ISSN 0105-8517

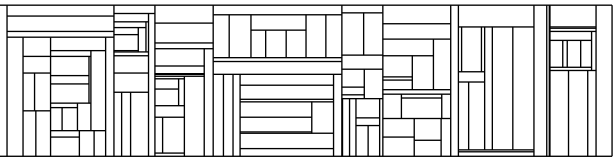
Ninth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools

Aarhus, Denmark, October 20-22, 2008

Kurt Jensen (Ed.)

DAIMI PB - 588
October 2008

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



Preface

This booklet contains the proceedings of the Ninth Workshop on Practical Use of Coloured Petri Nets and the CPN Tools, October 20-22, 2008. The workshop is organised by the CPN group at Department of Computer Science, Aarhus University, Denmark. The papers are also available in electronic form via the web pages: www.daimi.au.dk/CPnets/workshop08/.

Coloured Petri Nets and the CPN Tools are now licensed to more than 7,200 users in 138 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
João Paulo Barros, Protugal
Jörg Desel, Germany
Joao M. Fernandes, Portugal
Jorge de Figueiredo, Brazil
Monika Heiner, Germany
Thomas Hildebrandt, Denmark
Kurt Jensen, Denmark (chair)
Ekkart Kindler, Denmark
Lars M. Kristensen, Denmark
Charles Lakos, Australia
Johan Lilius, Finland
Daniel Moldt, Germany
Laure Petrucci, France
Rüdiger Valk, Germany
Lee Wagenhals, USA
Karsten Wolf, Germany
Jianli Xu, Finland

The programme committee has accepted 10 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 eight 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 been published in four special sections in the International Journal on Software Tools for Technology Transfer (STTT). For more information see: sttt.cs.uni-dortmund.de/. After an additional round of reviewing and revision, some of the papers from this years workshop will be published in Transactions of Petri Nets and Other Models of Concurrency (ToPNoC) which is new journal subline of Lecture Notes in Computer Science. For more information see: www.springer.com/lncs/topnoc.

Kurt Jensen
PC and OC chair

Table of Contents

Invited Tutorials:

Regular Papers:

The ASAP Platform: Next Generation Tool Support for State Space Analysis of CPN Models

Michael Westergaard and Sami Evangelista, University of Aarhus, Denmark

Abstract

State space exploration is one of the main approaches to model-based verification of concurrent systems and it has been one of the most successfully applied analysis methods for Coloured Petri Nets (CPNs). The basic idea of state space exploration and analysis is to compute all reachable states and state changes of the concurrent system under consideration and represent these as a directed graph. Based on state space exploration it is possible to automatically reason about a wide range of properties concerning the behaviour of concurrent systems.

In this talk we present the ASCoVeCo State Space Analysis Platform (ASAP) which is currently being developed in the context of the ASCoVeCo research project. ASAP represents the next generation of computer tool support for state space exploration of CPN models. The vision of the ASAP platform is to provide an open platform suited for research, education, and industrial use of state space exploration and model checking. We present the ASAP platform architecture, the support for state space exploration methods, and give a demonstration of the graphical user interface of ASAP which is based on the Eclipse Rich Client Platform. Finally, we end with an outlook on the future development of ASAP. Version 1.0 of the ASAP platform has recently been released, and we will release an updated version 1.1 after the workshop.

For further information on the ASCoVeCo project, see

http://www.daimi.au.dk/~ascoveco

To download ASAP, see

http://www.daimi.au.dk/~ascoveco/download.html

Bigraphical Business Processes Execution

Thomas Hildebrandt, IT University of Copenhagen, Denmark

Abstract

The model of Bigraphical Reactive Systems (BRSs) has been proposed by Milner as a formal meta-model for global ubiquitous computing that encompasses process calculi for mobility, notably the π-calculus and the Mobile Ambients calculus, as well as graphical models for concurrency such as Petri Nets.

In this presentation we demonstrate that BRSs also allow natural formalizations of languages used in practice by providing a direct and extensible formalization of a subset of WS-BPEL as a binding bigraphical reactive system.

The formalization exploits the close correspondence between bigraphs and XML to provide a formalization and execution format very close to standard WS-BPEL syntax.

In the talk we will comment on the potential use of the BRS metamodel to relate different formalizations of BPEL, for instance formalizations based on Petri Net, the π-calculus or the more direct bigraphical representation of the BPEL syntax as in the presented formalization.

We will also comment on its use to provide a completely formalized and extensible business process engine within the Computer Supported Mobile Adaptive Business Processes (www.CosmoBiz.org) research project at the IT University of Copenhagen.

Building upon the formalization of WS-BPEL we have at COORDINATION 2008 proposed and formalized HomeBPEL, a higher-order WS-BPEL-like business process execution language where processes are first-class values that can be stored in variables, passed as messages, and activated as embedded sub-instances. A sub-instance is similar to a WS-BPEL scope, except that it can be dynamically frozen and stored as a process in a variable, and then subsequently be thawed when reactivated as a sub-instance.

The formalization has been implemented in the BPL-Tool developed in the Bigraphical Programming Languages (BPL) project. The tool allows for compositional definition, visualization and simulation of the execution of bigraphical reactive systems.

Model Transformations for Business Process Analysis and Execution (Tutorial)

Marlon Dumas

University of Tartu, Estonia & Queensland University of Technology, Australia
marlon.dumas@ut.ee

Abstract

A business process model is a representation of the way an organization operates to achieve a goal, such as delivering a product or a service. For example, an order-to-cash business process describes the activities that take place within a company from the moment a purchase order is received until its fulfillment and the settlement of the associated invoice. Business process models have at least two classes of users. On the one hand, business and system analysts use process models to identify and to evaluate business improvement options or to define system requirements. On the other hand, software developers are concerned with the automated execution of business processes based on detailed models. Depending on the purpose, a business process may be modeled at different abstraction levels and using different languages.

In this tutorial, we will review various model transformatios aimed at bridging between different business process modeling languages. The tutorial will discuss transformations from popular business process modeling notations (e.g. BPMN and BPEL) to Petri nets and state machines for the purpose of automated analysis [1, 5, 2, 8, 9]. We will also discuss transformations from business-oriented to IT-oriented process modeling languages to support system implementation. In particular, we will review techniques for transforming graph-oriented process models expressed in BPMN into block-structured process definitions in BPEL [4, 6, 3, 7]. Finally, we will discuss open issues related to the definition of reversible transformations and round-tripping between high-level and executable process modeling languages.

References

1. T. Bultan, X. Fu, and J. Su. Tools for automated verification of web services. In *Proceedings of the 2nd International Conference on Automated Technology for Verification and Analysis (ATVA), Taipei, Taiwan*, pages 8–10. Springer, October 2004.
2. H. Foster, S. Uchitel, J. Magee, and J. Kramer. Ws-engineer: A model-based approach to engineering web service compositions and choreography. In L. Baresi and E. Di Nitto, editors, *Test and Analysis of Web Services*, pages 87–119. Springer, 2007.

