the mining algorithm. For instance, real-life logs can be incomplete and/or contain noise. Thus, a more common approach is to first test the accuracy of new process mining algorithms in logs created via simulation. This allows the researcher to have more control about the properties of the event log and to fine tune his/her mining algorithm. Besides, having the original model (the simulated one) may also be a useful aid to assess the quality of the mining algorithm. In our research group, we work with the ProM framework [6] mining tool which receives as input an XML event log. This XML format is also supported by process mining tools of other research groups [4]. This paper shows how to extend CP-nets to generate XML event logs that can be mined by process mining tools supporting this format. This way we benefit from the simulation capabilities of CPN Tools and, therefore, we avoid reinventing the wheel. The extension we made consisted of implementing (i) some ML functions that can be used to annotate the CP-net, and (ii) a ProM$_{import}$-framework [2] plug-in that bundles up the files (generated by the CP-net simulation) into a single XML file that is ready to be mined.

1 Introduction

Process mining targets the *automatic* discovery of information from an event log. This discovered information can be used to deploy new systems that support the execution of business processes or as a feedback tool that helps in analyzing and improving enacted business processes. The starting point of any process mining technique is an event log. This log can have lots of data, such as the tasks that are executed, their time of execution, the person/system that performed them, the data fields related to these tasks, and so forth. For instance, consider the event log in Table 1. This log has four executions (cases) of a process that handles fines for drivers in a certain province.

Case ID	Task Name	Event Type	Originator	Timestamp	Extra Data
1	File Fine	Completed	Anne	20-07-2004 14:00:00	...
2	File Fine	Completed	Anne	20-07-2004 15:00:00	...
1	Send Bill	Completed	system	20-07-2004 15:05:00	...
2	Send Bill	Completed	system	20-07-2004 15:07:00	...
3	File Fine	Completed	Anne	21-07-2004 10:00:00	...
3	Send Bill	Completed	system	21-07-2004 14:00:00	...
4	File Fine	Completed	Anne	22-07-2004 11:00:00	...
4	Send Bill	Completed	system	22-07-2004 11:10:00	...
1	Process Payment	Completed	system	24-07-2004 15:05:00	...
1	Close Case	Completed	system	24-07-2004 15:06:00	...
2	Send Reminder	Completed	Mary	20-08-2004 10:00:00	...
3	Send Reminder	Completed	John	21-08-2004 10:00:00	...
2	Process Payment	Completed	system	22-08-2004 09:05:00	...
2	Close case	Completed	system	22-08-2004 09:06:00	...
4	Send Reminder	Completed	John	22-08-2004 15:10:00	...
4	Send Reminder	Completed	Mary	22-08-2004 17:10:00	...
4	Process Payment	Completed	system	29-08-2004 14:01:00	...
4	Close Case	Completed	system	29-08-2004 17:30:00	...
3	Send Reminder	Completed	John	21-09-2004 10:00:00	...
3	Send Reminder	Completed	John	21-10-2004 10:00:00	...
3	Process Payment	Completed	system	25-10-2004 14:00:00	...
3	Close Case	Completed	system	25-10-2004 14:01:00	...

Table 1. Example of an event log.

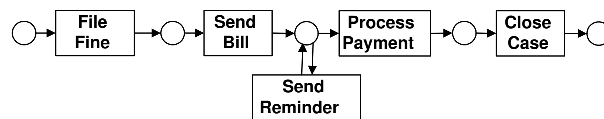


Fig. 1. Petri net illustrating the control- ow perspective that can be mined from the event log in Table 1.

The amount of data in the event log determines which *perspectives* of process mining can be discovered. If the log provides the tasks that are executed in the process and it is possible to infer their order of execution, the *control-flow perspective* can be mined. The log in Table 1 has this data (cf. fields "Case ID", "Task Name" and "Timestamp"). So, for this log, mining algorithms could discover the process in Figure 1. Basically, the process describes that after a fine is entered in the system, the bill is sent to the driver. If the driver does not pay the bill within one month, a reminder is sent. When the bill is paid, the case is archived. If the log provides information about the persons/systems that executed the tasks, the *organizational perspective* can be discovered. The organizational perspective discovers information like the social network in a process, the transferring of work etc. For instance, the log in Table 1 shows that "Anne" transfers work for both "Mary" (case 2) and "John" (cases 3 and 4), and "John" sometimes transfers work for "Mary" (case 4). Besides, by inspecting the log, the mining algorithm could discover that "Mary" never has to send a reminder more than once, while "John" does not seem to perform as good. The managers could talk to "Mary" and check if she has another approach to send reminders that "John" could benefit from. This can help in making good practices a common knowledge in the organization. When the log contains more details about the tasks, like the values of data fields that the execution of the task modifies, the *case perspective* can be discovered. So, for instance, a forecast for executing cases can be made based on previous already completed cases, exceptional situations can be discovered etc. In our particular example, logging information about the profiles of drivers (like age, gender, car etc) could help in assessing the probability that they would pay their fines on time. Moreover, logging information about the places where the fines were applied could help in improving the traffic measures in these places.

Having these three perspectives in mind, and the different mining tools[1] to tackle one or more of these perspectives, an effort was made in [4] to define a single XML format, called the Mining XML (MXML) format, that could be used as input to the different tools. By converting simulated or real-life logs to the MXML format, one could use the mining techniques in multiple contexts. Therefore, we test our mining algorithms with logs that have the MXML format.

Real-life logs typically contain some subset of data fields that allow for mining. However, real-life logs are not always the best data to first test a mining algorithm that is being developed because real-life logs may be incomplete and/or contain noise. So, a better approach to test mining algorithms is to use simulated data. Note that the use of simulated data gives the researcher more control over event log properties. This more controlled environment may help with tuning the

[1] Examples of mining tools are InWolvE [8, 10], Process Miner [11], EMiT [7], Little Thumb [12], MiSoN [3] and ProM framework [6]. The InWolvE, Process Miner, EMiT and Little Thumb mine the control-flow perspective. The MiSoN mines the organizational perspective. The ProM framework has plug-ins that to mine all three perspectives. Actually, the mining tools EMiT, MiSoN and Little Thumb were respectively implemented as the ProM mining plug-ins "Alpha algorithm", "Social network miner" and "Heuristics miner".

algorithm under development. After being able to correctly mine simulated data, the algorithm can be tested with real-life logs.

CPN Tools supports the modelling, execution and analysis of *Coloured Petri nets* (CP-nets) [9]. Additionally, there is a fair amount of CPN models that can be used as input to test mining algorithms. Thus, we decided to extend CPN Tools to support the creation of MXML logs. The main idea is to create random MXML logs by simulating CP-nets in CPN Tools. The extension is fairly simple and took us no more than twelve man-hours to finish it. The first part of the extension consisted of implementing the ML functions to support the logging from a CP-net. The second part consisted of implementing the "CPN Tools" plug-in in the the ProM_{import} framework [2] to bundle the logged files into a single MXML file.

In short, two steps are necessary to create MXML logs using CPN Tools:

1. Modify a CP-net to invoke the set of ML functions that will create logs for every case executed by the CP-net. This step involves modifying the *declarations* of the CP-net and the *input/output/action* transition inscriptions.
2. Use the CPN Tools plug-in, in the ProM_{import} framework, to group the logs for the individual cases into a single MXML log.

The rest of this paper is organized as follows. Section 2 explains the MXML format. Understanding how this format supports the different perspectives of mining helps in understanding how CPN Tools was extended. Section 3 describes how to modify a CP-net to create partial MXML logs during its simulation (Step 1 above). Section 4 shows how to use the ProM_{import} framework to bundle these partial MXML logs into a single log that can be mined (Step 2 above). Section 5 presents some conclusions and future work.

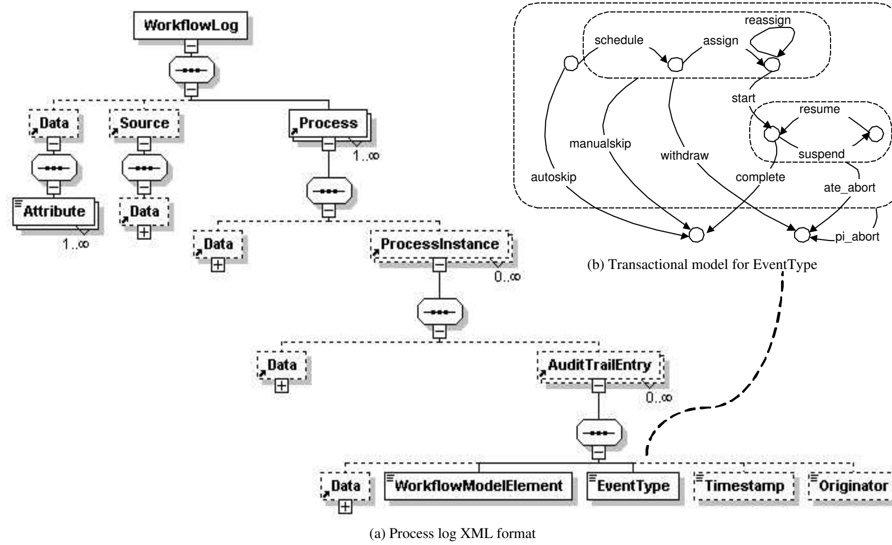
2 The MXML format

The Mining XML[2] format (MXML) started as an initiative to make different mining tools have a common input format [4]. This way, event logs could be shared among different mining tools. Actually, defining a common input format like MXML is the first step towards the creation of a repository on which process mining researchers can test their algorithms. In this section we explain the MXML format because this helps to understand the ML functions defined for the extension of CP-nets. The schema for this MXML format (depicted in Figure 2) is available at http://www.processmining.org/WorkflowLog.xsd.

As can be seen in Figure 2, an event log (field *WorkflowLog*) has the execution of one or more processes (field *Process*), and optional information about the source program that generated the log (field *Source*) and additional data elements (field *Data*). Every process (field *Process*) has zero or more cases or process instances (field *ProcessInstance)* [3]. Similarly, every process instance has zero or more tasks (field *AuditTrailEntry*). Every task or audit trail entry (ATE) should at least

[2] More information about the Extensible Markup Language (XML) can be found in [1].

[3] In the rest of this document, the words "execution", "case" and "process instance" are interchangeable.



(b) Transactional model for EventType

(a) Process log XML format

Fig. 2. The visual description of the schema for the Mining XML (MXML) format.

have a name (eld *WorkflowModelElement*) and an event type (eld EventType). The event type determines the state of the tasks. There are 13 supported event types: schedule, assign, reassign, start, resume, suspend, autoskip, manualskip, withdraw, complete, ate_abort, pi_abort and unknown. The other task elds are optional. The *Timestamp* eld supports the logging of time for the task. The *Originator* eld records the person/system that performed the task. The *Data* eld allows for more logging of additional information. More details about the MXML format can be found in [6].

Mapping the MXML format to the three mining perspectives, we see that the *control-flow* perspective mainly focuses on the Work owModelElement, the EventType and the Timestamp [4] elds. The *organizational* perspective chie y depends on the Originator eld. The *case* perspective especially relies on the extra Data elds.

Note that in CPN Tools the process corresponds to the CP-net, the tasks (or ATEs) are the transitions in the CP-net, and each simulation of the CP-net corresponds to the creation of a process instance.

3 Extending a CP-net to Produce MXML Event Logs

This section shows how to annotate a CP-net with the ML functions that we created to log MXML les. The ML functions can be downloaded from the section

[4] When the Timestamp eld is not logged, the sequence in which the tasks appear in the log is used to infer their order of execution.

"Tools"[5] in [2]. To illustrate the extension process, we use the CP-net in Figure 3. The extension of this CP-net involves editing its declaration and transition inscriptions.

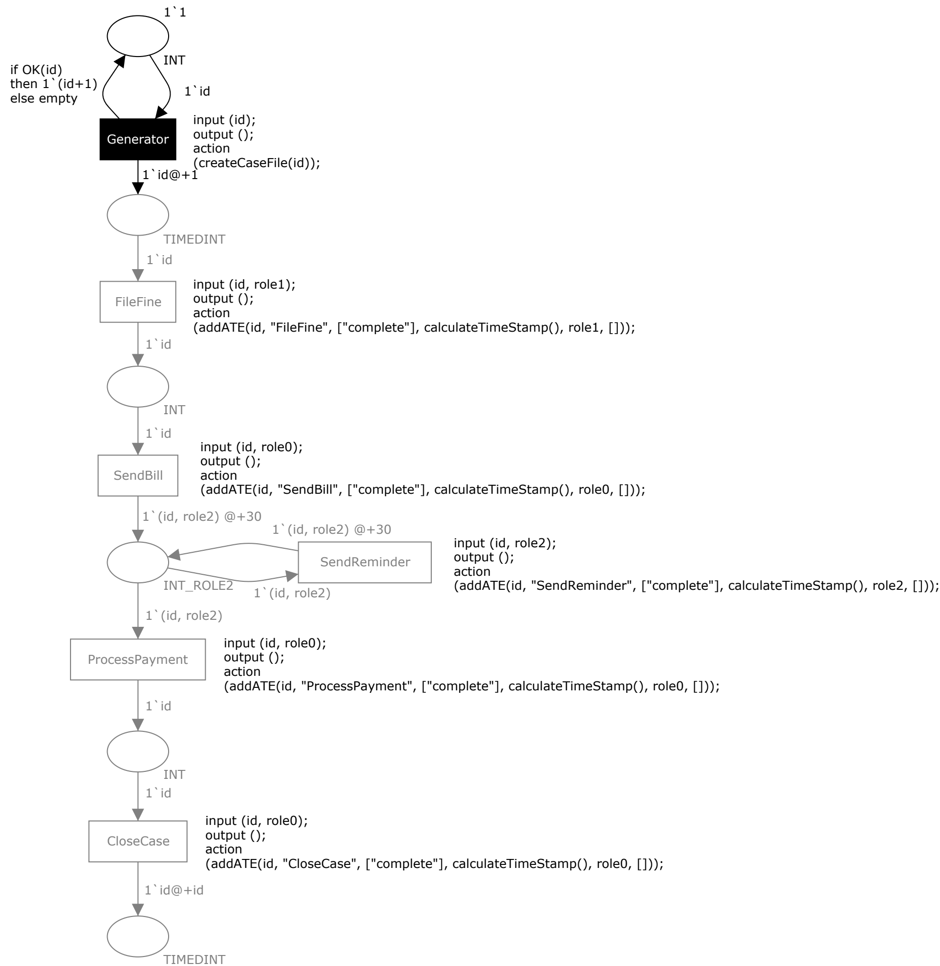


Fig. 3. Example of an extended CP-net for the process described in Figure 1. The highlighted transition "Generator" and the input/output/action inscriptions were added during extension of the CP-net.

CPN Declarations The declarations of a CP-net need to be modi ed to import the ML functions to log transitions. These functions are in the le *logging-FunctionsMultipleFiles.sml*. The ML functions in this le use two constants:

[5] Search for the link to the le "CPNToolsConverter.zip".

FILE and *FILE_EXTENSION*. The constant *FILE* sets the location and the name pre x of the MXML les that the CP-net will create for every case it executes. The constant *FILE_EXTENSION* sets the extension that these created les have. For instance, to (1) create the XML log les for every case at the subdirectory *logs* from the directory where the CP-net is located and name every log with the pre x *logsCPN*; and (2) assign the extension *.cpnxml* to every created log, the following should be declared:

1. *val FILE = “./logs/logsCPN”*
2. *val FILE_EXTENSION = “.cpnxml”*
3. *use “loggingFunctionsMultipleFiles.sml”*;

Note that the use of the le *loggingFunctionsMultipleFiles.sml* **must be declared after** declaring the constants *FILE* and *FILE_EXTENSION*. Table 3 shows what these declarations look like in the CP-net of Figure 3. Additionally, be aware that ML is case sensitive and the subdirectories provided in the constant *FILE* should already exist.

```
(* Standard declarations *)
   colset E = with e;
   colset INT = int;
   colset BOOL = bool;
   colset STRING = string;
(* Net declarations *)
   colset TIMEDINT = int timed;
   colset ROLE0 = subset STRING with [“system”];
   colset ROLE1 = subset STRING with [“Anne”];
   colset ROLE2 = subset STRING with [“Mary”, “John”];
   colset INT_ROLE2 = product INT * ROLE2 timed;
   var id: INT;
   var role0: ROLE0;
   var role1: ROLE1;
   var role2: ROLE2;
   fun OK(id) =
     if id < 1000 then true
     else false;
(* Log declarations *)
   val FILE = “./logs/logsCPN”
   val FILE_EXTENSION = “.cpnxml”
   use “loggingFunctionsMultipleFiles.sml”;
```

Table 2. Declarations for the CP-net in Figure 3. The declarations in **bold** were used to extend the model to log MXML les.

CPN Transitions Once the declarations of the CP-net have been updated, the *input/output/action* inscriptions of transitions can be modi ed to invoke the

logging functions. The CP-net will create a partial MXML log for *every* case that it executes. In the example in Figure 3, the transition *Generator* creates the unique case identifiers. After a partial MXML log has been created, the transitions (or tasks) executed for a case are written to its partial MXML log. Thus, two ML functions are provided: createCaseFile and addATE.

The function createCaseFile(int caseId) opens the log file for a case. This function should be invoked only *once per case*, and before the function addATE is invoked for this same case. The transition *Generator* in Figure 3 illustrates how to use the function createCaseFile. Note that this function receives an *integer* (the case identifier!) as input.

The function addATE(int caseId, String transitionName, ListOfStrings eventType, StringTimestamp timestamp, String originator, ListOfStrings data) logs the execution of a transition to the log of a case. For instance, the transition *FileFine* in Figure 3 has an invocation of this function. The parameters of the function addATE are:

1. caseId: *integer* that uniquely identifies a case. In Figure 3, the case id is given by the variable *id*.
2. transitionName: *string* that has the name of the transition to log. Note that all strings in ML should be in quotes (“”). In Figure 3, the task name for transition *FileFine* is *“FileFine”*.
3. eventType: *list of strings*. If the event type is supported, the list should contain a *single* element and have the format [**name**], where *name* in {“assign”, “withdraw”, “reassign”, “start”, “suspend”, “resume”, “complete”, “autoskip”, “manualskip”, “pi_abort”, “ate_abort”} . If the event type is *unknown*, this list should have *two* elements and the format [**“unknown”, “name”**], where *name* is the unknown event type name. In Figure 3, the event type for transition *FileFine* is [*“complete”*].
4. timestamp: *string* that represents the date and time in which the task was executed. The timestamp has the XML pre-defined format *dateTime* [5]. For instance, a valid timestamp string is “2005-06-30T14:55:00.000+01:00”. The function calculateTimeStamp() is provided to automatically calculate the timestamp field based on the current time (in minutes) of a CP-net. However, this is a relative time because it always starts at the year 0. The function calculateTimeStamp() is included in the file *loggingFunctionsMultipleFiles.sml*. In Figure 3, the function calculateTimeStamp() was used to provide the timestamp.
5. originator: *string* that has the name of the originator (person or system) that executed the transition. In Figure 3, the user with *role1* executed the task *FileFine*.
6. data: *list of strings* containing the additional data fields that may be associated to a task. This list must have the format [attributeName1, attributeValue1, attributeName2, attributeValue2, ..., attributeNameX, attributeValueX]. In Figure 3, no extra data is associated to the task *FileFine*.

The parameters timestamp, originator and data can be empty (see optional fields in Figure 2). The first two are empty if the string “” is given as input. The *data* parameter is empty when [] is given as input.

The simulation of the extended CP-net will create the partial MXML log files whose aggregation is described in the next section.

4 Final Log Aggregation

Once the CP-net has been simulated with logging extensions enabled, the generated log output has to be correctly aggregated and converted, such that it can be read and interpreted by mining tools like the ProM framework. As explained in Section 3, simulating a CP-net with logging extensions included and enabled will yield one log file per invocation of createCaseFile, including all logged events. The task of log aggregation is now to combine these files as process instances within one single MXML log file, representing all logs for the respective process model (i.e., the simulated CP-net).

This aggregation pass has been implemented as the “CPN Tools” import plug-in for the ProM_{import} framework [2]. This framework has been developed to serve as a common environment for converting and importing logs from all kinds of information systems, and subsequently creating MXML compliant log files from them. The actual procedures for importing logs from a specific source system can be implemented as plug-ins, which can be dynamically loaded and removed from the running framework. The framework has a common graphical user interface for configuring and controlling import plug-ins, keeps all configuration data persistent, and provides a set of useful classes which can be used by all import plug-ins in order to ease development.

As the implemented ML functions createCaseFile and addATE (cf. Section 3) write MXML compliant log fractions during simulation runs, all that is left to do is to generate a common enclosing log file. For each file created by the simulation runs, a process instance is created within that common log file. In this process instance, all log events from one input file are added in their given order. In the end, the output log file thus includes all data from the simulation run logs in an aggregated manner, ready to be analyzed by mining tools like the ProM framework.

Using the “CPN Tools” import plug-in is fairly straightforward: The configuration pane (see Figure 4) has two filter properties that allow the user to select the input directory for the partial MXML files and the suffix of these files. After running the simulation passes in CPN Tools, the files created for each trace of one process model can be selected here. As this plug-in is geared towards aggregating logs that have resulted from executing one process model, it will accordingly write one single output log file. In this, the import framework provides the choice between writing the log into a compressed ZIP file, or as plain XML file. In either case, the resulting file can subsequently be loaded from within, for instance, the ProM framework, where a multitude of Process Mining plug-ins are available for analysis.

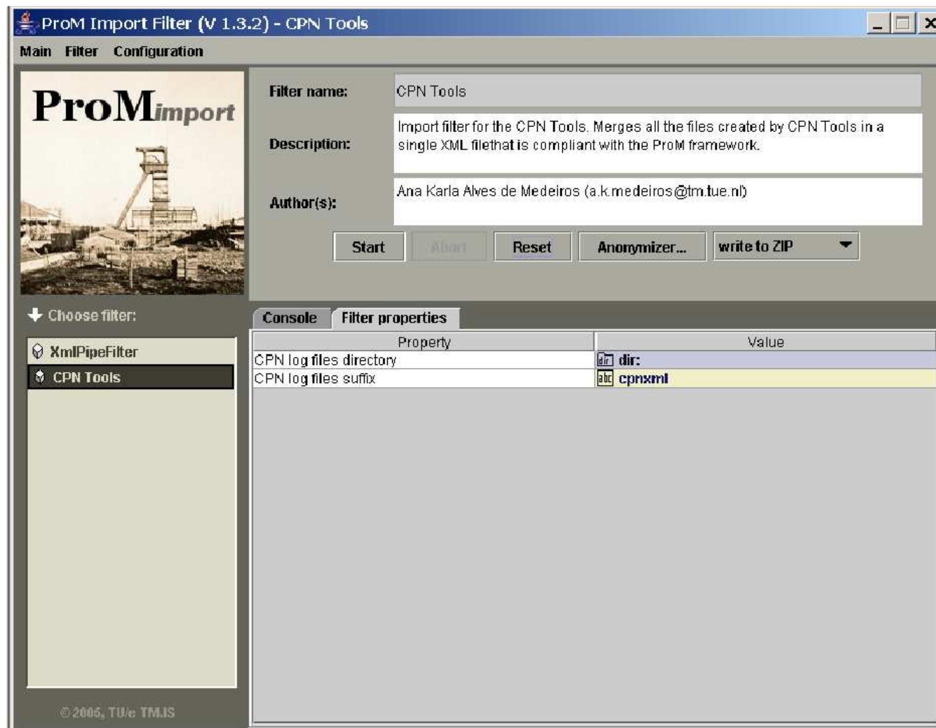


Fig. 4. Screenshot of the CPN Tools plug-in. This plug-in is implemented in the ProM$_{import}$ framework.

C:\TEMP\Temporary Directory 1 for fineCPN2005.zip\CpnToolslog.xml - Microsoft Internet Explorer

File Edit View Favorites Tools Help


```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <WorkflowLog xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://tmitwww.tm.tue.nl/research/processmining/WorkflowLog.xsd" description="Log
    extracted from CPN Tools">
    <Source program="CPN Tools" />
  - <Process id="CpnToolsLog" description="Log file created in CPN Tools">
    - <ProcessInstance id="1" description="">
      - <AuditTrailEntry>
          <WorkflowModelElement>FileFine</WorkflowModelElement>
          <EventType>complete</EventType>
          <Timestamp>0000-01-01T00:01:00.000+01:00</Timestamp>
          <Originator>Anne</Originator>
        </AuditTrailEntry>
      - <AuditTrailEntry>
          <WorkflowModelElement>SendBill</WorkflowModelElement>
          <EventType>complete</EventType>
          <Timestamp>0000-01-01T00:01:00.000+01:00</Timestamp>
          <Originator>system</Originator>
        </AuditTrailEntry>
      - <AuditTrailEntry>
          <WorkflowModelElement>ProcessPayment</WorkflowModelElement>
          <EventType>complete</EventType>
          <Timestamp>0000-01-01T00:31:00.000+01:00</Timestamp>
          <Originator>system</Originator>
        </AuditTrailEntry>
      - <AuditTrailEntry>
          <WorkflowModelElement>CloseCase</WorkflowModelElement>
          <EventType>complete</EventType>
          <Timestamp>0000-01-01T00:31:00.000+01:00</Timestamp>
          <Originator>system</Originator>
        </AuditTrailEntry>
      </ProcessInstance>
    - <ProcessInstance id="10" description="">
      - <AuditTrailEntry>
          <WorkflowModelElement>FileFine</WorkflowModelElement>
          <EventType>complete</EventType>
          <Timestamp>0000-01-01T00:01:00.000+01:00</Timestamp>
          <Originator>Anne</Originator>
        </AuditTrailEntry>
      - <AuditTrailEntry>
          <WorkflowModelElement>SendBill</WorkflowModelElement>
```


Done   My Computer

Fig. 5. Excerpt of the MXML log that was aggregated for a simulation of the CP-net in Figure 3.

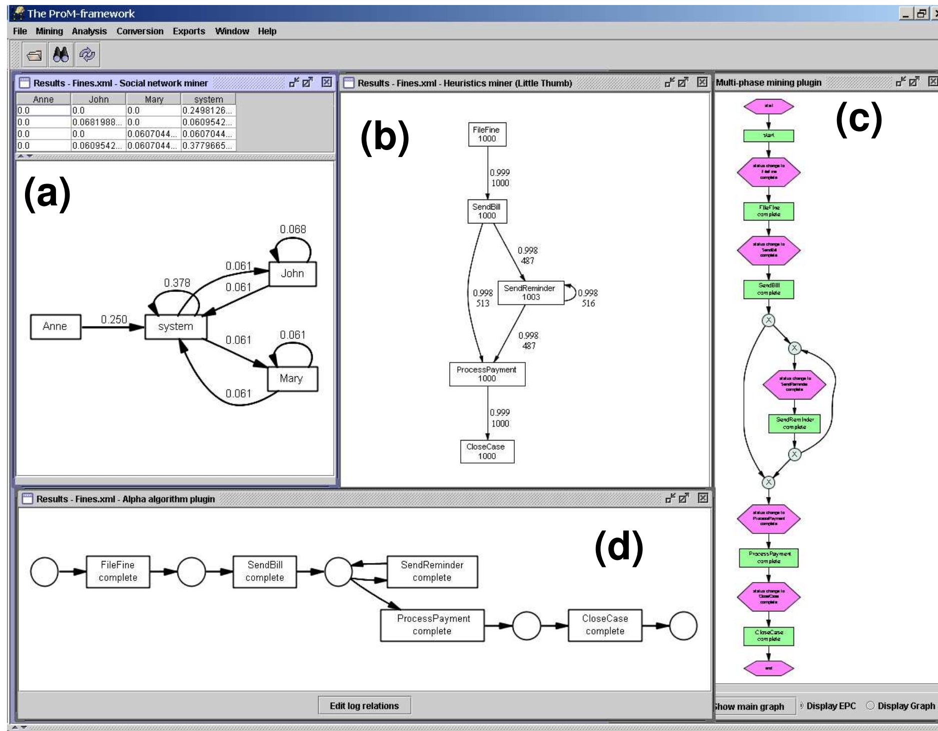


Fig. 6. Screenshot of the mined Petri net for the in Figure 5. The shown mining plug-ins are: (a) *Social network miner*, (b) *Heuristics miner*, (c) *Multi-phase mining* and (d) *Alpha algorithm*. The *Social network miner* plug-in can mine the *organizational perspective* (cf. Section 1) of an event log. Here we show the *handover of work* setting, considering only direct succession. Note that the users "John" and "Mary" never transfer work to each other. This is compatible with the CP-net in Figure 3. The other plug-ins in this gure can mine the *control-flow perspective* of the event log. As expected, all mined control- ow structures are like the CP-net in Figure 3.

As an illustration, Figure 5 shows an excerpt of the MXML log that was aggregated for the simulation of the CP-net in Figure 3. Additionally, Figure 6 shows a screenshot with the results of applying four di erent ProM mining plugints to this MXML log.

5 Conclusions and Future Work

This paper demonstrates how to bene t from the CPN Tools simulation capabilities to build event logs that can be used in the process mining research. The extension to CPN Tools consisted of implementing (i) a set of ML functions to create log les and (ii) a ProM$_{import}$ plug-in. the ML functions can be called from the input/output/action transition inscriptions of a CP-net. When the CP-net is simulated, partial logs are created. These partial logs are bundled into a single MXML log by using the ProM$_{import}$ CPN Tools plug-in. The resulting log can be mined by mining tools like the ProM framework. All the tools/ les necessary for extending a CP-net to create MXML logs can be found at http://www.processmining.org.

As future work, we are interested in creating a repository for MXML logs. This repository will contain logs with di erent properties. In addition, such a repository will allow researchers to test and compare their algorithms in a uni ed way and with logs that others may have created.

Acknowledgments

The authors would like to thank Wil van der Aalst for suggesting the use of CPN Tools to create random event logs in the MXML format. They also would like to thank Wil van der Aalst and Boudewijn van Dongen for the discussions about how to log from CPN Tools models. Last but not least, the authors would like to thank the anonymous reviewers for their useful remarks on how to improve this paper readability.

References

1. Extensible Markup Language (XML). http://www.w3.org/XML/.
2. Process mining website. http://www.processmining.org.
3. W.M.P. van der Aalst and M. Song. Mining Social Networks: Uncovering interaction patterns in business processes. In J. Desel, B. Pernici, and M. Weske, editors, *International Conference on Business Process Management (BPM 2004)*, volume 3080 of *Lecture Notes in Computer Science*, pages 244–260. Springer-Verlag, Berlin, 2004.
4. W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A.J.M.M. Weijters. Work ow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003.
5. P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes (Second Edition). http://www.w3.org/TR/xmlschema-2/, 2004.

6. B.F. van Dongen, A.K.A. de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. The ProM framework: A new era in process mining tool support. In G. Ciardo and P. Darondeau, editors, *International Conference on Applications and Theory of Petri Nets (ATPN 2005)*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer-Verlag, Berlin, 2005.
7. B.F. van Dongen and W.M.P. van der Aalst. EMiT: A process mining tool. In J. Cortadelle and W. Reisig, editors, *International Conference on Applications and Theory of Petri Nets (ATPN 2004)*, volume 3099 of *Lecture Notes in Computer Science*, pages 454–463. Springer-Verlag, Berlin, 2004.
8. J. Herbst and D. Karagiannis. Work ow mining with inwolve. *Computers in Industry*, 53(3):245–264, 2004.
9. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1.* EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1997.
10. J. Herbst M. Hammori and N. Kleiner. Interactive work ow mining. In B. Pernici J. Desel and M. Weske, editors, *International Conference on Business Process Management (BPM 2004)*, volume 3080 of *LNCS*, pages 211–226. Springer Verlag, January 2000.
11. G. Schimm. Mining exact models of concurrent work ows. *Computers in Industry*, 53(3):265–281, 2004.
12. A.J.M.M. Weijters and W.M.P. van der Aalst. Rediscovering Work ow Models from Event-Based Data using Little Thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.

Distributed and Modular State Space Exploration for Timed Petri Nets*

C. Lakos[1] and L. Petrucci[2]

[1] University of Adelaide
Adelaide, SA 5005
AUSTRALIA
Charles.Lakos@adelaide.edu.au
[2] LIPN, CNRS UMR 7030, Université Paris XIII
99, avenue Jean-Baptiste Clément
F-93430 Villetaneuse, FRANCE
petrucci@lipn.univ-paris13.fr

Abstract. This paper extends modular state space construction for concurrent systems to cater for timed systems. It identifies different forms of timed state space and presents algorithms for computing them. These include uniprocessor algorithms inspired by conservative and optimistic approaches to discrete event simulation, and also a distributed algorithm. The paper includes performance results for a simple case study.

1 Introduction

State space exploration is a convenient technique for the analysis of concurrent and distributed systems. Its chief disadvantage is the so-called state space explosion problem where the size of the state space can grow exponentially in the size of the system.

One way to alleviate the state space explosion problem is to use modular analysis, which takes advantage of the modular structure of a system specification. Here, the internal activity of the modules is explored independently rather than in an interleaved fashion. Experiments have indicated [11] that modular analysis can produce a significant reduction in the size of the state space, particularly for systems where the modules exhibit strong cohesion and weak coupling.

This paper extends modular state space exploration [3, 9] to timed systems. The introduction of time raises an interesting challenge since, by its very nature, time is a global entity rather than having local significance. This paper examines whether modular analysis algorithms can be modified to cater for time and still reap the benefits already demonstrated for untimed systems [9, 11].

This work applies to Coloured Petri nets as well as to Place/Transition nets. For the sake of readability, we present the algorithms for Place/transition nets. Their extension to CP-nets is straightforward.

* This work is supported by the Australian Research Council Linkage International grant LX04544639, and the French-Australian Science and Technology programme FR040062.

The paper is organised as follows. Section 2 presents preliminary definitions of timed and modular Petri Nets. Section 3 provides uniprocessor algorithms for modular state space exploration of timed systems, while Section 4 provides a distributed algorithm. In section 5 we present some experimental results obtained by applying the timed modular state space technique to a case study, and compare them to the flat state space. The conclusions are presented in Section 6.

2 Background

We commence with some modified definitions of Petri Nets and their state spaces. Adapting the notation of Jensen [6], we write $5@2+2@3$ for 5 tokens (or values) available from time 2 and 2 tokens available from time 3. If we remove from this multiset, 3 tokens at time 4, we could end up with $4@2$ or $2@2+2@3$ or some other combination. We capture these notions formally as follows:

Definition 1. *A* **time set** *TS is a set of numeric values. For much of this paper, the time values will be integral, i.e. $TS = \mathbb{N}$, but in general they could be positive real numbers, i.e. $TS = \mathbb{R}^+$. Markings and arc inscriptions will be given by multisets over TS, written as TS_{MS}. We also extend operations over multisets to take time into account:*

1. *Given $m_1, m_2 \in TS_{MS}$, $m_1 \geq_T m_2$ iff $m_2 = \emptyset$ or $\exists m'_1, m'_2 \in TS_{MS}, m_{1i}, m_{2i} \in TS$ such that $m_1 = m'_1 + 1@m_{1i}$ and $m_2 = m'_2 + 1@m_{2i}$ and $m_{1i} \leq m_{2i}$ and $m'_1 \geq_T m'_2$.*
2. *Given $m_1, m_2, m_3 \in TS_{MS}$, $m_1 -_T m_2 = m_3$ iff $m_2 = \emptyset$ and $m_1 = m_3$ or $\exists m'_1, m'_2 \in TS_{MS}, m_{1i}, m_{2i} \in TS$ such that $m_1 = m'_1 + 1@m_{1i}$ and $m_2 = m'_2 + 1@m_{2i}$ and $m_{1i} \leq m_{2i}$ and $m'_1 -_T m'_2 = m_3$.*
3. *Given $m_1, m_2, m_3 \in TS_{MS}$, $m_1 +_T m_2 = m_3$ iff $m_1 + m_2 = m_3$.*
4. *Given $m \in TS_{MS}$ and $k \in TS$, $k +_T m = \sum_{m_i \in m} 1@(k + m_i)$ and $k -_T m = \sum_{m_i \in m} 1@(k - m_i)$.*

The comparison operator is interpreted as the multiset m_1 having elements which have been accessible for at least as long as the demands specified by m_2. In other words, it must be possible to pair elements of m_2 with elements of m_1 such that the elements of m_1 are less than those of m_2, i.e. they have been accessible longer than the requirements. This interpretation of comparison is then used to define subtraction (between timed multisets) — $m_1 -_T m_2$ is only defined if $m_1 \geq_T m_2$. In general, $m_1 -_T m_2$ is not uniquely defined unless we insist that the element m_{1i} is always chosen to be the maximum value that is less than the corresponding m_{2i}. This is the approach taken by Jensen [6] in order to ensure that $(m_1 -_T m_2) -_T m_3 = (m_1 -_T m_3) -_T m_2$ which is required for the diamond rule to hold. For completeness, we define addition of timed multisets but this is the same as multiset addition. Adding and subtracting multisets to scalars is similar to the scaling function of van der Aalst [1].

Definition 2. *A* **Timed Petri Net** *is a tuple $PN = (P, T, W, M_0)$, where P is a finite set of* **places**, *T is a finite set of* **transitions** *such that $T \cap P = \emptyset$, W*

is the **arc weight function** *mapping from* $(P \times T) \cup (T \times P)$ *into* TS_{MS}*, and* M_0 *is the* **initial marking***, namely a function mapping from* P *into* TS_{MS}*.*

For a Timed Petri Net, each token has a time attribute, which indicates the earliest time that it is accessible. The output arcs of a transition indicate the time delays for generated tokens — an output arc with the inscription 5@2 would indicate that 5 tokens are added to a place and they will become accessible 2 units of time in the future. We allow similar inscriptions on input arcs — an inscription 5@2 would indicate the consumption of 5 tokens which have been accessible since (at least) 2 units of time in the past.

It would be possible to specify time delays only on the input arcs or only on the output arcs. If time delays are only specified on output arcs, then the time at which a transition can fire depends solely on the accessibility of the tokens. If time delays are only specified on the input arcs, then the time at which a transition can fire depends on transition-specific information. Our approach has the advantage of symmetry and generality. By contrast, the approaches of van der Aalst [1] and Jensen [6] only specify delays on output arcs.

Definition 3. *A* **marking** *is a function* M *mapping from* P *into* TS_{MS}*. The set of all markings is denoted by* $\mathbb{M}$*. A transition* t *is* **time-enabled** *at time* k *in a marking* M*, denoted by* $M[t\rangle_k$*, iff* $\forall p \in P : M(p) \geq_T k -_T W(p,t)$*. When a transition* t *is enabled in a marking* M_1 *at time* k*, it may* **occur***, changing the marking* M_1 *to another marking* M_2*, defined by:* $\forall p \in P : M_2(p) = (M_1(p) -_T (k -_T W(p,t))) +_T (k +_T W(t,p))$*. This is denoted by* $M_1[t\rangle_k M_2$*. The set of markings* **reachable** *from a marking* M*, denoted* $[M\rangle$*, is given by the smallest set satisfying* $M \in [M\rangle$ *and* $M' \in [M\rangle \wedge M'[t\rangle_k M'' \implies M'' \in [M\rangle$*.*

Definition 4. *The* **timed state space** *for a Timed Petri Net* $PN = (P,T,W,M_0)$ *is a tuple* $TSS = (V,E)$ *where* $V = [M_0\rangle$ *and* $E = \bigcup_{m \in V} \{(m,t,m')_k \mid m[t\rangle_k m'\}$*.*

It is common to require that timed transitions fire at the earliest possible time of enabling. Accordingly, we define an earliest time state space.

Definition 5. *The* **earliest time state space** *for a Timed Petri Net* $PN = (P,T,W,M_0)$ *is a tuple* $ESS = (V,E)$ *where* $V = [M_0\rangle$ *and* $E = \bigcup_{m \in V} \{(m,t,m')_k \mid m[t\rangle_k m', \nexists k' < k : m[t\rangle_{k'}\}$*.*

Finally, we define a reduced earliest time state space as a graph where the transitions in conflict at any given marking all have the same time.

Definition 6. *The* **reduced earliest time state space** *for a Timed Petri Net* $PN = (P,T,W,M_0)$ *is a tuple* $RSS = (V,E)$ *where* $V = [M_0\rangle$ *and* $E = \bigcup_{m \in V} \{(m,t,m')_k \mid m[t\rangle_k m', \nexists t', k' < k : m[t'\rangle_{k'}\}$*.*

Note that t' could be t firing at an earlier time.

Note that the standard definition for state spaces for timed systems (as in Design/CPN [4] for example) are equivalent to our definition of a reduced earliest time state space. Unlike Jensen, however, we do not attach a time to a marking

(to indicate the time of firing of the last transition), and then require that subsequent transitions should have a greater or equal time. This constraint is imposed to ensure that time cannot *go backwards*. Thus, if a marking M_0 enables independent transitions t_1 at time 2 and t_2 at time 5, then our definition of *ESS* would allow t_1 to be fired at time 2 *followed by* t_2 at time 5, *or vice versa*. With Jensen's constraint, the first alternative would still apply, but the second would have t_1 firing at time 5 after t_2. Our approach is somewhat anomalous, but we retain it because the anomalies will be eliminated in forming a *RSS* — t_1 will not be allowed to fire *after* t_2. Furthermore, for a modular system, it will be necessary to consider an *ESS* as a stepping stone to a *RSS*, and the standard unfolding of an earliest time modular state space will produce an *ESS*.

It should also be noted that, for a timed net, the *ESS* is a subgraph of the *TSS* since some edges are removed, and this may also make some nodes unreachable. Similarly, the *RSS* is a subgraph of the *ESS*. A *partially reduced earliest time state space* is also possible. This would be a subgraph of the *ESS* and a supergraph of the *RSS*.

Definition 7. *A* **Timed Modular Petri Net** *is a pair $MN = (S, TF)$, where:*

1. *S is a finite set of* **modules** *such that:*
   - *Each module, $s \in S$, is a Timed Petri Net: $s = (P_s, T_s, W_s, M_{0_s})$.*
   - *The sets of nodes corresponding to different modules are pair-wise disjoint: $\forall s_1, s_2 \in S : [s_1 \neq s_2 \Rightarrow (P_{s_1} \cup T_{s_1}) \cap (P_{s_2} \cup T_{s_2}) = \emptyset]$.*
   - *$P = \bigcup_{s \in S} P_s$ and $T = \bigcup_{s \in S} T_s$ are the sets of all places and all transitions.*
2. *$TF \subseteq 2^T \setminus \{\emptyset\}$ is a finite set of non-empty* **transition fusion sets**.

The above definition of a Timed Modular Petri Net is identical to existing definitions [3, 9] except for the introduction of time. Each module is a Timed Petri Net, and the modules interact via transition fusion — the elements of a fusion set fire as a single transition.

3 Modular Timed State Space Exploration

In this section we present two algorithms for modular state space exploration for a uniprocessor. (In section 4 we consider an algorithm suitable for a distributed environment.) The first algorithm is based on a conservative approach, which only explores transitions if their firing is consistent with the *RSS*. The second algorithm is based on an optimistic approach, which explores transitions if their firing is consistent with the *ESS*, with reduction left till later. These algorithms are consistent with the distinction between conservative and optimistic algorithms for distributed discrete event simulation [5]. Before handling the modular cases, we first present the algorithms for a flat timed net.

In this paper, we focus on state spaces built with a predetermined time limit, as computed by some tools such as Design/CPN. To remove this limitation, it would be necessary to construct classes of timed markings, as in e.g. [2]. As we

```
 1: TS system_limit ←??
 2: set Waiting ← ∅
 3: NODE.ADD(M_0)
 4: repeat
 5:    for all M ∈ Waiting do
 6:       Waiting ← Waiting \{M}
 7:       for all t ∈ T, ∃k ≤system_limit: M[t⟩_k M′ and ∄k′ < k : M[t⟩_k′ do
 8:          NODE.ADD(M′)
 9:          ARC.ADD(M, t, M′)_k
10:       end for
11:    end for
12: until stable
```

Fig. 1. Algorithm for earliest time state space of a timed net.

aim to construct a tool which might, in a distributed version, have the underlying structure of the distributed state space construction from [8], this limitation is not yet an issue.

3.1 Algorithms for a flat net

An algorithm to generate an earliest time state space for a timed net is given in Fig. 1. It maintains a set `Waiting` of as-yet unexplored markings. At each iteration of the *repeat* loop, the current elements of `Waiting` are removed and examined for enabled transitions. As usual, function NODE.ADD(M') adds a node labelled with M' to the graph and state M' to set `Waiting`, provided it does not already exist. Similarly, ARC.ADD$(M, t, M')_k$ adds an arc to the graph. The algorithm terminates when stability is reached, i.e. when `Waiting` is empty.

The algorithm includes a time limit (called `system_limit`) beyond which we do not explore transitions, and we always pick the earliest time at which a transition is enabled. It is these two aspects which differentiate this algorithm from the traditional algorithm for reachability analysis of an untimed net.

More significant modifications are required to generate the reduced earliest time state space. If we restrict our attention to integral time, then we can maintain a variable (called `system_time`) which is the time for which we are prepared to consider transition enablings. Essentially, the *repeat* loop of lines 4-12 of Fig. 1 can be nested within a loop that increments `system_time` at each iteration. This guarantees that in a given marking, we will consider the transitions that can fire earliest. The problem now is that the markings in the set `Waiting` cannot be discarded immediately because, while they may not currently enable a transition, they may enable a transition at some future time.

Fig. 2 contains our modified algorithm for producing a reduced earliest time state space. States are only removed from set `Waiting` once we have found an enabled transition. If a state does *not* enable any transition at any time in the future, then it will remain in set `Waiting` forever, which is clearly inefficient.

```
1: TS system_limit ←??
2: TS system_time ← 0
3: set Waiting ← ∅
4: NODE.ADD(M_0)
5: while system_time ≤ system_limit do
6:   repeat
7:     for all M ∈ Waiting do
8:       for all t ∈ T, M[t⟩_{system_time} M' do
9:         NODE.ADD(M')
10:        ARC.ADD(M, t, M')_{system_time}
11:        Waiting ← Waiting \{M}
12:      end for
13:    end for
14:  until stable
15:  system_time ← system_time + 1
16: end while
```

Fig. 2. Algorithm for reduced earliest time state space of a timed net.

3.2 Modular algorithms

In order to highlight the issues pertinent to the modular analysis of timed systems, we consider the simple example of Fig. 3. The transition input arcs indicate delays on tokens. Transitions t_1, t_2 and t_3 are local transitions, while transition *tf* is fused, with the occurrence in module A needing to synchronise with one of the occurrences in module B. The corresponding local state spaces and the synchronisation graph are shown in Fig. 4, where the states indicate the marked places and the token timestamp(s).

In modular analysis, we explore the local state space of each module. The synchronisation graph captures the synchronisation points between the modules. The states of the synchronisation graph are system states, each of which is a tuple of module states. Thus, the state labelled *A1,0 B1,0* corresponds to the system state with module A in state *A1* at time *0*, and with module B in state *B1* at time *0*. The arcs of the synchronisation graph are labelled with the

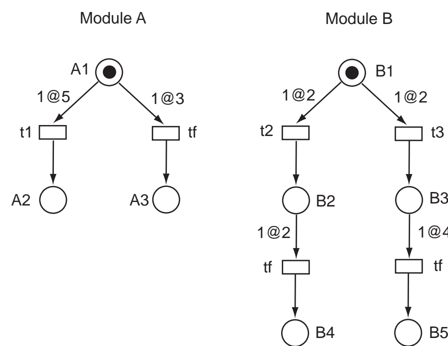


Fig. 3. Example of two timed modules

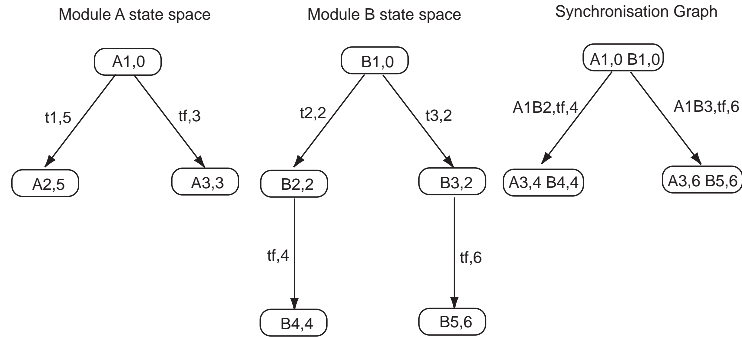


Fig. 4. Modular state space for two timed modules

fused transitions together with their time of firing. Since the fused transition will normally fire only after some internal activity of the modules, the arcs are also labelled with the local states which enable the fused transition. Thus, the arc labelled *A1B2,tf,4* indicates that transition *tf* can fire at time *4* when module A is in state *A1* and module B has reached state *B2*. Thus, while module A can fire *tf* at time *3*, it needs to wait till time *4*, when module B is also ready.

The fact that a system state consists of a tuple of module states also means that the state of one module may be combined with multiple other states, and decisions about reduced earliest time state spaces cannot be made merely at the local level. In the example, we cannot tell whether transition t_1 is preempted by transitions t_2 and t_3. Similarly, while we can determine if a local transition preempts a fused transition (in the same module), the converse is not possible. Thus, in module A, transitions t_1 and *tf* are in conflict. In the local state space, *tf* could preempt t_1, but *tf* needs to synchronise with *tf* in module B, which delays its firing time. With a short delay, *tf* may preempt t_1; with a longer delay *tf* may be preempted by t_1.

In other words, questions about which transition preempts another can only be finalised in an unfolded state space as shown in Fig. 5. The broken arcs indicate transitions that are preempted in the unfolded state space. Thus, the local state space needs to be an earliest time state space rather than a reduced earliest time state space.

3.3 Conservative Algorithm

The algorithm of Fig. 2 is now adapted to cater for modular analysis. The state space we construct is the timed extension of modular state spaces [3]. It consists of one local state space per module, describing only the module's behaviour, and a synchronisation graph capturing the interactions between modules.

Fig. 6 presents the algorithm for computing the synchronisation graph, while Fig. 7 presents the algorithm for computing the local state space of a module. As in Fig. 2, both the local state space and the synchronisation graph are computed

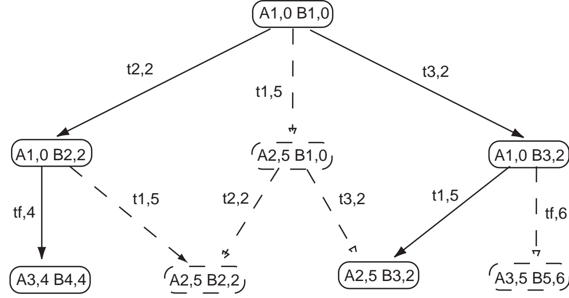


Fig. 5. Unfolded state space for two timed modules

in lock step — all activity at a time point is explored before time is advanced. This is the conservative approach.

Thus, in Fig. 6, `system_time` is incremented by one each time round the outer loop (up to `system_limit`) and all possible activity is investigated at each time point. Given that we are considering integral time, this means that we will consider transition firings at the earliest possible time. Note that the markings M are immediately removed from `Waiting` even though they may enable a fused transition some time in the future. This is dealt with by having the local state space exploration retain the local markings between calls.

The function EXPLORE (in Fig. 7) returns a set of triples — the first element is a synchronisation node, the second is the local marking reachable from the first, and the third is the fused transition which is locally enabled at this reachable marking. This approach is required because EXPLORE only examines reachable markings in time slices up to the current `system_time`. In other words, EXPLORE will examine the local markings reachable from a synchronisation node over several calls to the function, and it is necessary to relate the locally reachable marking to the synchronisation node from which it was derived.

Thus, each call to EXPLORE returns such a set of triples in variable $\texttt{trysynch}_i$. From the results returned for all the modules, we build global marking pairs (M, M'), where the first element of the pair is a node in the synchronisation graph, and the second element corresponds to a marking locally reachable from there. If the second element enables a fused transition at the current time, then we add the appropriate node and arc to the synchronisation graph.

The logic of function EXPLORE$(S_i, Current, local_limit_i)$ is based on the one for a flat net, presented in Fig. 2. The variables, such as $\texttt{trysync}_i$, are subscripted to emphasise their local significance, the variable $\texttt{local_time}_i$ replaces `system_time`, and the marking M is replaced by marking pairs (M, M_i'). Variable $\texttt{Waiting_future}_i$ holds the marking pairs that might enable a transition in the future, while $\texttt{Waiting_current}_i$ holds markings to be examined for transition enabling at the current time. Because these variables hold pairs of markings, NODE.ADD (at the local level) needs to have two arguments — it adds the second to the local state space, and the pair of markings to both $\texttt{Waiting_future}_i$ and $\texttt{Waiting_current}_i$. Finally, lines 18-20 consider the enabling of the local

```
1: TS system_limit ←??
2: TS system_time ← 0
3: set Waiting ← ∅
4: Node.Add(M_0)
5: while system_time ≤ system_limit do
6:    repeat
7:       ∀i: trysynch_i ← Explore(S_i, Waiting, system_time)
8:       Waiting ← ∅
9:       for all tf ∈ TF do
10:         for all (M, M') s.t. (M, M'_i, tf) ∈ trysynch_i ∨ M'_i = M_i ∧ tf ∩ T_i = ∅ do
11:            if M'[tf⟩_system_time M'' then
12:               Node.Add(M'')
13:               Arc.Add(M, (M', tf), M'')_system_time
14:            end if
15:         end for
16:      end for
17:   until stable
18:   system_time ← system_time + 1
19: end while
```

Fig. 6. Algorithm for synchronisation graph.

component of a fused transition. Where such an enabling is found, the relevant triple is added to variable $\texttt{trysynch}_i$, which is then returned as the function result. These possible synchronisations will be repeatedly returned until they are preempted by local transitions.

By exploring local activity at each consecutive time point, Explore caters for transitions preempting others at the local level. As noted in section 3.2, it cannot determine whether a local transition in one module preempts a local transition in another, nor whether a fused transition preempts a local transition. These issues can only be resolved in an unfolded state space.

3.4 Optimistic Algorithm

In view of the limitations of the conservative algorithm (noted in section 3.3), we now consider an alternative algorithm, inspired by the optimistic approach to distributed discrete event simulation [5]. Here, we acknowledge that we can only guarantee producing a reduced earliest time state space at the local level or at the global level but not local relative to global.

Consequently, each call to Explore examines the local state space up to `system_limit` and not just `system_time`. Since we explore all relevant activity from a synchronisation node, and not just some small time slice, we can simply return pairs giving the locally reachable marking and the fused transition which it enables. The modified algorithm is presented in Figs. 8 and 9. Note that if Explore is called multiple times with the same marking, then the locally reachable markings need to be recalculated or else some caching regime is required [10].

```
1: static TS local_time_i ← 0
2: static set Waiting_future_i ← ∅
3: set Waiting_current_i ← ∅
4: set trysynch_i ← ∅
5: ∀M ∈ Current: Node.Add(M, M_i)
6: Waiting_current_i ← Waiting_future_i
7: if local_time_i < local_limit_i then
8:    local_time_i ← local_time_i + 1
9: end if
10: repeat
11:    for all (M, M'_i) ∈ Waiting_current_i do
12:       Waiting_current_i ← Waiting_current_i \ {(M, M'_i)}
13:       for all t_i ∈ T_i \ TF, M'_i[t_i⟩_{local_time_i} M''_i do
14:          Node.Add(M, M''_i)
15:          Arc.Add(M'_i, t_i, M''_i)_{local_time_i}
16:          Waiting_future_i ← Waiting_future_i \ {(M, M'_i)}
17:       end for
18:       for all tf ∈ TF ∩ T_i, M'_i[tf⟩_{local_time_i} do
19:          trysynch_i ← trysynch_i ∪ {(M, M'_i, tf)}
20:       end for
21:    end for
22: until stable
23: return trysynch_i
```

Fig. 7. Algorithm for local state space — $\text{Explore}(S_i, Current, local_limit_i)$.

The optimistic approach simplifies both parts of the algorithm, but the cost is that it may produce larger local state spaces which will require further reduction in an unfolding of the modular state space.

4 Distributed Exploration

4.1 Basic Algorithm

The distributed algorithm in this section follows the paradigm of section 3.3 and relies on an architecture similar to that of [8]: several processes compute local parts of the state space while a single process handles the synchronisations.

Fig. 10 presents the algorithm to compute the local state space of module i. It focusses on the messages exchanged with the synchronisation process. The local generation, per se, is handled by function LOCAL_GENERATION, in Fig. 11.

The synchronisation process is also presented in two parts: the main one (Fig. 12) handles the communications with other processes and ensures the termination of all processes (if necessary), while function SYNCHRONISE, in Fig. 13, computes the synchronisation transitions enabled at the current time.

The different processes communicate by exchanging messages, as in table 1.

Let us now explain the different algorithms. There is one local process per module. Initially, the current time (variable current_time) is set to 0 — it will

```
1:  TS system_limit ←??
2:  set Waiting ← ∅
3:  Node.Add(M_0)
4:  repeat
5:     for all M ∈ Waiting do
6:        Waiting ← Waiting \{M}
7:        ∀i : trysynch_i ← Explore(S_i, M_i, system_limit)
8:        for all tf ∈ TF do
9:           for all M′ s.t. (M′_i, tf) ∈ trysynch_i ∨ M′_i = M_i ∧ tf ∩ T_i = ∅ do
10:             if M′[tf⟩_k M″ and ∄k′ < k : M′[t⟩_k′ then
11:                Node.Add(M″)
12:                Arc.Add(M, (M′, tf), M″)_k
13:             end if
14:          end for
15:       end for
16:    end for
17: until stable
```

Fig. 8. Algorithm for synchronisation graph.

Message	Local	Sync.	Meaning
SYNC	⟶		Send a marking
WAITING	⟶		Has sent markings, waits for reply
STATE	⟵		Synchronisation possible, send a marking
SENT	⟵		All new states have been sent
TIMEINC	⟵		No new synchronisation, hence increment time
STOPPED	⟶		Nothing enabled at current_time
STUCK	⟶		No future enabling
STOP	⟵		Computation finished

Table 1. Messages exchanged

be incremented when the synchronisation process sends the order to do so. The local process executes a loop until the synchronisation process instructs it to stop. This loop receives markings and calls function LOCAL_GENERATION to explore transitions for current_time. Two sets are used: Waiting_current contains the marking pairs that may enable a transition at the current time, and Waiting_future contains the marking pairs that enable a transition later on. Note that the function creating a node, Node.Add adds elements to both these sets.

After the local generation at current_time is completed, any states enabling a synchronised transition at the current_time are sent to the synchronisation process, followed by a WAITING message. The local process is then ready to receive new states obtained by synchronisation, which it will explore, or a TIMEINC message, indicating that no synchronisation was possible at the current time, and hence the time can be incremented. Otherwise, if the local process can neither fire a synchronised transition nor a local one at current_time, then either there are not enough tokens to enable a transition even in the future, in which

```
 1: set Waiting_i ← ∅
 2: set trysynch_i ← ∅
 3: Node.Add(M_i)
 4: repeat
 5:    for all M'_i ∈ Waiting_i do
 6:       Waiting_i ← Waiting_i \ {M'_i}
 7:       for all t_i ∈ T_i \ TF, M'_i[t_i⟩_k M''_i, ∄k' < k : M'_i[t_i⟩_k' do
 8:          Node.Add(M''_i)
 9:          Arc.Add(M'_i, t_i, M''_i)_k
10:       end for
11:       for all tf ∈ TF ∩ T_i, M'_i[tf⟩_k M''_i, ∄k' < k : M'_i[tf⟩_k' do
12:          trysynch_i ← trysynch_i ∪ {(M'_i, tf)}
13:       end for
14:    end for
15: until stable
16: return trysynch_i
```

Fig. 9. Algorithm for local state space — Explore($S_i, M_i, system_limit$).

case it tells the synchronisation process that it is STUCK, or else it says that it is STOPPED and waits until it is told to increment its time.

The synchronisation process also starts at current_time 0 and assumes that the status of all local processes is RUNNING. It performs a loop until all processes are STUCK or some maximum time system_limit has been reached. When this is the case, it tells all processes to STOP. In the loop, the synchronisation process receives and handles all messages sent by RUNNING local processes. These messages can either be states at which a synchronisation might be possible, or the new status of a local process. Function synchronise is the core of the process. For each synchronised transition tf, it checks if they can occur, it computes and sends the resulting states to the local processes concerned. The status of these processes is updated to RUNNING (which is really the case when all the necessary operations have taken place). When all synchronisations have been done, a SENT message is sent to all processes that were involved, so that they can pursue their local construction at the same current_time. Their markings are removed from the appropriate set of trysync states. If no synchronisation has occurred, the set of trysync states is reinitialised, the current_time is incremented, and all local processes are told to increment their time.

4.2 Correctness

To prove the correctness of the distributed algorithm, we will explain which markings and firings are handled by each part of each process.

The local processes construct only the local parts of the state space. Function local_generation uses a set Waiting_current of markings which possibly enable a transition at the current_time. All markings M of this set are dealt with one by one, and then deleted from Waiting_current. If M does not

```
 1: current_time ← 0
 2: repeat
 3:    repeat
 4:       message ← RECEIVE()
 5:       if message == STATE M then
 6:          NODE.ADD(M, M_i)
 7:       else if message == TIMEINC then
 8:          current_time ← current_time +1
 9:       end if
10:    until message ! = STATE M
11:    if message ! = STOP then
12:       sync ← LOCAL_GENERATION(current_time)
13:       if sync == now then
14:          SEND(WAITING i)
15:       else if sync == future then
16:          SEND(STOPPED)
17:       else
18:          SEND(STUCK)
19:       end if
20:    end if
21: until message == STOP
```

Fig. 10. Algorithm for local state space of module i.

enable any transition, independently of the time, it is also removed from the set `Waiting_future` of markings to be examined later. If M enables a transition at `current_time`, its successors are built and added to both sets of markings. As the transitions enabled from M are dealt with at `current_time`, M can be removed from `Waiting_future` because future transitions have been preempted.

Hence, if no synchronised transition is enabled, the local state space is generated up to `current_time`. The local process then waits for instructions from the synchronisation process, which are either to increment time, if a transition may still be enabled in the future, or stop otherwise.

If synchronised transitions are enabled at the `current_time`, the states enabling them are sent to the synchronisation process. Synchronisations may lead to new states which in turn may enable local transitions at `current_time`. Therefore, the local process waits for all states sent by the synchronisation process and then computes again the local parts at `current_time`.

The synchronisation process has the same `current_time` as the other processes. When it receives states, it tries to synchronise shared transitions, and eventually sends back the newly created states. If no synchronisation can occur, a message TIMEINC is sent to all local processes and `current_time` is incremented. The synchronisation process stops either when all processes are stuck or a maximum time is reached.

We conclude that transitions are handled in an *earliest firing time fashion.*

```
1: Waiting_current ← Waiting_future
2: sync ← none
3: for all (M, M'_i) ∈ Waiting_current do
4:    for all t_i ∈ T_i \ TF, M'_i[t⟩_current_time M''_i do
5:       Waiting_future ← Waiting_future \{(M, M'_i)}
6:       Node.Add(M, M''_i)
7:       Arc.Add(M'_i, t_i, M''_i)_current_time
8:    end for
9:    if ∃t, enabled_untimed(M'_i, t) then
10:      sync ← future
11:   else
12:      Waiting_future ← Waiting_future \{(M, M'_i)}
13:   end if
14:   for all tf ∈ T_i ∩ TF, M'_i[tf⟩_current_time do
15:      sync ← now
16:      send(SYNC M M'_i tf)
17:   end for
18:   Waiting_current ← Waiting_current \{(M, M'_i)}
19: end for
```

Fig. 11. Function LOCAL_GENERATION(current_time)

One of the key problems is the termination of the distributed algorithm. When a local process has finished its computation, it sends a message to the synchronisation process. The local process can either have sent markings on which a synchronisation is possible (it is then WAITING for an answer), or is STOPPED (nothing is enabled at the current time) or even STUCK (nothing will ever be enabled). When in state WAITING, if a synchronisation is possible, new markings will be sent to the local process, otherwise, a TIMEINC message will eventually be sent. The same message is sent when the process is STOPPED, so has markings to handle at a future time. Finally, when all local processes are STUCK or the time limit has been reached, all local processes receive a STOP message and end their computation. The synchronisation process stops as well.

5 Experiments

5.1 Case Study: the Timed Protocol

In this section, we apply the conservative uniprocessor algorithm to a variation of the timed protocol from [7], page 160, figure 5.4. This variant is depicted in Fig. 14. The main differences with the original model are that it is now split up into modules communicating via shared transitions, and that the random choices for time delays have been constrained so as to limit the state space explosion.

The case study is a simple timed protocol composed of 3 modules: a *sender*, a *receiver* and a *network* that connects them. The sender sends numbered packets, according to increasing sequence numbers, with a processing time of `Tsp` per

```
 1: ∀i, trysync[i] ← ∅
 2: ∀i, status[i] ← RUNNING
 3: current_time ← 0
 4: system_limit ←??
 5: NODE.ADD(M0)
 6: ∀i, SEND(i, STATE M0)
 7: ∀i, SEND(i, SENT)
 8: while ∃i, status[i] != STUCK ∧ current_time ≤ system_limit do
 9:    for all i do
10:       while status[i] == RUNNING do
11:          message ← RECEIVE(i)
12:          if message == SYNC M M'_i tf then
13:             trysync[i] ← trysync[i] ∪ {(M, M'_i, tf)}
14:          else
15:             status[i] ←message
16:          end if
17:       end while
18:    end for
19:    SYNCHRONISE(∀i trysync[i],current_time)
20: end while
21: ∀i, SEND(i, STOP)
```

Fig. 12. Algorithm for the synchronisation process.

packet. A packet can be retransmitted if an additional delay of `Twait` has elapsed before the corresponding acknowledgement is received. Transitions `Send packet` in the sender and the network are fused together. A packet can then either be lost (transition `Lose Packet`) or be transmitted to the receiver. The latter is achieved by transitions `Transmit Packet` in the network module and `Receive Packet` in the receiver, which are fused together. Upon reception of a packet, either it has the expected sequence number, in which case it is delivered, or else it is a duplicate, in which case it is discarded. In both cases, the receiver sends an acknowledgement indicating the next sequence number expected. Both transmission and loss of packets have a propagation time across the network of `5*ran'Trans()`. The transmission of acknowledgements from the receiver to the sender through the network is similar.

5.2 Implementation Issues

The conservative algorithm for modular state space construction from Section 3.3 was simulated in the *Maria* tool [10]. *Maria* supports modular state space construction but not time. However, it does support transition priorities, and these were used to simulate the timing mechanism.

All transitions from the model in Fig. 14 were allocated the highest priority of 9. In other words, if one of the model transitions (whether local to a module or fused) was enabled, then it would fire. Then, each module maintained a local

1: new_sync ← false
2: **for all** $tf \in TF$ **do**
3: **if** $\forall i$ synchronising on $tf, \exists(M, M'_i, tf) \in$ trysync[i] **then**
4: {a synchronisation is possible}
5: **for all** M' such that $\forall i, (M, M'_i, tf) \in$ trysync[i] : $M'[tf\rangle_{current_time}M''$ **do**
6: NODE.ADD(M'')
7: ARC.ADD$(M, (M', tf), M'')_{current_time}$
8: $\forall i$ synchronising on tf, SEND(i, STATE M''_i)
9: **end for**
10: new_sync ← true
11: $\forall i$ synchronising on tf, status[i] ← RUNNING
12: **end if**
13: **end for**
14: **if** new_sync **then**
15: **for all** i such that status[i] == RUNNING **do**
16: trysync[i] ← 0
17: SEND(i, SENT)
18: **end for**
19: **else if** current_time < system_limit **then**
20: current_time ← current_time +1
21: ∀ i, SEND(i, TIMEINC)
22: **end if**

Fig. 13. Function SYNCHRONISE($\forall i$ trysync[i],current_time)

timer consisting of a local time and a local time limit. Similarly, a global clock maintained the global time and the global time limit. At a priority of 2, the global clock was prepared to synchronise with the module timers. Finally, at the lowest priority, the global clock was allowed to advance.

The priorities thus ensured that if a transition in the model was enabled, then it would occur. Only when no such model transitions were enabled, would the global clock synchronise with the modules and/or advance to the next time point.

5.3 Experimental Results

The results are given for total time periods of 50 up to 300. The timeout period for the sender is `Tsp+Twait=58`. When `Trans={4}`, the propagation time is fixed at 20 for each direction and the round trip delay is `Tsp+Trp+2*5*ran'Trans()=56`. This just preempts the timeout retransmission of a message.

The experiments also vary the propagation delay with `Trans` selecting values from the ranges 4 to 4, 3 to 5 and 2 to 6. This gives corresponding propagation delays chosen from the sets $\{20\}$, $\{15, 20, 25\}$ and $\{10, 15, 20, 25, 30\}$.

The experimental results are given in Figs. 15, 16, and 17. Each one displays the number of nodes in the synchronisation graph (abbreviated *Synch*), the total number of nodes in the modular state space (abbreviated *Total*), and the

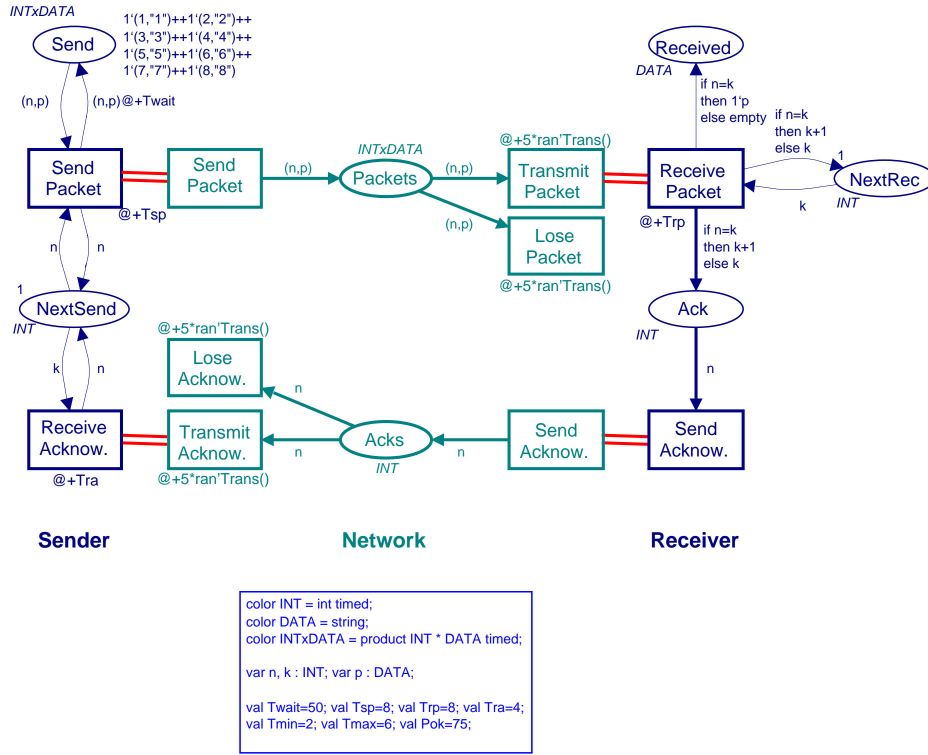


Fig. 14. Case study: the timed protocol

number of nodes in the flat state space (abbreviated *Flat*). It should be noted that the number of nodes was considered to be a more appropriate measure than the actual time, given that the algorithms are currently simulated in *Maria*. Secondly, it is worth noting that the total number of nodes for the modular state space is of the same order of magnitude as the number of nodes in the synchronisation graph. Thirdly, the results clearly indicate that as the variability in the system increases — whether due to the variability in the roundtrip delay, or in the time period over which the system is explored — so the benefits of the modular state space construction are increasingly apparent.

6 Conclusions and Future Work

This paper has extended the definition of state spaces to cater for timed systems. It has identified an earliest time state space, where all transitions enabled in a marking occur at the earliest time possible. It has also identified a reduced earliest time state space, where transitions enabled at a state are preempted by others which can occur earlier. The latter is the more traditional approach to

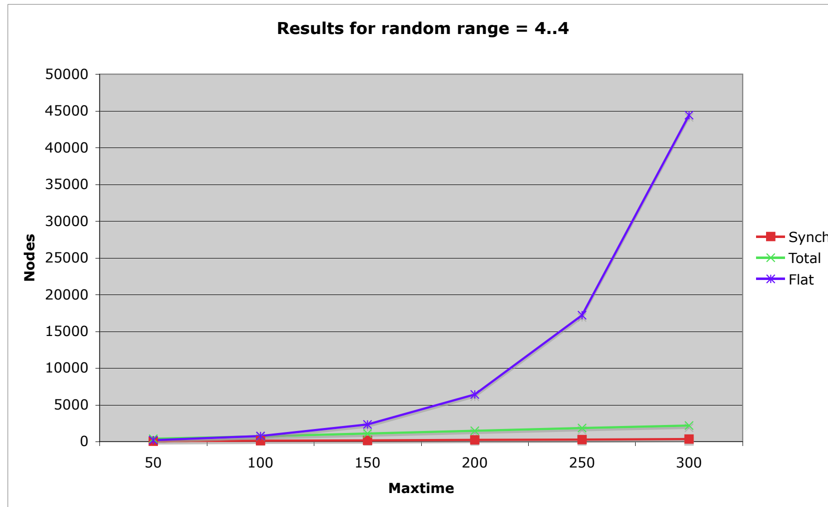


Fig. 15. Results for `Trans={4}`

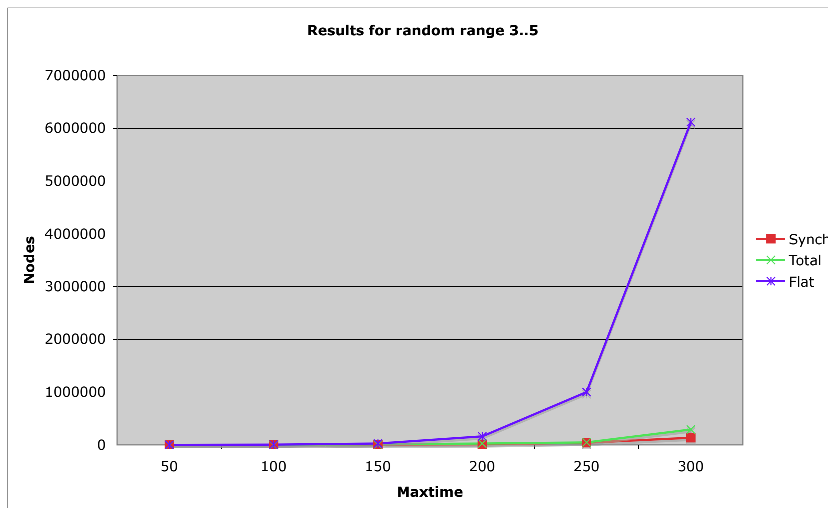


Fig. 16. Results for `Trans={3,4,5}`

timed state spaces. However, for modular systems, the independent analysis of the modules means that a reduced earliest time state space cannot be determined except in the unfolded state space. This being the case, it is appropriate to generate an earliest time state space as part of modular exploration.

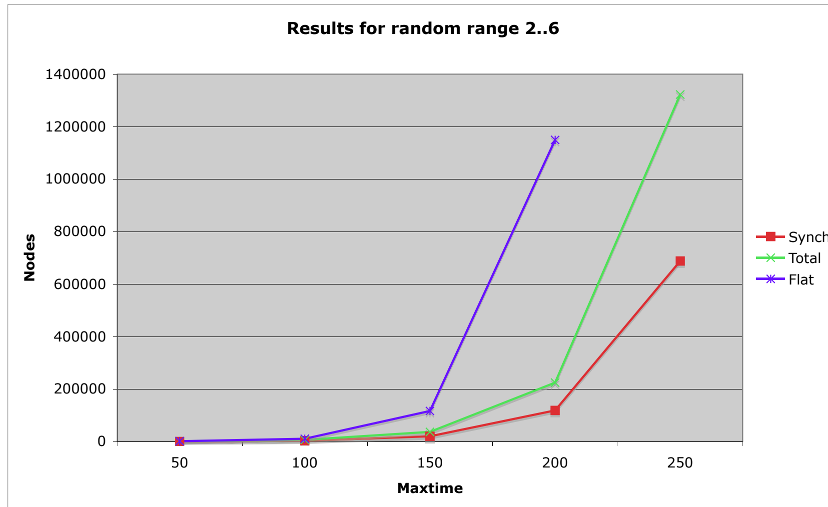


Fig. 17. Results for `Trans={2,3,4,5,6}`

The above raises the challenging question whether, as for earlier results [9], timing properties can be determined from the timed modular state space *without* unfolding. This is an important issue for future work.

This paper has presented two algorithms for a uniprocessor and one for a distributed environment which have generated the timed modular state space. The uniprocessor algorithms were inspired by the conservative and optimistic approaches to distributed discrete event simulation. The optimistic approach is based on generating the earliest time state space.

The formal definitions allow for dense time but the algorithms are defined for integral time. The algorithms can be modified to handle dense time by maintaining a schedule of pending events, e.g. in a priority queue. While this does not constitute a significant change, it would unnecessarily clutter the presentation of the algorithms. A more challenging issue for further work consists in adapting the algorithms to the generation of state class graphs, as in [2]. This would alleviate the maximum system time constraint.

Experimental results have been presented for the conservative uniprocessor algorithm. These demonstrate the value of modular analysis for timed systems. We have also derived preliminary results for the optimistic uniprocessor algorithm, and they indicate that this approach does reduce the amount of synchronisation between modules without necessarily resulting in a lot of superfluous exploration of the local state space. However, the optimistic algorithm does need to be paired with the use of a schedule of pending events, or else each module from each synchronisation node will increment time up to the time limit looking for possible enabled transitions. It will be important to perform further experiments to see whether the preliminary results for the optimistic algorithm carry

over to more realistic case studies, particularly when the schedule of pending events is incorporated. It will also be important to experiment with a number of generalisations and optimisations that we have identified, in order to see whether they are of value in fine-tuning the algorithms.