3. L. García-Bañuelos. Pattern Identification and Classification during the Translation from BPMN to BPEL. In *Proceedings of the On The Move to Meaningful Internet Systems (OTM) Confederated Conferences, Monterrey, Mexico*. Springer, October 2008.
4. R. Hauser and J. Koehler. Compiling process graphs into executable code. In *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE 2004), Vancouver, Canada*, pages 24–28. Springer, October 2004.
5. S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri nets. In W.M.P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Proceedings of the International Conference on Business Process Management (BPM2005)*, volume 3649 of *Lecture Notes in Computer Science*, pages 220–235, Nancy, France, September 2005. Springer-Verlag.
6. J. Mendling, K.B. Lassen, and U. Zdun. On the transformation of control flow between block-oriented and graph-oriented process modeling languages. *International Journal of Business Process Integration and Management*, 3(2), September 2008.
7. C. Ouyang, W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, and J. Mendling. From business process models to process-oriented software systems. *ACM Transactions on Software Engineering Methodology*, 2009. Preprint available at: http://eprints.qut.edu.au/archive/00005266/01/5266.pdf.
8. C. Ouyang, H.M.W. Verbeek, W.M.P. van der Aalst, S. Breutel, M. Dumas, and A.H.M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67(2-3):162–198, 2007.
9. W. M. P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek. Conformance checking of service behavior. *ACM Transactions Internet Technology*, 8(3), 2008.

JoSEL: A Job Specification and Execution Language for Model Checking

Michael Westergaard and Lars Michael Kristensen

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

Abstract. Model checking tools and techniques are being applied for verification of concurrent systems by users having different skills and background. This ranges from formal methods experts with detailed knowledge of the inner workings of the tools over students learning about model checking techniques to engineers that are mostly interested in applying the technology as a black-box. This paper proposes JoSEL, a visual language for specification of executable model checking jobs. JoSEL makes it possible to work at different levels of abstraction when interacting with model checking tools and thereby support the different kinds of users in a coherent manner. A verification job in JoSEL consists of tasks, ports, and connections describing the models to be verified, the behavioural properties to be checked, and the model checking techniques to be applied. A job can then be mapped onto an underlying model checking tool for execution. We introduce the syntax of JoSEL, informally define its semantics, and describe how it has been realised in the ASAP model checking platform.

1 Introduction

Model checking [1] is a very useful technique for validating the correctness of concurrent systems. Algorithms for performing model checking can be highly automated and a multitude of model checking tools have been implemented. In order to cope with the inherent state explosion problem, a large number of reduction techniques have been devised [9]. Reduction techniques typically exploit characteristics of systems to represent states more efficiently or to store only some of the states, but no reduction technique performs well for all systems. Also, many reduction techniques can be parametrised, and the chosen parameters can greatly affect the performance of the model checker. A useful model checker must therefore allow the user to select between a large number of reduction techniques and set parameters for each. Furthermore, new reduction techniques are constantly being developed, and a model checker must be able to incorporate new techniques easily. This means that it must be possible to augment the model checker, and that it must be possible for the user to access new techniques in a uniform way.

Model checkers can be used in at least three settings: by researchers (who develop reduction techniques), by students (who learn about model checking

and reduction techniques), and by engineers (who analyse real-life systems). The three kinds of users have different backgrounds and approaches to using model checkers. Researchers may need extremely fine-grained control over all of the parameters, know in detail how the reduction techniques work, and often need to experiment with new reduction techniques and compare them to existing techniques. A student in the process of learning initially knows little about reduction techniques and wishes to experiment without pre-existing intuition about the techniques. An engineer will seldom learn the reduction techniques in detail so the model checker should basically be a push-button technology. Sometimes an engineer needs to fine-tune a few parameters. This means that users of a model checker work on different levels and need different abstractions. It is therefore desirable that a model checker makes it possible to hide details of the supported techniques, but also to fine-tune details when required.

The contribution of this paper is to propose the verification Job Specification and Execution Language (JoSEL) aimed at supporting users with different background when applying and experimenting with model checkers to solve concrete verification problems. A verification job in JoSEL consists of tasks, input/output ports, and connections. A job typically specifies model(s), properties to be verified, and model checking techniques to be applied. The tasks correspond to software components of an underlying model checker and the input ports of a task specify required parameters to the component. The output ports of a task specify the results produced when the component is executed. Output produced by one component can be used as input for other components by connecting the corresponding tasks using connections. Examples of components are storages, which can store reachable states, queues containing states that need to be processed, queries (e.g., in a temporal logic) that express behavioural properties, and hash-functions for storing states in hash-tables. JoSEL is independent of any concrete model checking tool, but for a verification job to be executed the tasks must be mapped onto components of a concrete model checking tool. This can be done in a manner which is fully transparent to the user, and in this paper we show how this has been implemented in the ASAP model checking platform [6].

JoSEL should be viewed as a flexible graphical alternative to hard-coding the supported model checking techniques into dialog boxes in a graphical user interface or relying on complex command-line arguments to manipulate the parameters of the model checker. To allow users to abstract away details that may not be required for everyone, JoSEL has macro tasks that basically represent a set of tasks and their interconnections. If a macro is given a meaningful name, it is possible to ignore the details of the compound task it represents. Still, it is easy to allow users to manipulate the details if needed. To make specifications created in JoSEL less prone to human errors, we assign types to the input and output ports of tasks. Finally, in JoSEL the user does not have to worry about the order in which tasks are to be executed as we can do a simple dependency analysis and execute components in a correct order.

It is possible to specify verification tasks using other techniques, e.g., a textual description like shell scripts, Makefiles or custom scripting languages, such as

SVL [4]. Textual languages have several disadvantages making them difficult to use for inexperienced users. Firstly, the user needs to remember keywords and names of the available components, and, secondly, textual languages are very prone to typing errors. Furthermore, and more importantly, textual descriptions have a built-in order, and it may be confusing if it does not correspond to the order of execution. Hence, at least at some level, users need to worry about the order in which the components are executed. Graphical alternatives exist as well. One example is a workflow specification language [10], but they are not really tailored to dealing with model checking, and we find that a language designed specifically for model checking is preferable. An example of a language for this purpose is jETI [7], whose main objective is to easily compose web-services. jETI does not, however, make it easy to specify several similar tasks, and is more tailored to a write once, run many times paradigm, whereas we seek a language that makes it possible for experienced users to make building blocks that can be used and modified by less experienced users that are not necessarily programmers. JoSEL addresses the above problems. As JoSEL is a graphical language it can show users all available components and only allow the user to select legal components, so the user does not have to remember the names of the components, thereby alleviating the problems with textual languages. JoSEL also takes care of the order of execution so that all input values required for each part of a task are guaranteed to be available. Furthermore, JoSEL is designed with abstraction mechanisms that allow experienced users to design components that can be used directly as black boxes or adapted by other users.

The rest of this paper is structured as follows: Section 2 introduces the constructs of JoSEL and defines its syntax. In Sect. 3, we informally define the semantics of JoSEL and indicate how we have realised JoSEL in the ASAP model checking platform. We sum up our conclusions in Sect. 5.

2 Syntax

The Job Specification and Execution Language (JoSEL) is inspired by workflow specifications [11] and data-flow graphs [8]. We do not use these languages directly as we aim at a language tailored for specification of verification jobs. Furthermore, we want our job specifications to be executable which means that we must be able to distinguish the inputs and outputs of processes, and we need to be able to instantiate and synchronise processes based upon the data content. The two latter requirements means that we cannot use data-flow graphs directly. In this section, we formally introduce the graphical syntax of JoSEL; in the next section we consider the execution of JoSEL specifications.