References

1. W. van der Aalst. Interval Timed Coloured Petri Nets and their Analysis. In M.A. Marsan, editor, *Internation Conference on the Application and Theory of Petri Nets*, volume 961 of *LNCS*, pages 453–472, Chicago, 1993. Springer.
2. G. Berthelot and H. Boucheneb. Occurrence graphs for Interval Timed Coloured Nets. In *Proc. 15th Int. Conf. Application and Theory of Petri Nets (ICATPN'1994), Zaragoza, Spain, June 1994*, volume 815 of *LNCS*, pages 79–98. Springer, June 1994.
3. S. Christensen and L. Petrucci. Modular analysis of Petri nets. *The Computer Journal*, 43(3):224–242, 2000.
4. Design/CPN *online*. `http://www.daimi.au.dk/designCPN`.
5. R.M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, 1990.
6. K. Jensen. *Coloured Petri Nets: Basic concepts, analysis methods and practical use. Volume 1: Basic Concepts.* Monographs in Theoretical Computer Science. Springer, 1992.
7. K. Jensen. *Coloured Petri Nets: Basic concepts, analysis methods and practical use. Volume 2: Analysis Methods.* Monographs in Theoretical Computer Science. Springer, 1994.
8. L. Kristensen and L. Petrucci. An approach to distributed state space exploration for coloured Petri nets. In *Proc. 25th Int. Conf. Application and Theory of Petri Nets (ICATPN'2004), Bologna, Italy, June 2004*, volume 3099 of *LNCS*, pages 474–483. Springer, June 2004.
9. C. Lakos and L. Petrucci. Modular analysis of systems composed of semiautonomous subsystems. In *Proc. 4th Int. Conf. on Application of Concurrency to System Design (ACSD'04), Hamilton, Canada, June 2004*, pages 185–194. IEEE Comp. Soc. Press, June 2004.
10. M. Mäkelä. Model Checking Safety Properties in Modular High-Level Neets. In W. van der Aalst and E. Best, editors, *24th International Conference on the Application and Theory of Petri Nets*, volume 2679 of *LNCS*, pages 201–220, Eindhoven, The Netherlands, 2002. Springer.
11. L. Petrucci. Cover picture story: Experiments with modular state spaces. *Petri Net Newsletter*, 68:Cover page and 5–10, April 2005.

Modeling the Case Handling Principles with Colored Petri Nets

Christian W. Günther and Wil M.P. van der Aalst

Department of Technology Management, Eindhoven University of Technology
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands
{c.w.gunther, w.m.p.v.d.aalst}@tm.tue.nl

Abstract. *Case handling* is a new paradigm for supporting flexible and knowledge intensive business processes. It is strongly based on data as the typical product of these processes. Unlike workflow management, which uses predefined process control structures to determine what *should* be done during a workflow process, case handling focuses on what *can* be done to achieve a business goal. While classical Petri nets are a good theoretical foundation for workflow management, the data-intensive nature of case handling does not allow for the abstraction of data. Therefore, we use *Colored Petri Nets* (CPNs) as a foundation for case handling. This paper models the key principles of case handling in terms of CPNs and uses state-space analysis and simulation to validate the concepts. Moreover, we also link the CPN model to *process mining* and show that it is possible to rediscover case handling processes based on the event log of a CPN simulation.

1 Introduction

Although workflow management concepts and technology [3, 13, 20, 21] have been applied in many enterprise information systems in the last decade, there appears to be a severe gap between the promise of workflow technology and what systems really offer. As indicated by many authors, workflow management systems (WFMSs) are too restrictive and have problems dealing with change [2, 5, 7, 9, 11, 12, 17, 18, 24].

Most of the workflow management systems consider workflows to be production processes, directly driven by structured process models. Such an approach is solely on the concept of routing, i.e., shifting work among resources based on causal relationship between activities. In [6] it is argued that for some applications the approach is suitable, but that for many other applications the use of control-flow as the primary enactment mechanism may result in the following four problems:

- Distributed handling of work requires its being partitioned, or *straight-jacketed into activities*. This fundamental principle, vital for the WFMS, can hardly ever comply with the way workers organize their tasks. Usually activities are performed at a far more fine-grained level than proposed by a process model.

– Typical WFMSs make no distinction between *authorization* and *distribution* of work, i.e., a worker is always offered to pick up *any task he is authorized to do*. This property leads to e.g. overcrowded in-trays for employees in higher positions, as their role typically includes many others.
– Strong control-flow orientation of WFM systems tends to blind out the context of tasks to be performed, most notably data created at earlier points in the process. This leads to the phenomenon of *context tunneling*, i.e., a worker has only access to data deliberately provided, a handicap that hinders efficiency and quality of work.
– The *push-oriented nature of routing* leaves hardly any decision to the user, so that he does not even have a means of making small ad-hoc adjustments to the process. That way workflows become unnecessarily inflexible, and small errors can spawn great problems.

To overcome this problem, we proposed *case handling* as a new paradigm for supporting knowledge-intensive business processes [1, 6]. This paradigm has proven its value in the tool FLOW*er* [1, 8, 22]. This tool is one of the most successful products on the Dutch workflow market and has demonstrated its value in situations requiring more flexibility. The core features of case handling are:

– avoid context tunneling by providing all information available (i.e., present the case as a whole rather than showing just bits and pieces),
– decide which activities are enabled on the basis of the information available rather than the activities already executed,
– separate work distribution from authorization and allow for additional types of roles, not just the execute role,
– allow workers to view and add/modify data before or after the corresponding activities have been executed (e.g., information can be registered the moment it becomes available).

Since case handling is a combination of mechanisms, it is not easy to define this new paradigm in a rigorous manner. The existing definitions given in [1, 6] are too informal to allow for any form of analysis. The purpose of this paper is to "formalize" the case handling concept in terms of *Colored Petri Nets* (CPNs) [15, 19]. CPNs are a natural extension of the classical Petri net [23]. There are several reasons for selecting CPNs as the language for modeling the case handling paradigm. First of all, CPNs have formal semantics and allow for different types of analysis, e.g., state-space analysis and invariants [16]. Second, CPNs are executable and allow for rapid prototyping, gaming, and simulation. Third, CPNs are graphical and their notation is similar to existing workflow languages. Finally, the CPN language is supported by CPN Tools[1] – a graphical environment to model, enact and analyze CPNs. The model was created by one person, having a software engineering background and basic knowledge about classical Petri nets, in about four man-weeks. A significant amount of this time was spent

[1] CPN Tools can be downloaded from wiki.daimi.au.dk/cpntools/.

getting acquainted with CPN Tools as an application in general, and Standard ML for e.g. functions in particular.

The remainder of this paper is organized as follows. First, the case handling paradigm is introduced, emphasizing its specific features in contrast to traditional Workflow Management. Subsequently, Section 3 introduces a CPN model representing the basic case handling functionality in an abstract manner, including the results of a state space analysis showing model correctness. Section 4 discusses the applicability of the presented model by inspecting its alignment to an industrial case handling system and discussing limitations. Further, an extension to the model is presented that allows for generating artificial enactment logs for process mining research, followed by a conclusion.

2 Case Handling

In contrast to the strongly process oriented view of production workflow, emphasizing the routing between atomic activities, the case handling paradigm focuses mainly on the *case* itself. The case is the primary object to be manufactured in any kind, e.g. the outcome of a lawsuit or the response to a customer request. Resulting from that, single activities diminish in importance in favor of the larger context. They are no longer considered atomic steps that have to be performed in an "all or nothing" manner, but rather serve as logical partitions of work between which a transition from one worker to another is possible.

As in traditional WFM there exists a set of precedence relations between single activities making up a process, using well-known patterns [4] for that. However the primary driver for progress is no longer the event of explicitly finishing activities but the availability of values for data objects. While production workflow clearly separates the process from associated data, case handling integrates both far more closely, using produced data not only for routing decisions but also for determining which parts of the process have already been accomplished. With case handling, each task has associated with it data objects for three distinct purposes, while the first association is between a task and all data objects that are accessible while performing it. Further on, all data objects that are *mandatory* for a task have to be set (i.e., bound to a value) before the task itself is considered to be accomplished by the system. Finally, every data object can have a random number of tasks to which it is *restricted*, meaning that it can only be altered while performing one of these tasks. The *mandatory* and *restricted* properties are independent from each other, however reason dictates to have the last (in the sense of a causal chain) task featuring a data object as *restricted* also declare it as *mandatory* (to make sure it will be provided). User-interactive tasks are connected to a *form*, each providing access to a selection of data objects. Note that one form can be associated with multiple tasks; on the other hand it is also possible to associate a form to the case itself, so that it can be accessed at any point in time.

To introduce the case handling principles, Figure 1 shows a simplified example of a case type, the case handling analogy to a workflow process definition:

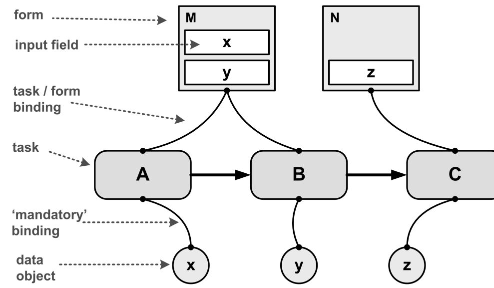


Fig. 1. Simplified example case type

Three tasks A, B and C are making up the process, sequentially chained by causal relationships denoted by connecting arrows. Their *mandatory* relationships to the three data objects x, y and z below are denoted by curved arcs, correspondingly they are associated with the forms M and N above. As can be seen in the illustration, tasks A and B share the same form M, providing access to data objects x and y. If a properly authorized worker now starts handling task A, the associated form M will open and he will start providing values for the presented data objects. In a traditional WFMS activity A would not be finished before form M is closed, however the case handling system regards A as finished as soon as a value for x has been provided (and confirmed appropriately), automatically enabling task B in the background. If the worker would now close form M, another employee could pick up the case where he left it, starting task B, which would provide the same form M with x having a value filled in (that could now be changed again). Another possibility is, however, that the first worker keeps on handling the form, providing also a value for y. This would correspondingly trigger the *auto-completion* of task B (as all associated mandatory data elements, in this case only y, have been provided) and activate task C. Note that if a worker closes a form after filling out only parts of the mandatory data fields of a task, despite the task not being considered finished data already entered is not lost but will be presented to the person continuing work on that task.

Such closely intertwined relationship between data and process obviously abandons their, often unnatural, separation so rigidly pursued in traditional workflow management. With the status of case data objects being the primary determinant of case status, this concept overcomes a great deal of the problems described in the introduction:

- Work can now be organized by those performing them with a far higher degree of freedom. Activities can either be performed only partly, without losing intermediary results, or multiple related activities can be handled in one go, surpassing the considerably weakened border between single tasks.
- The phenomenon of *context tunneling* can be remedied by e.g. providing overview forms directly associated with the case. Every authorized worker

can consult such form at any point in time, ensuring he is aware of the context where necessary.

– Routing is no longer solely determined by the process model. Case types can be designed in such a way, that multiple activities become enabled concurrently, providing different ways of achieving one goal. It is up to the user to decide which way to go, with the system "cleaning up behind", i.e., disabling or auto-completing tasks that have not been chosen.

In addition to the *execute* role, specifying the subset of resources allowed to handle a specific task, the case handling paradigm introduces two further roles crucial for operation. The *skip* role allows workers to bypass a selected task, which could be interpreted as an *exception*. When one thinks of real business processes an exception, like skipping an activity that deals with thoroughly checking the history of a client before granting a mortgage for well-known and trusted clients, is likely to occur quite frequently. The ability to grant the *skip* role to a senior worker renders the necessity for implementing such bypass obsolete, thus greatly simplifying the whole case type. It has to be noted that in order to skip a task all preceding tasks that have not been completed yet have to be skipped (or completed) beforehand. Traditional workflow definitions use loops for repeating parts of the process, e.g. because they have not yielded an expected result. In a case handling system, such construct has been made obsolete as well by the introduction of a *redo* role, enabling its bearer to deliberately roll the case's state back and make a task undone. In doing so, the values provided for data objects during this task are not discarded but merely marked as *unconfirmed*, so that they serve as kind of template when re-executing the affected task. Similar to skipping, before a task can be *redone* all subsequent tasks that have already been completed need to be rolled back as well before. Roles in a case handling system are *case specific*, i.e., having assigned the role "manager" for a case type A does not imply that one can play the same role for another case B. They can be specified in form of a *role graph*, where single role nodes are connected to each other by arcs, symbolizing *is-a* relationships; i.e., being authorized to play a role also implies the authorization to play all roles connected as child nodes.

Intertwining authorization with distribution of activities has been one major flaw of traditional workflow technology. In a case handling system, the former *in-tray*, i.e., a list of all activities the user is authorized to perform and that he can choose from, has been replaced by a sophisticated *query mechanism*. This tool can be used to look for a specific case, based on certain features (e.g. case data, or enactment meta-data like the start date of a case instance). Moreover it can be used to create predefined queries tailored to each worker (or, group of workers). A manager is no longer constantly flooded with all possible activities that he can perform, but only those which require a certain role, or e.g. case instances of an order where the combined value exceeds $1000. Obviously the query mechanism can also be used to perfectly imitate a classic in-tray, be it required.

3 CPN Model

This section introduces the CPN model of a case handling system, thus making the case handling paradigm explicit. We first discuss the color sets and then show the top level view of the model. The top level model contains two substitution transitions, which are also discussed. Finally, the model is evaluated using the state space tool of CPN Tools.

3.1 Color Sets

Before showing the top level view of the model, we present some of the color sets used in the model (cf. Figure 2). Data elements are represented by the color set `DATUM` which is composed of a string that denotes its (unique) name, a boolean flag symbolizing if it is enabled (i.e., if its value can currently be set or unset), and a second boolean flag representing whether the value has actually been set. The place `all_data` contains complete information about all data elements as one single token of color set `DATUMLIST`, a list of `DATUM` instances, serving as central repository and interface between all three parts of the model. Color set `RESOURCE` is composed of a string containing the unique name of the resource, and a list of strings denoting the roles this resource can play. The most complex color set within this model is the `TASK`, composed as follows: One string contains the unique name of the task, followed by a list of strings representing the (names of the) data elements that can be accessed and another list of strings representing the mandatory data elements of this task (i.e., those that have to be set in order to complete a task). Two further string lists contain the names of preceding and successive tasks, thus making up the control flow of the case, and three strings specify the respective roles necessary for executing, skipping and redoing the task.

```
color STRINGLIST = list STRING;

color DATUM = product
  (* name *) STRING *
  (* enabled *) BOOL *
  (* isset *) BOOL;
color DATUMLIST = list DATUM;

color ROLE = STRING;

color RESOURCE =  product
  (* name *) STRING *
  (* roles *) STRINGLIST;

color MUTEX = unit with null;

color TASKSTATE =
  with initial | enabled | finished;

color TASK = product
  (* name *) STRING *
  (* data *) STRINGLIST *
  (* mandatory *) STRINGLIST *
  (* previous *) STRINGLIST *
  (* successors *) STRINGLIST *
  (* exec role *) STRING *
  (* skip role *) STRING *
  (* redo role *) STRING;

color TASKxRESOURCE =
  product TASK * RESOURCE;

color TASKENTRY = product
  (* name *) STRING *
  (* state *) TASKSTATE;

color TASKLIST = list TASKENTRY;
```

Fig. 2. Color set definition used in the CPN model

3.2 Top-Level View

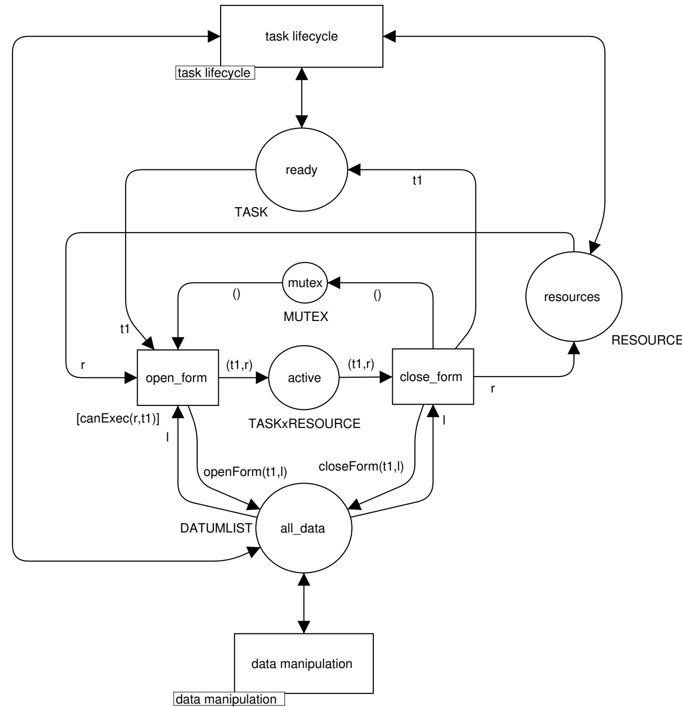


Fig. 3. Top-level view of the model

All basic functionality available to the user of a case handling system is included within the top level view. The `ready` place is the main interface to one further part of the model describing the lifecycle of tasks; when a task is ready to be handled it is transported to this place by the system, respectively tasks whose postcondition has been satisfied get removed from this place by the system. Users are now able to activate tasks that are being located in place `ready` for handling, which usually means opening the respective form in order to change associated data. This activity corresponds to firing transition `open_form`, consuming a task from `ready` and a resource token from the central repository place, both being stored intermediately in place `active` (as a product of both colors). When fired, `open_form` will also consume the token from `all_data` and set all data elements defined in the opened task as enabled. Subsequently firing transition `close_form` will adversely disable all data elements specified by the now closed task, transport the task token itself back into `ready` and free the associated resource, i.e., put it back into the central resource repository. Note that opening and closing forms is the only way to enable or disable data elements

for manipulation and, the other way round, apart from that no change in the overall state is performed.

One thing that has to be noted is that case handling systems usually not allow for real concurrency. To avoid context tunneling it will provide access to all case data, and therefore limit parallel updates of data values. If several users could open forms on one single case in parallel, the outcome of this would be impossible to determine, as they could enter contradictory values for one data field on different forms. This behavior is modeled by place `mutex` which contains exactly one black token. Once this token has been consumed by firing `open_form`, no further form can be opened unless firing `close_form` has produced a new token in `mutex`.

To avoid context tunneling and provide to the experienced user a higher degree of freedom, case handling systems allow to define forms that are associated directly to the case (in contrast to task-associated forms). These forms can be opened at any time and allow direct access to a random choice of case data elements. In this model, case forms are represented by tasks having no predecessor and successor tasks and no mandatory data elements. They reside in the `ready` place and will not leave it, due to their not being connected to other tasks from a control-flow point of view.

3.3 Data Manipulation

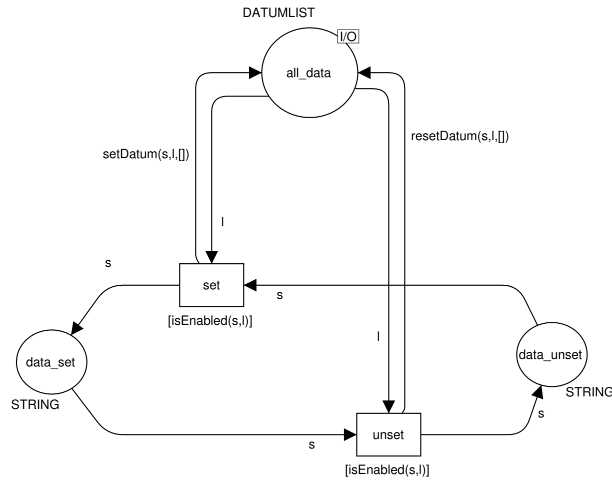


Fig. 4. Data manipulation subsection of the case handling model

This part of the model represents the setting and unsetting of data values dependent on their being enabled by currently open forms. Single data elements are represented by simple string tokens (containing their name) and can either

reside in places `data_set` or `data_unset`, so that their state is symbolized by their position within the model. A transition between the states of being set or unset — i.e., between the two respective places — can be achieved by firing transitions `set` and `unset`. The precondition for firing these transitions is, that for the data element to be (un-)set the "enabled" flag is set to true within place `all_data`, a condition checked by function `isEnabled`. In effect, firing either `set` (`unset`) executes function `setDatum` (`resetDatum`), toggling the boolean flag denoting the value-bearing status of the respective data element within `all_data`.

3.4 Task Lifecycle

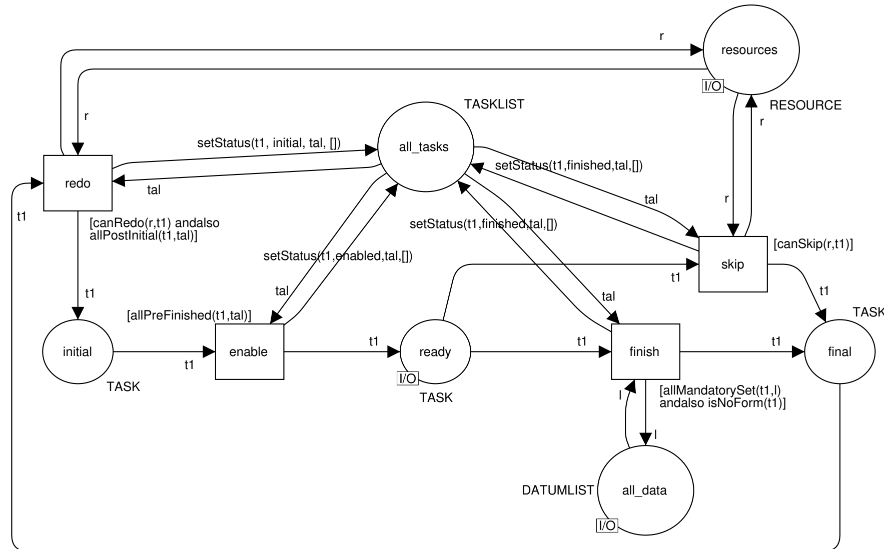


Fig. 5. Part of the model representing the task lifecycle

Handling the lifecycle of tasks is essential to the case handling mechanism, for this is possibly the point where such system differs most substantially from a conventional WFMS. The model is built to reflect the current lifecycle status of each task as its respective position in one of three places of type `TASK` (or four, if you count the `active` place in the top view part). Any task is either in state (place) `initial`, `ready` or `final`. One place `all_tasks` serves as a central repository for task state information, replicating this for the sake of model readability[2]. State transitions for tasks are dependent on the states of

[2] It would have been possible to test multiple places as precondition for firing a transition instead of just polling `all_tasks`. However, by using this central replication of state information it is possible to avoid a large amount of arcs crossing each other.

their respective predecessor and successor tasks, as well as it is dependent on the state of the case's data elements, i.e., which of them have already been set. When a case is started, all tasks contained will reside within place `initial`, waiting to be enabled by firing transition `enable`. The precondition for that is that all tasks that are direct predecessors have already been finished, it is checked by function `allPreFinished` which therefore retrieves the central token from `all_tasks` and puts it back, not before having changed the respective status of the enabled task.

After having been enabled the task resides at the `ready` place, from which basically three possibilities exist for progressing further. The task token can transcend into the top-level part of the model by being activated, i.e., opening its adjacent form and starting to change data values associated. This will, however, finally result in the form being closed again, returning the task into place `ready`. If the user decides so and has sufficient rights (i.e., a resource having the necessary role is available) he can deliberately `skip` a task (implemented by a transition of that very name), transporting the respective token directly into place `final`. This will adjust the task's status within the central repository `all_tasks` and free the resource again immediately afterwards.

The usual way, however, for a task to progress from `ready` to `final` is by firing transition `finish`. This transition gets enabled as soon as all mandatory data elements of a task have been set, further it is ensured that the respective task is "real", and not in fact a case form (by making sure it has predecessor and successor tasks) and the task state is adjusted in `all_tasks`. By solely depending on the status of data elements for finishing tasks this transition also implements the "autoskip" functionality of case handling systems, i.e., the possibility to finish tasks without even having touched them, solely by satisfying their mandatory data requirements otherwise (e.g. using case forms).

Redoing tasks requires, opposite to enabling them, that all successors of the respective task are in place `initial` (i.e., that they have previously been rolled back). The `redo` transition checks this property, and further needs to acquire a suitable resource that is freed immediately afterwards. The possibility to redo tasks closes the circular lifecycle, allowing tasks to progress through this an arbitrary number of times, under the sole premise that the basic (half-)ordering of tasks be maintained.

3.5 Evaluation

In order to both verify the correctness of the model and ensure alignment with the case handling principles that were intended to be modeled, a state space analysis of the model was performed within CPN Tools. However, before such analysis was feasible we were forced to reduce the size of the example case handling process and abstract from forms. Without these changes, CPN Tools was unable to construct the full state space. A token from place `ready` (symbolizing a form directly associated with the case) was removed and the represented process was simplified into a case handling process of only two consecutive steps `task1` and `task2`, i.e., the initial state of place `initial` was changed and the token in

place all_tasks was adjusted accordingly as well. However, the basic principles of case handling remain untouched by these alterations. As a result, the analysis in remainder remains relevant for other case handling processes.

Place	Upper Integer Bound	Lower Integer Bound
active	1	0
all_data	1	1
mutex	1	0
resources	3	2
ready	1	0
data_set	4	0
data_unset	4	0
all_tasks	1	1
final	3	1
initial	3	1

Table 1. Integer bounds of simplified case handling model

Boundedness Properties — One property the state space analysis can yield is the upper and lower integer bound of tokens per place, i.e., the maximum and minimum number of tokens of the appropriate color each place can contain for a given initial marking [14]. With respect to the scenario described above, the results are documented in Table 1. One first property that can be noted is that every place has an upper bound, i.e. the net is bounded. This corresponds to our expectations, as the model is not intended to generate additional tokens, so that the overall number of tokens contained within the complete net is expected to remain unchanged by firing any transitions[3]. Further it can be observed that places all_tasks and all_data will always contain exactly one token, as these are to serve as central information repositories which are not to be moved permanently. The difference in upper and lower bounds for place resources of merely 1 corresponds to the upper bound of 1 for place active, emphasizing the nature of a case handling system locking the complete case for exclusive access to one user at a time. Finally, the lower bounds of 1 for places initial and final are due to the virtual START and END tokens of color TASK which remain in their respective places infinitely as guaranteed process boundaries.

Home Properties — The case handling paradigm allows for a maximal degree of freedom with respect to navigating within a process. Given that he has sufficient permissions, the user can freely move back and forth within a case type's

[3] An exception to this is place mutex, which is empty as long as an activity or form is open.

process model by executing, skipping and redoing tasks at will. As extreme examples consider at the one hand finishing a case by merely skipping all available tasks, and on the other hand rolling an otherwise completed case completely back by redoing the tasks contained in reverse order. This behavior, i.e., the ability to reach any state from every other possible state, is reflected in the state space analysis labeling all possible markings of the net as Home Markings, i.e., markings that it is always feasible to reach [14].

Liveness Properties — Liveness denotes the remaining active of a set of binding elements, i.e., every transition can become enabled by firing an arbitrary number of transitions[14]. In contrast to this, a dead marking denotes a state of the net in which no transition is enabled. With respect to the model presented, the only acceptable dead marking, i.e., a state in which no further action is possible, would be a state in which the process is completely finished. However, given the fact noted above that case handling provides unlimited navigation within the process by means of skipping and redoing tasks, neither should there exist dead markings nor dead transitions (i.e., single transitions that can become permanently disabled by any firing sequence) for the model. This expectation has been acknowledged by the state space analysis, yielding no dead markings and denoting all transition instances as live.

Fairness Properties — Fairness is an indicator of how often different binding elements occur[14]. For the given model, the state space analysis denotes transitions `enable` and `redo` as fair, so an infinite number of enablings for these transitions implies an infinite number of occurrences. This corresponds to the intended behavior, as these transitions will actually remain enabled until fired. With the exception of `close_form`, all further transitions are just, i.e., persistent enabling of these implies an eventual occurrence. This property can be deduced from the free-choice character of the model, i.e., choices between two concurrently enabled transitions depend solely on user decision and are also intended to reflect these. Finally, transition `close_form` has no fairness. This is due to the fact that, in case this transition is enabled, it is always possible to fire `set` and `unset` in an infinite, alternating sequence, so that firing `close_form` can be effectively evaded. However, such behavior would correspond to a user having one specific form open infinitely long and keeping on changing values without eventually closing it again. Therefore this property reflects an intended model behavior, and a real-life occurrence of such potential infinite loop can safely be ruled out by common sense.

4 Applicability

The purpose of this section is to probe the real-life applicability and relevance of the presented model. In order to more illustrate the concept of case handling, and also to bridge the gap between the abstract model presented and real life

application, the first section compares the CPN model to the real case handling system FLOW*er* [1, 8, 22]. The second part discusses limitations of the presented model and discrepancies with respect to FLOW*er*, while the last part introduces an extension of the model for creating enactment logs during simulation.

4.1 Mapping to FLOW*er*

In order to more illustrate the concept of case handling, and also to bridge the gap between the abstract model presented and real life application, this section compares the CPN model to the real case handling system FLOW*er*.

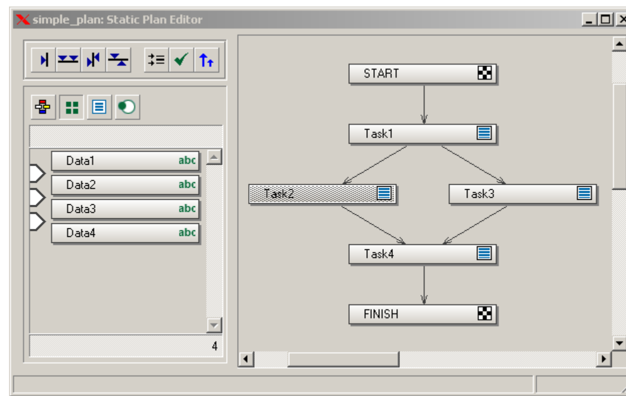


Fig. 6. Process model ("plan") in FLOW*er*

FLOW*er* features a graphical process designer tool (FLOW*er* Studio), in which so-called `case types` can be defined, Figure 6 shows a screenshot of this application with a simple example process[4]. On the left side four rectangular boxes can be seen, symbolizing the four data objects *Data1* to *Data4* used in this case type. To the right, the process structure is depicted, showing the tasks *Task1* to *Task4* as rectangular nodes, connected with arcs denoting causal relations between them. The start and end of the process is marked with so-called *Milestones*, which are no real tasks but symbolize defined states of the process and have been included here to make a connection to the fixed `START` and `END` tokens within the CPN model.

Apart from the relationships visible in Figure 6, the following has been defined: All four tasks use the same form (*Form1*), providing access to all four data objects; this form is also connected to the case type itself, i.e., it can be accessed at any time during handling a case. *Task1* has *data1* as mandatory, *Task2* and *Task3* both have *data2* and *data3* specified as mandatory and *Task4* requires *data4* respectively.

[4] In FLOW*er*, the process part of a case type is called a *plan*.

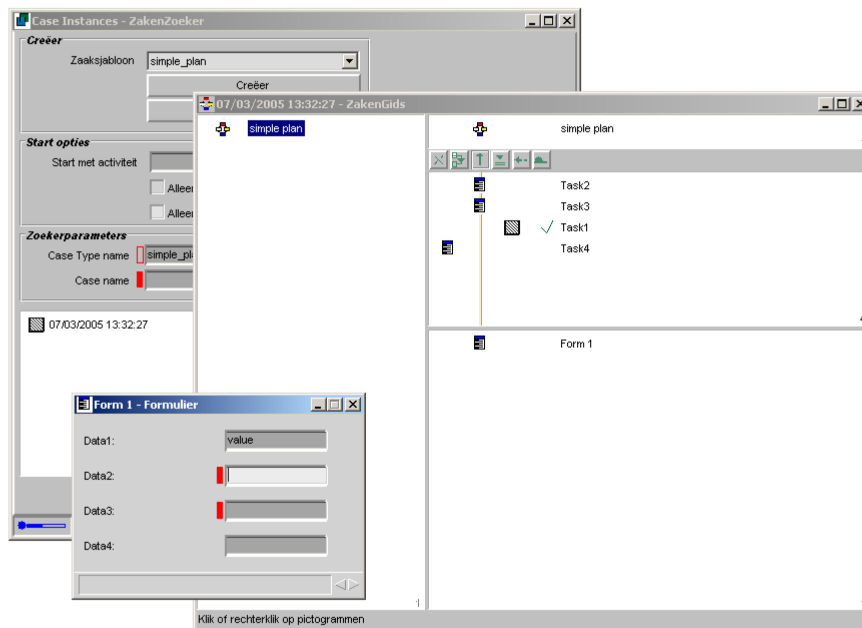


Fig. 7. User perspective of the FLOW*er* system

Figure 7 shows the user interface of FLOW*er*. In the front, the single form has been opened in reaction to executing *Task1* and a value has been provided and acknowledged for *data1*. The window in second order shows the main user interface of FLOW*er* which provides an overview of the case currently handled by the user. On the left, the single plan, i.e., process model, of the currently handled plan has been selected, thus it is also displayed on the top third of the right part. In the middle part of the right side the so-called *wavefront* is displayed, which is mainly a vertical line. Tasks that are being displayed on the left of this line are not yet ready to be executed, enabled tasks are right on the wavefront and completed tasks are shown to the right of it. Thus, tasks "travel" from left to right over the wavefront correspondingly to their lifecycle status (and they move back to the left again when they are being redone). Below the wavefront part of the interface, *Form1* is explicitly depicted and can be opened any time while handling this case.

As in this example a value has already been provided for *data1*, the first task (*Task1*) has already been finished (as *data1* was the only mandatory data object for this task) and has moved to the right of the wavefront. Corresponding to this, *Task2* and *Task3* have been enabled and are positioned right on the wavefront. A closer look at the opened form reveals red square icons to the left of the entry fields for *data2* and *data3*: Both *Task2* and *Task3* that are now enabled have set these data objects as mandatory, thus the system now signals (by the square

indicators left to each input field) that, by providing values for these fields, the next step could be accomplished.

Notice the non-intrusive nature of guiding user interaction performed by the case handling system, which is not enforcing or suggesting any specific behavior but rather presenting possible options. The user is not controlled by the system in a push-oriented manner, but he potentially has a great degree of freedom in his actions (depending on the design of the case type). Meanwhile the system will constantly observe which parts of the process have already been accomplished and track progress, providing more of a help in direction than restricting to predefined paths.

Table 2. Model place contents in discussed state

Place	Tokens
all_data	`1‘[("data1", true, true), ("data2", true, false), ("data3", true, false), ("data4", true, false)]`
all_tasks	`1‘[("START", finished), ("task1", finished), ("task2", enabled), ("task3", enabled), ("task4", initial), ("END", initial)]`
initial	`1‘("END", [], [], ["task4"], [], "", "", "")++ 1‘("task4", ["data1", "data2", "data3", "data4"], ["data2", "data3", "data4"], ["task2", "task3"], ["END"], "role3", "role3", "role2")`
ready	`1‘("task3", ["data1", "data2", "data3", "data4"], ["data2", "data3"], ["task1"], ["task4"], "role1", "role1", "role2")`
active	`1‘(("task2", ["data1", "data2", "data3", "data4"], ["data2", "data3"], ["task1"], ["task4"], "role2", "role2", "role3"), ("resource1",["role1","role2"]))`
final	`1‘("task1", ["data1", "data2", "data3", "data4"], ["data1"], ["START"], ["task2", "task3"], "role1", "role1", "role2") ++ 1‘("START", [], [], [], ["task1"], "", "", "")`
resources	`1‘("resource2",["role2","role3"]) ++ 1‘("resource3",["role3","role1"])`
data_set	`1‘"data1"`
data_unset	`1‘"data2" ++ 1‘"data3" ++ 1‘"data4"`

If the example case type and situation as described above is translated into the introduced CPN model, the marking of the net after executing *Task1* and providing a value for *data1* is given in Table 2. In the token for `all_data` it can be observed that all four data elements have been enabled (with their second element, a boolean, set to `true`, while merely `data1` has its third element set to `true`, corresponding to its having been set. This last information is also explicitly reflected in the distribution of tokens between `data_set` and `data_unset`.

4.2 Limitations

The mapping performed in the last section shows that the presented CPN model is suitable for introducing and analyzing the basic features of the case handling paradigm, enabling the implementation and analytic enactment of simple case types. However, compared to the presented commercial system FLOW*er* it has several subtle discrepancies and limitations, which shall be discussed within this section.

Regarding case type design, the model does not allow for alternative branches within the process structure. Such feature would require the interpretation of Boolean statements on data objects, which is beyond the scope of the presented model. Another fundamental limitation is concerning process structure as well, for constructs like arbitrarily instantiated sub-processes[5] cannot be represented with this model. This does, however, have no influence on the correctness of the model, for these can be interpreted as abstract tasks executed in parallel.

Naturally, FLOW*er* as a commercial system allows for a much wider range of task definitions beside form actions. Tasks can also use database interactions, arbitrary code execution and other non-interactive methods of data processing. On the conceptual level of the model these differences are, however, completely irrelevant — all task types available in FLOW*er* share the same basic property relevant for the scope of the model, i.e., they all access and potentially modify case data objects.

Other constructs present in FLOW*er* can be seen as helpful abstractions, which can be implemented using the model by breaking them down into more low-level constructs. One example are role hierarchies; these are not supported by the model, but can be implemented by granting every resource that has role r also each child role of r. Regarding the *restricted* feature of data objects, which is also not directly implemented in the model, it is possible to make the respective data object available only for those tasks to which it shall be restricted, resulting in the very effect.

One aspect in which the behavior of the presented model deviates significantly from that of FLOW*er* is concerning task lifecycle transition. While the model treats task and associated form as unity, this association is implemented much less coherent in FLOW*er*. While, after providing a value for *data1* in the example of the last section, *Task1* has already transitioned to the final state, i.e., to the right of the wavefront, in the model this transition can only happen after the form has been closed. This discrepancy extends onto the time of enabling subsequent *Task2* and *Task3*, which is self-evident as they have to wait until *Task1* is finished. While this difference shows in the delayed transition of tasks, it has no influence on the general order of these transitions, such that the overall behavior of the model is consistent with FLOW*er*. Due to this property, results gained from the model can be applied with little to no restriction.

[5] These serve for executing a specific subprocess multiple times in parallel, whereas the number of subprocess instances is only determined during super-process enactment (e.g., collecting witness statements for an accident).

4.3 Process Log Synthesis

The presented CPN model obviously does not directly represent a business process, but it rather describes the abstract process of handling, or executing, such business processes within a case handling system. The executed process is thus not explicitly visible in this *meta* model, it is encoded in tokens of type *TASK*. Only when the model is executed, the handled process becomes observable.

To make such observation easier to achieve, and to allow for a more detailed analysis of the handled processes, the model was enhanced with the logging extensions of CPN Tools[10]. These are three ML functions, `createCaseFile`, `addATE` and `calculateTimeStamp`, that create process enactment logs of a model in MXML, the ProM[6] format, when the model is simulated in CPN Tools. An extra transition has been added to the model, activating the logging function `createCaseFile(int caseId)` before each execution of the model. This will create a new log file containing all events recorded for this particular simulation run (i.e., case), while the parameter *caseId* has to be manually set to a unique number in order to distinguish between different cases.