To introduce JoSEL we use an example of a verification job where we want to verify two safety properties of a given (formal) model. The properties we wish to check are that the model contains no dead-locks (states without enabled transitions), and that it is impossible to reach a state where a certain buffer overflows. We assume that dead-lock freeness is a standard property built into the underlying model checking tool, and that the buffer overflow property can be expressed

as a state predicate in a propositional logic. Executing this job consists of systematically traversing all reachable states of the model while storing already encountered states in a storage. If a violation is found, i.e., if we encounter a dead-locked state or a state where the buffer has overflowed, an error-path leading to the violating state should be reported to the user. We introduce JoSEL by constructing the example job bottom-up from scratch, i.e., we deal with the most specific details first. In a real model checker, it is rarely required to specify all parts as some building blocks would be available beforehand. We build the job to illustrate how JoSEL supports accessing the detailed parameters of the model checker components when required.

2.1 Tasks

The basic unit of computation in JoSEL is a *task*, which corresponds to a process in data-flow graphs. The reason for the different terminology is in part due to workflow specifications, whose terminology we have adopted, and due to the fact that a task can be instantiated more than once when a job is executed, thereby giving rise to multiple *processes*. As we need to graphically fit information for inputs and outputs into each task, we do not use a circle to represent tasks, but rather a rounded rectangle. Figure 1 shows a task named Instantiate Hash Table Storage. The task in Fig. 1 is able to instantiate the storage required for the verification job.

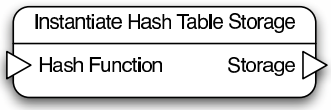


Fig. 1: A task with one input port and one output port.

Associated with tasks are two kinds of *ports*: *input ports* and *output ports*. Input ports specify the input data required for a task and are by convention located on the left-hand side of a task. Output ports specify the output data produced by a task and are located on the right-hand side of a task. Ports are represented by triangles; input ports point into the task, symbolising values going into the task, and output ports point out of the task. The name of a port is written inside the task next to the port itself. The task in Fig. 1 has one input port and one output port. The set of input ports of a task is denoted I and the set of output ports of a task is denoted O. In the example we have $I = \{\text{Hash Function}\}$ and $O = \{\text{Storage}\}$. This specifies that in order to generate a storage (the output port) that store states in a hash table, we need a hash function (the input port).

Associated with each port of a task is a *port type* specifying the kind of data that can be consumed/produced at the port. This is represented by a mapping

$\tau : P \to \Sigma$ from the set of all ports $P = I \cup O$ of the task to a set of types Σ. The actual types allowed are implementation specific, but can, e.g., be simple strings, integers and files, or more complex data-structures like hash-tables or representations of formal models and state space graphs. The type of each port is not reflected in the graphical representation, since in practice the type of a port can be inferred from the name of the port and task. In the example in Fig. 1, it is fairly clear that the input has a type that is a hash function and the type of the output port is storage. The main reason for omitting the types from the graphical representation is simplicity of the drawing, but tools implementing JoSEL are free to reveal the types if desired. Associated with each port is also a *port mode* which is either unit, iterator, or collection. The port mode describes how data is consumed (input ports) and produced (output ports), and is specified by a port mode mapping PM. Basically, unit ports only produce/consume one value, whereas iterator and collection ports can consume/produce more than one value. We explain port modes in more detail in Sect. 2.4 and Sect. 3 when discussing the execution of tasks. The following summarises the definition of tasks.

Definition 1. *A **task** is a tuple $T = (I, O, PM, \Sigma, \tau)$ where:*

- *I is a finite set of **input ports**,*
- *O is a finite set of **output ports** such that $I \cap O = \emptyset$,*
- *$PM : P \to \{\mathsf{unit}, \mathsf{iterator}, \mathsf{collection}\}$ assigns to each port a **port mode**, where $P = I \cup O$.*
- *Σ is a set of types and $\tau : I \cup O \to \Sigma$ assigns to each port a **port type**.* □

2.2 Jobs

When we have a task able to generate a hash function for use by the task in Fig. 1, we would like to be able to specify that the hash function produced by that task should be passed to the Instantiate Hash Table Storage task. A *job* consists of a set of tasks $\mathcal{T}$ connected by a set of *connections* C describing how the output produced by one task is used as input to other tasks. A connection is a pair (c_O, c_I) connecting an output port c_O of one task with an input port c_I of another task. We represent a connection by a line from the output port to the input port. In Fig. 2, the output port of Instantiate Hash Function is connected to the input port of Instantiate Hash Storage. In this case Instantiate Hash Function is able to generate a hash function. This is done from a specific model that must be provided. The generated hash function is passed to Instantiate Hash Storage, which can generate a storage, which is able to store states of the model given to Instantiate Hash Function. Input ports can be connected to output ports with the same type. It is, however, easy to allow sub-typing, by loosening this requirement such that the type of the output port is only required to be a sub-type of the type of the input port. This is also how we have implemented it in ASAP [6] (using the type hierarchy of Java as the sub-typing relation), but we have omitted it in the formal definition for simplicity. Furthermore, we require that the directed graph induced by the tasks and connections in a job is acyclic

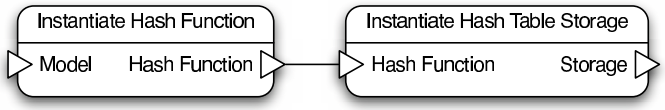


Fig. 2: A job consisting of two tasks.

to ensure termination. As we require that input ports are always to the left and output ports are to the right of the task, this naturally leads to a flow from the left to the right of the job. The following summarises the definition of a job:

Definition 2. *A **job** is a tuple $J = (\mathcal{T}, C)$ where:*

- $\mathcal{T} = \{T_i = (I_i, O_i, PM_i, \Sigma_i, \tau_i)\}_{1 \leq i \leq n}$ *is a finite set of **tasks** satisfying $i \neq j \Rightarrow P_i \cap P_j = \emptyset$, where $P_i = I_i \cup O_i$*
- $C \subseteq \mathcal{O} \times \mathcal{I}$ *is a set of **connections** (where $\mathcal{I} = \bigcup_{T_i \in \mathcal{T}} I_i$, $\mathcal{O} = \bigcup_{T_i \in \mathcal{T}} O_i$) satisfying that the directed graph induced by tasks and connections is acyclic,*
- *For all $c_O \in O_i$ and $c_I \in I_j$ such that $(c_O, c_I) \in C$ we have $\tau_i(c_O) = \tau_j(c_I)$.* □

It should be noted that the definition allows multiple connections to and from each port. This is, e.g., useful if we want to use the value produced at an output port in multiple locations or if we want to consume values from multiple sources or instantiate processes for values calculated by different processes. It is possible for tasks in a job to have *free ports*, i.e., ports that are not assigned via connections. The set of free input ports of a job J is denoted $Free_I(J)$. A job is *closed* if it has no free input ports, otherwise it is *open*. The job in Fig. 2 is open as the input port Model of Instantiate Hash Function is free.