The actual log events, describing the life-cycle change of the task in question, are recorded by the function `addATE(int caseId, String transitionName, ListOfStrings eventyType, StringTimestamp timestamp, String originator, ListOfStrings data)`, which is called when firing transitions *enable*, *finish*, *skip*, and *redo*. Parameter *caseId* is set according to the value provided in `createCaseFile`, while the value for *transitionName* corresponds to the task name contained in the processed *TASK* token. The *eventType* is determined by the fired transition, i.e. transition *enable* will set the event type to *schedule*, *finish* to *complete* and so on. Notice that while the names are different, this is only due to a different terminology between the case handling principle and the MXML format, the semantics are perfectly in line. For setting the *timestamp* parameter, the helper function `calculateTimestamp` is called, creating ascending, logical timestamps. Finally, the *originator* parameter is set to the name of the *RESOURCE* token used (if any, otherwise it is left blank), while the *data* field is left blank.

Multiple simulations of the same process model can be aggregated into one log file, using a plugin in the ProM*import* framework[6]. This is an application that allows for importing MXML logs from all sorts of process-aware information systems. The aggregated log is then ready to be analyzed using the wide palette of Process Mining techniques available within ProM. Figure 8 shows the results of mining 20 simulation runs of the model with the Alpha algorithm. The process model encoded in the *TASK* tokens corresponds to the case type shown in Figure 6, except for one minor adaption: In order to ease mining and produce a sound WF-Net a task *INIT* has been prepended to the process. It cannot be skipped and rolled back, thus ensuring exactly one defined start condition without incoming arcs.

[6] ProM is the process mining framework. Both ProM and ProM*import* can be freely downloaded at http://www.processmining.org/.

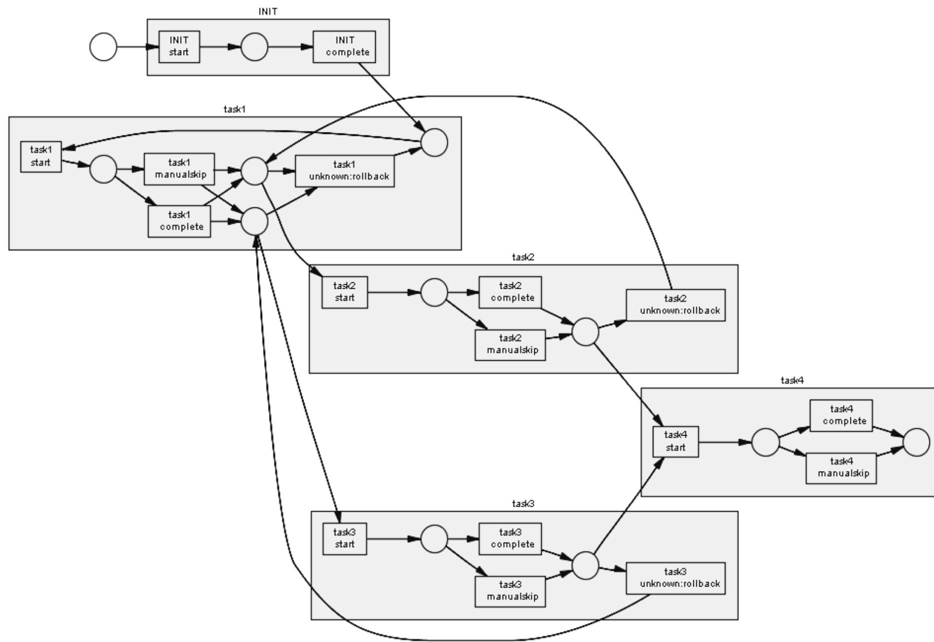


Fig. 8. Example process model mined from simulation logs

It is worthwhile pointing out that the mined process model, which is a WF-Net satisfying the soundness criteria, describes possible paths of execution in a very concise and accurate manner. For each task a similar sub-process has been discovered, showing the typical case handling lifecycle of a task with scheduling, skipping or completing, and rolling back. This accuracy allows for the model to be used for the sake of synthetic case handling log creation by simulation. In order to research new methods for Process Mining in the field of case handling systems a wide variety of enactment logs is required, and log synthesis by simulation can deliver these in a fast and automatized manner. Notice further that the ProM*import* framework also includes a plugin for importing logs from FLOW*er*. Actual case handling processes can be remodelled in the CPN model to research all potential enactment paths in synthesized logs, and the results can be compared to those obtained from analyzing real FLOW*er* logs.

5 Conclusion

In this paper we have discussed the deficiencies contemporary Workflow Management Systems suffer from and that hinder their wider application in industry, boiled down to four crucial problems. Subsequently, the case handling paradigm has been presented as potential remedy to these structural problems. Using the technique of Colored Petri Nets, an abstract model of case handling has been

introduced, incorporating all significant features of the way a case is being processed in such system.

This model can on the one hand be employed to understand the functionality of case handling and trace process enactment down to fine-grained steps. On the other hand, close alignment of the model to a real case handling system, FLOW*er*, has been shown in all significant aspects. Together with a state space analysis, proving basic correctness and expected behavior of the model, these properties can be extended onto the principles of case handling as a whole. Taking into account the mentioned limitations of the model and its discrepancies with respect to a real system like FLOW*er*, it can be employed to conduct more thorough research on specific properties of case handling. Especially the extension of the model with logging capabilities can provide valuable artificial logs, which are important for process mining research on case handling systems.

6 Acknowledgements

This research is supported by the Technology Foundation STW, applied science division of NWO and the technology programme of the Dutch Ministry of Economic Affairs.

The authors would like to thank Kurt Jensen for his support in the early stages of this article and his comments, and Ana Karla Alves de Medeiros for implementing the logging extensions to CPN Tools. Last but not least, the authors would like to thank the anonymous reviewers for their useful remarks.

References

1. W.M.P. van der Aalst and P.J.S. Berens. Beyond Workflow Management: Product-Driven Case Handling. In S. Ellis, T. Rodden, and I. Zigurs, editors, *International ACM SIGGROUP Conference on Supporting Group Work (GROUP 2001)*, pages 42–51. ACM Press, New York, 2001.
2. W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors. *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2000.
3. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2002.
4. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
5. W.M.P. van der Aalst and S. Jablonski. Dealing with Workflow Change: Identification of Issues and Solutions. *International Journal of Computer Systems, Science, and Engineering*, 15(5):267–276, 2000.
6. W.M.P. van der Aalst, M. Weske, and D. Grünbauer. Case Handling: A New Paradigm for Business Process Support. *Data and Knowledge Engineering*, 53(2):129–162, 2005.
7. A. Agostini and G. De Michelis. Improving Flexibility of Workflow Management Systems. In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 218–234. Springer-Verlag, Berlin, 2000.

8. Pallas Athena. *Case Handling with FLOWer: Beyond workflow*. Pallas Athena BV, Apeldoorn, The Netherlands, 2002.
9. F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow Evolution. In *Proceedings of ER '96*, pages 438–455, Cottubus, Germany, Oct 1996.
10. A.K. Alves de Medeiros and C.W. Günther. Process mining: Using cpn tools to create test logs for mining algorithms. *http://is.tm.tue.nl/research/processmining/tools/ProM/cpnToolConverter.zip*, 2005.
11. C.A. Ellis and K. Keddara. A Workflow Change Is a Workflow. In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 201–217. Springer-Verlag, Berlin, 2000.
12. T. Herrmann, M. Hoffmann, K.U. Loser, and K. Moysich. Semistructured models are surprisingly useful for user-centered design. In G. De Michelis, A. Giboin, L. Karsenty, and R. Dieng, editors, *Designing Cooperative Systems (Coop 2000)*, pages 159–174. IOS Press, Amsterdam, 2000.
13. S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996.
14. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1992.
15. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1*. EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1997.
16. K. Jensen and G. Rozenberg, editors. *High-level Petri Nets: Theory and Application*. Springer-Verlag, Berlin, 1991.
17. M. Klein, C. Dellarocas, and A. Bernstein, editors. *Proceedings of the CSCW-98 Workshop Towards Adaptive Workflow Systems*, Seattle, Washington, November 1998.
18. M. Klein, C. Dellarocas, and A. Bernstein, editors. *Adaptive Workflow Systems*, volume 9 of *Special issue of the journal of Computer Supported Cooperative Work*, 2000.
19. L.M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.
20. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999.
21. D.C. Marinescu. *Internet-Based Workflow Management: Towards a Semantic Web*, volume 40 of *Wiley Series on Parallel and Distributed Computing*. Wiley-Interscience, New York, 2002.
22. Pallas Athena. *Flower User Manual*. Pallas Athena BV, Apeldoorn, The Netherlands, 2002.
23. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.
24. M. Weske. Formal Foundation and Conceptual Design of Dynamic Adaptations in a Workflow Management System. In R. Sprague, editor, *Proceedings of the Thirty-Fourth Annual Hawaii International Conference on System Science (HICSS-34)*. IEEE Computer Society Press, Los Alamitos, California, 2001.

Execution of UML Models with CPN Tools for Workflow Requirements Validation *

Ricardo J. Machado[1], Kristian Bisgaard Lassen[2]
Sérgio Oliveira[1], Marco Couto[1], Patrícia Pinto[1]

[1] Dept. of Information Systems, University of Minho, Portugal
[2] Dept. of Computer Science, University of Aarhus, Denmark

Abstract. Requirements validation is a critical task in any engineering project. The confrontation of stakeholders with static requirements models is not enough, since stakeholders with non computer science education are not able to discover all the inter-dependencies between the elicited requirements. Even with simple UML (unified modelling language) requirements models it is not easy for the development team to get confidence on the stakeholders' requirements validation. This paper describes an approach, based on the construction of executable interactive prototypes, to support the validation of workflow requirements, where the system to be built must explicitly support the interaction between people within a pervasive cooperative workflow execution. A case study from a real project is used to illustrate the proposed approach.

1. INTRODUCTION

Clients (normally, stakeholders) and developers (system designers and requirements engineers) have, naturally, different points of view towards requirements. A requirement can be defined as "something that a client needs" and also, from the point of view of the system designer or the requirements engineer, as "something that must be designed". The IEEE 610 standard [1] defines a requirement as: (1) a condition or capability needed by a user to solve a problem or achieve an objective; (2) a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed documents; (3) a documented representation of a condition or capability as in (1) or (2).

Taking into account these two distinct perspectives, two different categories for requirements can be conceived:

- *User requirements* result directly from the requirements elicitation task [2], as an effort to understand the stakeholders' needs. They are, typically, described in natural language and with informal diagrams, at a relatively low level of detail. User requirements are focused in the problem domain and are the main communication medium between the stakeholders and the developers, at the analysis phase.
- *System requirements* result from the developers' effort to organize the user requirements at the solution domain. They, typically, comprise abstract models of the system [3], at a relatively high level of detail, and constitute the first system representation to be used at the beginning of the design phase.

* This work has been supported by projects uPAIN (AdI/IDEIA/70/2004/3.1B/00364/007) and STACOS (FCT/POSI/CHS/48875/2002).

The correct derivation of system requirements from user requirements is an important objective, because it assures that the design phase is based on the effective stakeholders' needs. Some existent techniques [4-7] can be used to support the transformation of user requirements models into system requirements models, by manipulating the corresponding specifications. This also guarantees that no misjudgement is arbitrarily introduced by the developers during the process of system requirements specification.

However, this effort of maintaining the model continuity by applying transformational techniques can prove to be worthless, if the user requirements models are not effectively validated. Typically, the confrontation of stakeholders with static requirements models is not enough, since stakeholders with non computer science education are not able to discover all the inter-dependencies between the elicited requirements. Even with simple UML requirements models (use case diagrams and some kind of sequence diagrams) it is not easy for the development team to get confidence on the stakeholders' requirements validation. In fact, according to [8] there are three kinds of analysis that should be accomplished before a workflow is put into production: (1) validation, to check if the workflow behaves as expected; (2) verification, to study the correctness of a workflow; (3) performance analysis, to estimate the solution conformance with throughput times, service levels, and resource utilization. This paper is solely devoted to the first kind of analysis at the process level; i.e., we are neither considering the resource dimension where resources estimation is supposed to be reached, nor the case dimension where a concrete instance of a workflow process is analysed both in its commonalities and exceptions.

This paper describes the usage of CPN Tools [9] in the generation of interactive prototypes to allow stakeholders to be confronted with executable versions of previously elicited UML use case and sequence diagrams. This approach towards user requirements validation is illustrated with a real case study where a healthcare information system must be built to explicitly support the interaction between people within a pervasive workflow execution.

The remaining of this paper is organized as follows. Section 2 presents the case study by informally describing the purposes of the uPAIN system. Some UML models of the uPAIN system are also presented to support the discussion on the difficulties of achieving effective requirements validation based on static user requirements models. In section 3, we describe the construction of coloured Petri nets for animation of the dynamic properties of UML models. Here, the relation between the adopted stereotyped UML sequence diagrams and the coloured Petri nets is explained. Section 4 contains the global architecture of the tool environment used to generate the interactive animation prototype. Since some interoperability issues are not technological transparent when CPN Tools are used together with the *BRITNeY Animation tool*, some examples of the required XML files to perform the integration are discussed. In section 5, the strategies used to design the graphical user interface of the interactive animation prototype are discussed and some usability issues are referred. This section is devoted to the discussion of the efforts that must devote to obtain an animation artefact that effectively involves the stakeholders in the workflow requirements validation. Section 6 concludes the paper with some final remarks, mainly devoted to the

synthesis of the proposed approach limitations and of the accomplishments achieved. Future work is also briefly referred.

2. REQUIREMENTS MODELING

The case study considered in this paper consists of an information system (uPAIN system) whose main concern is the process of pain control of patients in a hospital, who are subjected to relatively long periods of pain during post surgery recovery. When a surgery is concluded, the patient enters a recovery period, during which analgesics must be administered to him in order to minimize the pain that increases as the effects of the anaesthesia gradually disappear. This administration of analgesics must be controlled according to a program which depends on factors like some personal characteristics of the patient (weight, age …) and the kind of surgery to which the patient has been submitted. The quantity of administered analgesics must be high enough to eliminate the pain, but low enough to avoid exaggerated or dangerous sedation states. This controlled analgesia is supplied to the patient by means of specialized devices called PCAs (patient controlled analgesia). PCA is a medication-dispensing unit equipped with a pump attached to an intravenous line, which is inserted into a blood vessel in the patient's hand or arm. By means of a simple push-button mechanism, the patient is allowed to self administer doses of pain relieving medication (narcotic) on an "as need" basis. This is called a bolus request.

The motivation for the development of the uPAIN system arises from the fact that different individuals feel pain and react to it very differently. Also, although narcotic doses are predetermined as mentioned previously, there is a considerable variability of their efficiency from patient to patient. This is why anaesthesiologists are interested in monitoring several variables, in a continuous manner during patients' recovery, in order to increase their knowledge on what other factors, besides those already known, are relevant to pain control, and in what measure they influence the whole process. To achieve this, the main idea behind the uPAIN system is to replace the PCA push-button by an interface on a PDA (personal digital assistant), which still allows the patient to request doses from the PCA, but with the addition of the functionality of creating records in a database of all those requests, along with other data considered relevant by the medical doctors, like the values of some pre-determined physiological indicators measured by a monitor, and/or other data related to a particular patient's state, symptoms, etc. These questions may be automatically asked by the system, via the PDA, when the patient requests a dose or at regular time intervals, or even when a medical doctor decides to ask for it.

So, the uPAIN system is intended to provide a platform that enables the registration of patients' pain levels and the occurrence of several symptoms related with analgesia processes, as frequently as desired, while allowing the medical staff to be permanently aware of the occurrence of all the relevant facts of the patients' recovery and pain control processes and, simultaneously, allowing permanent remote wireless communication among system, patients and medical staff.

Requirements elicitation is all about learning and understanding the needs of users and project sponsors with the ultimate aim of communicating these needs to the system developers [2]. Getting the right requirements is considered as a vital and difficult part of soft-

ware development projects. Modelling and model-driven approaches provide ways of representing the existing or future processes and systems using analytical techniques with the intention of investigating their characteristics and limits [3].

UML use case diagrams are a quite adequate tool to describe user requirements at a first high-level of abstraction. These diagrams constitute a suitable means for delimiting the system boundaries, for identifying the functionalities that should be provided by the system, and for affecting external actors with specific use case functionalities. Additionally, brief textual descriptions may be provided in natural language for each use cases. These diagrams are normally constructed by the developers in a tentative to document the elicited requirements. Stakeholders can read and use these diagrams to recognize the main functional areas of the system to be designed.

General functionalities of the uPAIN system are inscribed in the UML use case diagram depicted in Fig. 1. A set of additional use case diagrams have been constructed to refine some of the use cases existent in Fig. 1. The corresponding textual descriptions have also been obtained.

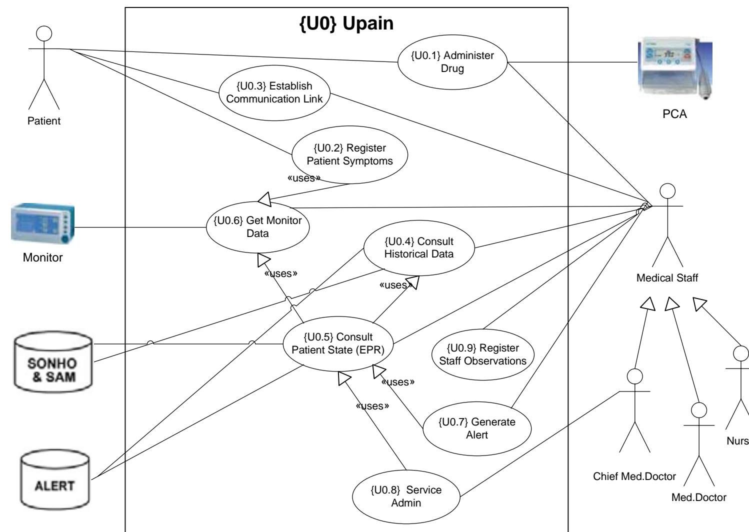


Fig. 1. UML use case diagram for the uPAIN system.

With the exception of just a few *«uses»* and *«extends»* relationships that may already be shown between use cases, it turns out to be obvious that use case diagrams do not practically say anything about how the system should be designed, in order to supply the identified functionalities. A further step on that direction may be provided by sequence diagrams in order to illustrate the desired dynamic behaviour in what concerns its functional interaction with the environment. These diagrams are also to be constructed by the developers. Stakeholders can also read them. However, they are not comfortable with all the details these diagrams can entail.

Fig. 2 depicts one UML sequence diagram for the uPAIN system that describes one macro-scenario where a patient requests a bolus. That request may be accepted by the system or originate a request for an explicit medical decision. In the later case, the

doctor may decide to authorize the bolus or to reconfigure the PCA parameters. The integration of several scenarios into only on sequence diagram (for a macro-scenario) is possible due to the new mechanisms of UML 2.0 in supporting different kinds of frames.

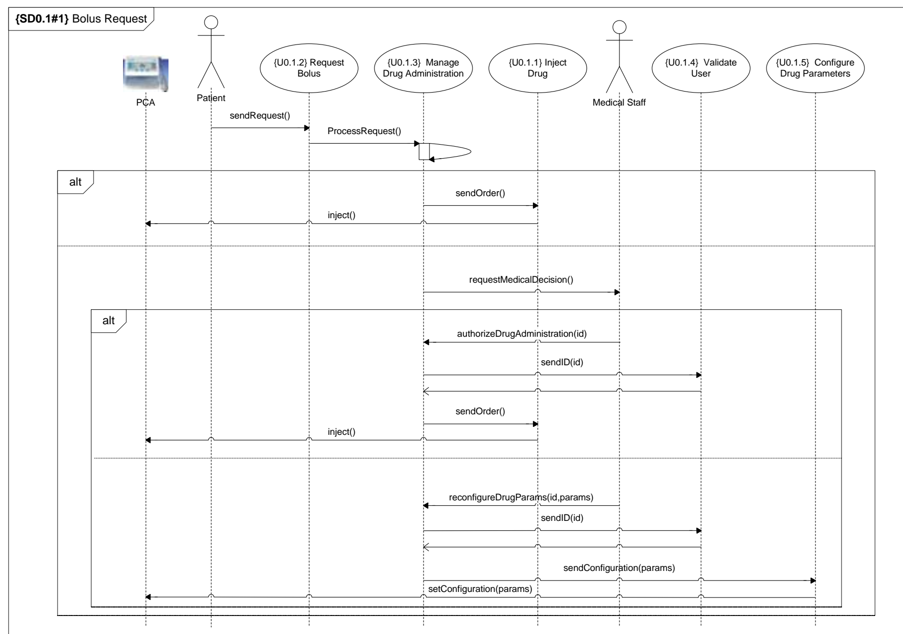


Fig. 2. UML sequence diagram for the uPAIN system.

Some more UML sequence diagrams have been constructed to capture the main system scenarios. At the analysis phase of system development, we adopt a stereotyped version of UML sequence diagrams, where only actors and use cases are involved in the sequences, since no particular structural elements of the systems are known yet. This kind of sequence diagrams allow a pure functional representation of behavioural interaction with the environment and are particularly appropriate to illustrate workflow user requirements.

Our stereotyped UML sequence diagrams contrast with the traditional ones that already involve system objects in the interaction with external actors, implying that those objects must be previously identified. One important issue concerning objects identification and building object diagrams is that they already model structural elements of the system, which is clearly beyond the scope of the user requirements. Additionally, the use of this kind of traditional sequence diagrams at the first stage of analysis phase (user requirements modelling and validation) require a deeper intervention of modelling skills that are hardly understandable to most stakeholders, making more difficult for them to establish a direct correspondence between what they initially stated as functional requirements and what the model already describes. So, a validation of the user requirements resulting from such an advanced model is not only

more difficult to achieve, but also less trustworthy and less ensuring that the resulting system will correspond effectively to the stakeholders expectations.

3. CPNs FOR ANIMATION PROTOTYPES

The effort to use only elements from the problem domain (external actors and use cases) in the user requirements models (use case and stereotyped sequence diagrams) and to avoid any reference to elements belonging to the solution domain (objects and methods) is not enough to obtain requirements models that are capable of being fully understandable by common stakeholders. This difficulty is mainly observable in what concerns the comprehension of the dynamic properties of the system within its interaction with the environment. This means that, even with the referred efforts, those static requirements models should not be used to directly base the validation of the elicited user requirements by the stakeholders. Instead, we use those static requirements models to derivate animation prototypes.

User friendly visualizations of the system behaviour, automatically translated from formal systems' models specifications, accepting user interaction for validation purposes, have been generically called animations. Despite seeming a good idea, in a context where IT is offering more and more powerful multimedia capabilities, the use of animation in user requirements validation, as a means of improving the understandability of systems' models by stakeholders, has been considered by only a small number of researchers.

In [10] an empirical study has been carried out to comparatively evaluate the effectiveness of animation and narration (voice recordings of diagram explanations complemented with *PowerPoint* slides) in the process of communication of domain information to stakeholders for validation purposes, which may be seen as a sign that animation is increasingly drawing the software engineers' attention as a potentially valuable instrument for user requirements validation. The results of that empirical study were inconclusive about the effectiveness of animation, as opposed to the success of narration, but in our opinion that was due to the fact that, instead of using a meaningful user interface, the animations were of a very rudimentary type by highlighting the graphical elements of the diagrams while narration is being executed.

Some other papers have been published, reporting the use of animations to ease validation by stakeholders, as is the case in [11], where a *CORBA* API has been used to directly interpret *VDM-SL* specifications of requirements to generate a graphical user interface. Scenario-based approaches have also been used in [12] as a means of ensuring user-orientation, and also in [13], where *fluents* (boolean system states that model pairs of system actions) have been used to relate goals with scenarios and, simultaneously, support animation. In [14], virtual reality is used to support animation techniques when modelling high consequence systems (systems where errors in development have consequences of high cost).

The behaviour of the animation prototypes (proposed in this paper) results from rigorous translations of the sequence diagrams into Coloured Petri Nets (CPNs) [15, 16]. The transitions of these CPNs present a strict one-to-one relationship with the messages in the sequence diagrams. So, for each message in a sequence diagram, one transition, in the corresponding CPN, is created. The name of each transition matches

exactly the corresponding name of the message in the sequence diagram. Two simple rules were used for that translation: (1) Fig. 3 illustrates the rule for translating two successive messages in a sequence diagram (Fig. 3a) into a CPN (Fig. 3b); (2) Fig. 4 illustrates the rule for translating an alternative block in a sequence diagram (Fig. 4a) into a CPN (Fig. 4b).

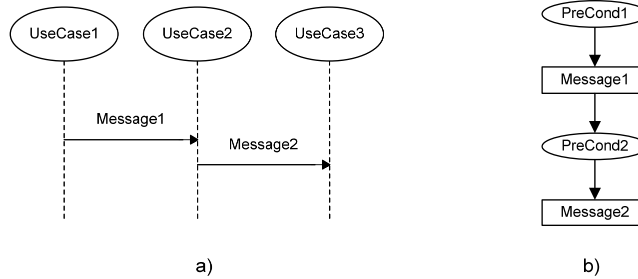


Fig. 3. Transformation of successive messages.

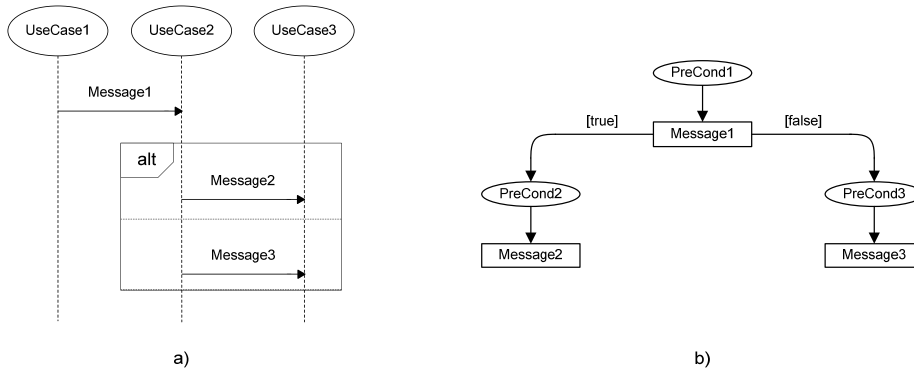


Fig. 4. Transformation of an alternative block.

Each output place of a transition (corresponding to a message in a sequence diagram that ends in a life line of a use case) represents the reaction of the system to the message request. Although CPN Tools do not support the creation of pages for the addition of refinement subnets for places (that is the reason why we use the expression “refinement subnets”, instead of “refinement subpages”), those output places may be replaced, at the same hierarchy level, by refinement sub-nets (composed of one input place, one output place, and one substitution transition between them) to support the refinement of use cases. The refinement subpage for each substitution transition describes one refined use case.

Typically, the refinement subnets will be built after the application of the 4SRS (4 step rule-set) technique [7] that transforms users requirements into architectural models representing system requirements, by mapping use cases into system-level objects within a four step approach: (1) object creation, (2) object elimination, (3) object packaging and aggregation, and (4) object association. Therefore, each transition in those subnets will correspond to the invocation of a method of a system object.

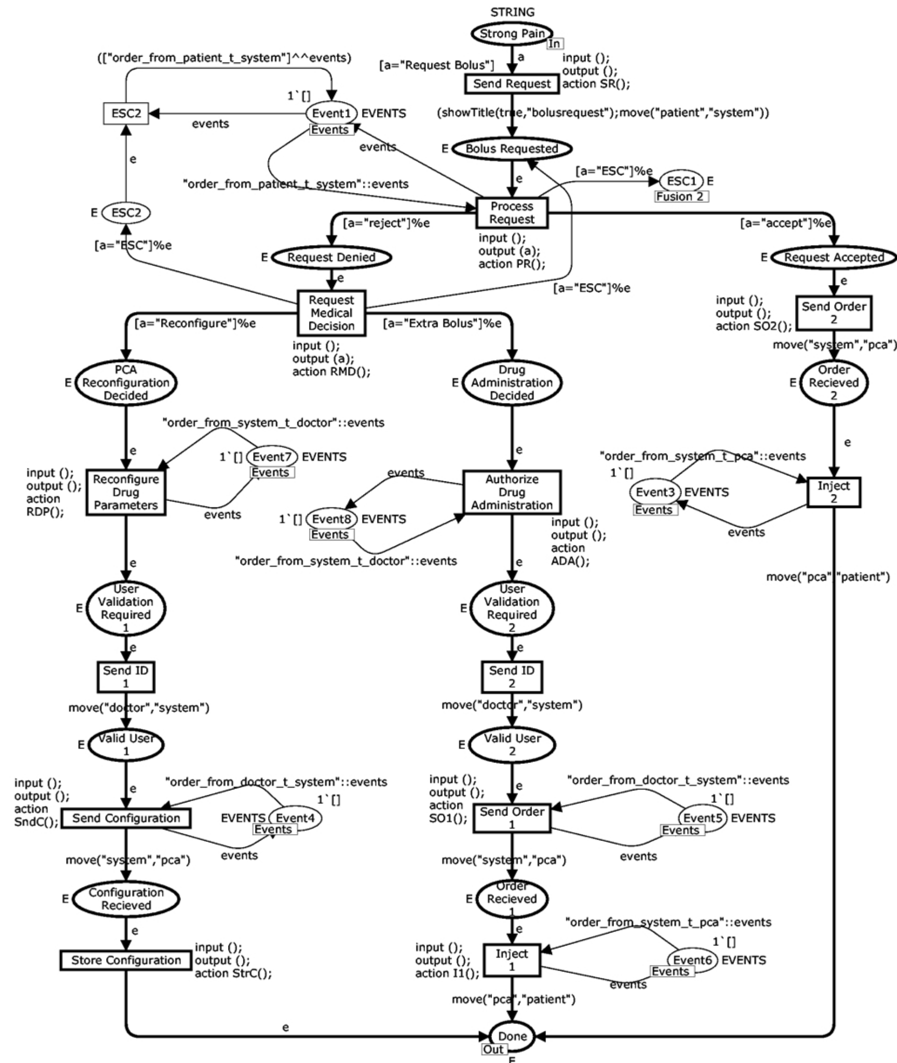


Fig. 5. CPN responsible for the animation of the use case *{U0.1} administer drug*.

The CPN of Fig. 5 is responsible for the animation of the use case *{U0.1} administer drug* (see Fig. 1) by executing three different sequence diagrams, each one corresponding to one of the three branches of the CPN. The middle branch is responsible for the execution of the sequence diagram of Fig. 2. Those nodes and arcs drawn with thinner lines were added in a later phase, and have no semantic correspondence to the sequence diagrams. They were included for the purpose of tools interoperability, as explained in section 4.

The CPN represented in Fig. 6 corresponds to the top-level net of the animation prototype for the uPAIN system. Thick lines were used to represent the elements that correspond to the main animation paths. Most of the transitions of this CPN

correspond to the use cases in Fig. 1. The refined CPN of the substitution transition *bolus request* of Fig. 6 corresponds to Fig. 5.

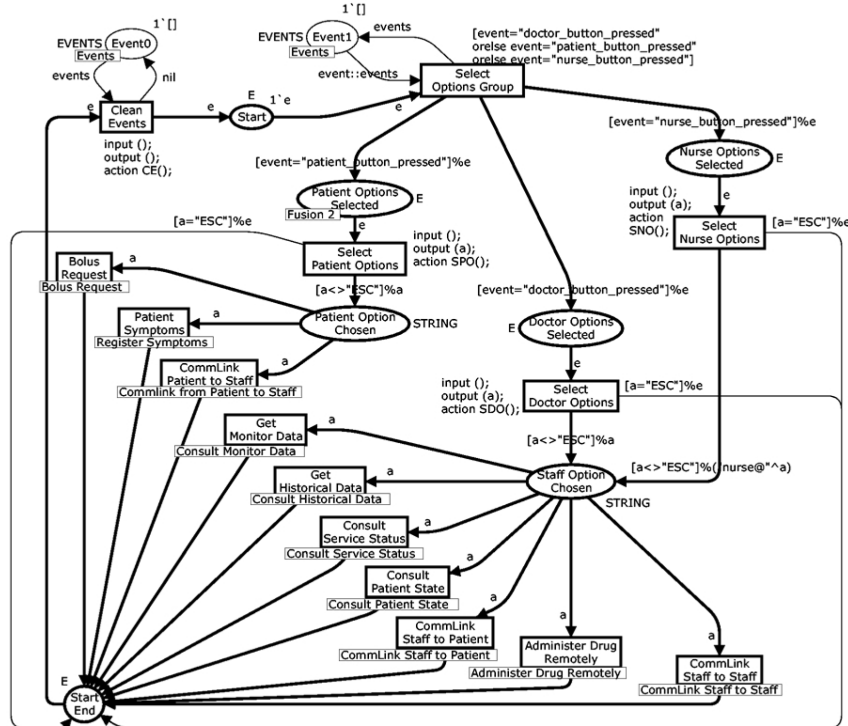


Fig. 6. Top-level CPN of the animation prototype for the uPAIN system.

With the transformation rules depicted in Fig. 3 and Fig. 4, direct links between the use case diagram (Fig. 1) and the CPNs (Fig. 5 and Fig. 6) are not intended, nor considered of interest. Instead, the link between the UML diagrams and the CPNs is obtained in two steps: the first is supported by the fact that the sequence diagrams are directly derived from the use cases; the second is ensured by a direct transformation of the sequence diagrams into CPNs.

Sequence diagrams transmit partial views for the interaction between the system and its environment, allowing the adoption of an evolutionary approach, by considering a set of sequence diagrams to have a partial evaluation of the requirements and then progress with more detailed requirements. In the uPAIN system, the animation prototype reflects only a top-level description of the system. After the validation of this top-level model, a set of additional animations, based on refined sequence diagrams at the solution level (where objects would already appear), can be constructed.

4. TOOLS INTEGRATION

The implementation of the interactive animation prototype demanded the usage of several technologies. The integration of tools was mainly based on XML files. Fig. 7 shows the global architecture of the tool environment used to generate the animation prototype. It is composed of a model executor and an animation tool. The model executor includes a *CPN editor* and a *CPN simulator*, both from CPN Tools. The animation tool used corresponds to the *BRITNeY Animation tool* [17].

With *BRITNeY Animation tool* it is possible to use pre-defined plug-ins (or write our own plug-ins) for executing some animation behaviour in the model. The pre-defined plug-ins include *SceneBeans* [18] (an animation framework), message sequence charts (for displaying the passing of messages) and plot graphs. The writing of our own plug-ins involves the coding of Java classes and the creation of an XML description of the plug-in. *BRITNeY Animation tool* will automatically generate the code needed for the simulator to know of and use those plug-ins.

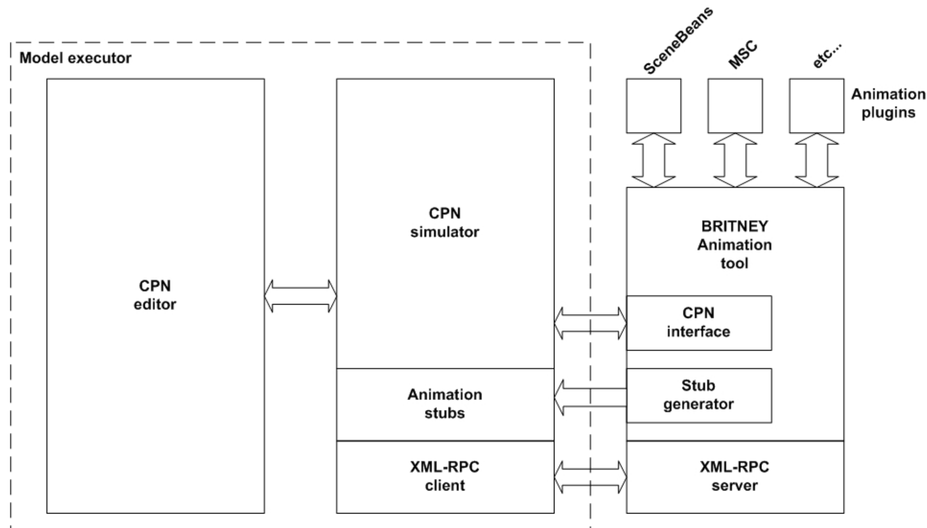


Fig. 7. Global architecture for prototype animation.

It is possible to execute behaviours in the *BRITNeY Animation tool* while simulating models in CPN Tools. Behaviours are executed through certain SML functions which in turn call the corresponding Java methods. The names of the functions correspond to those of the Java methods. When an SML function calls a Java method it simply corresponds to the logic of an RMI call. The method name and arguments are passed over to the interface of the *BRITNeY Animation tool* and the return value of the executed method is passed back from the interface. If the method *M* in class *C* has the signature *int M (int x, string y)*, then it could be invoked as *C.M (42, "Hello World")*. However, this is just an example to explain the way to use the Java methods in the CPN model (see [17] for complementary explanations). These behaviours, or methods, can be executed anywhere in the CPN model where an expression is allowed. So, it can be on an arc expression, code segments on transitions (these are specific for CPN Tools), and so on.

This is a nice feature for debugging and for understanding the way the model affects the animation.

The *BRITNeY Animation tool* can also be executed as a standalone program, using e.g. Java WebStart to enable web browser integration. This feature is very useful to generate an autonomous animation prototype which allows stakeholders to "play with" without the interference and the presence of elements from the development team. This approach to validation was experimented with and proved to be very effective. This empowerment of the stakeholders promoted a deeper involvement of them in the analysis phase that not only assured better validation results, but also allowed the complementary elicitation of workflow requirements.

The interactive animation prototype for the uPAIN system is depicted in Fig. 8. The usage of *SceneBeans* allowed the animation of actors and message passing. *SceneBeans* provides a parser that translates XML documents into animation objects. A *SceneBeans* document is contained within a top-level *<animation>* element that contains five types of sub-elements: (1) a single *<draw>* element defines the scene graph to be rendered (e.g. the representation of the doctor, the nurse, the patient); (2) *<define>* elements define named scene graph fragments that can be linked into the visible scene graph; (3) *<behaviour>* elements define behaviours that animate the scene graph (e.g. the animation of the drug injection from PCA to the patient); (4) *<event>* elements define the actions that the animation performs in response to internal events (e.g. the cleaning of the info text at the end of the drug injection animation); (5) *<command>* elements name a command that can be invoked upon the animation and define the actions taken in response to that command (e.g. the invocation of the behaviours responsible for the drug injection animation).

Fig. 9 shows an example of a code segment that was used in our *<draw>* element. This code segment is responsible for the creation of the icons for the patient, the uPAIN system and the black ball that represents the messages between actors. To animate the ball, we used the *<animate>* element, that means that parameters *x* and *y* will be animated by the behaviours *xf_patient_to_system* and *yf_patient_to_system*, respectively.

Fig. 10 shows the behaviours, the commands and the events that are responsible for moving the ball from the patient to the uPAIN system. When the simulator invokes the command *f_patient_t_system_cmd*, the behaviours corresponding to the movement of the ball and the displaying of its textual info are started. When the execution of a behaviour ends, the animation will trigger the associated event, and this will start other behaviours, like *xball_out* (to hide the ball), *fadeout_info* (to hide the textual info), or *hide_patientpda_icon* (to hide the patient PDA icon). At the end, the event will be announced, which is crucial, because it allows the *CPN simulator* to capture it.