2.3 Macro Tasks

In order to support different levels of abstraction, we introduce *macro tasks* (or macros for short). A macro task is a high-level representation of a job with free ports. A macro task can be thought of as a component of a job which is intended to be re-used in different settings and therefore does not have all parameters defined immediately. For example, we may want to reuse the job from Fig. 2 in several different job descriptions, as it provides an implementation of a way to generate a storage that is able to store states of a given model. Instead of copying the entire job, we just draw a special macro task. A macro task is graphically represented like an ordinary task except that we draw its outline using double lines. An example of a macro task can be seen in Fig. 3 (top). In order to specify the input and output ports of a macro, we introduce a special kind of port, which states that when a job is used as a macro, this port should be available to the user of the macro. Such ports are called *exported ports* (exported input/output ports), and are drawn using a double outline, as in Fig. 3 (bottom), where Model and Storage are exported. Outside the task, next to an exported port,

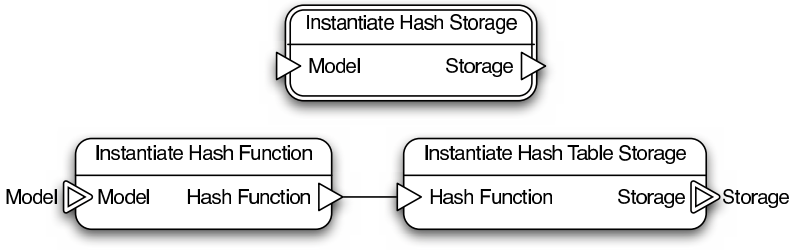


Fig. 3: A macro task (top) and the corresponding job (bottom).

we write the *exported name* of the port. The name is chosen by the user, but will often be the same as the name of the port. The macro task in Fig. 3 (top) can represent the job in Fig. 3 (bottom). The exported name becomes the name of the port in the macro. Macros are purely syntactical, and can be removed by repeatedly replacing macro tasks with the tasks of underlying jobs, moving connections to/from a port on a macro task to the corresponding exported port in the job, and removing all exported ports. We require that the set of exported input ports includes all the free input ports of the job and that the exported ports are contained in the ports of the underlying job. Port modes and types of an exported job are inherited from port modes and types of the underlying job. The following summarises the definition of a macro task.

Definition 3. *Let $J=(\mathcal{T},C)$ be a job and $\mathcal{T}=\{T_i=(I_i,O_i,PM_i,\Sigma_i,\tau_i)\}_{1\leq i\leq n}$. J can be represented by a* ***macro task*** *which is a task $M=(I,O,PM,\Sigma,\tau)$ where:*

- *I is a set of* ***exported input ports*** *such that $I\subseteq\mathcal{I}$ and $Free_I(J)\subseteq I$, where $\mathcal{I}=\bigcup_{T_i\in\mathcal{T}} I_i$.*
- *O is a set of* ***exported output ports*** *such that $O\subseteq\mathcal{O}$, where $\mathcal{O}=\bigcup_{T_i\in\mathcal{T}} O_i$.*
- *$PM(p)=PM_i(p)$ for all $p\in P\cap P_i$ and $1\leq i\leq n$, where $P=I\cup O$ and $P_i=I_i\cup O_i$.*
- *$\Sigma=\bigcup_{1\leq i\leq n}\Sigma_i$ and $\tau(p)=\tau_i(p)$ for all $p\in P\cap P_i$ and $1\leq i\leq n$.* □

2.4 Hierarchical Jobs

In our running example we have until now constructed a means to store states of a given formal model (cf. Fig. 3). We now hierarchically construct the remainder of the verification job for checking the two safety properties. The next step is to construct a job for checking a safety property of a model. Such a job is shown in Fig. 4. The job has a Waiting Set Exploration task which traverses all states, stores all states we have already visited in a Storage, and stores all discovered but not yet processed in a Waiting Set. The task is parametrised to allow flexibility. For example, we can use a storage storing states in a balanced tree or on disk instead, or we can use a different waiting set to impose a different

traversal order. The typing of ports makes sure that only storages and waiting sets that are actually usable can be connected. Our previous macro task from Fig. 3, Instantiate Hash Storage, takes care of creating a storage, and the task Instantiate Queue takes care of instantiating a specific waiting set, implemented as a queue, which imposes a breadth-first traversal of the states. As the model has to be used both for the Waiting Set Exploration and as input for instantiating the storage and waiting set, we use a Multiplex task, which just lets its input flow unaltered to the output port. Here we have used the ability to connect an output port to multiple input ports. The result of the Waiting Set Exploration is a Traversal, an abstract result, which can be used by the On-the-fly Safety Checker to check properties. We see that the job in Fig. 4 exports four ports, Model, Properties, Answer, and Error trace. The Instantiate Queue, Waiting Set Exploration, and On-the-fly Safety Checker are all components of the underlying model checker.

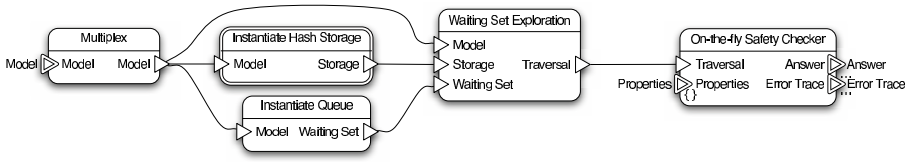


Fig. 4: Specification of a verification job for checking safety properties.

The exported port Properties in Fig. 4 has port mode *collection*. We indicate that a port has mode collection by annotating it with "{ }" (like a set). On-the-fly Safety Checker takes care of actually checking the properties given. It produces an Answer (a boolean value; did the property hold) and a path to violating states (if any). In order to do this efficiently, it needs the set of all safety properties when the task is started which is exactly what collection ports specify. As the On-the-fly Safety Checker task can take more than one property as input, it may also produce more than one output for both Answer and Error trace. We could assign both of these mode collection, but it is possible that we can provide an answer for one of the safety properties earlier than we can for other properties. The Answer and Error trace ports therefore have port mode *iterator*, which indicate that multiple outputs may be produced, and that they are produced one at a time as they become available. Iterator ports are annotated with "...". Letting Answer and Error trace be iterator ports allows us to show a user that one of the properties has been violated before the execution of the task has completed. We also allow iterator ports to be input ports (indicating that we can consume values one at a time) and collection ports to be output ports (indicating that all results are returned when the task has completed). Ports that are neither collection ports nor iterator ports are *unit* ports which means that they consume/produce exactly one value per task instantiation.

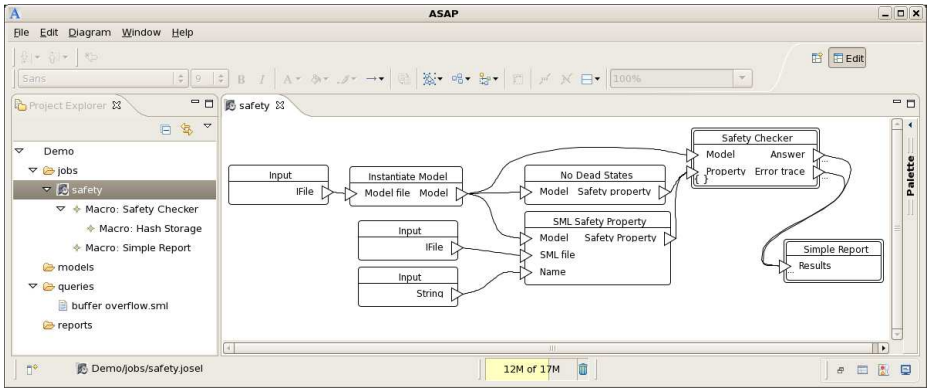


Fig. 5: Using the ASAP tool to model check using JoSEL.