Communication between *SceneBeans* objects in the animation and the CPN model can be done in two ways: (1) asynchronously, here the CPN model simply invokes a command on a *SceneBeans* object and proceeds simulating, not caring for the moment when the animation behaviour that was executed terminates; (2) synchronously, here the CPN model, again, invokes a command on a *SceneBeans* object, but, instead of just proceeding, the CPN model waits for a particular event to arrive (e.g. the event "ball moved from patient to system"). This event would be broadcasted by the animation command that was executed when it terminates to let the CPN model know that this animation has completed. Synchronous interactions with *SceneBeans* objects must be

carefully analyzed; otherwise, animations that should be executed in sequence will be executed concurrently. It is necessary to determine which animation behaviours are to be completed before any other can proceed (synchronous) and those which can occur in any order (asynchronous). Invocations on *SceneBeans* objects are asynchronous in the sense that, per default, they do not broadcast any event; this has to be specified in the *SceneBeans* XML specification.

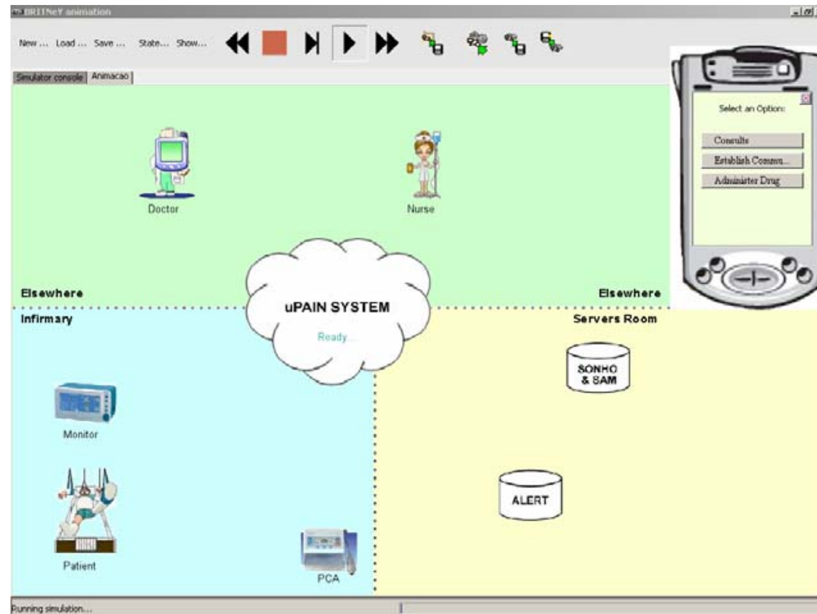


Fig. 8. Interactive animation prototype for the uPAIN system.

After creating all the behaviours, commands and events, which allow the animation to announce events and receive commands from the *CPN simulator*, the next step is to create Java classes. *SceneBeans* have the limitation of not allowing the user to input dynamic contents. In fact, *SceneBeans* only allows the creation of animations, based on static behaviours, defined in an XML file. The Java classes we created are responsible for showing the graphical interfaces of the PDAs and for sending the corresponding user (of the animation prototype) inputs to the *CPN simulator*. For instance, the Log Window that shows all the messages sent between actors demanded the creation of a Java class (*Messenger*) that receives the messages from the *CPN simulator*. To add a new message to the list of messages of the Log Window, we simply invoke *Messenger.createAndShowGUI("message")*. After creating the Java classes, an XML description must be constructed so that the *BRITNeY Animation tool* recognizes them as plug-ins (see Fig. 11).

```
<draw>
...
<transform type="translate">
      <param name="translation" value="(${systemX},${systemY})"/>
     <transform type="scale">
     <param name="x" value="1"/>
     <param name="y" value="1"/>
       <primitive type="sprite">
           <param name="src" value="upainimages/system.gif"/>
       </primitive>
    </transform>
 </transform>
...
<input type="mouseClick" id="patient_button">
 <param name="releasedEvent" value="patient_button_pressed"/>
 <transform type="translate">
      <param name="translation" value="(${patientX},${patientY})"/>
       <primitive type="sprite">
           <param name="src" value="upainimages/patient.gif"/>
       </primitive>
 </transform>
 </input>
<transform type="translate" id="ball">
      <param name="translation" value="(-50,-200)"/>
      <animate param="x" behaviour="xf_patient_t_system"/>
      <animate param="y" behaviour="yf_patient_t_system"/>
      ...
      <primitive type="circle">
            <param name="radius" value="10"/>
      </primitive>
 </transform>
...
</draw>
```

Fig. 9. Drawing in *SceneBeans*.

```
<!-- behaviours -->
<co id="f_patient_t_system" event="msg_f_patient_t_system" state="stopped">
<behaviour algorithm="move" id="xf_patient_t_system">
    <param name="from" value="${patientX}+40"/>
    <param name="to" value="${systemX}+60"/>
    <param name="duration" value="${ballSpeed}"/>
</behaviour>
<behaviour algorithm="move" id="yf_patient_t_system">
    <param name="from" value="${patientY}+70"/>
    <param name="to" value="${systemY}+60"/>
    <param name="duration" value="${ballSpeed}"/>
</behaviour>
</co>
<!-- commands -->
<command name="f_patient_t_system_cmd">
  <reset behaviour="f_patient_t_system"/>
  <start behaviour="f_patient_t_system"/>
  <set object="info" param="text" value="Receiving request..."/>
  <reset behaviour="fadein_info"/>
  <start behaviour="fadein_info"/>
</command>
<!-- events -->
<event object="f_patient_t_system" event="msg_f_patient_t_system">
  <reset behaviour="xball_out"/>
  <start behaviour="xball_out"/>
  <reset behaviour="fadeout_info"/>
  <start behaviour="fadeout_info"/>
  <reset behaviour="hide_patientpda_icon"/>
  <start behaviour="hide_patientpda_icon"/>
  <announce event="order_from_patient_t_system" />
</event>
```

Fig. 10. Defining behaviours, commands, and events in *SceneBeans*.

To access public methods of previously written Java classes (*Chat*, *ScenarioSelector*, *Messenger*, *Ppda*, *Dpda*, and *Npda*) and to invoke commands of the *SceneBeans* object (object *anim*) from the CPN Tools, plug-ins must be declared and instantiated as objects in the index of CPN Tools (see Fig. 12).

Java classes in defined animation plug-ins can be instantiated through SML (Standard Meta Language) functors that *BRITNeY Animation tool* generates. SML functors are “abstract” SML structures which can be instantiated. A Java object is instantiated by, e.g., *structure anim = SceneBeans(val name = "Name")*, which instantiates an object from the *SceneBeans* class. Methods on the instantiated *anim* are accessed as public methods defined in the *SceneBeans* class. Another example is the function *SPO ()* in the transition *Select Patient Options* of Fig. 6 that contains the following code to invoke methods to our Java objects:

```
Messenger.cleanText ( );
Ppda.createAndShowGUI ("mainmenu");
Ppda.getValueString ( );
```

SceneBeans objects provide also some methods to control the animation. For instance, the calling of function *move ()* in the CPN of Fig. 5 consists in an invocation of the method *invokeCommand* to the *SceneBeans* object *anim* (Fig. 12), which is responsible for invoking the previously defined commands in the XML file (Fig. 10). In this case, the invoked command corresponds to the movement of the black ball between the actors of the animation.

```
<?xml version="1.0" ?>
<!DOCTYPE plugin PUBLIC "-//JPF//Java Plug-in Manifest 0.3" "http://jpf.sourceforge.net/plugin_0_3.dtd">
<plugin id="UPAIN" version="0.1.0">
        <requires>
                <import plugin-id="AnimationTools"/>
        </requires>
        <runtime>
                <library id="ScenarioSelector" path="UPAIN.jar" type="code">
                        <export prefix="*"/>
                </library>
        </runtime>
        <extension plugin-id="AnimationTools" point-id="Animation" id="ScenarioSelector">
                <parameter id="class" value="views.ScenarioSelector"/>
        </extension>
        <extension plugin-id="AnimationTools" point-id="Animation" id="Ppda">
                <parameter id="class" value="views.Ppda"/>
        </extension>
        <extension plugin-id="AnimationTools" point-id="Animation" id="Dpda">
                <parameter id="class" value="views.Dpda"/>
        </extension>
        <extension plugin-id="AnimationTools" point-id="Animation" id="Npda">
                <parameter id="class" value="views.Npda"/>
        </extension>
        <extension plugin-id="AnimationTools" point-id="Animation" id="Messenger">
                <parameter id="class" value="views.Messenger"/>
        </extension>
</plugin>
```

Fig. 11. Defining Java classes as plug-ins of *BRITNeY Animation tool*.

Additionally, it is possible to capture events announced by the animation. In our animation prototype we included one CPN subpage called *Events* (Fig. 13) that is composed by two distinct parts: one is responsible for the initial loading of the XML animation description (places *Start* and *Running*, and transition *Init*); the other part includes the transition *Capture Event* (captures all the events announced by the *SceneBeans* animations and places them, in the form of a string list, in the place *Events*) and the place *Events*. This place *Events* can be cloned and connected to any transition where the capture of specific events is required (see, for instance, the nodes and arcs drawn with thinner lines in Figs. 5-6). These cloned places are named *EventN* (where *N* is a digit that serves only as a distinguishing character, because CPN Tools do not accept places with the same name, in the same page) and have no semantic meaning from the workflows’ point of view. They are only needed for tool interoperability.

```
▶Tool box
▶Help
▶Options
▼SD0.1k.cpn
   Step: 0
   Time: 0
 ▶Options
 ▶History
 ▼Declarations
   ▶Standard declarations
   ▶Custom Declarations
   ▼Animation setup
     ▼structure anim = SceneBeans(val name = "Animacao");
     ▼structure chat = Chat(val name = "Chat")
     ▼structure scenarioselector = ScenarioSelector(val name = "SCENARIO SELECTOR");
     ▼structure messenger = Messenger(val name = "LOG WINDOW");
     ▼structure ppda = Ppda(val name = "PATIENT PDA");
     ▼structure dpda = Dpda(val name = "DOCTOR PDA");
     ▼structure npda = Npda(val name = "NURSE PDA");
   ▶fun readyRunning
   ▶fun showHabRects
   ▼fun move(from:STRING, to:STRING)=
     (anim.invokeCommand("f_"^from^"_t_"^to^"_cmd");e)
   ▶fun moveMessage
   ▶fun notify
   ▶fun showPdaIcon
   ▶fun showTitle
   ▶fun delim
```

Fig. 12. Declaring and instantiating objects in CPN Tools.

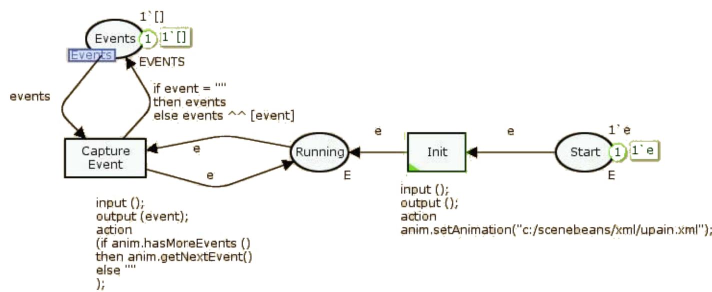


Fig. 13. *Events* CPN subpage.

5. USABILITY ISSUES

According to [19], usability is considered "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use". This means that, besides all the technical efforts described in the previous two sections of this paper, the effectiveness of the implemented animation prototype to involve stakeholders in the interactive execution of the elicited sequence diagrams, complementary elicitation of workflow requirements and validation of the requirements model was also a result of a strong investment in using usability techniques in the construction of this software artefact, namely in what concerns its GUI (graphical user interface) and the comfort of exploitation of the animation prototype.

The adopted GUI makes use of eight icons on the display: three proactive actors (one patient, one medical doctor and one nurse); four reactive actors (one monitor, one PCA device and two databases in use at the hospital); and the uPAIN system represented by a cloud. The adopted GUI should be obvious and intuitive to the

stakeholders and thus, with the exception of the cloud and the databases, we opted for "concrete" icons. When real-world objects are represented in an icon ("concrete" icon), individuals are likely to find it more meaningful, are often familiar with the items depicted, and find it easy to make links between what is shown in the icon and the function it is supposed to represent [20]. To symbolize the uPain system (a concept which is difficult to materialize and to represent), we chose a cloud which constitutes an "abstract" icon. Forming strong systematic relations between icons and functions is very important, particularly when there are no pictorial alternatives for a given icon function [21]. To represent the uPain system we wanted an icon that emphasized its pervasive and wireless nature. Databases are also represented through an "abstract" icon which is a standard way to represent software-technology databases. We also opted for uniform icons in terms of size because we wanted to avoid stakeholders focusing on some of the icons and not others due to size differences; we wanted them to have, at the first glance, the notion of the whole GUI. On the other hand, the real sized PDA is the bigger element and the only one which detaches from the GUI in terms of size, in order to improve the legibility of its contents.

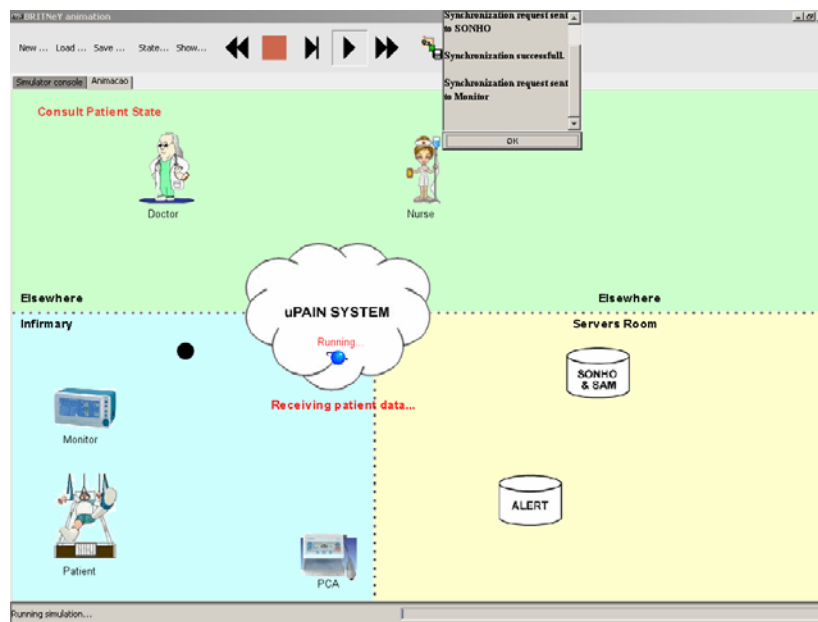


Fig. 14. Message passing in the animation prototype for the uPAIN system.

Whenever one proactive actor is clicked with the mouse, a PDA icon appears above it and then, a real sized version of the PDA is also displayed, showing the predefined options, corresponding to possible requests (see Fig. 8). Through each proactive actor's PDA, the stakeholder just has to select the desired option and then the corresponding sequences are executed (each one of these is formally related with one of the UML stereotyped sequence diagrams).

Each time one of the proactive actors is clicked, a black ball (representing the actor's request) is sent from the actor towards the cloud. In Fig. 14, the stakeholder interacting with the animation prototype chose the "consult patient state" option by using the PDA of the medical doctor. The snapshot in Fig. 14 corresponds to the exact moment in which the monitor is sending to the uPAIN system some physiological indicators about the patient; this data exchange is graphically represented by the black ball trajectory in the display. This snapshot also shows a log window, where all the requests and interactions are registered. At the same time, underneath the cloud, a textual expression "receiving patient data" identifies the ongoing request/interaction. A caption, identifying the selected option is displayed during the whole action in the upper left corner of the display to prevent stakeholders from forgetting the task at hand and to provide them feedback, a golden rule of GUI design suggested in [22]. It is crucial for stakeholders that the animation prototype lets them know at what point they are, at any given time in a clearly understandable way. Additionally, in Fig. 8 it is possible to observe a green coloured "Ready…" message informing that the animation prototype is ready to accept one mouse click in one of the buttons of the displayed PDA. If, in any point of the simulation, the actor "uPAIN system" is in processing state, then a spinning globe appears inside the cloud and a red coloured "Running…" message is presented (see Fig 12).

To assure that the purpose of any graphical entity is clearly apparent and inferred (an important cognitive dimension in GUI design to deal with expressiveness [23]), a green dashed line contour was added around each proactive actor to make clear that only these are the proactive actors on which it is possible to click to produce some kind of interaction (Fig. 15). The green coloured "Ready…" message also appears when these green dashed line contours are displayed. We also used a dashed line and different background colours to help delimit the three main areas of the GUI and grouping actors in a logical way, according to the areas in the hospital where they may be: the patient, the monitor and the PCA are always in the infirmary; the two databases are installed in the server's room; the medical doctor and the nurse can be elsewhere due to the nature of uPain system (ubiquitous); and the uPain system is "everywhere" in the hospital and so the cloud is placed in the middle of the three dashed areas. Below each actor, and to ensure that the actor is clearly identified immediately, the respective caption was added, since good labelling can guide stakeholders through the GUI with minimal search time. We also labelled the three dashed areas. This approach in GUI design contributes for a reduced cognitive load and immediate recognition in detriment of recalling in order to let stakeholders make optimal use of their high level cognitive abilities and save them to perform the essence of work; i.e., using the high level cognitive capacity for the more demanding work tasks such as workflow requirements validation, which is the real aim of the animation prototype.

The reduction of short-term memory load [23] was another intended goal, once in that part of memory only few information elements (typically, 5 to 8) can be stored simultaneously and the decay time is short (approximately 15 sec.). Thus, we avoided a dense area with many elements and presented only the necessary information.

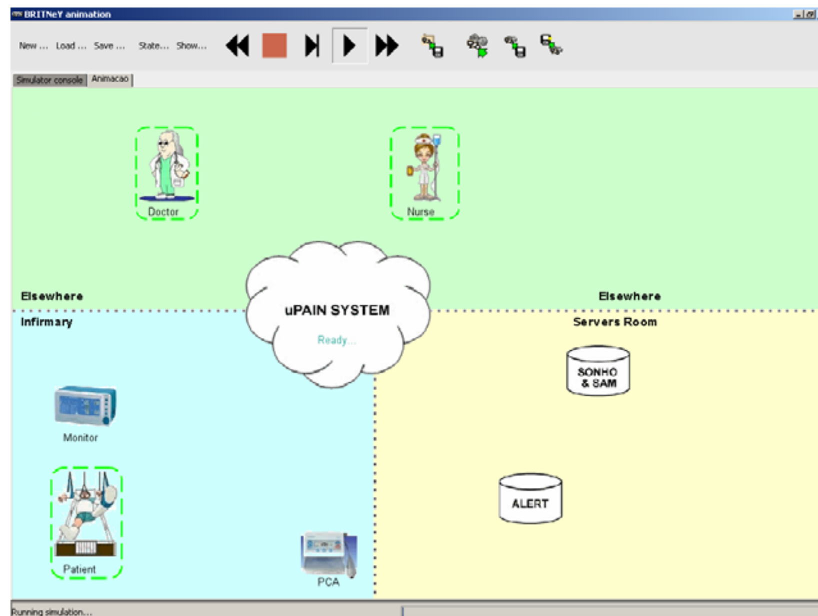


Fig. 15. Dashed line contours in the animation prototype for the uPAIN system.

The animation prototype was first demonstrated to the stakeholders with a strong involvement of the developers to explain the main approach to its usage as a software artefact to support the early execution of functional requirements. After that, the stakeholders have been given a standalone version of the animation prototype. This usage of the animation prototype has enabled the effective validation of requirements, since stakeholders generate frequently change requests to incorporate new scenarios and to adjust others already elicited, which has definitively contributed to the rapid evolution of the requirements model maturity, prior to design phase. We believe the usability concerns we adopted in designing the whole animation prototype was determinate to the success of the uPAIN project.

6. CONCLUSIONS

Static requirements models should not be used to directly base the validation of the elicited user requirements by the stakeholders, since the effort to use only elements from the problem domain in the user requirements models and to avoid any reference to elements belonging to the solution domain is not enough to obtain requirements models that are capable of being fully understandable by common stakeholders. The stakeholders' comprehension of the dynamic properties of the system within its interaction with the environment is better assured if animation prototypes, formally deduced from the elicited static requirements models, are used.

The behaviour of the animation prototypes can be specified by using CPNs rigorously translated from use case and stereotyped sequence diagrams. An effective execution of UML models can be achieved by using CPN Tools to operationally imple-

ment the interaction with the stakeholders within their efforts to validate the previously elicited workflow requirements models. Presently, the referred transformations are executed manually, which can be considered a major drawback of the proposed approach when the system to animate is of large dimension, presenting a great number of use cases and a large amount of behavioural scenarios to transform into CPNs.

The generation of standalone versions of the interactive animation prototypes motivates stakeholders to get a deeper involvement in the analysis phase (without the interference of the development team). Usability features of the animation prototypes must also be carefully studied and experimented, before reaching the final version of the prototype in supporting the interactive execution of the elicited sequence diagrams, complementary elicitation of workflow requirements and validation of the requirements models. CPN Tools and *BRITNeY Animation tool* should evolve to support better the transparent generation of this kind of standalone versions and to allow a simpler start-up of an animation.

As future work, we intend to automatically generate CPN skeletons from workflows requirements models (use case and stereotyped sequence diagrams). Additionally, we will study the possibility of using CPNs, constructed for specifying the behaviour of the animation prototype, to base the behavioural specification of the elements that will compose the architecture of the system within the design phase. If data-flow languages (such as LabVIEW, as described in [24, 25]) are used to develop the semantic layer responsible for integrating the whole ubiquitous system (embedded and mobile devices, database accesses and a service-oriented architectural platform), the asynchronous nature of CPNs will smooth the transition from analysis to design phases in what regards behavioural models. A semantic layer in the Arena environment [26], capable of accepting CPN-based workflow specifications, will also be developed to allow the stochastic execution of workflow scenarios as a complement to the current validation approach based on CPN Tools.

References

1. IEEE 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology, 1990.
2. D. Zowghi, C. Coulin. Requirements Elicitation: A Survey of Techniques, Approaches, and Tools. In A. Aurum and C. Wohlim (Eds.), Engineering and Managing Software Requirements, pp. 19–46, Springer-Verlag, July, 2005.
3. R.J. Machado, I. Ramos, J.M. Fernandes. Specification of Requirements Models. In A. Aurum and C. Wohlim (Eds.), Engineering and Managing Software Requirements, pp. 47-68, Springer-Verlag, July, 2005.
4. Y. Liang. From Use Cases to Classes: a Way of Building Object Model with UML. Information and Software Technology, no. 45, pp. 83–93, 2003.
5. J. Whittle, R. Kwan, J. Saboo. From Scenarios To Code: An Air Traffic Control Case Study. Software and Systems Modeling, vol. 4, no. 1, pp. 71-93, Springer-Verlag, Feb/2005.
6. I. Krüger, R. Grosu, P. Scholz, M. Broy. From MSCs to Statecharts. In F.J. Rammig (Ed.), Distributed and Parallel Embedded Systems, pp. 61-72, Kluwer Academic Publishers, 1999.
7. R.J. Machado, J.M. Fernandes, P. Monteiro, H. Rodrigues. Transformation of UML Models for Service-Oriented Software Architectures. 12th IEEE Int. Conference on the Engineering of Computer-Based Systems (ECBS 2005), Greenbelt, Maryland, U.S.A., pp. 173-182, IEEE CS Press, April, 2005.

8. W.M.P. van der Aalst. Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management. In J. Desel, W. Reisig, G. Rosenberg (Eds.), Lecture Notes in Computer Science 3098, pp. 1-65, Springer-Verlag, 2004.
9. M. Beaudouin-Lafon, W.E. Mackay, P. Andersen, P. Janecek, M. Jensen, M. Lassen, K. Lund, K. Mortensen, S. Munck, A. Ratzer, K. Ravn, S. Christensen, K. Jensen. CPN/Tools: A Post-WIMP Interface for Editing and Simulating Coloured Petri Nets. 22nd International Conference on Applications and Theory of Petri Nets (ICATPN 2001), Newcastle upon Tyne, UK, June, 2001.
10. A. Gemino. Empirical Comparisons of Animation and Narration in Requirements Validation. Requirements Engineering, vol. 9, pp. 153-168, Springer-Verlag, November, 2003.
11. P. Fenkam, H. Gall, M. Jazyeri. Visual Requirements Validation: Case Study in a Corba-supported Environment, IEEE Joint International Conference on Requirements Engineering (RE'2002), 2002.
12. M.B. Ozcan, P.W. Parry, I.C. Morrey, J. Siddiqi. Requirements Validation Based on the Visualisation of Executable Formal Specifications. International Conference on Computer Software & Applications, pp. 381-386, Austria, IEEE CS Press, 1998.
13. S. Uchitel, R. Chatley, J. Kramer, J. Magee. Fluent-based Animation: Exploiting the Relation between Goals and Scenarios for Requirements Validation, 12th IEEE Requirements Engineering International Conference (RE'04), 2004.
14. V. Winter, D. Desovski, B. Cukic. Virtual Environment Modeling for Requirements Validation of High Consequence Systems, Proceedings of the IEEE International Conference on Requirements Engineering, pp. 23-30, 2001.
15. K. Jensen. Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volumes 1-3. Monographs in Theoretical Computer Science. Springer-Verlag, 1992-1997.
16. L.M. Kristensen, S. Christensen, K. Jensen. The Practitioner's Guide to Coloured Petri Nets. International Journal on Software Tools for Technology Transfer, no. 2, pp. 98-132, 1998.
17. BRITNeY Animation tool. wiki.daimi.au.dk/tincpn
18. N. Pryce, J. Magee. SceneBeans: A Component-Based Animation Framework for Java. http://www-dse.doc.ic.ac.uk/Software/SceneBeans/
19. ISO 9241-11: Guidance on Usability, 1998.
20. S.J.P. McDougall, M.B. Curry, O. de Bruijn. Exploring the Effects of Icon Characteristics on User Performance: The Role of Icon Concreteness, Complexity, and Distinctiveness. Journal of Experimental Psychology: Applied, vol. 6, no. 4, pp. 291-306, 2000.
21. S.J.P. McDougall, M.B. Curry, O. de Bruijn. The Effects of Visual Information on Users` Mental Models: An Evaluation of Pathfinder Analysis as a Measure of Icon Usability. International Journal of Cognitive Ergonomics, vol. 5, no. 1, pp. 59-84, 2001.
22. M. Welie, G. van der Veer, A. Eliëns. Breaking Down Usability. Interact 99, Edinburgh, Scotland, 1999.
23. J.F. Pane. A Programming System for Children that is Designed for Usability. PhD Thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, USA, May, 2002.
24. R.J. Machado, J.M. Fernandes.. Heterogeneous Information Systems Integration: Organizations and Methodologies. In M. Oivo, S. Komi-Sirviö (Eds.), 4th International Conference on Product Focused Software Process Improvement (PROFES'02), pp. 629-643, Rovaniemi, Finland, Lecture Notes in Computer Science Series 2559, Springer-Verlag, December, 2002.
25. R.J. Machado, J.M. Fernandes. Integration of Embedded Software with Corporate Information Systems. In A. Rettberg, M.C. Zanella, F.J. Rammig (Eds.), From Specification to Embedded Systems Application, IFIP Series vol. 184, Springer-Verlag, September, 2005.
26. W.D. Kelton, R.P. Sadowski, D.A. Sadowski. Simulation With ARENA, 2nd edition, McGraw-Hill, 2002.

Analysis of resource-constrained processes with Colored Petri Nets

Mariska Netjes, Wil M.P. van der Aalst, Hajo A. Reijers

Eindhoven University of Technology, Faculty of Technology and Management,
(PAV J5,) PO Box 513, NL-5600 MB Eindhoven, The Netherlands
Contact: m.netjes@tm.tue.nl

Abstract. Formal models of business processes support the performance analysis of processes and the evaluation of redesign alternatives. This paper presents a formal model to analyze the behavior of resource-constrained processes. The model is developed using Colored Petri Nets (CPN or CP nets) and the supporting software package CPN Tools. In our approach, a business process consists of tasks and resources able to perform one or more tasks in the process. We developed a task building block to model tasks and a resource module to model the allocation of resources with different allocation methods. The opening of a bank account is used as an example process to investigate two so-called "best practices" while using the simulation facility of CPN Tools. First, we explore the specialist-generalist trade-off, i.e., finding the optimal ratio of specialists and generalists. Then we explore the flexible assignment policy, i.e., a strategy to deploy specialists first to preserve operational flexibility.

1 Introduction

Organizations are constantly looking for ways to improve their performance. By emphasizing business processes rather than hierarchies and by putting a focus on customer satisfaction, organizations tend to attain better overall performance, a better *esprit de corps* and less interfunctional conflicts [16]. A survey among 90 Swedish organizations, ranging from small consulting firms, hospitals and state-owned companies to multinational companies such as ABB, Ericsson, Saab and Volvo, pointed out that organizations seem to be quite satisfied with the effects of such a process orientation and that they were planning to allocate more resources to their process initiatives [7].

A process initiative can take on different forms, for example, as a project to rethink the underlying process structure, to change resource allocation strategies, or by the introduction of new technology such as workflow management systems. In this paper, we focus on a number of *best practices* in this area. A best practice prescribes a historical solution that seems worthwhile to replicate in another situation or setting, although it may need to be adapted in skilful ways in response to prevailing conditions. A collection of best practices for business process improvement is given in [18]. It should be noted here that many of the best practices lack adequate (quantitative) support. It is our objective to investigate two of these best practices in more detail in this paper, i.e., the so-called *specialist-generalist trade-off* and the *flexible assignment policy*. We would like to obtain insights in the underlying mechanisms and see under which conditions the best practices indeed provide improvements.

The *specialist-generalist trade-off* aims at *finding the optimal ratio of specialists and generalists* in a process. Given a fixed number of resources, the question is how many of the resources should be specialized resources and how many of them should be generic resources. We define a specialist as a resource able to perform exactly one task. Routine is built up in doing this task and as a result he or she performs the task faster. A generalist is able to perform more tasks and adds flexibility to the process leading to a better utilization of resources [18].

Using a *flexible assignment policy* means that resources are *assigned to the work in such a way that maximal flexibility is preserved for the near future*. When both specialists and generalists are available to perform a certain task the most specialized resource is assigned to the task. By doing this, more generic resources are 'saved' for other tasks for which this specialized resource might

not be suitable. In this way, the waiting time for a suitable resource may be reduced. Another advantage is that the specialized resource is more skilled and will need less time to perform the task [18].

In this paper, we focus on an abstraction of an actual business process as a collection of a number of inter-related *tasks*, where a limited number of *resources* (people, systems, machines, etc.) is available to execute the process. To be more precise, a task is an "atomic" and logical unit of work, which is carried out in full or not at all. A task which is just about to be executed for a specific case (or process instance) is called a *work item*. Each work item should be performed by one resource suited for its execution. This implies that, while there may be several resources able to perform a given work item, exactly one resource should be assigned to perform the work item. Note that we reserve the term *activity* for the actual execution of a work item [3]. Resources are grouped into roles to facilitate the mapping of resources on work items. A role is a group of resources with similar characteristics. A resource has one or more roles and each role may be performed by many resources [12]. The objective of this paper is to give an approach for analyzing the performance of such a *resource-constrained process* and to show how a choice between alternative designs for a process using best practices can be supported with this kind of analysis.

For the modeling and analysis of resource-constrained processes, CPN Tools is used. This tool provides support for the construction, simulation and performance analysis of high-level Petri nets [11]. CPN Tools is chosen, because it combines the strength of Petri nets with the strength of programming languages. The reasons for using Petri nets to model business processes are stated in [1]. The basic behavior of a process is modeled and visualized with Petri nets and more sophisticated behavior is added through ML functions. It is possible to debug the model and validate the correct behavior of the model with a step-by-step simulation of the model. Alternatively, the functional correctness of the system can be validated and verified with state space analysis. Once the model is validated, it is easy to make adaptations to it and create alternatives for the original model. The simulation environment in CPN Tools also has the capability to perform an automatic sequence of firings to examine the behavior of a model in the long run [20]. The combination of time and simulation in CPN Tools provides the possibility to analyze the performance of the modeled process. For an introduction into high-level Petri nets and CPN Tools the reader is referred to [8,9,11,20].

The structure of the paper is now as follows. Section 2 describes the developed CPN model and explains the task building block, the resource module and the different allocation methods in more detail. Section 3 describes the application of the developed CPN model to investigate the mechanisms of the specialist-generalist trade-off and the flexible assignment policy. Section 4 discusses the related work and section 5 gives the conclusions and proposes future work.

2 Development of the CPN model

Before investigating the *specialist-generalist trade-off* and the *flexible assignment policy*, we describe the developed CPN model. Both best practices aim at the improvement of resource-constrained processes. The purpose of the model is to provide an easy way to model such a resource-constrained process and change it to evaluate the use of the best practices under different circumstances. In this section we first give an overview of the top level of the developed model to show its main parts. Further, we explain the sub models which represent the different elements of a general resource-constrained process. These elements are the generic task building block, the generic resource module and different resource allocation methods.

2.1 Overview of CPN model

In Figure 1, an example of the top level or main page of the CPN model (for a process consisting of two sequential tasks) is shown. The different parts of the CPN model are the generator, the process (a number of tasks in some relation to each other, e.g. sequential) and the generic resource module for the allocation of resources to the process.

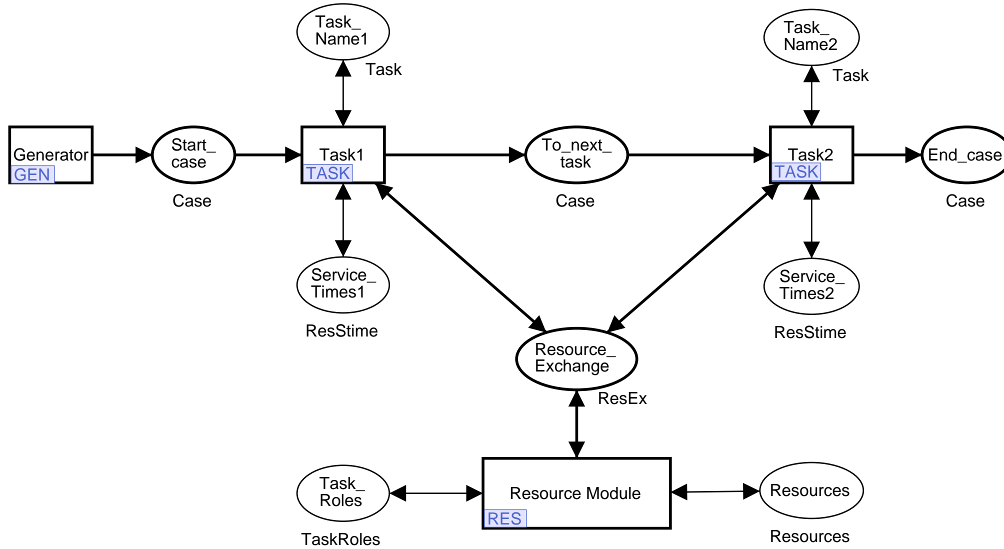


Fig. 1. Top level

We give a global description of each of the parts to explain how the model works. The `Generator` generates cases which flow through the process. The process consists of `tasks` placed in some configuration, allowing for both sequential and parallel configurations. To ensure rapid and straightforward modeling of (large) processes a generic task sub model is developed. This sub model gives each task the same layout and functionality, because for each additional task an instance of the generic task sub model is used. At the top level the specific task variables need to be configured, because these specific variables distinguish one task instance from another. For each task the task name needs to be added to the place `Task_Name` and the service times (one for each role that may perform the task) need to be added to the place `Service_Times`. The `Task` and `Resource Module` substitution transitions communicate and exchange resources through the place `Resource_Exchange`. The `resource module` arranges the allocation of resources to work items in a generic way. The resource module is generic, i.e., the number of tasks and resources in the process is not predefined. Only a specification of the roles able to perform a specific task and the available resources is required. For each task, the associated roles need to be specified in the place `Task_Roles` and the available resources need to be placed in the place `Resources`. Available resources are named according to their role, because the role of the resource is the only attribute required for the mapping of resources to tasks. According to the standard definition of a role [3], it is possible that resources have more than one role, but in this model the assumption is made that every resource fulfills exactly one role. This is not necessarily a limitation, because resources with more roles can be linked to a 'meta'-role enclosing these roles. For example, if a resource has roles r1 and r2, we could add a new role r12 which is attached to all tasks that require r1 or r2.

2.2 Task building block

A process model consists of a number of tasks placed in a certain order. When a work item flows through the process, service times could differ per task and depend on the type of resource performing the task. Each task has the same basic functionality with a different value for the service time and different resource types. A generic task sub model enclosing the required functionality is developed. This model is called a task building block, because a process can easily be built with these blocks. It is possible to place the building blocks in a sequential order or to put them in parallel. The task building block is depicted in Figure 2. The main functions of the task block are requesting a suitable resource and putting a time delay on the case and the involved resource. An explanation of the color sets used in Figure 2 is given in Table 1.

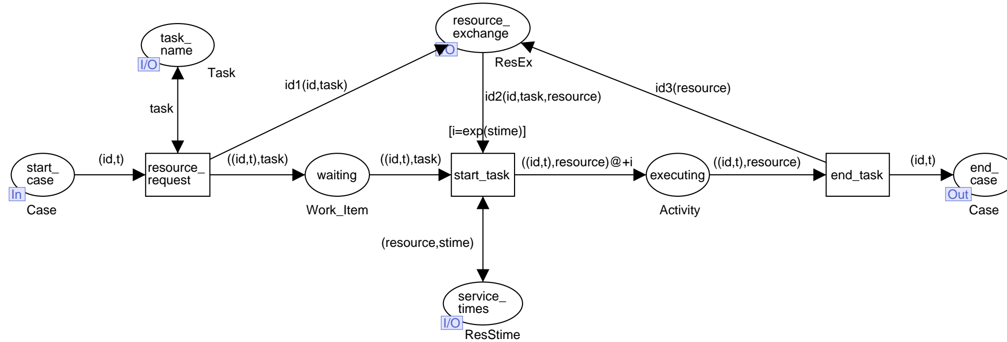


Fig. 2. Task building block

Table 1. Color sets

Color sets task building block	**Color sets resource module**
color I = int	color ID = int timed
color ID = int timed	color Task = string
color T = int	color Prerequest = product ID * Task
color Case = product ID * T timed	color Role = string
color Task = string	color Roles = list Role
color Work_Item = product Case * Task	color TaskRoles = product Task * Roles
color Stime = int	color Request = product ID * Task * Roles
color Role = string	color Requests = list Request
color Resource = Role	color Resource = Role
color ResStime = product Resource * Stime	color Resources = list Resource
color Activity = product Case * Resource timed	color Match = product ID * Task * Resource
var i:I	color ResEx = union id1:Prerequest+id2:Match +id3:Resource
var id:ID	var id:ID
var t:T	var task:Task
var task:Task	var roles:Roles
var stime:Stime	var request:Request
var resource:Resource	var requests:Requests
	var resource:Resource
	var resources:Resources
	var match:Match

Before we describe the details of the task block we will look at the interaction between a task building block and the resource module. A task building block communicates with the resource module through the place `resource_exchange`. The purpose of the communication is obtaining a suitable resource from the resource module to perform a specific work item. After completion of the task the resource is returned to the resource module. In Figure 2 we see two arcs going to and one coming from the place `resource_exchange`. A token travelling the left arc going to the place `resource_exchange` represents the request for a resource and for requesting two attributes, the case id and the task name, are required. The middle arc returns a token representing the request and the resource attached to it. The attributes on this arc are the case id, the task name and the resource. With the right arc the resource returns to the resource module. We see that the sets of attributes, or color sets, on the inputs and output of the place `resource_exchange` differ. However, the place `resource_exchange` should have a color set consistent with each of these input and output colors. We make a *union* place from the place `resource_exchange` uniting the different color sets to have one consistent color set. The color set of the place is a union of the different input and output color sets. Each arc has an identifier with the required attributes to specify the color set related to the arc. With the place `resource_exchange` specified as a *union* place different tokens with different color sets may pass it.