To check the two safety properties for a given model we create the job in Fig. 5 which constitute the *root job* of our example verification job. Figure 5 is a snapshot from the implementation of JoSEL in ASAP. The Safety Checker macro task represents the job in Fig. 4. In addition, we have tasks to instantiate a safety property for checking dead-locks (No Dead States), and to instantiate a safety property from a user-supplied file (for checking buffer overflow; two Inputs, one for specifying the file and one to give a descriptive name of the property, and SML Safety Property). Both of these properties are input to the Safety Checker. Additionally, we have tasks to instantiate a model from a user-supplied file (Input and Instantiate Model). The model is also passed to the Safety Checker. Furthermore, we want to generate a report which shows whether the properties were violated and, if so, how we can reach a state where the property is violated. At our disposal we have a Simple Report macro which is able to generate such reports. The task is passed the values produced by the Safety Checker. The Simple Report task is able to update the report as answers arrive, so we can see partial results during the calculation. The Simple Report macro is an example of connecting more than one output port to a single input port. We need not worry about the actual implementation of the Simple Report macro as long as we are aware that it creates a report and displays the values passed to it. The Simple Report task is in fact implemented so that it is possible to extend its capbilities to show values of various types, but, as the actual operations of the task is out of scope of this paper, we can just assume that the task is able to perform reasonable post-processing and display any value passed to it.

Our example shows that JoSEL can express rather complex jobs at different levels of abstraction. The top-level view (Fig. 5) allows us to specify which properties to check of which model and what to do with the results. The next level (Fig. 4) allows us to determine the traversal strategy (using different waiting set implementations) and how to store states (by using another storage implementation). At this level we can also swap the safety checker for, e.g., an LTL

model checker, which is able to check more complex properties. At the lowest level (Fig. 3) we can change which hash function is used to store states in a hash table. The three levels together forms a *hierarchical job* as defined below.

Definition 4. *A **hierarchical job** is a tuple $H = (\mathcal{J}, J_r, \mathcal{T}, SJ)$ where:*

- $\mathcal{J} = \{J_i = (\mathcal{T}_i, C_i)\}_{1 \leq i \leq n}$ *is a finite set of jobs such that $\mathcal{T}_i \cap \mathcal{T}_j = \emptyset$ and $C_i \cap C_j = \emptyset$ for $i \neq j$.*
- $J_r \in \mathcal{J}$ *is a distinguished **root job**,*
- $\mathcal{T} = \{T_k = \{I_k, O_k, PM_k, \Sigma_k, \tau_k\}\}_{1 \leq k \leq m}$ *is a set of macro tasks such that for all $T_k \in \mathcal{T}$ there exists a $J_i \in \mathcal{J}$ such that $T_k \in \mathcal{T}_i$,*
- $SJ : \mathcal{T} \to \mathcal{J}$ *assigns jobs to macro tasks such that SJ is onto $\mathcal{J} \setminus \{J_r\}$, T_k is a macro task for $SJ(T_k)$ for all $T_k \in \mathcal{T}$ (cf. Def. 3), and the graph induced by $\mathcal{J}$ and SJ is a tree rooted in T_r.*
- *For $J_i, J_j \in \mathcal{J}$ with $i \neq j$, $(\mathcal{P}_i \setminus \mathcal{P}'_i) \cap (\mathcal{P}_j \setminus \mathcal{P}'_j) = \emptyset$, where $\mathcal{P}_i$ are the ports of tasks of J_i and $\mathcal{P}'_i$ are the ports of macro tasks of J_i.* □

The last requirement in the above definition ensures that only exported ports are shared between jobs. A hierarchical job can be flattened by replacing all macro tasks with the corresponding jobs and moving the connections to the correct ports as defined below.

Definition 5. *Let $H = (\mathcal{J}, J_r, \mathcal{T}, SJ)$ be a hierarchical job with $\mathcal{J} = \{J_i = (\mathcal{T}_i, C_i)\}_{1 \leq i \leq n}$. The corresponding **flattened job** is $J = \{T, C\}$ where $T = \bigcup_{1 \leq i \leq n} T_i \setminus \mathcal{T}$ and $C = \bigcup_{1 \leq i \leq n} C_i$.* □

A hierarchical job is said to be closed when the root job is closed. Whenever a hierarchical job is closed, so is the corresponding flattened job as stated in the following proposition.

Proposition 1. *If $H = (\mathcal{J}, J_r, \mathcal{T}, SJ)$ is a hierarchical job. Then the corresponding flattened job is closed if and only if H is closed.*

Proof. Any free input ports of a job $J_j = SJ(T)$ with $T \in \mathcal{T}_i$ is either connected in T_i or a free input port of T_j (Def. 3). Thus any free port of J_r is a free port of the flattened job as no connections are broken during flattening, and H is closed if and only if J_r is closed. □

3 Execution of JoSEL Specifications

In this section we outline the semantics of JoSEL and outline how this has been implemented in the ASAP model checking platform [6]. The semantics define the dynamics of the language. We can formally define the semantics, but for an intuitive understanding of JoSEL this is not necessary, and may in fact be counter-productive due to the level of detail required to fully specify JoSEL. Instead we intuitively describe how jobs are executed with particular focus on how instantiation works when ports with different modes interact.

The basic idea in the semantics of JoSEL is to instantiate each task such that whenever a task is instantiated, all preceding tasks (tasks with a connection to an input port of the task in question) have been instantiated and completed (at least all values for non-iterator input ports). For non-hierarchical jobs, this can be ensured by performing a topological sorting of the tasks where a task $T_1 = (I_1, O_1, PM_1, \Sigma_1, \tau_1)$ is sorted before $T_2 = (I_2, O_2, PM_2, \Sigma_2, \tau_2)$ if there is a connection (c_O, c_I) from T_1 to T_2 ($c_O \in O_1$ and $c_I \in I_2$). As this graph is acyclic (cf. Def. 2) this sorting is well-defined and yields a total order of all tasks (tasks that using the aforementioned ordering are equal or incomparable are simply taken in an arbitrary order). If we further assume that the job is closed, executing jobs according to this order will ensure that all values are available when we want to execute a task. If we are given a hierarchical job, we construct (at least conceptually) the corresponding flattened job according to Def. 5. Due to Prop. 1, we easily see that a hierarchical job can be executed when it is closed.

3.1 Port Modes and Instances of Tasks

If ports with different modes are connected, we may either instantiate a task more than once or synchronise a number of tasks depending on the port modes. The effects of all possible combinations of ports are summarised in Table 1. Basically, whenever an input port has mode unit, we start an instance for each value it is given. If we have multiple connections with different modes to an input port with mode unit, we simply start a thread for each value arriving from each connection, essentially implementing a fork semantics. Similarly, we always pass exactly one collection or iterator to input ports with mode collection and iterator, respectively. The result of having multiple connections to such a port is thus to create exactly one collection containing all data or one iterator iterating over all values. Iterator output ports differ from collection output ports in that values can be passed before the task producing them has terminated. This is, e.g., useful for data collection. Table 1 shows what happens when a single output port is connected to a single input port.

Based on the intuition of the semantics, we obtain the following proposition which states that if the execution of all tasks terminate for all inputs and produce only a finite amount of data, execution of the job always terminates.

Proposition 2. *Let H be a closed hierarchical job. If all tasks of jobs in H terminate for any input and all iterators produce only a finite number of data-elements, the execution of H eventually terminates.*

Proof. The proof is in the structure of the flattened job corresponding to H. If we assume that the predecessor of a task T produce only a finite number of data elements and that predecessor tasks can only be instantiated a finite number of times, T can only be instantiated a finite number of times. □

3.2 Implementation in ASAP

The outlined semantics assume that we wait for all instances of a task to run to completion before starting subsequent tasks. As most modern computers have

Table 1: The effect of connecting different kinds of ports.

Output	Input		
	Unit	Iterator	Collection
Unit	Directly pass value	Create iterator over one value	Create a collection with one value
Iterator	Fork as values arrive	Pass all values to a single task instance as they arrive	Synchronise; wait until all values have arrived and pass in a collection
Collection	Fork for each value in collection	Create iterator over collection and pass to a single task instance	Pass all values to a single task instance

CPUs with two, four, or more cores, this is not optimal. Instead, we have made a more efficient implementation in ASAP, which basically allows tasks to start as soon as all the input values they require are available. From Prop. 2, we see that this strategy eventually will lead to executing all tasks.

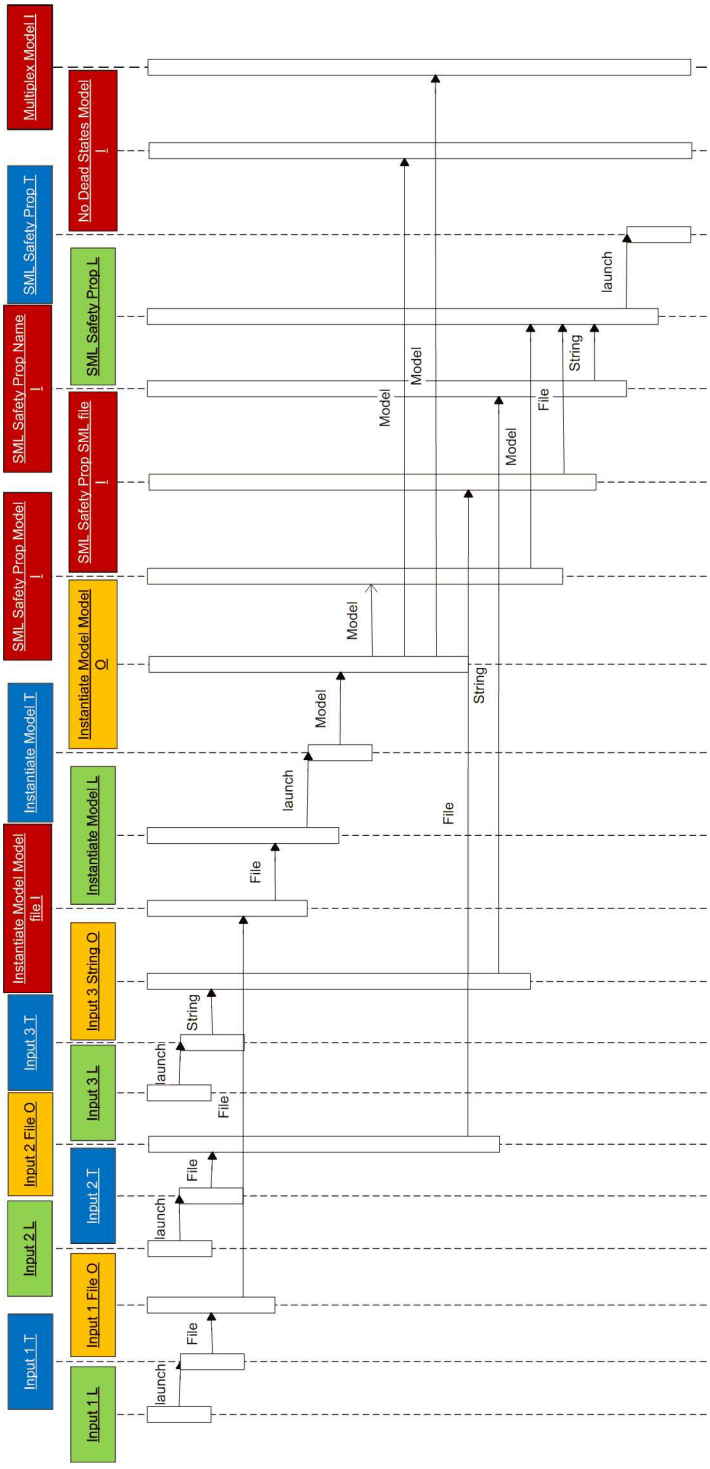
The implementation of JoSEL in ASAP is done by implementing the data-structures corresponding to the tuples in the previous section using the Eclipse Modeling Framework (EMF) [3], implementing a simple object oriented representation of tasks, connections, and jobs. On top of that, we have developed a graphical editor using the Eclipse Graphical Modeling Framework [2], which semi-automatically generates a graphical editor that automatically maintains the object oriented representation of JoSEL, so we only need to consider an abstract representation of JoSEL when dealing with the execution of jobs. As we only deal with the abstract representation it is in principle possible to develop other concrete syntaxes for JoSEL if one, e.g., prefers a textual syntax.

Our implementation of execution of JoSEL specifications aims to increase concurrency and observes that starting many threads that are just waiting to be executed is relatively inexpensive. The idea is to create a thread for each task (a *task thread*) which is intended to contain the actual implementation of the task, a thread for each output port (an *output thread*) to distribute the output to multiple recipients, a thread for each input port (an *input thread*) to collect and synchronise input from multiple sources and put the data to the correct format, and finally a thread to spawn the appropriate task thread and pass parameters (a *launcher thread*). We may start multiple instances of a task thread, one for each execution of the task, but only one instance of each output, input, and launcher thread. Additionally, we have a communication channel for each output port, allowing task threads to transmit values to the correct output threads, a channel for each connection, allowing output threads to transmit data to the correct input threads, and one for each input port, allowing input threads to pass data to the correct launcher thread. Finally, We need to keep track of how many instances of each task thread we have started and when we will start no

more task threads for a specific task thread. This information allows subsequent output threads to realise when they have received all output values from task threads.

In the figure on page 14, we see some of the threads instantiated in ASAP to execute the JoSEL specification in Figs. 3, 4, and 5. Threads are named after their task and, in the case of input and output threads, after their port as well. Threads are identified by an L, a T, an I, or an O for launcher, task, input, and output threads, respectively. We have numbered the three Input tasks in Fig. 5 from the top to the bottom in order to better be able to distinguish them. Active threads are marked by a white band. We notice that in the beginning all threads except for task threads are active (but waiting). Initially, the launcher thread for the Input 1 task realises that it has all input values required to start an instance of the task, so it launches a task thread for the task. This thread runs to completion and produces a single output value, which is transmitted to the output thread for the File output port of the Input 1 task. Independently, the launcher, task, and output threads for the Input 2 and Input 3 task do the same thing. The single connection from the File output port of the Input 1 task to the Model file input port of the Instantiate Model task, makes the output thread transmit the value to the input thread of the Model file input thread, which is then awoken. The value needs no translation, so it is just transmitted directly to the launcher thread of Instantiate Model, which realises that it has received enough input values to launch an instance of the Instantiate Model task thread. This runs and produces a Model output value, which is transmitted to the output thread. The corresponding output port has three connections, so the value is transmitted to the three corresponding input threads. At some point, the File and String values have been transmitted to the SML file and Name input threads of the SML Safety Property task. The three input threads transmit the values to the launcher thread of the SML Safety Property task, which realises, it has enough information to launch a SML Safety Property task thread. The computation continues in this manner.