Let us now consider the task building block in more detail. A case enters the task block via the place `start_case`. A case has attributes specifying a unique case id and the time the case was generated. The task name is stored in the place `task_name`. The task name is added to the case attributes, because it is necessary for the resource request. By adding the task name the case is changed to a work item, which needs to be executed. A resource request stating the case id and the task name is sent to the resource module via place `resource_exchange`. The work item waits in the place `waiting` until a work item with the case id and the same task name (together with the allocated resource) is available in the place `resource_exchange`. The mapping on the waiting work item is done based on both the case id and the task name to allow for parallel execution of tasks for the same case. Recall that the term activity [3] refers to the actual execution of a work item. The average service time needed by the resource to complete the activity is received from the place `service_times`. The duration or actual service time is modeled as a negative exponentially distributed delay. When the delay time of the activity is passed, the resource is returned to the place `resource_exchange` and the case leaves the task block via the place `end_case`.

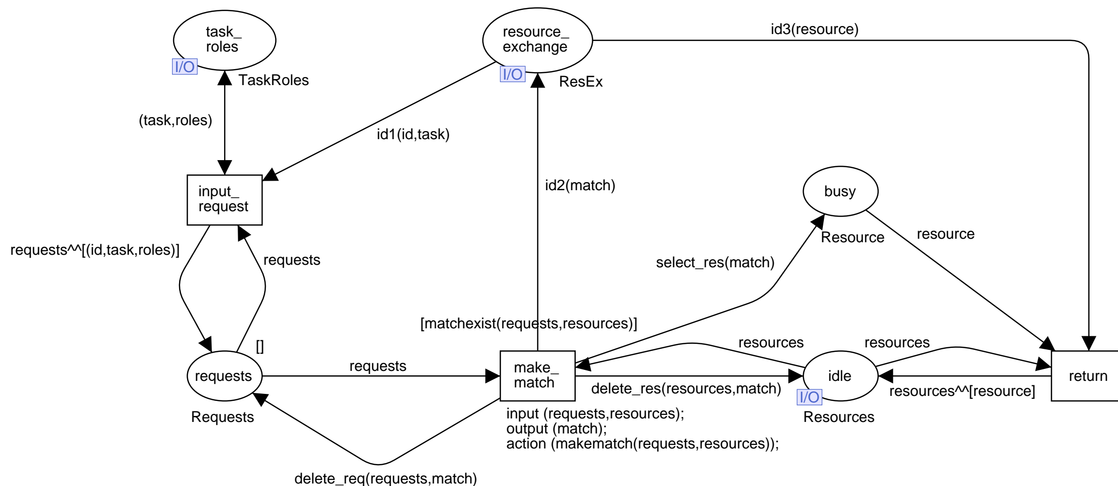


Fig. 3. Resource module

2.3 Resource module

The resource module has been developed to allocate resources to work items. In the module, each resource is assigned to exactly one role and the resource name is equal to the assigned role. For instance, a resource with role *postman* is called a postman. If there is more than one postman in the process, then all resources with role *postman* are called postman. A role is linked to one or more tasks, i.e., a resource with a certain role is capable of performing one or more tasks. Resources are allocated to work items by mapping the role of the resource with the task that needs to be executed for the work item.

The resource module is a generic solution, i.e., the use of the model is not limited to a specific number of tasks or resources and the functionality is the same for every resource-constrained process model. Figure 3 shows the resource module and Table 1 contains the color sets of the resource module. We will explain the resource module in more detail starting in place `resource_exchange` and following the arcs from there.

A resource request for a certain work item is received in the place `resource_exchange`. The resource request has the attributes case id and task name, because both attributes are necessary to return the filled request to the right work item. At the first transition, `input_request`, the roles that may perform the work item are added to the request. The request (with attributes case id, task name and roles) is put in a list with requests ordered in a First Come First Serve order. This is done by putting an arriving request at the end of the requests list. The requests will be matched with the available resources starting from the top of the requests list. The combination of the case id and the task name from the request and the allocated resource is called a match. When more role types are able to perform a certain task an allocation method is necessary to allocate role types in a specified order. Note that in the next section different allocation methods and the associated functions are described. The available resources are stored in the place `idle`. If a match exists, the transition `make_match` is enabled. When enabled, an attempt is made to match the first request from the requests list with an available resource. When a resource with the required role is available a match is found. A match is transferred to the place `resource_exchange` and from there to the corresponding task. The matching request and resource are deleted from the requests list and the resources list. Next to this, busy resources are stored in the place `busy_res`. This place can be perceived as redundant, but it is added so show which resources are actually busy. From head to tail the requests list is evaluated and matches are made. Requests for which no resource is available wait and stay in the requests list. When a resource is finished working on a task, it is put back in the resources list via place `resource_exchange` and transition `return`.

One could think that the model has some restrictions. One possible restriction is that it is not possible to withdraw an assignment of some work to a resource. This functionality is not added, because human resources will become demotivated when work is taken from them just before they will start working on it. More on assignment policies, the possible choices and the subsequent consequences can be found in [23]. Another point that can be seen as a limitation is that a resource can not work on several activities at the same time. This limitation follows from the definition of a task as an "atomic" piece of work. Once started, the task needs to be finished before the resource will be able to start another task [3].

2.4 Resource allocation methods

The resource module allocates resources to work items based on a certain allocation method. An allocation method specifies the order in which the allocation of different resource types should take place. Two allocation methods, priority based allocation and random allocation, are modeled for incorporation in the resource module. We have chosen to define allocation methods with ML functions. The functions are made on two levels, the top level functions are the same for each allocation method and these functions call lower level functions which define the specific allocation method. By doing so, the CPN sub model representing the resource module is the same regardless of the chosen allocation method, because the resource module only refers to the top level functions. We will briefly discuss the top level functions, they are all mentioned in Figure 3. All functions are

connected to the transition make_match in the resource module. This transition is enabled when the precondition in the guard is satisfied. The precondition function matchexist evaluates if at least one possible match exists between the list with requests and the resources. When the transition make_match is enabled it performs an action. The action part of the transition make_match is used to model the action and the action part is placed under transition make_match (see Figure 3). In the action part the input, the output and the actual action are specified. The inputs for the action part are the list with requests and the resource list. The actual action is the evaluation of the function makematch. When evaluated this function calls the lower level functions for a specific allocation method. The output of the action part is a match between a request and a suitable resource allocated with the underlying allocation method. This output, match, is used to put the right value on each arc going from transition make_match to one of the surrounding places. The match itself, with attributes case id, task name and resource, is returned to the requesting task via the place resource_exchange. The function delete_req evaluates the requests list and deletes the request which is in the match. The function delete_res deletes the matching resource from the resources list. The function select_res places the resource assigned to the match in the busy place.

We have predefined an allocation method to assign resources on priority and a method to assign resources randomly. The general idea of both methods will be described. We will first explain the priority based allocation and then the random allocation.

With priority based allocation it is already known before the actual allocation in which order resources will be assigned to work items. The roles able to perform a task are prioritized and the order in which resources are assigned is based on the priority of their role. The priority of the roles is defined in the place task_roles. For each task the roles able to perform the task are specified. The roles are ordered from high priority to low priority. From the available resources the resource with the highest priority role is selected and assigned.

With random allocation the available resources with a suitable role all have an equal change of being selected. For each resource request a preselection of all resources with a suitable role is made. From this preselection a resource is randomly chosen and allocated.

For validation purposes a tandem M/M/1 queueing system has been modeled with the task building blocks and the resource module. The simulation results from this model are equal to the results calculated with queueing theory [10]. This validation suggests the model is working as we expected. To collect the statistics we have used the monitoring facility of CPN Tools. This facility has not yet been released. However, we were able to use a prerelease. The facility allowed us to collect measurement data without changing the functional model and thus very helpful in maintaining the readability of the models.

3 Application of the CPN model

In this section we will use the developed CPN model to investigate how process performance is influenced by the ratio of specialized and generic resources in the process and the allocation method applied to these resources. In this section we will first give a short introduction on the best practices and the example process on which they are applied. Secondly, we will apply the specialist-generalist best practice to find the optimal ratio of specialists and generalists. Next to that, we will investigate when flexible assignment of resources should be applied.

3.1 Introduction

We would like to have more (quantitative) insight in the best practices for business process improvement. In this paper we consider *two best practices related to the assignment of specialized or generic resources.* We will investigate under which conditions process improvements are achieved with the application of the best practices. We look for improvements of the average throughput time of the process and this measure is used as performance indicator. With the developed CPN

model we model an example process on which we apply the best practices. This example process deals with the opening of a bank account. A short description of this bank process is as follows. When a registration for a bank account is received the data is entered into the system. Secondly, the bank account is initiated and finally a letter is sent to the customer. The process model is built with three task building blocks and the resource module and is depicted in Figure 4.

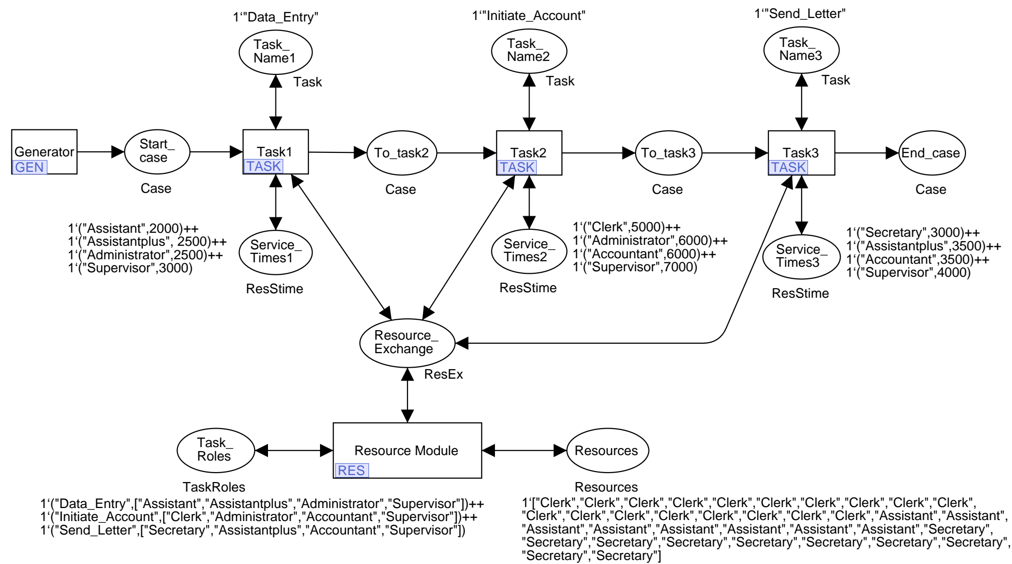


Fig. 4. Process model: Opening a bank account

In the model each task has two places specifying the associated task variables. The task names are stated above place `Task_Name`. The service times are stated next to place `Service_Times`. For instance, the first task named 'Data_Entry' can be executed by one resource with any of the four stated roles. Each role needs a different average service time to execute the task (e.g. the first role 'assistant' needs on average 2 minutes to perform a data entry). The resource module also has two specification places. The place `Task_Roles` describes for each task which roles are able to execute it. The task 'Data_Entry' is linked to the roles 'assistant', 'assistantplus', 'administrator', and 'supervisor'. The roles are ordered from specialized resources to generalists. The assistant is specialized in entering data, the assistantplus can enter data and send letters, the administrator could do data entries and account initiations and the supervisor is a generalist and is able to do all three tasks. The place `Resources` contains the resources available for the process. In Figure 4 there are 36 specialists available, namely 18 clerks, 8 assistants, and 10 secretaries.

3.2 Specialist-generalist trade-off

In this section we will investigate one of the two best practices, the specialist-generalist trade-off, and its underlying mechanisms. This best practice aims at *finding the optimal ratio of specialized and generic resources*. This implies that the total number of resources is kept constant, while it is decided how many of the resources should be specialized and how many should be generic to have an optimal process performance. The performance, indicated by the average throughput time, of a mixture of specialists and generalists is determined by two mechanisms.

- The specialist performs the work faster and using specialists will lower the average service time spent on a case. A generalist needs more time for service and this first mechanism is in favor of the specialized resources.

– According to our definition a specialist can only perform one task. When the utilization of the resources increases, work items will have to wait longer for 'their' specialist. The use of generalists will add flexibility and a better utilization of resources to the process. The second mechanism favors generic resources, because they are suited for more tasks and using generalists will lower the average waiting time of a case.

We would like to have more insight in the trade-off between these two mechanisms and find the optimal ratio of specialists and generalists for the bank process.

According to the description of the specialist-generalist trade-off we should try to *find the optimal ratio of specialists and generalists*. In the bank process there are 36 resources available and each resource could perform one out of seven roles. This means we can form approximately 6 billion different ratios of specialists and generalists, which all should be evaluated. And each result should be compared with the results for the other ratios to see which alternative has the optimal ratio of specialists and generalists. We think a more pragmatic and more efficient approach can distinguish at least one ratio that is favorable compared to other ratios. For the bank process we would like to find an alternative process that has *a throughput time significantly lower than the initial situation with only specialists*. To find this alternative with a *distinctive ratio of specialists and generalists* we first perform a global search for alternatives that seem fruitful for further investigation. From these alternatives we will derive promising alternatives to evaluate and enclose the alternatives with a *distinctive* ratio.

For the global search we make several process alternatives with specialist-generalist ratios ranging from mainly specialists to mainly generalists. In our initial process model (see Figure 4) only specialized resources are available to execute the process. The roles of resources stored in place `Resources` are easily changed to create an alternative process model. Note we have two types of generalists in the bank process. There are generalists like the administrator which are able to perform two tasks and this type is called `2task-generalist`. The other type of generalists is able to perform three tasks. Since there is only one such generalist type in the bank process we name this type `supervisor`. We form five alternative process models with different ratios of specialists and generalists. The initial process is called process alternative 0. The six process alternatives are stated in Table 2 with the ratio of specialists, 2task-generalists, and supervisors per alternative and the available resources.

Table 2. Definition of process alternatives

Alternatives	Ratio of spec-gen	Resources
0	36 specialists	8 assistants, 18 clerks, 10 secretaries
1	27 specialists 9 2task-generalists	6 assistants, 14 clerks, 7 secretaries 3 administrators, 3 accountants, 3 assistantsplus
2	18 specialists 18 2task-generalists	4 assistants, 9 clerks, 5 secretaries 6 administrators, 6 accountants, 6 assistantsplus
3	9 specialists 18 2task-generalists 9 supervisors	2 assistants, 4 clerks, 3 secretaries 6 administrators, 6 accountants, 6 assistantsplus 9 supervisors
4	9 specialists 9 2task-generalists 18 supervisors	2 assistants, 4 clerks, 3 secretaries 3 administrators, 3 accountants, 3 assistantsplus 18 supervisors
5	18 2task-generalists 18 supervisors	6 administrators, 6 accountants, 6 assistantsplus 18 supervisors

Each alternative allocates resources randomly, i.e., flexible assignment is not used. For each of the alternatives we perform a simulation. The arrival intensity is 200 arrivals per hour (Poisson arrival process) and for each alternative we run a simulation run divided in 30 sub runs handling 1000 cases each. We measure the average throughput time and the average utilization and for the

average throughput time we calculate a 95%-confidence interval. The results are presented in Table 3. For each alternative the 95%-confidence interval for the average throughput time and the average utilization rate are given. The average throughput time for two alternatives differ significantly if the confidence intervals do not overlap. The results show us that alternative 1 with nine 2task-generalists has the lowest average throughput time, although the difference is not significant when compared to alternative 0 (the initial process). Next to this, we see that alternative 5 (with only generic resources) could be significantly improved by specializing some resources, leading to alternative 4. The same holds for alternative 4 for which making more resources specialist would lead to the improved performance of alternative 3. The same argumentation could be applied to alternative 2 and 3.

Table 3. Results for random allocation

Alternatives	Throughput time	Utilization rate
0	[16.134 ; 18.604]	0.84
1	[14.541 ; 16.451]	0.89
2	[16.438 ; 19.086]	0.89
3	[29.694 ; 34.838]	0.92
4	[36.390 ; 43.550]	0.93
5	[51.831 ; 58.895]	0.94

It seems that a process with mainly specialists and some generalists could perform significantly better than alternatives with other ratios of specialists and generalists. Our next step is to focus our search on alternatives with approximately the same ratio as alternative 1 to find a *distinctive ratio* of specialists and generalists. Two configurations of a process with 6 2task-generalists and 30 specialists have an average throughput time which is significantly lower than the average throughput time of the initial process. The configuration with 2 administrators, 2 accountants, 2 assistantsplus, 8 secretaries, 16 clerks, and 6 assistants is called alternative 1a. The configuration with 2 administrators, 2 accountants, 2 assistantsplus, 9 secretaries, 16 clerks, and 5 assistants is called alternative 1b. The results for the alternatives 1a and 1b are stated in Table 4 and for comparison the result for the initial process is included. We can see that alternative 1a and alternative 1b outperform the initial process (and also alternatives 2-5).

Table 4. Results for distinctive ratios

Alternatives	Throughput time	Utilization rate
0	[16.134 ; 18.604]	0.84
1a	[13.529 ; 14.867]	0.86
1b	[13.852 ; 15.538]	0.86

The specialist-generalist trade-off suggests *the improvement of a process by using a distinctive ratio of specialists and generalists*. Our conclusion is that a *distinctive ratio* of specialists and generalists is a ratio with mainly specialists and one or a few generalists. With this ratio the best trade-off is made between the shorter service times of the specialists and the flexibility offered by a generalist. When applying the best practice to the problem the total number of resources has been kept constant. A similar problem would have been encountered if the total salary or another variable had been kept constant.

A precondition for finding a *distinctive ratio* is a high resource utilization, because then the advantage of shorter service times for specialists does not compensated for the flexibility offered

by one or a few generalists. When the utilization rate is lower suitable specialists will always be available and the flexibility is not necessary. Next to this, the distribution of the resources over the different specialized resource types should be proportional, so each resource type has an equal utilization rate. We also made the explicit assumption that generalists need more time to perform a task than specialists. If specialists and generalists have an equal service time, a process with only generic resources will have the best performance.

3.3 Flexible assignment policy

With the flexible assignment of resources *the most specialized (available) resource will be assigned to a work item*. In this way, the more generic resources are preserved and the possibilities for having a suitable resource for the next work item are maximal. There is a smaller chance that a case has to wait for a specific resource and it is expected that the overall queueing time in the process will be reduced. Next to the flexibility, a specialist performs a task faster than a generalist leading to lower service times. When we compare flexible assignment with random allocation we expect the following for the initial process and the five process alternatives (see Table 2 for the definition of the process alternatives). For the initial process both allocations will perform the same, because there are only specialists to allocate. Also, in the first alternative there are mostly specialists available and no difference is expected. In the second alternative the ratio specialists-generalists is half-half and it could be that flexible assignment will select a specialist more frequently than would happen randomly. For the alternatives 3 and 4 there are more generalists than specialists available and it is likely that flexible assignment will allocate significantly more specialists than random allocation would. We expect that allocating the specialist will lead to lower queuing and service times for the process. So, we expect a reduced average throughput time for process alternatives 3 and 4. Alternative 5 has two types of generic resources and maybe allocating the least generic resource will lead to an improvement.

We apply the flexible assignment policy to the six process alternatives specified in Table 2. We also apply it to the alternatives 1a and 1b (with a *distinctive ratio* of specialists and generalists). We use priority based allocation to model flexible assignment. In place *Task_Roles* (see Figure 4) the roles for each task are already prioritized with the specialized resource having the highest priority, the 2task-generalist having a middle priority and the supervisor having a lower priority. Through this priority setting an available specialist will be assigned and generic resources are preserved for the near future. The results of the application of flexible assignment to the 8 alternatives are presented in Table 5.

Table 5. Results for flexible assignment

Alternatives	Throughput time	Utilization rate
0	[16.703 ; 19.731]	0.84
1	[14.569 ; 16.613]	0.88
1a	[14.066 ; 16.158]	0.87
1b	[14.149 ; 15.687]	0.87
2	[15.817 ; 17.410]	0.90
3	[29.131 ; 34.279]	0.93
4	[38.070 ; 44.872]	0.92
5	[47.594 ; 54.428]	0.94

We see that alternatives 1a and 1b again have a significantly lower throughput time than the initial process and also alternative 1 performs significantly better. Next to this, we compare the results of flexible assignment (Table 5) with the results of random allocation (Table 3 and Table 4). Flexible assignment does not seem to improve process performance, because for all alternatives both allocation methods lead to similar results.

We expected that flexible assignment would have an impact on the performance of processes with many generalists and just a few specialists, because the outcome of the allocation methods would be different. With random allocation there is a high chance that a generalist would be selected, while flexible assignment would favor the selection of a specialist. Alternative processes 3 and 4 have 9 specialists and 27 generalists, but the results with flexible assignment do not differ from random allocation. When we look at the results for these alternatives we see that the utilization rate is rather high. When 36 resources have an utilization rate of 0.92 there are on average 33 resources busy and three resources available. Only a quarter of the total amount of resources is specialist and specialists can only perform one of the three tasks. It follows that most of the time only suitable generalists will be available when an allocation has to be made. Because of this, most of the time flexible assignment will also select a generic resource. This leads to the conclusion that flexible assignment will probably act the same as random allocation for processes with a high utilization rate, because in most cases no choice is offered between specialists and generic resources.

We perform new simulations for the alternatives 3 and 4 with less cases arriving to lower the utilization rate of the resources. Table 6 shows the results for an arrival intensity of 120 and an arrival intensity of 150 cases per hour (Poisson arrival process).

Table 6. Significant results for flexible assignment policy

Arrival rate	Alt.	Allocation method	Throughput time	Util. rate
120 cases per hour	3	Random allocation	[12.075 ; 12.249]	0.64
		Flexible assignment	[11.312 ; 11.478]	0.60
	4	Random allocation	[12.489 ; 12.647]	0.67
		Flexible assignment	[11.707 ; 11.993]	0.62
150 cases per hour	3	Random allocation	[12.276 ; 12.614]	0.78
		Flexible assignment	[11.814 ; 12.186]	0.76
	4	Random allocation	[13.076 ; 13.664]	0.81
		Flexible assignment	[12.563 ; 13.019]	0.79

We see that flexible assignment outperforms random allocation for both alternatives 3 and 4 and with both arrival rates. Another interesting point is that although the average throughput time is lower with an arrival intensity of 120 cases per hour the difference between flexible assignment and random allocation is larger. It seems that the proposed relationship between the utilization rate and having a choice between a specialist and a generalist is correct. Flexible assignment does not perform better than random allocation for processes with a high utilization rate, because then there is no choice between suitable specialists and generalists when an allocation has to be made.

With the application of the flexible assignment policy we have gained insight in when to use a flexible allocation method. The flexible assignment policy suggests *the improvement of a process by allocating a specialist* when a choice has to be made between specialized and generic resources. The application of flexible assignment to the bank process led to an improvement of process alternatives with more generic than specialized resources. Flexible assignment will select a specialist more frequent than would happen randomly if there is at least one generic resource. So, in general flexible assignment could improve each process with both specialized and generic resources.

Note that the benefits of flexible assignment will only be observed if the process is not loaded too heavily. There should be enough free specialists in the process to have a choice between suitable specialists and generalists. Next to this, the actual improvement gained with flexible assignment is dependent on the difference in service time between a specialist and a generalist. When the

specialist works faster there is not only the benefit of the preserved flexibility, but also of lower service times.

4 Related Work

In this section we briefly discuss related work on the two main topics of this paper. The first topic is the analysis of business processes with simulation. Regarding the analysis of business processes we describe its position in Business Process Management, the broad use of simulation, and the application of the simulation facility of CPN Tools to business processes. The second topic is the application of best practices. We present their general background, the history of the two evaluated best practices, and an empirical study on the use of specialists and generalists in a call center.

The redesign of business processes based on best practices is part of Business Process Management (BPM). BPM is defined as "Supporting business processes using methods, techniques, and software to design, enact, control, and analyze operational processes..." [4]. Within the four defined phases the focus has traditionally been on enactment and control [4]. In this paper we give attention to (re)design and analysis.

The analysis of business processes is performed with simulations. From practice, the use of simulation is advocated to compare the "to be" alternatives and to understand the "what" and the "why" of the current process and possible alternatives [5]. From a scientific point of view, simulation is used to evaluate quantitative criteria such as throughput time or costs as input for design decisions [6].

For the analysis of the best practices the simulation facility of CPN Tools was used. CPN Tools and Design/CPN, the predecessor of CPN Tools, have been applied in various industrial projects [24], but we have found only two projects for which a business process has been analyzed with simulation. In one project a planning process was modeled with Design/CPN and this process was simulated with the Design/CPN simulator [14]. In another project CPN Tools was used to study the bullwhip effect in supply chains. For this project the supply chain was modeled with CPN Tools and with simulation the bullwhip effect, inventory increase in the chain, was demonstrated [13].

Best practices should be seen as independent rules of thumb helping practitioners in their redesign effort. Best practices are often derived from practical experience within companies and 29 of these best practices have been collected in [18].

In this paper we have evaluated the application of two of these best practices, the specialist-generalist trade-off [17,19] and the flexible assignment policy [3]. [17] identifies cross-training as a process improvement point and refers to cross-training as *training staff to make them capable of performing each others' jobs*. When a specialist should become more generic cross-training is required. A large survey on techniques used for Business Process Re-engineering (BPR) showed that companies consider the training of employees as the most important technique [21]. Toyota, for instance, used a system of cross-training and job rotation to enrich the jobs of their employees [15].

So far, both best practices have been described qualitatively, but little research has been conducted on the underlying mechanisms and the actual improvement gained when applied. An experimental study on the use of specialists and generalists in a call center has been performed by [22]. With simulation it is evaluated if 1) a *one-level* design should be used with only specialists or 2) a *two-level* design with generalists on the first level receiving the call which could be forwarded to specialists on the second level. The results show that a *one-level* design leads to better quality measures and a more regular load sharing between employees, but to higher labor costs.

5 Conclusions and Future Work

In this paper we developed a model for the analysis of resource-constrained processes and the evaluation of process alternatives. The model represents an abstraction of a business process

consisting of a process structure formed by task building blocks placed in sequence or parallel, a generic resource module and a method to allocate resources. While applying the specialist-generalist trade-off and the flexible assignment policy, many process alternatives were modeled and evaluated. The developed CPN model turned out to be very useful for comparing the performance of many process models in a short time. The addition of more tasks or a change in the ordering of the tasks may also be done in a short period of time.

With the application of the specialist-generalist trade-off to a process, a distinctive ratio of specialists and generalists can be found. The distinctive ratio is a combination of mainly specialists and one or a few generalists. Using this ratio instead of the current ratio could improve the process significantly. With a flexible assignment policy a specialist is assigned to a work item when both suitable specialized and suitable generic resources are available. The application of this policy to a process with both specialized and generic resources could lead to an significant improvement of the process. It is remarkable that the use of the specialist-generalist trade-off and the flexible assignment policy is influenced by the same mechanisms, but that the working of these mechanisms is opposite to each other. A useful application of the specialist-generalist trade-off requires high utilization rates and approximate equal service times for specialists and generalists to be effective, while the flexible assignment policy pays off if the utilization rates are lower and the service times of specialists and generalists differ.

We see opportunities for the extension of the developed CPN model. Other process characteristics like different case types and different customer classes influence the allocation of resources and could be added to the model. Next to this more performance indicators could be taken into account. The current model focuses on the timing aspect, but also the performance on cost, quality or flexibility could be interesting. In our model the requests for resources are handled in a FIFO order. The use of other priority rules could also change the performance of the process. With an extended model more insight will be gained on how to allocate resources in an optimal way.

Acknowledgement

We would like to thank Kurt Jensen and Lisa Wells from the University of Aarhus. We appreciate the help Kurt gave in the early stages of the research presented in this paper and we are grateful that Lisa let us use the monitors.

This research is supported by the Technology Foundation STW, applied science division of NWO and the technology programme of the Dutch Ministry of Economic Affairs.

References

1. W.M.P. van der Aalst. Three Good Reasons for Using a Petri-net-based Workflow Management System. In: T. Wakayama et al., editors. Information and Process Integration in Enterprises: Rethinking Documents. Volume 428 of The Kluwer International Series in Engineering and Computer Science. Boston, Kluwer Academic Publicers, pp.161-182, 1998.
2. W.M.P. van der Aalst. Workflow verification: finding control-flow errors using Petri-net-based techniques. In: W.M.P. van der Aalst et al., editors. Business process management: models, techniques, and empirical studies. Lecture Notes in Computer Science 1806. Berlin, Springer-Verlag, pp.161-183, 2000.
3. W.M.P. van der Aalst, K.M. van Hee. Workflow management: models, methods and systems. London, MIT Press, 2002.
4. W.M.P. van der Aalst, A.H.M. ter Hofstede, M. Weske. Business process management: a survey. In: Proceedings of the 2003 International Conference on Business Process Management (BPM2003). Lecture Notes of Computer Science 2678. Berlin, Springer-Verlag, pp.1-12, 2003.
5. F. Ardhaldjian, M. Fahner. Using simulation in the business process reengineering effort. Industrial Engineering, 26(7), pp.60-61, 1994.
6. J. Desel, T. Erwin. Modeling, Simulation and Analysis of Business Processes. In: W.M.P. van der Aalst et al., editors. Business process management: models, techniques, and empirical studies. Lecture Notes in Computer Science 1806. Berlin, Springer-Verlag, pp.129-141, 2000.

7. T. Forsberg, L. Nilsson, M. Antoni. Process orientation: the Swedish experience. Total Quality Management, Volume 10(4-5/July), pp.540 - 547, 1999.
8. K. Jensen. Colored Petri Nets: Basic Concepts, Analysis Methods and Practical Use Volume 1. Berlin, Springer-Verlag, 1992.
9. K. Jensen. Colored Petri Nets: Basic Concepts, Analysis Methods and Practical Use Volume 2. Berlin, Springer-Verlag, 1995.
10. L. Kleinrock. Queueing Systems, Volume 1: Theory. New York, John Wiley and Sons, 1975.
11. L.M. Kristensen, S. Christensen, K. Jensen. The Practitioner's Guide to Colored Petri nets. International Journal on Software Tools for Technology Transfer, 2(1998), pp.98-132, 1998.
12. A. Kumar, W.M.P. van der Aalst, H.M.W. Verbeek. Dynamic Work Distribution in Workflow Management Systems: How to balance quality and performance? Journal of Management Information Systems, 18(3), pp.157-193, 2002.
13. D. Makajic-Nikolic, B. Panic, M. Vujosevic. Bullwhip effect and supply chain modelling and analysis using CPN Tools. CPN Workshop 2004.
14. B. Mitchell, L.M. Kristensen, L. Zhang. Formal Specification and State Space Analysis of an Operational Planning Process. CPN Workshop 2004.
15. R. Muramatsu, H. Miyazaki, K. Ishii. A Successful Application Of Job Enlargement/Enrichment At Toyota. IIE Transactions, 19(4), pp.451-459, 1987.
16. K. McCormack. Business process orientation: Do you have it? Quality Progress, Volume 34(1), pp.51-58, 2001.
17. G. Poyssick, S. Hannaford. Workflow reengineering. Mountain View, Adobe Press Editions, 1996.
18. H.A. Reijers, S. Limam Mansar. Best practices in business process redesign: an overview and qualitative evaluation of successful redesign heuristics. Omega, Volume 33(4), pp.283-306, 2005.
19. A. Seidmann, A. Sundararajan. Competing in information-intensive services: analyzing the impact of task consolidation and employee empowerment. Journal of Management Information Systems, 14(2), pp.33-56, 1997.
20. A. Vinter Ratzer, L. Wells, H.M. Lassen et al. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In: W.M.P. van der Aalst, E. Best (Eds.). Applications and Theory of Petri Nets 2003. Lecture Notes in Computer Science 2679. Berlin, Springer-Verlag, pp. 450-462, 2003.
21. M. Zairi, D. Sinclair. Business process re-engineering and process management: a survey of current practice and future trends in integrated management. Business Process Re-engineering & Management Journal, 1(1), pp.8-30, 1995.
22. M. Zapf, A. Heinzl. Evaluation of Generic Process Design Patterns: An Experimental Study. In: W.M.P. van der Aalst et al., editors. Business process management: models, techniques, and empirical studies. Lecture Notes in Computer Science 1806. Berlin, Springer-Verlag, pp.83-98, 2000.
23. M. zur Muehlen. Organizational Management in Workflow Applications - Issues and Perspectives. Information Technology and Management, 5(2004), pp.271-291, 2004.
24. http://www.daimi.au.dk/CPnets/intro/example_indu.html

Colored Petri Net based model checking and failure analysis for E-commerce protocols

Panagiotis Katsaros[1] Vasilis Odontidis[2] Maria Gousidou-Koutita[2]

[1] Department of Informatics, Aristotle University of Thessaloniki,
54124 Thessaloniki, Greece
katsaros@csd.auth.gr
http://delab.csd.auth.gr/~katsaros/index.html

[2] Department of Mathematics, Aristotle University of Thessaloniki,
54124 Thessaloniki, Greece
gousidou@math.auth.gr

Abstract. We present a Colored Petri Net approach to model check three atomicity properties for the NetBill electronic cash system. We verify that the protocol satisfies money atomicity, goods atomicity and certified delivery in the presence of potential site or communication failures and all possible unilateral transaction abort cases. Model checking is performed in CPN Tools, a graphical ML-based tool for editing and analyzing Colored Petri Nets (CP-nets). In case of property violation, protocol failure analysis aims in exploring all property violation scenarios, in order to correct the protocol's design. Model checking exploits the provided state space exploration functions and the supported Computation Tree like temporal logic (CTL). On the other hand, protocol failure analysis is performed by inspection of appropriately selected markings and if necessary, by interactively simulating certain property violation scenarios. In e-commerce, Colored Petri Net model checking has been used in verifying absence of deadlocks, absence of livelocks and absence of unexpected dead transitions, as well as in verifying a protocol against its service. To the best of our knowledge, our work is the first attempt to employ CP-nets for model checking atomicity properties. We believe that the described approach can also be applied in model checking other functional properties that are not directly related to the structural properties of the generated state space graph.

1 Introduction

E-commerce systems offer considerable potential, but they are accompanied by a broad range of often unprecedented risks. A lack of system security or reliability can cause the system to behave differently than its stakeholders expect and can lead to loss of physical assets, digital assets, money, and consumer confidence. Model checking refers to a set of mechanized techniques, which are used to automatically discover any scenarios, in which the actual system behavior and the behavior of the stakeholders' model diverge from one another. These scenarios identify potential failures and pinpoint areas where design changes or revisions should be considered.

In e-commerce, most published model checking approaches ([10], [16], [23], [24]) use a Communicating Sequential Processes (CSP) system description (`SYSTEM`) and verification is performed by the Failure Divergence Refinement (FDR) model checker. The properties to be checked are also expressed as CSP processes (`SPEC`) and FDR checks if the set of behaviors generated by `SYSTEM` is a subset of those generated by `SPEC`.

In the field of e-commerce, CP-nets ([12]) and CPN Tools ([7]) have been used to model check the absence of deadlocks and livelocks and the absence of unexpected dead transitions and inconsistent terminal states, for the Internet Open Trading Protocol ([18], [20], [21]). Also, CP-nets have been used in verifying a protocol against its service ([19]). The correctness properties considered in [21] depend on certain structural properties of the state space graph, like for example the absence of self-loops.

However, in e-commerce we are also interested to model check functional properties that are not directly related to the structural properties of the state space graph. CSP-based model checking is broadly used due to its success in verifying security properties, like confidentiality ([22], [26]), message authentication ([26]), anonymity ([26]), integrity ([22], [26]) and information flow security ([14]). CSP-based model checking can also be applied in verifying validated receipt ([23], [24]), accountability ([13]), personalization potentiality ([9]) and atomicity properties, like withdrawal atomicity, payment atomicity and deposit atomicity ([15], [27], [29]), in the presence of site or communication failures and all possible unilateral transaction abort cases.

In this work, we use CPN Tools to model check payment atomicity for the NetBill ([6]) electronic cash system. We show how to exploit the provided state space exploration functions and the supported Computation Tree like temporal logic (CTL), in order to verify ([4], [5]) three different levels of payment atomicity (money atomicity, goods atomicity and certified delivery). In case of property violation, we propose protocol failure analysis to be based on inspection of appropriately selected markings and if necessary, to exploit the CPN Tools advanced graphical environment to interactively simulate the actions performed in one or more property violation scenarios.

Although it was already known that NetBill possesses the three levels of payment atomicity ([10]), we are not aware of published results that model check certified delivery. Moreover, the applied CP-net approach is characterized by the comparative advantage of interactively simulating the developed model and makes CPN Tools an attractive alternative (over CSP-based model checking), for model checking the forenamed security and reliability properties. Finally, the proposed protocol failure analysis is possible to be applied to other electronic cash systems, like for example Digicash ([2], [3]), Payword ([25]), MicroMint ([25]) and MiniPay ([11]), which fail to provide a certified delivery mechanism, for selling and delivering digital goods and services.

Section 2 provides a compact description of the NetBill electronic cash system and the three levels of payment atomicity. Section 3 introduces the adopted modeling assumptions and the proposed CP-net. Section 4 focuses on model checking the three levels of payment atomicity (money atomicity, goods atomicity and certified delivery). Section 5 introduces the proposed protocol failure analysis and the paper concludes with a summary of our CP-net model checking experience and its potential impact.

2 The NetBill electronic cash system

The NetBill transaction model involves three participants: the consumer (*C*), the merchant (*M*) and the bank server (*B*). A transaction involves three phases: price negotiation, goods delivery and payment. We consider the selling of information goods, in which case the NetBill protocol links goods delivery and payment into a single atomic transaction. We use the notation "$X \Rightarrow Y$ *message*" to indicate that *X* sends the specified *message* to *Y*. The basic protocol consists of the following messages:

1. $C \Rightarrow M$ Price request
2. $M \Rightarrow C$ Price quote
3. $C \Rightarrow M$ Goods request
4. $M \Rightarrow C$ Goods, encrypted with a key *K*
5. $C \Rightarrow M$ Signed Electronic Payment Order (*epo*)
6. $M \Rightarrow B$ Endorsed Electronic Payment Order (including the key *K*)
7. $B \Rightarrow M$ Signed result (including *K* in case of successful payment)
8. $M \Rightarrow C$ Signed result (including *K* in case of successful payment)

C and *M* interact with each other in the following way:

- *C* issues a price request for a particular product (1) and *M* replies with the requested price (2),
- *C* either aborts the transaction or issues a goods request to the merchant (3),
- in the second case, *M* delivers the requested goods encrypted with a key *K* (4).

The goods are cryptographically checksummed in order to be able to confirm that received goods are not affected by a potential transmission error and that they are not subsequently altered. The bank (*B*) is not involved until the payment phase:

- *C* sends to *M* a signed electronic payment order (5) including all necessary payment details and the received product checksum,
- *M* validates the received electronic payment order (*epo*) and checksum information and either aborts the transaction or endorses it by sending the received payment order and the associated decryption key *K* to *B* (6),
- *B* responds to *M* (7) with the payment result and the decryption key *K* (in case of successful payment), which are finally forwarded to *C* (8) to terminate the transaction.

NetBill provides protection for *C* against fraud by *M* in the following ways:

- the key *K*, which is needed to decrypt the goods is registered with *B* and if *M* does not respond in a valid payment as expected, the consumer asks the key from *B*, that in fact acts as a trusted third party,
- if there is a discrepancy between what *C* ordered and what *M* delivered, *C* can easily demonstrate this discrepancy to the trusted third party, since the payment order received by *B* includes all details about what exactly was ordered, the amount charged, the key *K* reported by *M* and the checksum of the delivered encrypted goods. Thus, if the goods are faulty it is easy to demonstrate that the problem lies with the goods as sent and not with any subsequent alteration (that would produce different checksum information).

NetBill is one of the first electronic cash systems that provides the three levels of payment atomicity, in the presence of potential site or communication failures and all possible unilateral transaction abort cases.

Money atomicity is the basic level that should be ensured by all transactions exchanging electronic cash. It means that the payment transaction ensures that there is no possibility of creation or destruction of money, while electronic cash is being transferred.

Goods atomicity ensures money atomicity and also ensures that there is no possibility of paying without receiving goods or vice versa.

Most electronic payment systems ensure money and goods atomicity, but fail to ensure certified delivery, which is the highest level of atomicity. Certified delivery includes both money and goods atomicity and also allows both participants to prove the details of the transaction. In the related bibliography, certified delivery is also mentioned as non-repudiation of transactions ([28]) or strongly fair exchange ([1]).

3 A Colored Petri Net model for NetBill

In this section, we introduce a CP-net for the NetBill electronic cash system. The adopted modeling assumptions take into account consumer and merchant site failures and non-reliable communication between the protocol's participants, including potential message losses. We assume that both, consumer's account debit and merchant's account credit take place at the same site (bank server) that provides trivial transaction atomicity guarantees. Thus, we omit modeling bank site failures, since this would burden the CP-net with details that are not part of the NetBill protocol, but concern the provided transaction processing mechanism. This implies the property of money atomicity, but we show how to express it by exploiting the provided state space exploration functions and the supported Computation Tree like temporal logic (CTL).

Compared to the CP-net proposed for the Internet Open Trading Protocol ([17]) our model differs in how the protocol's participants are represented. In our model, the participant processes correspond to substitution transitions that include potential site and communication failures and all possible unilateral transaction abort cases. In [17], the participants are represented by places, whose color sets include token colors that correspond to all possible participant states. Also, we do not use places that implement reliable FIFO communication channels between the protocol participants. In our case, a protocol message loss terminates the corresponding protocol session. We aim in model checking the forenamed payment atomicity properties that are not directly related to the structural properties of the generated state space graph. On the other hand, the CP-net of [17] has been used in verifying the protocol against its service and in model checking correctness properties that depend on certain structural properties of the model's state space graph.

The proposed CP-net consists of a number of hierarchically related pages (Figure 1), which model the protocol's behavior in different levels of abstraction.

∨Protocol
ConsumerProcess
MerchantProcess
BankProcess

Fig. 1. The hierarchy page of the NetBill CP-net

In the highest level of abstraction (Figure 2), we model the protocol's participants and message exchanges. We omit the protocol steps 1 and 2, since they are not significant for checking payment atomicity. Figure 3 summarizes the color and variable declarations used for the transition and arc inscriptions of the described CP-net. Our model is important to reflect all possible protocol execution scenarios. We adopt a compact representation of all distinct transaction abort cases by specifying them as `request` typed tokens that encode the following execution scenarios:

1. *C* sends to *M* a valid goods request (`gReq=v`)
2. *C* sends to *M* an invalid goods request (`gReq=i`)
3. the encrypted goods received by *C* are the requested ones (`enGoods=v`)
4. the encrypted goods received by *C* are affected by an occurred data transmission error (`enGoods=i`)
5. *C* sends to *M* a valid electronic payment order (`epoReq=v`)
6. *C* sends to *M* an invalid electronic payment order (`epoReq=i`)

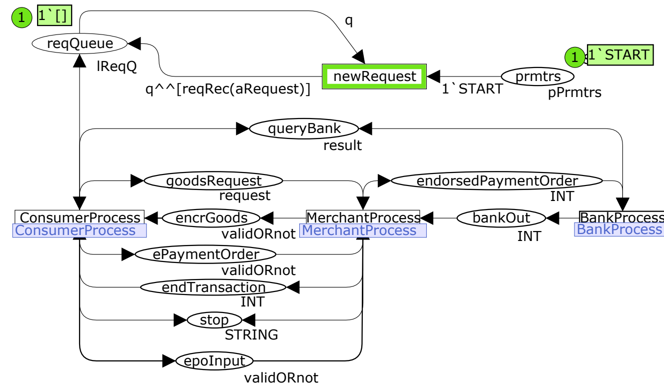


Fig. 2. Top-level structure of the NetBill CP-net

```
colset pPrmtrs        =with START;
colset validORnot     =with v | i;
colset request        =record       gReq:validORnot*
                                    enGoods:validORnot*
                                    epoReq:validORnot;
colset sRequest       =union reqRec:request;
colset lReqQ          =list sRequest;
colset result         =with noFunds | paymentReceipt | noRecord;
var aRequest          :request;
var q                 :lReqQ;
var valCode           :validORnot;
var intVar            :INT;
var res               :result;
```

Fig. 3. Color sets and variables used in the NetBill CP-net

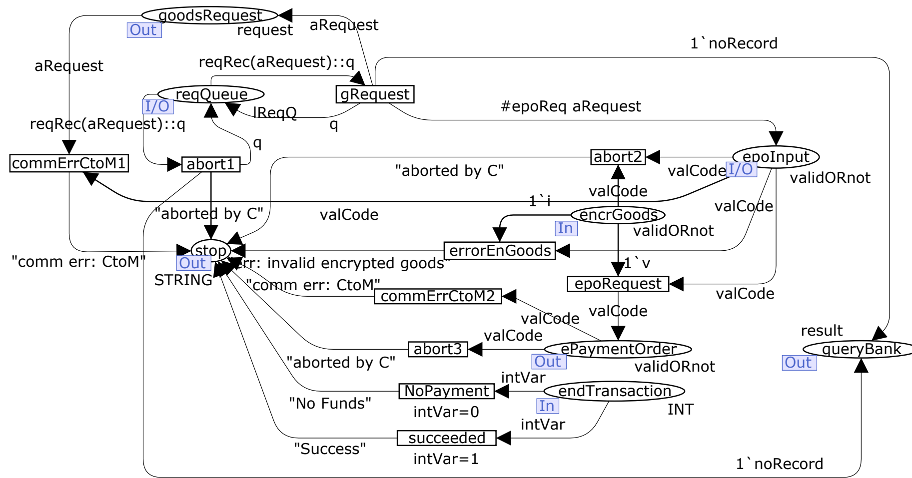


Fig. 4. The Consumer (*C*) process page of the NetBill CP-net

An electronic payment order (*epo*) is not valid, when it is not signed or includes invalid payment details, like for example a product checksum that does not coincide to the one assigned to the encrypted goods sent to *C*.

Figure 4 introduces the consumer process page that corresponds to the `Consumer-Process` substitution transition of Figure 2. Input and output places were assigned to the synonyms shown in the top-level CP-net. Firing of transition `gRequest` places the `result` typed token `noRecord` at the place `queryBank`. This models the possibility of *C* querying *B* (trusted third party) for the result of the ongoing transaction. The `request` typed token `aRequest` is passed to the place `goodsRequest` and it is then used non-deterministically to fire either the `commErrCtoM1` transition (communication error: C to M) or the one corresponding to its reception by the merchant process (merchant process page).

We note that *C* can abort the executed transaction any time up to the submission of *epo*. Potential unilateral abort decisions and consumer site failures are modeled by transitions named as `abort#` and terminate the protocol by placing an appropriate diagnostic string at the output place `stop`. Unilateral transaction aborts are also represented by transitions like the one called `errorEnGoods`, which correspond to the validation actions performed by the protocol participants. Dispatch of the signed *epo* by *C* signifies the commitment of *C* to the executed transaction request.

The diagnostic strings placed at the place `stop` indicate protocol termination, but only `"No Funds"` and `"Success"` are reported to the end user. In case of site or communication failure, *C* is informed for the transaction result by querying *B*.

Figure 5 introduces the merchant process page that corresponds to the `Merchant-Process` substitution transition of Figure 2. On reception of a `request` typed token at the place `goodsRequest` the merchant process either aborts the transaction (site failure), terminates the protocol (because of an invalid goods request) or proceeds to the processing of the received request (transition `receiveGoodsReq`).

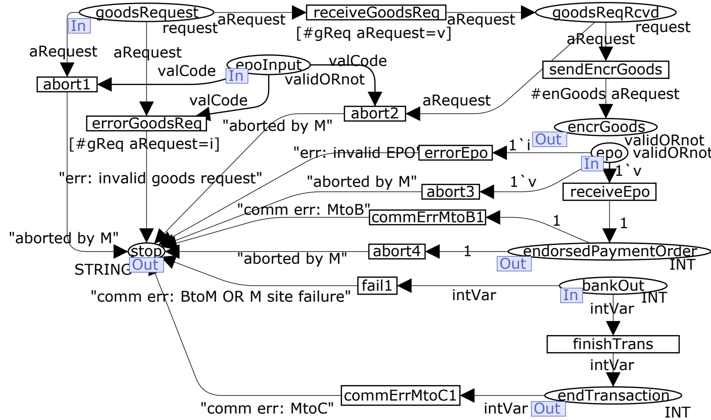


Fig. 5. The Merchant (*M*) process page of the NetBill CP-net

The merchant process responds to the consumer process by dispatching an encrypted version of the ordered goods (transition `sendEncrGoods`). On reception of the signed *epo* at the place `epo`, the merchant process either terminates the transaction in case of invalid *epo*, aborts the transaction due to a site failure (transition `abort3`) or endorses the received *epo* (attaches the required decryption key) and forwards it to the bank server process through the place `endorsedPaymentOrder`.

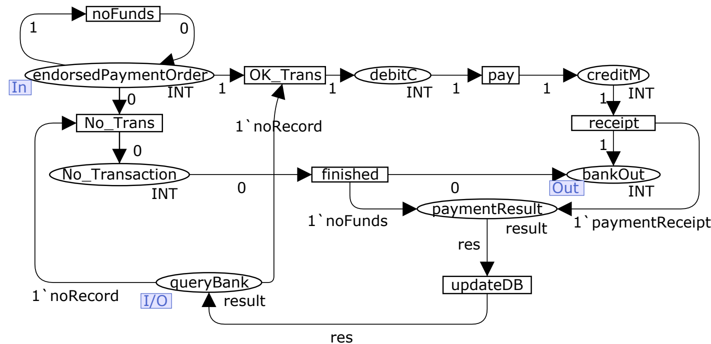


Fig. 6. The Bank (*B*) process page of the NetBill CP-net

Figure 6 introduces the bank process page that corresponds to the `BankProcess` substitution transition of Figure 2. On reception of an endorsed payment order at the place `endorsedPaymentOrder` the bank server either proceeds to the debit of the consumer's account or discovers a low balance account (because of the transition `noFunds`) and does not charge *C*. Both transaction cases are (assumed to be) executed atomically and this is modeled, by initially removing the `result` typed token found in place `queryBank` and finally placing an appropriate token value in it. Although this makes money atomicity self-proved, we are still interested to demonstrate the potentiality of the Computation Tree like temporal logic (CTL) supported in CPN

Tools to express and prove all payment atomicity properties, including money atomicity.

4 State space analysis and model checking

Figure 7 shows the standard report generated for the state space analysis of the described NetBill CP-net.

```
 Statistics
------------------------------------------------------------------------
  State Space
    Nodes:  59
    Arcs:   103
    Secs:   0
    Status: Full

  Scc Graph
    Nodes:  59
    Arcs:   103
    Secs:   0

Boundedness Properties
------------------------------------------------------------------------
  Best Integers Bounds                 Upper          Lower
  BankProcess'No_Transaction 1         1              0
  BankProcess'creditM 1                1              0
  BankProcess'debitC 1                 1              0
  BankProcess'paymentResult 1          1              0
  ConsumerProcess'epoInput 1           1              0
  MerchantProcess'goodsReqRcvd 1       1              0
  Protocol'bankOut 1                   1              0
  Protocol'ePaymentOrder 1             1              0
  Protocol'encrGoods 1                 1              0
  Protocol'endTransaction 1            1              0
  Protocol'endorsedPaymentOrder 1      1              0
  Protocol'goodsRequest 1              1              0
  Protocol'prmtrs 1                    1              0
  Protocol'queryBank 1                 1              0
  Protocol'reqQueue 1                  1              1
  Protocol'stop 1                      1              0

Home Properties
------------------------------------------------------------------------
  Home Markings:  None

Liveness Properties
------------------------------------------------------------------------
  Dead Markings:  13 [59,658,57,56,55,...]
  Dead Transitions Instances: None
  Live Transitions Instances: None

Fairness Properties
------------------------------------------------------------------------
  No infinite occurrence sequences.
```

Fig. 7. The standard report generated for the state space analysis of the NetBill CP-net

The results for the standard properties checked in the shown report constitute a necessary input source, for correctly expressing the CTL formulae of the required correctness properties. We observe the absence of home markings and the absence of dead and live transitions instances. There are no infinite occurrence sequences and the protocol terminates in one of the 13 dead markings, with node numbers that are easily found by the provided state space exploration functions.

The ML-functions used in model checking payment atomicity are summarized in Table 1:

Table 1. State space querying functions

function description	use
Mark.<PageName>'<PlaceName> N M	Returns the set of tokens positioned in place <PlaceName> of the Nth instance of page <PageName> in the marking M
SearchNodes (<search area>, <predicate function>, <search limit>, <evaluation function>, <start value>, <combination function>)	Traverses the nodes of the part of the occurrence graph specified in <search area>. At each node the calculation specified by <evaluation function> is performed and the results of these calculations are combined as specified by <combination function> to form the final result. The <predicate function> maps each node into a boolean value and selects only those nodes, which evaluate to true. We use the value EntireGraph for <search area> to denote the set of all nodes in the occurrence graph and the value NoLimit for <search limit> to continue searching for all nodes, for which the predicate function evaluates to true.
List.nth(l,n)	Returns the nth element in list l, where 0 <= n < length l.

Function SearchNodes is used to detect the marking(s) right after the occurrence of a particular event, like for example a money transfer from *C*'s account to *M*'s account.

Table 2. CTL state formulae operators and model checking functions

state formulae syntax	meaning
NOT(A)	Boolean value that corresponds to the negation of A, where A is a CTL formula.
AND(A_1,A_2)	This formula is true if both A_1 and A_2 are true.
NF(<message>,<node function>)	A function that is typically used for identifying single states or a subset of the state space. Its arguments are a string and a function, which takes a state space node and returns a boolean. The string is used when a CTL formula evaluates to false in the model checker.
EV(A)≡FORALL_UNTIL(TT,A)	This formula is true if the argument A becomes true eventually (within a finite number of steps) starting from the state we are now. TT denotes the true constant value.
ALONG(A)≡NOT(EV(NOT(A)))	This formula is true if there exists a path for which the argument A holds for every state. The path is either infinite or ends in a dead state.
POS(A)≡EXIST_UNTIL(TT,A)	This formula is true if possible from the state we are now, to reach a state where the argument A is true.
EXIST_NEXT(A)	This formula is true iff there exists an immediate successor state, from where we are now, in which the argument A is true.
FORALL_NEXT(A)	This formula is true iff for all immediate successor states from where we are now the argument A is true.
eval_node <formula> <node>	The standard model checking function that takes two arguments: the CTL formula to be checked and a state from where the model checking should start.

Table 2 summarizes the CTL formulae used to express the required properties in terms of paths over the generated state space graph. A CTL expression that corre-

sponds to the required property is model checked by the `eval_node` function, starting from the node number that is passed as second argument. The ML statements need to activate the provided Computation Tree like temporal logic (CTL) are shown in Figure 8.

```
use (ogpath^"ASKCTL/BitArray.sml");
use (ogpath^"ASKCTL/ASKCTL.sml");
open ASKCTL;
```

Fig. 8. ML statements used to activate the provided CTL support

4.1 Model checking money atomicity

Figure 9 shows the model checking of the money atomicity property. We are interested to verify that money transfer takes place atomically that is, *for all paths* starting from the occurrence of the consumer's debit the protocol performs the corresponding credit to the merchant's account irrespective of the considered failure possibilities.

Value `firstdebitState` corresponds to the marking that signifies consumer's debit. This is the marking from where the model checking starts. Value `noDebit` is used to detect redundant debits before the occurrence of the expected credit. Value `moneyAtomicity` (`true`) ensures that for all immediate successors, `noDebit` is true for each state along the path, until the last state on the path, where `creditState` becomes `true`.

```
Binder 0
Money atomicity

fun debitDone n = (Mark.BankProcess'debitC 1 n = [1]);
val firstDebitState = List.nth(SearchNodes (
      EntireGraph,
      fn n => (debitDone n),
      NoLimit,
      fn n => n,
      [],
      op ::),0);
fun creditDone n = (Mark.BankProcess'queryBank 1 n = [paymentReceipt]);
val noDebit = NOT(NF("Double debit!",debitDone));
val creditState = NF("No credit!",creditDone);
val moneyAtomicity = FORALL_NEXT(FORALL_UNTIL(noDebit,creditState));
eval_node moneyAtomicity firstDebitState;

None
```

```
val debitDone = fn : Node -> bool
val firstDebitState = 36 : Node
val creditDone = fn : Node -> bool
val noDebit = NOT (NF ("Double debit!",fn)) : A
val creditState = NF ("No credit!",fn) : A
val moneyAtomicity =
  NOT
    (MODAL
      (NOT
        (OR
          (NOT TT,
           NOT
             (MODAL
               (NOT
                 (FORALL_UNTIL
                   (NOT (NF ("Double debit!",fn)),
                    NF ("No credit!",fn))))))))) : A
val it = true : bool
```

Fig. 9. Model checking money atomicity: true

4.2 Model checking goods atomicity

Figure 10 shows the model checking of non-atomic goods delivery. We are interested to verify that irrespective of the considered failures and unilateral abort decisions (i) when *C* signs a valid *epo* it is not possible to eventually perform *C*'s debit, without a subsequent protocol termination with a registered payment receipt (including the re-

quired decryption key) and (ii) when *C* sends a not necessarily valid *epo* it is not possible to eventually register a payment receipt, without having previously debited *C*'s account. The first mentioned guarantee ensures goods atomicity from the consumer's perspective and the second mentioned guarantee ensures goods atomicity from the merchant perspective.

Value `dispatchedEPOState` corresponds to the marking that signifies the dispatch of a valid signed *epo*. This is the marking from where the model checking of (i) starts. Value `debitState` is used to detect *C*'s debit. Value `notRegisteredDecrKey` is used to check the absence of a payment receipt. Value `noGoodsAtomicityA` (`false`) ensures that there is no path, for which it is possible to eventually occur *C*'s debit and at the same time in every state, to not register the expected payment receipt. Note that because of the absence of infinite paths (state space report), all paths quantified by `ALONG` end in a dead marking (protocol termination).

Value `dispatchedEPOState1` corresponds to the marking that signifies the dispatch of a valid signed *epo*. On the other hand, value `dispatchedEPOState2` corresponds to the marking that signifies the dispatch of an invalid *epo*. These are the markings from where the model checking of (ii) starts. Value `noDebitFound` is used to check the absence of *C*'s debit. Value `registeredDecrKey` is used to detect registration of an unexpected payment receipt. Value `noGoodsAtomicityB` (`false` in both model checking cases) ensures that there is no path, for which it is possible to eventually register a payment receipt and at the same time in every state, to not have performed *C*'s debit.

```
Binder 14
Goods atomicity

fun signedEPO n = (Mark.Protocol'ePaymentOrder 1 n = [v]);
val dispatchedEPOState = List.nth(SearchNodes (
      EntireGraph,
      fn n => (signedEPO n),
      NoLimit,
      fn n => n,
      [],
      op ::),0);
fun debitDone n = (Mark.BankProcess'debitC 1 n = [1]);
fun noTrans n = (Mark.Protocol'queryBank 1 n <> [paymentReceipt]);
val debitState = NF("No debit!",debitDone);
val notRegisteredDecrKey = NF("Found decryption key!",noTrans);
val noGoodsAtomicityA = ALONG(AND(EV(debitState),notRegisteredDecrKey));
eval_node noGoodsAtomicityA dispatchedEPOState;
```

```
val signedEPO = fn : Node -> bool
val dispatchedEPOState = 32 : Node
val debitDone = fn : Node -> bool
val noTrans = fn : Node -> bool
val debitState = NF ("No debit!",fn) : A
val notRegisteredDecrKey = NF ("Found decryption key!",fn) : A
val noGoodsAtomicityA =
  NOT
   (FORALL_UNTIL
     (TT,
      NOT
       (NOT
         (OR
           (NOT (FORALL_UNTIL (TT,NF ("No debit!",fn))),
            NOT (NF ("Found decryption key!",fn))))))) : A
val it = false : bool
```

```
fun sendEPO n = (Mark.Protocol'ePaymentOrder 1 n <> []);
val dispatchedEPOStates = SearchNodes (
      EntireGraph,
      fn n => (sendEPO n),
      NoLimit,
      fn n => n,
      [],
      op ::);
val dispatchedEPOState1 = List.nth(dispatchedEPOStates,0);
val dispatchedEPOState2 = List.nth(dispatchedEPOStates,1);
fun noDebitDone n = (Mark.BankProcess'debitC 1 n <> [1]);
fun succeedTrans n = (Mark.Protocol'queryBank 1 n = [paymentReceipt]);
val noDebitFound = NF("Debit found!",noDebitDone);
val registeredDecrKey = NF("Failed transaction!",succeedTrans);
val noGoodsAtomicityB = ALONG(AND(EV(registeredDecrKey),noDebitFound));
eval_node noGoodsAtomicityB dispatchedEPOState1;
eval_node noGoodsAtomicityB dispatchedEPOState2;
```

```
val sendEPO = fn : Node -> bool
val dispatchedEPOStates = [32,31] : Node list
val dispatchedEPOState1 = 32 : Node
val dispatchedEPOState2 = 31 : Node
val noDebitDone = fn : Node -> bool
val succeedTrans = fn : Node -> bool
val noDebitFound = NF ("Debit found!",fn) : A
val registeredDecrKey = NF ("Failed transaction!",fn) : A
val noGoodsAtomicityB =
  NOT
   (FORALL_UNTIL
     (TT,
      NOT
       (NOT
         (OR
           (NOT (FORALL_UNTIL (TT,NF ("Failed transaction!",fn))),
            NOT (NF ("Debit found!",fn))))))) : A
val it = false : bool
val it = false : bool
```

```
None
```

Fig. 10. Model checking the two parts of the non-atomic goods delivery: false

4.3 Model checking certified delivery

Figure 11 shows the model checking of a non-certified delivery. We are interested to verify that irrespective of the considered failures and unilateral abort decisions, the protocol does not fall in a state, where it is possible to end with a payment receipt, without *C* having previously obtained an encrypted version of the requested goods and the corresponding checksum number. The obtained checksum number can be used to prove potential discrepancy between what *C* ordered and what *M* delivered (if the number coincides with the checksum number written on the registered payment receipt).

Value `registerKey` is used to detect registration of the expected payment receipt. Value `noGoods` is used to check the absence of an encrypted goods delivery. Value `nonCertiefiedDelivery (false)`, when it is model checked starting from the initial node, ensures that there is no path for which it is possible to not have delivered the assumed encrypted goods and the protocol to terminate with having registered a payment receipt.

Model checking certified delivery from the merchant perspective is performed in a similar way.

```
Binder 13
Certified Delivery
fun registerKeyState n = (Mark.Protocol'queryBank 1 n = 1`paymentReceipt);
val registerKey = POS(EV(NF("err",registerKeyState)));
fun enGoodsTransferedState n = (Mark.Protocol'encrGoods 1 n = 1`v);
val noGoods = NOT(POS(EV(NF("err",enGoodsTransferedState))));
val nonCertifiedDelivery = EXIST_NEXT(AND(noGoods,registerKey));
eval_node nonCertifiedDelivery InitNode;
val registerKeyState = fn : Node -> bool
val registerKey = EXIST_UNTIL (TT,FORALL_UNTIL (TT,NF ("err",fn))) : A
val enGoodsTransferedState = fn : Node -> bool
val noGoods = NOT (EXIST_UNTIL (TT,FORALL_UNTIL (TT,NF ("err",fn)))) : A
val nonCertifiedDelivery =
  MODAL
    (NOT
      (OR
        (NOT TT,
         NOT
          (MODAL
            (NOT
              (OR
                (NOT
                  (NOT
                    (EXIST_UNTIL
                      (TT,FORALL_UNTIL (TT,NF ("err",fn))))),
                 NOT
                  (EXIST_UNTIL
                    (TT,FORALL_UNTIL (TT,NF ("err",fn)))))))))) : A
val it = false : bool
None
```

Fig. 11. Model checking non-certified delivery: false

5 Protocol failure analysis

In general, protocol failure analysis aims in exploring all property violation scenarios (if any) and pinpoints areas where design changes or revisions should be considered. Having shown that CP-net based model checking of payment atomicity is feasible, we can then exploit the CPN Tools advanced graphical environment, to interactively simulating the actions performed in possible property violation scenarios.

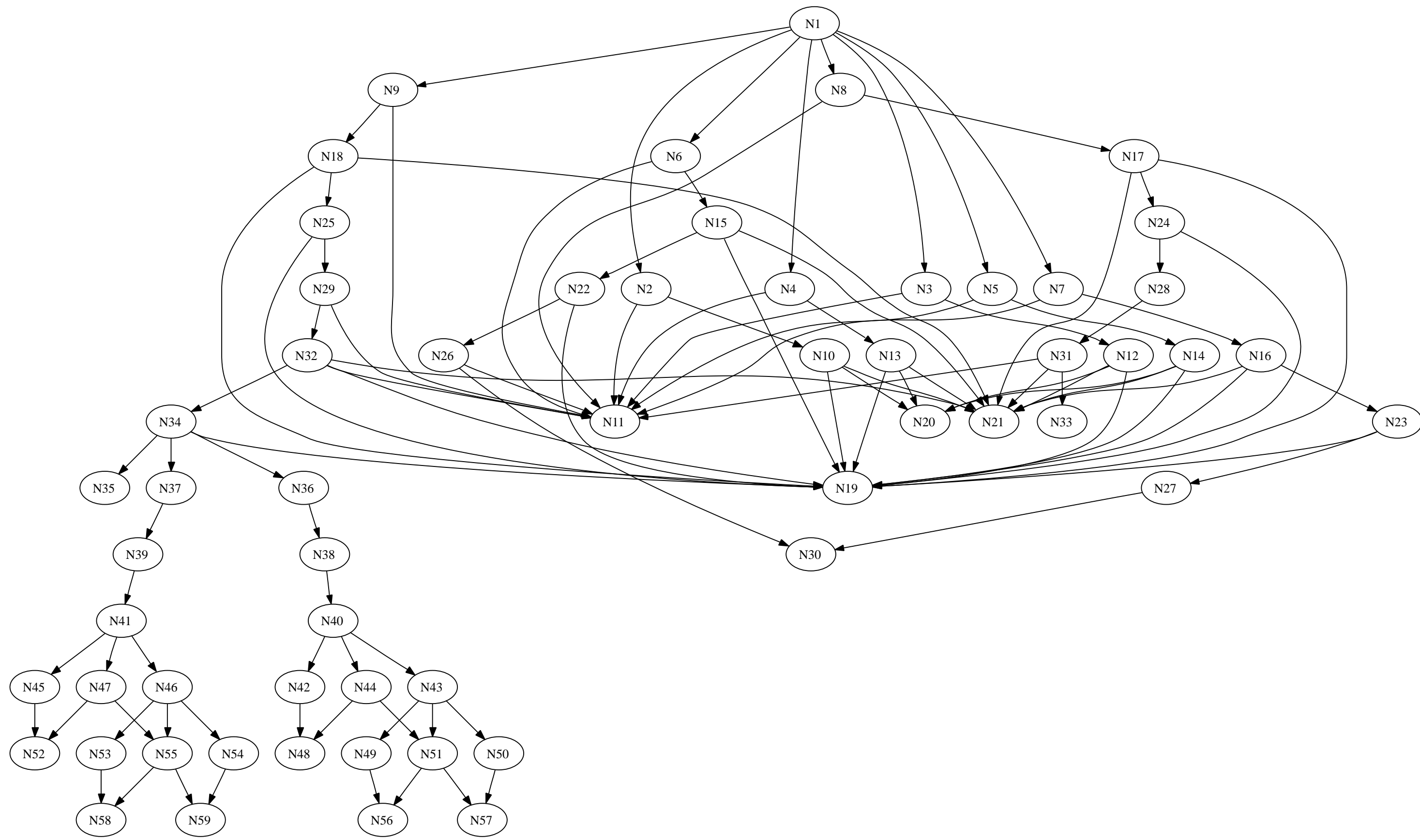


Fig. 12. NetBill's CP-net state space graph

Protocol failure analysis is based on inspection of the terminal markings (dead markings) in all property violation paths. The diagnostic strings positioned at places, like the place `stop` and the place `queryBank` in the top-level CP-net of NetBill (Figure 2), provide details for interactively simulating the corresponding protocol execution scenario. The simulation control functionality found in the latest version of CPN Tools allows firing transitions with an interactively chosen binding. Thus, the actions included in the scenario of interest are easily reproduced and the analyst explores all possible protocol revision prospects to repair the detected property violation. A necessary prerequisite is the selection of informative diagnostic strings in the phase of model development.

In what is concerned with NetBill, protocol failure analysis is not applicable, since we did not detect atomicity violation cases. However, we proceed to the inspection of the CP-net's terminal markings and the visualization of the generated state space graph.

CPN Tools exports the model's state space graph in a DOT language based text file that is then automatically visualized by an appropriate program, which implements well-tuned layout algorithms ([8]), for placing graph nodes and arcs. Figure 12 shows the generated state space graph for the described NetBill CP-net. Leaf nodes correspond to the dead markings to be inspected (Figure 13).

```
ListDeadMarkigs() -> val it =[59,58,57,56,55,
                              51,35,33,30,21,
                              20,19,10]:Node list
```

Fig. 13. Dead markings for the described NetBill CP-net

Finally, Table 3 provides a concise view of the performed protocol termination inspection.

Table 3. Protocol termination inspection

marking (N)	Mark.Protocol' stop 1 N	Mark.Protocol' queryBank 1 N	interpretation
59	`[“No Funds”]`	`[noFunds]`	No failures.
58	`[“comm err: MtoC”]`	`[noFunds]`	Communication failure: M fails to report the transaction result to C. C is informed for the result of the submitted transaction by querying B.
57	`[“Success”]`	`[paymentReceipt]`	No failures.
56	`[“comm err: MtoC”]`	`[paymentReceipt]`	Communication failure: M fails to report the transaction result to C. C obtains the product decryption key by querying B.
55	`[“comm err: BtoM or M site failure”]`	`[noFunds]`	M is not informed for the result of the submitted transaction due to a potential site or communication failure. C is informed for the result of the submitted transaction by querying B.
51	`[“comm err: BtoM or M site failure”]`	`[paymentReceipt]`	M is not informed for the result of the submitted transaction due to a potential site or communication failure. C obtains the product decryption key by querying B.
35	`[“comm err: MtoB”]`	`[noRecord]`	Communication failure: the signed payment order is not transmitted to B. C is informed that there is no transaction by querying B.
33	`[“err: invalid EPO”]`	`[noRecord]`	M aborts the transaction due to an invalid *epo*. C is informed that there is no transaction by querying B.

marking (N)	Mark.Protocol' stop 1 N	Mark.Protocol' queryBank 1 N	interpretation
30	["err: invalid encrypted goods"]	[noRecord]	C aborts the transaction due to reception of encrypted goods that are possibly affected by an occurred transmission error.
21	["comm err: CtoM"]	[noRecord]	Communication failure: the goods request or the signed payment order is not transmitted to M. C is informed that there is no transaction by querying B.
20	["err: invalid goods request"]	[noRecord]	M aborts the transaction due to an invalid goods request.
19	["aborted by M"]	[noRecord]	M aborts the transaction due to a potential site failure or due to a unilateral abort decision. C is informed that there is no transaction by querying B.
10	["aborted by C"]	[noRecord]	C aborts the transaction due to a potential site failure or due to a unilateral abort decision before being committed to it, by the dispatch of a signed payment order.

6 Conclusion

This paper introduces the use of CP-nets and CPN Tools to model check the three levels of payment atomicity, for an electronic cash system. The combined use of appropriate state space exploration functions and CTL formulae allowed us to express and model check money atomicity, goods atomicity and certified delivery.

Although it was already known that NetBill possesses these three properties, we are not aware of published works in e-commerce, where CP-nets are used to model check correctness properties that are not directly related to the structural properties of the generated state space graph. We believe that the described approach is also applicable in more complex system models and is also possible to be extended for model checking other reliability and security properties.

We also proposed the performance of protocol failure analysis, in order to explore potential property violation scenarios and pinpoint areas, where design changes or revisions should be considered. In protocol failure analysis, it is possible to exploit the advanced graphical environment of CPN Tools to interactively simulate the actions included in a protocol execution scenario.

Our model checking experience suggests that CPN Tools is an attractive alternative over CSP-based model checking in e-commerce problems.

Acknowledgments

We acknowledge the CPN Tools team at Aarhus University, Denmark for kindly providing us the license of use of the valuable CP-net toolset. We also acknowledge the anonymous referees for their helpful comments.

References

1. Asokan, N., Fairness in electronic commerce, PhD thesis, University of Waterloo, Ontario, Canada, 1998
2. Chaum, D., Fiat, A., Naor, M., Untraceable electronic cash, In: Proceedings of CRYPTO'88, Springer-Verlag, 1990, 200-212
3. Chaum, D., Security without identification: Transaction systems to make big brother obsolete, Communications of the ACM, Vol. 28, No. 10, 1985, 1030-1044
4. Cheng, A., Christensen, S., Mortensen, K. H., Model checking Coloured Petri Nets exploiting strongly connected components, In: Proceedings of the International Workshop on Discrete Event Systems, Edinburg, Scotland, UK, 1996, 169-177
5. Clarke, E. M., Emerson, E. A., Sistle, A. P., Automatic verification of finite state concurrent system using temporal logic, ACM Transactions on Programming Languages and Systems, Vol. 8, No. 2, 1986, 244-263
6. Cox, B., Tygar, J. D., Sirbu, M., NetBill security and transaction protocol, In: Proceedings of the 1st USENIX Workshop in Electronic Commerce, New York, NY, USENIX Association, California, 1995, 77-88
7. CPN Tools, http://wiki.daimi.au.dk/cpntools/cpntools.wiki
8. Gansner, E. R., Koutsofios, E., North, S. C., Vo, K.-P., A technique for drawing directed graphs, IEEE Transactions on Software Engineering, Vol. 19, No. 3, 1993, 214-230
9. Georgiadis, C. K., Manitsaris, A., Personalization in Mobile Commerce Environments: Multimedia Challenges, Cutter IT Journal, Vol. 18, No. 8, 2005, 36-43
10. Heintze, N., Tygar, J., Wing, J., Wong, H., Model checking electronic commerce protocols, In: Proceedings of the 2nd USENIX Workshop in Electronic Commerce, Oakland, CA, USENIX Association, California, 1996, 146-164
11. Herzberg, A. and Yochai, H., Minipay: Charging per click on the web, Computer Networks, Vol. 29, 1997, 939-951
12. Jensen, K., Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use, Volumes 1-3, Basic Concepts. Monographs in Theoretical Computer Science, Springer-Verlag, 1997
13. Kailar, R., Accountability in electronic commerce protocols, IEEE Transactions on Software Engineering, Vol. 22, No. 5, 1996, 313-328
14. Katsaros, P., On the design of access control to prevent sensitive information leakage in distributed object systems: a Colored Petri Net model, In: Proceedings of CoopIS/DOA/ODBASE, Lecture Notes in Computer Science 3761, Springer-Verlag, 2005, 945-962
15. Kempster, T. and Stirling, C., Modeling and model checking mobile phone payment systems, Proceedings of the FORTE 2003 Workshops, Lecture Notes in Computer Science 2767, Springer-Verlag, 2003, 95-110
16. Lu, S. and Smolka, S. A., Model checking the Secure Electronic Transaction (SET) protocol, In: Proceedings of the 7th International Symposium on Modeling, Analysis and Simulatio of Computer and Telecommunication Systems, 1999, 358-365
17. Ouyang, C., Kristensen, L. M. and Billington, J., A formal service specification for the Internet Open Trading Protocol, In: Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets, Lecture Notes in Computer Science 2360, Springer-Verlag, 2002, 352-373
18. Ouyang, C., Kristensen, L. M. and Billington, J., A formal and executable specification of the Internet Open Trading Protocol, In: Proceedings of EC-Web 2002, Lecture Notes in Computer Science 2455, Springer-Verlag, 2002, 377-387

19. Ouyang, C. and Billington, J., On verifying the Internet Open Trading Protocol, In: Proceedings of EC-Web 2003, Lecture Notes in Computer Science 2738, Springer-Verlag, 2003, 292-302
20. Ouyang, C. and Billington, J., An improved formal specification of the Internet Open Trading Protocol, In: Proceedings of the 2004 ACM Symposium on Applied Computing, Nicosia, Cyprus, 2004, 779-783
21. Ouyang, C. and Billington, J., Formal analysis of the Internet Open Trading Protocol, In: Proceedings of the FORTE 2004 Workshops, Lecture Notes in Computer Science 3236, Springer-Verlag, 2004, 1-15
22. Panti, M., Spalazzi, L. and Tacconi, S., Verification of security properties in electronic payment protocols, In: Proceedings of the ACM SIGPLAN and IFIP WG 1.7 Workshop on Issues in the Theory of Security, Oregon, 2002
23. Ray, I., Ray, I., Failure Analysis of an E-Commerce Protocol using Model Checking, In: Proceedings of the Second International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems, San Jose, CA, June 2000, 176-183
24. Ray, I., Ray, I., Natarajan, N., An anonymous and failure resilient fair-exchange e-commerce protocol, Decision Support Systems, Vol. 39, 2005, 267-292
25. Rivest, R. L. and Shamir, A., Payword and MicroMint: Two simple micropayment schemes, In: Security Protocols Workshop, 1996, 69-87
26. Schneider, S., Modelling security properties with CSP, Tech. Report CSD-TR-96-04, Dept. of Computer Science, Royal Holloway, University of London, 1996
27. Schuldt, H., Popovici, A. and Schek, H.-J., Execution guarantees in electronic commerce payments, In: Proceedings of the 8th International Workshop on Foundations of Models and Languages for Data and Objects, Lecture Notes in Computer Science 1773, Springer-Verlag, 2000, 193-202
28. Shyamasundar, R. K. and Deshmukh, B., MicroBill: An efficient secure system for subscription based services, In: Proceedings of ASIAN 2002, Lecture Notes in Computer Science 2550, Springer-Verlag, 2002, 220-232
29. Xu, S., Yung, M., Zhang, G. and Zhu, H., Money conservation via atomicity in fair off-line e-cash, In: Proceedings of the 2nd Int. Workshop of Information Security, Lecture Notes in Computer Science 1729, Springer-Verlag, 1999, 14-31

Application of Coloured Petri Nets in Cooperative Provision of Industrial Services

Katrin Winkelmann

Research Institute for Rationalization and Operations Management at Aachen University
Pontdriesch 14/16, D-52062 Aachen, Germany
katrin.winkelmann@fir.rwth-aachen.de

Abstract. Product-related services in the capital goods industry are characterised by international customer demands. Often these demands can only be met efficiently by cooperating in networks. However, the decision-making process for network configuration is still not sufficiently supported. Existing controlling approaches for networks and services are not suitable for an assessment of different alternatives prior to their implementation. Simulation approaches for production networks on the other hand, cannot be used for industrial services. This paper deals with an approach to overcome this problem and presents a simulation model based on Petri Net theory for the cooperative provision of industrial services. As a result machine and equipment producers will be able to assess their cooperation alternatives related to integrated service provision in advance and thus avoid cost-intensive false decisions.