As indicated from the example the threads collaborate to perform the entire computation specified by the specification. The different functions of the different kinds of threads have been summarised in Table 2. Task threads perform the actual execution of a single instance of a task. A task thread receives a correctly formatted value for each input port, executes the task with the given values and transmits values produced for each output port to the corresponding output process. Output threads take care of transmitting values to the correct input threads, and as such implement the connections. When there is only a single connection from an output port, this just consists of passing on the value. In the case where there are more outgoing connections (such as the Model output port of the Instantiate Model task in Fig. 5), the value must be copied, in particular if it is a port with collection or iterator mode. Input threads are responsible for receiving values from multiple output threads (as in the case of the Properties input port of the On-the-fly Safety Checker in Fig. 4, which is exported and connected to multiple output ports in Fig. 5). Input threads are also responsible for


Input 1 T
Input 1 L
Input 1 File O
launch
File
Input 2 L
Input 2 T
launch
File
Input 2 File O
Input 3 T
Input 3 L
launch
File
Instantiate Model file I
Input 3 String O
String
Instantiate Model T
Instantiate Model L
File
launch
SML Safety Prop Model I
Instantiate Model Model O
Model
Model
String
SML Safety Prop Name I
SML Safety Prop SML file I
Model
Model
File
SML Safety Prop T
SML Safety Prop L
String
launch
Multiplex Model I
No Dead States Model I

converting the received values to the correct format. In the example of the Properties input port of On-the-fly Safety Checker, the input port has mode collection, but it receives values from two output ports with mode unit. The input thread must therefore construct a collection containing the values received. A similar conversion must take place when the input port has mode iterator. If the input port has mode unit, values are simply transmitted one at a time as they are received, optionally extracting values from collections or iterators. Finally, input value(s) on the correct form are transmitted to the launcher thread. Launcher threads receive correctly formatted input values and as soon as a value is received for each input port, a new task thread is started with the correct parameters. As more than one value can be received on unit input ports (collection and iterator ports always receive exactly one collection or iterator), it is possible that more than one task instance needs to be started. The launcher thread takes care of starting new task threads as soon as possible by constructing new elements of the Cartesian product of the input values as soon as possible.

Table 2: The function to be performed by each kind of thread.

Thread	Function
Task	Run a single instance of the task with the provided parameters and transmit the produced results to the correct output threads.
Output	Receive values and transmit them to all input threads according to the connections.
Input	Collect values from multiple output processes and put them into the correct form according to the input port mode.
Launcher	Receive values from input threads and launch new task threads whenever enough values have been received to provide a full set of inputs.

4 Example

Let us see some examples of use of JoSEL in ASAP from the point of view of different users. Assume a person from the industry wishes to verify a concrete model of a network protocol. The task is to verify that the protocol contains no dead-locks and that the buffers of the participants do not overflow. As the model is of a real-life system, it is suspected that the state space may be large, and as the protocol works in steps, it is decided, guided by ASAP, to use the sweep-line state space reduction method, which exploits progress to only store parts of the state space in memory at any time. A standard JoSEL template specification for checking for dead-locks using the sweep-line method is created using a simple wizard, and a specification like the one in Fig. 6 is obtained (for legibility, we have removed all the input tasks; all unassigned input ports are connected to the output of a corresponding input task). Compared to the example from Fig. 5 the

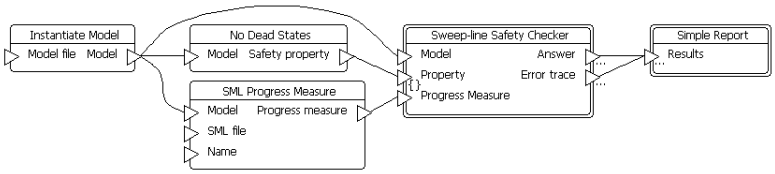


Fig. 6: Initial dead-lock freeness checker using sweep-line method.

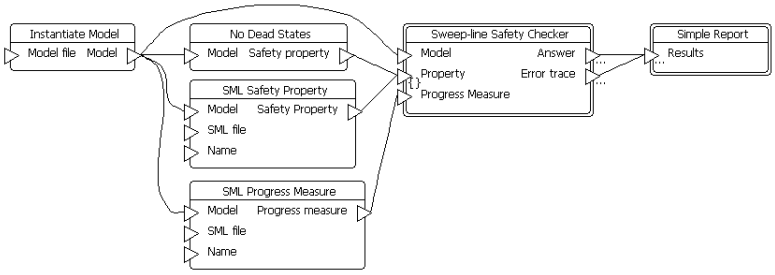


Fig. 7: Dead-lock and buffer overflow checker using sweep-line method.

only difference we see is that there is no SML Safety Property task and instead we have a SML Progress Measure, which is used by the sweep-line method to decide when states no longer need to be kept in memory.

Our industry person slightly modifies the task and adds tasks to specify the no buffer overflow property. The task now looks like the one in Fig. 7, which identical to the one in Fig. 5 except for the added progress measure. Another way to arrive at the job in Fig. 7 would be start from Fig. 5, observe that the verification is not able to go through as too much memory is consumed, and delete the Safety Checker macro and replace it with a Sweep-line Safety Checker built into the tool.

Assume now a student wants to experiment with the sweep-line method. The student starts with the same job from Fig. 6 and digs into the details of the Sweep-line Safety Checker and sees the implementation shown in Fig. 8. Compared to the implementation of the Safety Checker in Fig. 4, we have removed the Queue and use a Sweep-line Exploration instead of a Waiting Set Exploration. The queue is not needed as the sweep-line method imposes a certain order of traversal (least progress first). We also see that the Sweep-line Exploration task takes an additional input value, namely Persistent initial states, which is a heuristic improving speed for certain systems that return to the initial state after executing. This is usually a very cheap way to speed up a large class of models, and is set to true by default. In order to better understand the method, a student may try setting the value to false and observe the performance on different models.

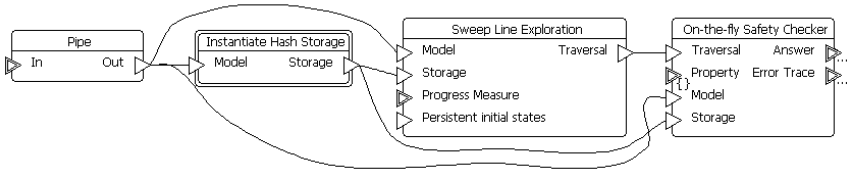


Fig. 8: Implementation of sweep-line safety checker.

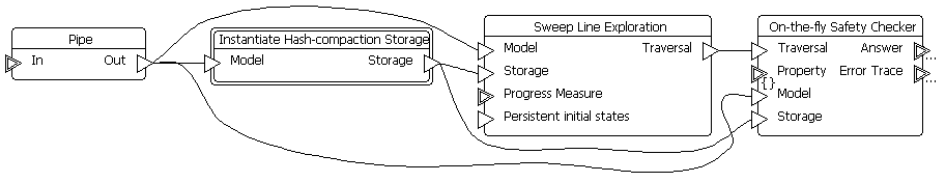


Fig. 9: Job for checking safety properties using a combination of the sweep-line method and hash compaction.