1 Motivation

1.1 Trend towards Cooperation and Need for Assessment Support

The global trend towards a service economy has become manifest in the German capital goods industry: German machine and equipment producers realise a rising percentage of their revenues with product-related services [17]. Product-related industrial services (e.g. maintenance or spare parts supply) are offered to (re)establish, ensure or enhance the long-term usability of industrial products. Since Germany's capital goods industry has an export rate of about 70 percent and capital goods are sold around the globe, product-related services also have to be offered internationally [5], [37]. However, international provision of high-quality services requiring expert knowledge proves to be rather cost-intensive and thus threatens the service profit margin. Therefore machine and equipment producers are looking for partners to form coalitions and thus improve their service business through cooperative advantages like more efficient deployment of experts, shorter response time or cost reduction. This means that the process of providing services for customers is not longer owned by one single company but by several partners. This shared process is referred to as cooperative service provision.

While companies have gained preliminary experience with cooperative service provision, problems in finding appropriate partners and organisational difficulties

indicate that companies lack support for the planning and configuration of service networks [38]. A variety of alternatives related to cooperations and configurations complicates a substantiated decision [1], [11], [15], [33], [36]. In this context, a possibility to assess different cooperation alternatives before implementation is sorely needed [7], [21]. Some approaches for network and service controlling exist, but these methods are not suitable for an assessment of different alternatives prior to their implementation, i.e. a prospective assessment. Companies are not able to evaluate, for example, whether a certain combination of partners in the service process leads to a shorter throughput time for service orders or not. To solve problems of this kind, a model for the prospective assessment of service network alternatives in the capital goods industry is needed. This model should represent all relevant elements and their correlations so that an assessment against the background of service objectives is possible.

1.2 Prospective Assessment of Service Network Alternatives through Simulation with Petri Nets

The target is therefore to enable the prospective assessment of different service network alternatives in the capital goods industry. The chosen method of resolution is a simulation with Petri Nets. Simulation is particularly suitable for assessing different alternatives before their actual implementation and has proven to be a successful method for prospective assessment of production network design [4], [23], [30], [39]. Modern systems are often so complex that system interactions can usually only be analysed using simulation techniques [3], [4]. First applications of simulation in service systems (e.g. queuing models at bank counters) have produced promising results [22], [32]. Petri Nets are a widely used method to model business processes [18], [19]. They are particularly useful for the case in hand, since they offer a broad range of analytical possibilities with regard to simulation as well as answers to "what-if"-questions, and revision of dynamic system characteristics [14], [29], [35].

2 Existing Approaches

Several existing approaches have been examined and evaluated:

- First, approaches based on simulation in different ranges of application have been evaluated. These approaches provide a valuable contribution since they show how to model certain problems and transfer them into a simulation model. Although their objectives differ significantly from the problem at hand, these approaches provide some useful pointers for the development of a simulation model for the case described in this paper.
- Second, to consider the special conditions concerning the cooperative provision of industrial services, approaches for assessing networks and industrial services were analysed. These approaches are not designed for a prospective assessment but deliver valuable information about relevant assessment criteria.

First simulation applications for services were developed by [22], [27] and [32]. Although the authors examined a prospective assessment for services, their approaches do not consider the cooperation aspect and characteristics of service provision in an industrial context. Advanced simulation models exist for different ranges of application, e.g. for product development [24] or for autonomous production cells [31]. However, they do not provide information about modelling cooperative service provision.

Approaches in the field of network controlling proved to be of little value to the problem at hand, because of their focus on a retrospective analysis and management. Concepts based on the Balanced Scorecard (e.g. [2], [26]) benefit from the need to define assessment criteria, but remain superficial about operationalisation of these criteria.

With regard to the controlling and quality management of industrial services, the following approaches are worth mentioning: [6], [8], [12], [16], [25] and [34]. These authors have developed assessment criteria for industrial services at different levels of detail, and their results will be taken into account in the development of the simulation model.

The analysis of relevant literature shows that some approaches exist for parts of the problem, but shortcomings prevail, especially concerning the prospective assessment of alternatives of cooperative service performance. The following deficiencies can be identified:

- Most of the approaches concerning prospective assessment based on simulation refer to manufacturing or other areas of application and do not consider the special characteristics of industrial services. The constitutive characteristics of services lead to significant differences between the manufacturing of material goods and the provision of services. This means that approaches that do not consider these characteristics of services cannot be used to solve the problem at hand. First simulation approaches for services differ importantly in terms of the underlying problem, so that their adoption for the problem is not possible.
- Approaches in the field of network controlling do not consider the special characteristics of services, either. In addition, concepts of controlling are not designed for a prospective assessment but rather for continuous management and retrospective analysis. They are therefore not applicable for a prospective assessment of service provision.
- Approaches to asses the quality of services or to control services are also designed for continuous management and retrospective analysis and cannot be adopted for a prospective assessment. In addition, the approaches focus on individual companies and neglect network aspects. These aspects, however, are very important for the current case.

But existing approaches also provide valuable information for solving different parts of the problem at hand:

- Simulation as a method of prospective assessment has proven to be of great value, especially for complex processes in networks. First promising applications for services suggest that using simulation for the problem at hand would be useful.

- Petri Nets in particular have proven to be a valuable modelling technique for complex processes in networks and are applicable to cooperative service provision. Their analysis potential meets the demands of the current case.
- Assessment criteria used in service controlling and to measure service quality represent an important input for the development of a model for the assessment of possible alternatives of cooperative service provision. Especially the assessment criteria developed for industrial services are relevant for the problem at hand.

Based on the deficiencies and contributions of existing approaches, the following tasks need to be addressed:
- A conceptual model for the prospective assessment of industrial services has to be developed.
- This model has to integrate characteristics of services while considering the dimensions of services [13]: structure, process and outcome, which are explained in more detail in the following section.
- The model has to be implemented into a simulation model based on the Petri Net notation.

3 Simulation Model of Cooperative Provision of Industrial Services

The model presented in this paper basically consists of three partial models that consider the following dimensions:
- *Process dimension*: Representation of the process flow in a form amenable to simulation and modelled according to Petri Net formalism.
- *Structure dimension*: Representation of the required resources (personnel and material) and resource consumption related to different service assignments and network configurations.
- *Outcome dimension*: Definition of assessment criteria that can be influenced by the process design within the network.

These partial models map different concepts of the whole system. A brief description of these partial models and how they are integrated into an overall simulation model is presented in the following sections.

3.1 Concepts on Which the Overall Simulation Model is Based

Process dimension. To represent the process of industrial service provision, reference models have been developed [6], [20]. These reference models have been adapted to fit the cooperative service provision and to fulfil the modelling requirements of the Petri Net notation. Furthermore, the resulting model has been structured in a hierarchy in order to model process elements in adequate detail while keeping complexity at the top level low (fig. 1). At the uppermost level, the whole process is mapped with low granularity. Each process step is then specified at a first level of detail. For some processes, this first level is sufficient. The process step *accomplish order* requires

further detailing at a second level. The process of resource allocation within *plan order* is a crucial point in order processing so this part is modelled at a third level of detail.

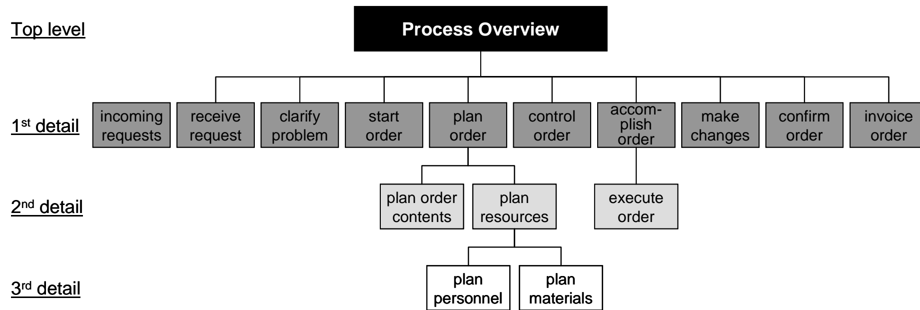


Fig. 1. Overview of process hierarchy

Structure dimension. The structure dimension represents the resources needed to execute the processes. Therefore, all relevant resources and their consumption have to be mapped. Mapped resources are different categories of personnel and material (fig. 2). Depending on the kind of order that has to be processed, different categories and quantities of resources are needed and consumed. Resource costs are calculated by assigning a cost value based on a single item to each of the categories of resources.

Categories of Resources	
Personnel	Material
• Service technicians (ST) • Support engineers (SE) • Client technicians (CT) • Service providers (SeP)	• Tools • Operating resources (OR) (lubricant, gaskets, etc.) • Spare parts (SpP)

Fig. 2. Categories of resources

Outcome dimension. Based on the assessment criteria for industrial services mentioned above, relevant criteria for the cooperative provision of industrial services have to be selected an implemented into the model. In order to do so, information stored in other dimensions have to be integrated and analysed in the outcome dimension, e.g. information about resource consumption. On a top level, criteria can be categorised into time (e.g. throughput time), cost (e.g. resource consumption and related costs) and quality criteria (e.g. first hit rate, the rate of orders that get processed without any changes, additions or rescheduling).

3.2 Implementation of the Model Using CPN Tools

To transfer the conceptual model into a simulation model, the software CPN Tools of the University of Aarhus was used [9], [10]. All aspects of the overall model were mapped into a directed graph consisting of places, transitions, arcs, and markings. The implementation of the main concepts is presented below.

Process modelling. First of all, the entire process was represented in Petri Net notation using places, transitions and arcs. The hierarchical concept (fig. 1) was realised using the technique of substitution transitions. A substitution transition represents another page (subpage) in the net which starts and ends at the same places as the substitution transition but which may contain a more detailed process comprising several other transitions and places. As depicted in fig. 3 the process overview maps ten substitution transitions.

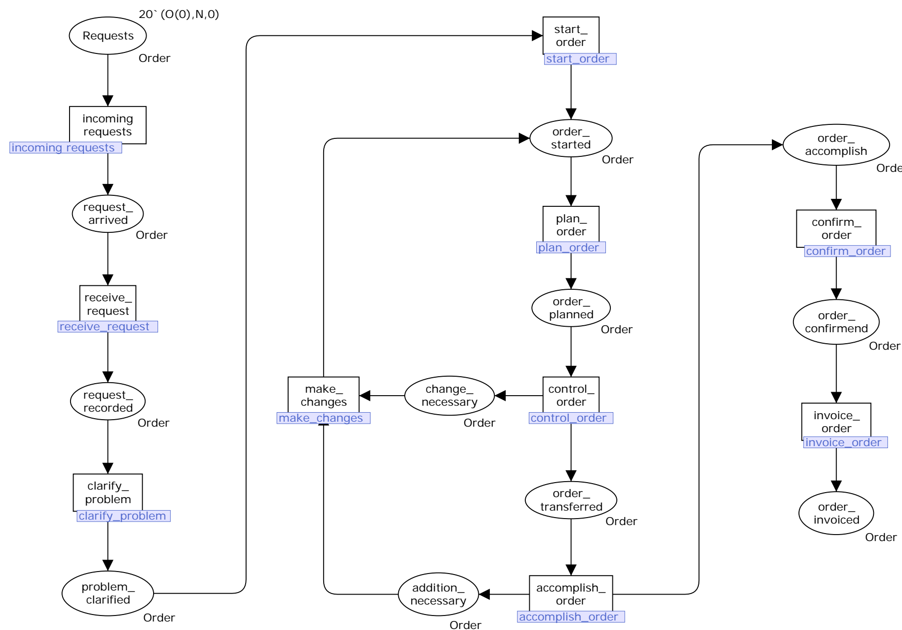


Fig. 3. Top page process overview

Most of the process steps on the top level are arranged in a sequential order except for the `make_changes` transition that can be triggered from the transitions `accomplish_order` and `control_order`. Place and transition names have been chosen accordingly to usual terminology in the area of application. Some of the subpages are explained in more detail in the following remarks.

Time modelling. All time delays of the model are using random distribution functions, because the duration of the process steps varies. Incoming requests are modelled using a queue timed with a Poisson process. All other process steps (transitions) that

are engaged with a certain time delay are timed using normal distributions. Since CPN Tools can handle time values only as integers, the real values of the normal distribution function have been rounded, e.g. `@+round(normal(48.0,20.0))`. Time values (mean and variance) are estimates and have been derived through interviews with experts in industrial service networks.

Order modelling. The primary markings in the net are the orders that pass through the process. The colour set of orders `(Order(i),j,k)` is declared as follows:

```
colset A = index A with 0..n;
colset B = with N|Ch|Add|N_r|Ch_r|Add_r; with
(N=normal, Ch=change, Add= addition, N_r=normal
rescheduled, Ch_r=change rescheduled, Add_r=addition
rescheduled)
colset Order = product A*B*INT;
```

Initially, each order (respectively request, since any order enters the process as a client's request and turns into an order later on) receives a one-to-one order number (position `k` in `Order(i),j,k`). This is an important requirement since, in order to model several concurrent processes, order copies (with the same order number) are created by the function

```
copy(Order(i),j,k)=(Order(i),j,k);.
```

When the concurrent processes become synchronised, the order numbers ensure that original and copy match the same order. (For a more detailed description of branching and synchronisation patterns see [40].) The function

```
getn(Order(i),j,k)=k;
```

extracts the order number from any order (original or copy) so that the numbers of original and copy can be matched.

The orders are indexed to represent different degrees of difficulty of service assignments (position `i` in `Order(i),j,k`). Depending on the degree of difficulty of the service assignment, different categories and quantities of resources have to be employed and certain process steps take different amounts of time.

Any order also carries information about its status regarding changes and additions in the service assignment as well as a notation to account for conducted rescheduling, since certain process steps only apply to orders not having passed through any changes, additions or rescheduling (position `j` in `Order(i),j,k`).

Resource modelling. The different resources described above are controlled by special resource places that monitor their availability and usage. Current information about availability of resources and reserved resources that are already on-site is monitored on a special resource subpage, where all relevant information is stored using the concepts of fusion places (fig. 4). Fusion places are duplicates of places to make the places accessible at different pages or different locations on one single page.

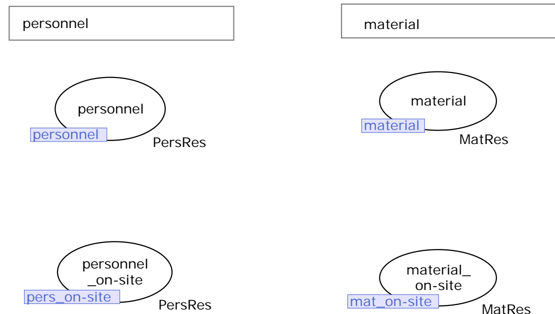


Fig. 4. Subpage `resources`

Planning and assigning of resources is performed as a function of the degree of difficulty of the respective orders. The subpage `plan_resources` (fig. 5) depicts the concurrent processes of planning the personnel and material. The two processes are started by the transition `plan_personnel_and_material` that creates a copy of the order. At the end, both processes are synchronised by the transition `end_planning`. The guard

```
getn(ord) = getn(ord_copy)
```

ensures that the order copy drawn from the place `end_of_material_planning` matches the original order drawn from `end_of_personnel_planning` by checking the order number.

`Start_planning_personnel` and `start_planning_material` are both substitution transitions that refer to subpages. Since both subpages are similar and differ only in the use of different resources, only the subpage `plan_personnel` is presented here[1]. One of the characteristics of resource planning is that the actual planning can be done quickly, but before the resources actually are available on-site, the resources have to be ordered and (eventually) travel to the site or be shipped there. This is modelled by separating the planning from the allocation, as depicted in fig. 6.

[1] Since both processes are similar, resources could have been modelled in Coloured Petri Nets using only one place containing personnel and material as different coloured tokens. In the field of industrial services however, personnel and material resources are commonly separated and treated quite differently. Thus, to keep comprehensibility of the model by users high, this additional complexity of having two separate places and sub-processes has been approved.

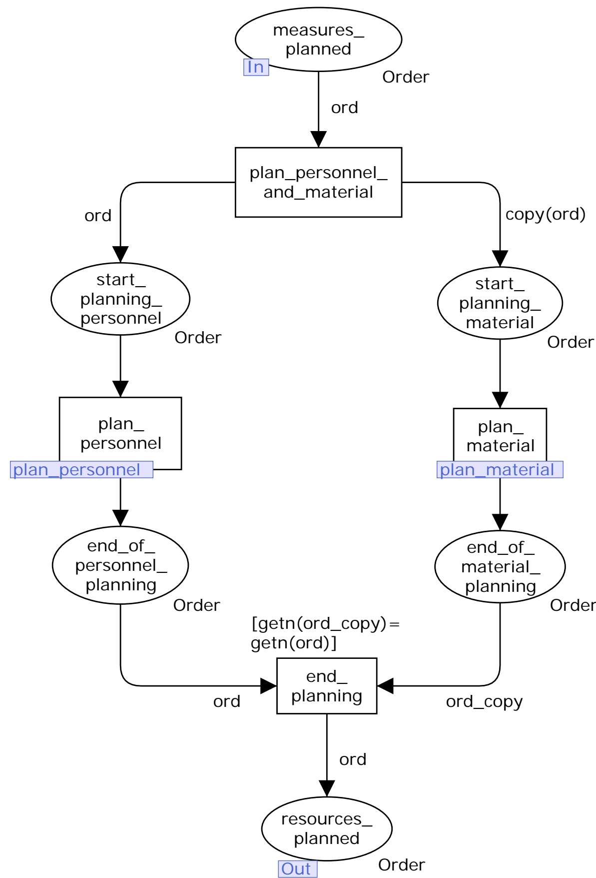


Fig. 5. Subpage `plan_resources`

The transition `plan_personnel` puts the order into the output place `end_of_personnel_planning`. This way the planning process can continue on a higher level omitting the further steps in allocating the resources. The mechanism for the allocation works as follows: Besides the two input places `personnel_planned` and `resources`, a third input place `no_rescheduling` is used. This is to make sure that in the concurrent process on the subpage `control_order`, possible rescheduling that can affect the required resources is initiated before the resources travel to the site. Again a guard ensures that the order copies drawn from the place `personnel_planned` and `no_rescheduling` belong to the same order by checking the order number. The arc weight of the resources consumed by the transition `personnel_travel` is calculated by the function

```
fun personnel(Order(i),j,k) = if(geti(Oder(i),j,k)=1)
then pers1 else (if(geti(Order(i),j,k)=2) then pers2
else pers3);
```

```
with pers1, pers2 and pers3 representing the personnel
needed for orders of the three different degrees of
difficulty.
```

This function takes into account that the amount of required resources depends on the degree of difficulty of the respective order and can be modified according to the specifications and qualifications of different partners and suppliers. The same arc inscription is used for the two output places: One place contains the resources that have travelled to the site (`personnel_on-site`), the other one (`personnel_consumption`) is a counter of the consumption of personnel.

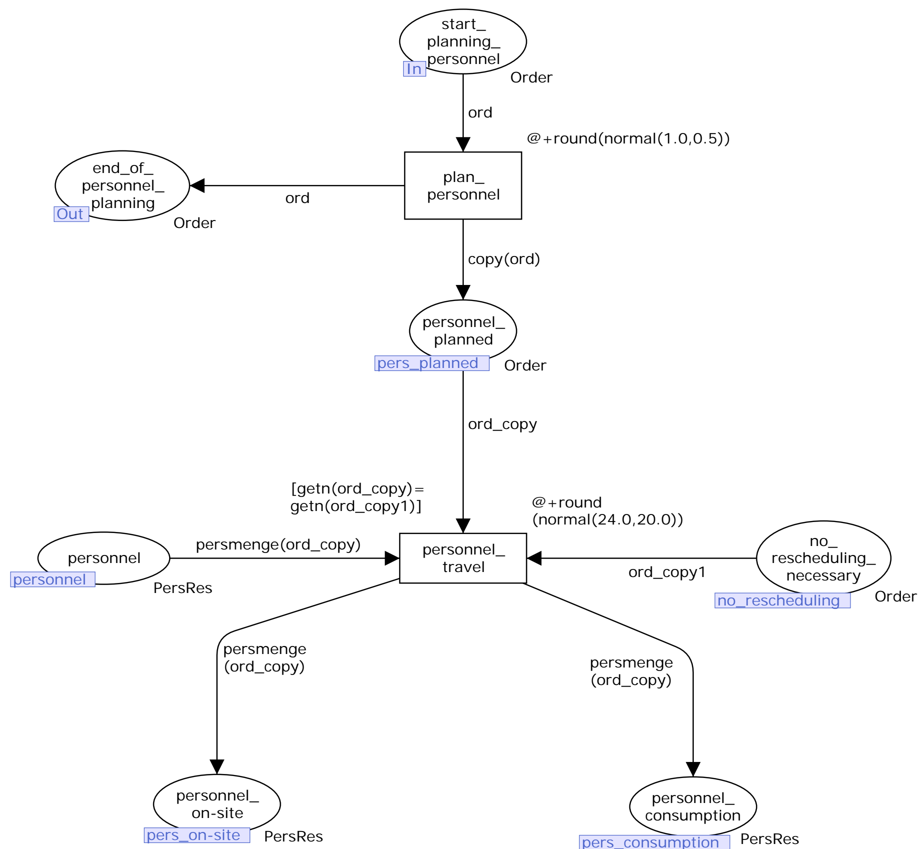


Fig. 6. Subpage `plan_personnel`

Control modelling. The central processes of assigning changes and rescheduling are implemented in the subpage `control_order` (fig. 7). After order costs are planned and the whole planning process is finished, the controlling of the order starts. The transition `control_order` assigns changes to an order according to the probability of changes as it is stored in the place `p_of_changes`. To make sure that

orders get assigned these changes only once (the first time they pass this step) the functions `case_x` check the orders for their status regarding changes, additions or rescheduling:

```
fun case_p((O(i),j,k),P)=
if(getj(O(i),j,k)=N) then
(if (P=p) then 1`(O(i),j,k) else empty)
else empty;

fun case_r((O(i),j,k),P)=
if(getj(O(i),j,k)=N) then
(if (P=r) then 1`(O(i),j,k) else empty)
else 1`(O(i),j,k);

fun case_r_copy((O(i),j,k),P)=
if(getj(O(i),j,k)=N) then
(if (P=r) then 2`(copy(O(i),j,k)) else empty)
else 2`(copy(O(i),j,k));
```

Another assignment of changes and rescheduling is performed by the transition `check_changes_etc`. The logic of this step is the same as before with the differences that another probability is used and that two order copies are created in the places `no_rescheduling`. These order copies are necessary to allocate resources correctly as described above.

Partner modelling. To account for the concept of cooperative service provision, the model has to change with regard to different cooperation alternatives. That means that different parts of the process may be provided by different partners and thus may differ in respect of time, cost and quality. Therefore all variable factors of the model have been assessed if they are influenced by a change in responsibility of certain process steps. These cooperation relevant factors have to be changed for every cooperation alternative that is to be assessed using the simulation model. For example, time delays for process steps, the probability of necessary changes and additions or the qualification and cost of personnel change depending on the responsible partner.

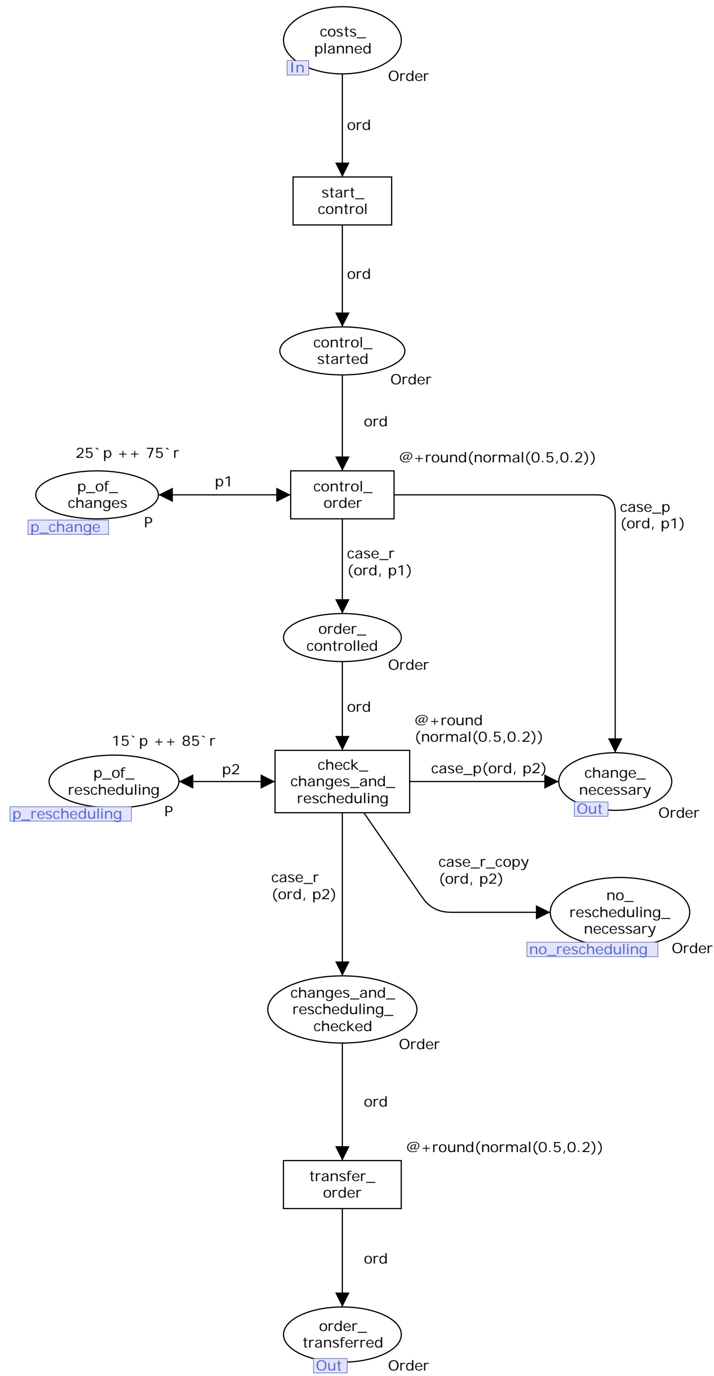


Fig. 7. Subpage `control_order`

4 Preliminary Results and Discussion

4.1 Preliminary Results

In order to verify and validate the model, several actions have been taken. Although these measures are presented at the end of this paper, it is stressed that model verification and validation is not a task to perform at the end of model development but rather accompanies the whole process of model development [4], [23], [28].

The full state space and strongly connected components graph were calculated in order to verify the model. To restrict complexity to a manageable level for this calculation the model was reduced in different respects: The amount of orders was reduced; probabilities for changes, additions, and rescheduling were modified so that only one of those concepts occurred at any one time (analyses were carried out for the several cases that this implies). The analysis shows that the net is bounded and the process ends only in desired dead states (`invoice_generated`, `no_action_necessary`, `other_kind_of_order` and `no_order`). Dead transition instances existed only for the transitions that had been disabled via probability to manage complexity. As soon as they were enabled, they were no longer listed as dead transition instances.

To validate the conceptual model potential users were involved in the development process of the model from the beginning. The use of common expressions in the names of places and transitions as well as the graphical representation and hierarchical structure of the model in CPN Tools have proven of immense value to facilitate the communication with potential users and their understanding of the model. Several workshops with experts in industrial service networks have been organised to validate the logic and assumptions of the model.

Simulation was then used to assess the behaviour of the model. Interactive simulation runs showed that the net behaved as expected. Terminating simulation runs with any amount of orders showed that all orders ended up in the desired dead states. As a result of these experiments, some minor shortcomings could be identified – and consequently eliminated. One of these shortcomings was the mechanism for resource allocation that was improved in the following way: In an early version of the implementation the need for rescheduling was not considered in the subpages `plan_personnel` and `plan_material`, leading to incorrect (double) resource allocation. After improving the model resources are allocated only after rescheduling has taken place.

To check the model for face validity, an example of one cooperation alternative was simulated and the output's plausibility checked by experts. The simulation results of the example proved to be consistent with perceived system behaviour.

4.2 Discussion and future research

This paper presents a conceptual and simulation model of cooperative service provision in order to allow prospective assessment of cooperation alternatives. The model reflects the major features of industrial services as well as their cooperative provision. It also allows the prospective assessment of process organisation alternatives by varying cooperation relevant factors and comparing the analysis results.

The use of Coloured Petri Nets as a modelling and simulation formalism involves several benefits: One the one hand, the graphical representation of the model leads to transparency, consistency and conformity with user's expectations so that understanding and acceptance of the model are facilitated. On the other hand, the maturity of the Petri Net formalism and the availability of appropriate tools allow a direct implementation into computer models and their analysis.

The model presented in this paper contains elements of some well-known approaches. In integrating many aspects of these different approaches, the new model considers aspects neglected by the others. The hierarchical structure and the partial models which properly map the cooperative provision processes of industrial services enable planners of service networks to try out alternatives in process organisation and assess the results before actually implementing the network. First results show that the model output is consistent with its perceived behaviour and enables the assessment of cooperation alternatives.

The primary areas for continued development are more detailed output analyses, sensitivity analyses to identify the most important cooperative relevant factors as well as closer matching distribution functions for time delays. Additionally, the integration of concepts which are not directly measurable, like cultural fit and trust between partners, would be very interesting, and research on how to implement these concepts should be worthwhile as well.

5 References

[1] Aharoni, Y. (Eds.): Coalitions and Competition: The Globalization of Professional Business Services, Routledge, London, New York, 1993.

[2] Arns, M.; Bause, F.; Kemper, P.; Schmitz, M.; Schweier, H.; Stüllenberg, F.; Völker, M.: Gestaltung von Beschaffungsnetzwerken auf Basis einer prozesskettenorientierten Modellierung, in: Industrie Management, 16 (2000)3, p. 33-36.

[3] Banks, J.; Carson II, J. S.; Nelson, B. L.; Nicol, D.: Discrete-event System Simulation, Pearson Prentice Hall, Upper Saddle River, NJ, 2001.

[4] Banks, J.: Principles of Simulation, in: J. Banks (Eds.): Handbook of Simulation, Wiley, New York, NY [u.a.], 1998, p. 3-30.

[5] Bienzeisler, B.; Meiren, T.: Trendstudie Dienstleistungen. Ergebnisse einer Befragung zu Dienstleistungsproduktivität und Dienstleistungsinternationalisierung, Fraunhofer-Institut für Arbeitswirtschaft und Organisation IAO, Stuttgart, 2005.

[6] Borrmann, A. F.: Service-Controlling für produzierende Unternehmen, Shaker, Aachen, 2003, also: Aachen, Techn. Hochsch., Diss., 2003.

[7] Braun, J.: Veränderte Blickwinkel auf Unternehmen, in: H. Warnecke; J. Braun (Eds.): Vom Fraktal zum Produktionsnetzwerk: Unternehmenskooperationen erfolgreich gestalten, Springer, Berlin [u.a.], 1999, p. 43-90.
[8] Bruhn, M.: Qualitätsmanagement für Dienstleistungen: Grundlagen, Konzepte, Methoden, Springer, Berlin [u.a.], 2004.
[9] CPN Tools: http://wiki.daimi.au.dk/cpntools/.
[10] Coloured Petri Nets at the University of Aarhus: http://www.daimi.au.dk/CPnets/.
[11] DIHT: Services Going Abroad, Deutscher Industrie- und Handelstag, Berlin, 2001.
[12] DIN (Eds.): Strukturmodell und Kriterien für die Auswahl und Bewertung investiver Dienstleistungen, PAS 1019, Beuth, Berlin, 2002.
[13] Donabedian, A.: Explorations in Quality Assessment and Monitoring. Vol. 1: The Definition of Quality and Approaches to its Assessment, Health Administration Press, Ann Arbor, Mich., 1980.
[14] Girault, C.; Valk, R.: Petri Nets for Systems Engineering: A Guide to Modeling, Verification and Applications, Springer, Tokyo, 2003.
[15] Gulati, R.; Nohria, N.; Zaheer, A.: Strategic Networks, in: Strategic Management Journal, 21 (2000)Special Issue, p. 203-215.
[16] Hlubek, W.; Pötzsch, G.; Kesting, J.: Certified Service, in: H. Luczak; V. Stich (Eds.): Betriebsorganisation im Unternehmen der Zukunft, Springer, Berlin [u.a.], 2004, p. 167-187.
[17] Hoeck, H.; Kutlina, Z.: Status quo und Perspektiven im Service 2004. Ergebnisse der Expertenbefragung Servicemanagement, Verlag Klinkenberg, Aachen, 2004.
[18] Jensen, K.; Rozenberg, G.: High-level Petri Nets, Springer, Berlin [u.a.], 1991.
[19] Jensen, K.: Coloured Petri Nets, Vol. 1-3, Springer, Berlin [u.a.], 1997.
[20] Kallenberg, R.: Ein Referenzmodell für den Service in Unternehmen des Maschinenbaus, Shaker, Aachen, 2002, also: Aachen, Techn. Hochsch., Diss., 2002.
[21] Klocke, F.: Produktion 2000 plus - Visionen und Forschungsfelder für die Produktion in Deutschland: Untersuchungsbericht zur Definition neuer Forschungsfelder für die Produktion nach dem Jahr 1999, Freundeskreis des Laboratoriums für Werkzeugmaschinen und Betriebslehre der RWTH Aachen, Aachen, 1998.
[22] Laughery, R.; Plott, B.; Scott-Nash, S.: Simulation of Service Systems, in: J. Banks (Eds.): Handbook of Simulation, Wiley, New York, NY [u.a.], 1998, p. 629-644.
[23] Law, A. M.; Kelton, W. D.: Simulation Modeling and Analysis, McGraw-Hill, Boston [u.a.], 2000.
[24] Licht, T.; Dohmen, L.; Schmitz, P.; Schmidt, L.; Luczak, H.: Person-Centered Simulation of Product Development Processes Using Timed Stochastic Coloured Petri Nets, in: P. Geril (Eds.): Proceedings of the European Simulation and Modelling Conference, ESM'2004, October 25-27, 2004, Paris, EUROSIS-ETI, Ghent, Belgien, 2004, p. 188-195.
[25] Luczak, H.; Drews, P. (Eds.): Praxishandbuch Service-Benchmarking, Service Verlag Fischer, Landsberg, 2005 (in print).
[26] Merkle, M.: Bewertung von Unternehmensnetzwerken: Eine empirische Bestandsaufnahme mit der Balanced Scorecard, St. Gallen, Univ., Diss., 1999.
[27] Mjema, E.: A Simulation Based Method for Determination of Personnel Capacity Requirement in the Maintenance Department, Shaker, Aachen, 1998, also: Aachen, Techn. Hochsch., Diss., 1997.
[28] Naylor, T.; Finger, J.: Verification of Computer Simulation Models, in: Management Science, (1967)2, p. B-92 - B-101.
[29] Peterson, J. L.: Petri Net Theory and the Modeling of Systems, Prentice-Hall, Englewood Cliffs, NJ, 1981.
[30] Pritsker, A.; Alan B.,: Principles of Simulation Modeling, in: J. Banks (Eds.): Handbook of Simulation, Wiley, New York, NY [u.a.], 1998, p. 31-51.

[31] Reuth, R.; Schlick, C.; Luczak, H.: A Simulation Approach: Comparative Assessment of Knowledge, Skills and Abilities in Autonomous Production Cells, in: M. J. Smith; G. Salvendy (Eds.): 'Systems, Social and Internationalization Design Aspects of Human-Computer Interaction', Proceedings of HCI International, 5.-10. August 2001, New Orleans, vol. 2, Lawrence Erlbaum, Mahwah, 2001, p. 207-211.

[32] Seel, C.: Visuelle Simulation von Dienstleistungsprozessen, Eul, Lohmar [u.a.], 2002, also: Saarbrücken, Univ., Diss., 2002.

[33] Sydow, J.: Network Development by Means of Network Evaluation? in: Human Relations, 57 (2004)2.

[34] VDI (Eds.): Richtlinie 2893: Auswahl und Bildung von Kennzahlen für die Instandhaltung, VDI-Handbuch Betriebstechnik, Teil 4 - Betriebsüberwachung/Instandhaltung, Beuth, Berlin [u.a.], 2003.

[35] Wang, J.: Timed Petri Nets: Theory and Application, Kluwer Academic Publishers, Boston, 1998.

[36] Wiertz, C.; Ruyter, K. d.; Streukens, S.: Cooperating for Service Excellence in Multichannel Service Systems: An Empirical Assessment, Maastricht, 2003.

[37] Wise, R.; Baumgartner, R.: Go Downstream. The New Profit Imperative in Manufacturing, in: Harvard Business Review, 77 (1999)5, p. 133-144.

[38] Zahn, E.; Stanik, M.: Wie Dienstleister gemeinsam den Erfolg suchen - Eine empirische Studie über Netzwerke kleiner und mittlerer Dienstleister, in: M. Bruhn; B. Stauss (Eds.): Dienstleistungsnetzwerke, Gabler, Wiesbaden, 2003, p. 593-612.

[39] Zhou, M.; Venkatesh, K.: Modeling, Simulation and Control of Flexible Manufacturing Systems: A Petri Net Approach, World Scientific, Singapore [u.a.], 1999.

[40] Van der Aalst, W. M. P.; Ter Hofstede, A. H. M.; Kiepuszewski, B.; Barros, A. P.: Workflow Patterns, in: Distributed and Parallel Databases, 14 (2003)3, p. 5-51.