Finally, let us consider a researcher who experiments with the sweep-line method, knowing that a good way to obtain even better performance is to combine different methods. In addition to the sweep-line method, the researcher also knows about hash compaction, which, instead of storing the full states, just stores a hash value. This saves a lot of memory, as the hash value is usually 4 or 8 bytes whereas the full state is usually thousands of bytes. The caveat is that the method may not explore all states as several states may have the same hash value. The researcher thinks that using the sweep-line method to remove states known not to be encountered again may reduce this probability and use less memory than using either method by itself. The reasearcer therefore deletes the Instantiate Hash Storage macro, replaces it with a Instantiate Hash-compaction Storage macro, and obtains Fig. 9. If it is found that this method performs well, the constructed task can be saved and added to the tool so others can benefit in the future.

Suppose that the researcher wishes to test the performance of the new method and eliminate as much overhead to get as accurate results as possible. Looking at the details of the Hash-compaction Storage in Fig. 10, the researcher observes that it is possible to tune the initial size of the hash-table used to store states. Initially, this value is set rather small (1000) as the overhead in real cases is relatively small. If the researcher knows that the models uses for testing have at most 100,000 states and that the used computer has sufficient memory to handle this, the values can be set to 200,000 so the hash table never needs to be re-balanced, which will improve the performance slightly and factor out the cost of re-balancing a hash table, which provides a more fair and accurate comparison between the standard sweep-line method and the new method.

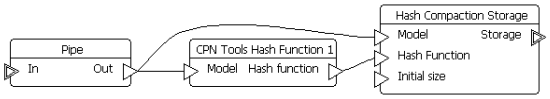


Fig. 10: Details of the hash compaction storage.

Finally, we may argue that even the task in Fig. 5 is not that simple as it contains eight tasks, of which only two or three are really interesting. We can instead create a new macro like the one in Fig. 11, where we have basically removed the inputs to model file and SML file from the task in Fig. 5 and exported the ports. We can go even simpler and remove the Property input port as well by replacing it with a set of standard properties that make sense for all models.

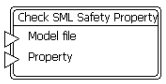


Fig. 11: More abstract top level of the safety checker from Figs. 3–5.

5 Conclusion and Future Work

In this paper we have proposed the JoSEL language for specification and execution of verification jobs in model checking. By means of an example we have demonstrated how JoSEL can be used to define a simple safety checker, which can be modified easily at different levels of abstraction. We have informally defined the semantics of JoSEL and indicated an efficient implementation in ASAP.

An important requirement for JoSEL was to support users at different experience levels and with different reasons to use verification tools. We find that JoSEL supports this because of the hierarchy concept, as is supported by the examples given in Sect. 4.

Future work includes considering extensions of the JoSEL language. One interesting extension is to allow cyclic connections, so that we can iteratively improve a verification result. This is interesting for implementing incremental improvement of approximative reduction techniques, e.g., by iteratively increasing the size of the table used for bit-state hashing [5] until we have refuted the property or a certain threshold has been reached. The reason for not just doing this is that it would make the implementation more complex and, more importantly, it would be easier for users to make mistakes, as Prop. 2 would no longer hold. It would also be interesting to add a way to terminate a task or a set of tasks. This has interesting applications to, e.g., state-of-the art model

checking, in which we start model checking with all or at least several available model checking techniques. As soon as the first technique provides a definitive answer we terminate the execution of the other techniques. This should be fairly easy to do, and the main issue is to find a good graphical representation for this mechanism. A variant of macro tasks may be used as a composition mechanism stating which tasks to terminate. Many control structures need not be part of the language but can be implemented as tasks that do not correspond to anything in the underlying model-checking platform. On such example is an If task, which takes as input a boolean value test and an arbitrary value value. Depending on the value of the test, the value is either transmitted to a then or an else output port.

Another interesting direction is to investigate scheduling across multiple machines. It should be possible to extend JoSEL with annotations describing the time-wise and space-wise cost of a task and the produced values. It should then be possible to intelligently distribute data to the appropriate machines, taking into account that it may sometimes be more efficient to re-calculate data than to transmit results.

References

1. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*, chapter 12. The MIT Press, 1999.
2. Eclipse Graphical Modelling Framework (GMF). `www.eclipse.org/modeling/gmf/`.
3. Eclipse Modelling Framework (EMF). `www.eclipse.org/modeling/emf/`.
4. H. Garavel and F. Lang. SVL: A Scripting Language for Compositional Verification. In *Proc. of FORTE'01*, volume 197 of *IFIP Conference Proceedings*, pages 377–394. Kluwer, 2001.
5. G.J. Holzmann. An Analysis of Bitstate Hashing. *Formal Methods in System Design*, 13:289–307, 1998.
6. L.M. Kristensen and M. Westergaard. The ASCoVeCo State Space Analysis Platform: Next Generation Tool Support for State Space Analysis. In *Proc. of 8th CPN Workshop*, volume 584 of *DAIMI-PB*, pages 1–6, 2007.
7. T. Margaria, R. Nagel, and B. Steffen. Remote Integration and Coordination of Verification Tools in JEIT. In *Proc. of ECBS'05*, pages 431–436. IEEE Comp. Soc. Press, 2005.
8. R.S. Pressman. *Software Engineering*. McGraw-Hill, 1997.
9. A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 429–528. Springer-Verlag, 1998.
10. W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
11. W.M.P. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.

Formal Modeling and Analysis by Simulation of Data Paths in Digital Document Printers*

Venkatesh Kannan, Wil M.P. van der Aalst, and Marc Voorhoeve

Department of Mathematics and Computer Science,
Eindhoven University of Technology, Eindhoven, The Netherlands.
{V.Kannan,W.M.P.V.D.Aalst,M.Voorhoeve}@tue.nl

Abstract. This paper reports on a challenging case study conducted in the context of the Octopus project where CPN Tools is used to model and analyze the embedded system of digital document printer. Modeling the dynamic behavior of such systems in a predictable way is a major challenge. In this paper, we present the approach where colored Petri nets are used to model the system. Simulation is used to analyze the behavior and performance. The challenge in modeling is to create building blocks that enable flexibility in reconfiguration of architecture and design space exploration. CPN Tools and ProM (a process mining tool) are used to collect and analyze the simulation results. We present the pros and cons of both the conventional presentation of simulation results and using ProM. Using ProM it is possible to monitor the simulation is a refined and flexible manner. Moreover, the same tool can be used to monitor the real system and the simulated system making comparison easier.

1 Introduction

The Octopus project is a co-operation between Océ Technologies, the Embedded Systems Institute (ESI), and several research groups in the Netherlands. The aim of the project is to define new methods and tools to model and design embedded systems like printers, which interact in an adaptive way to changes during their functioning. One of the branches of the Octopus project is the study of design of data paths in printers and copiers. A data path encompasses the trajectory of image data from the source (for instance the network to which a printer is connected) to the target (the imaging unit). Runtime changes in the environment may require use of different algorithms in the data path, deadlines for completion of processing may change, new jobs arrive randomly, and availability of resources also changes. To realize such dynamic behavior in a predictable way is a major challenge. The Octopus project is exploring different approaches to model and analyze such systems. This paper focuses on the use of colored Petri nets to model and study such systems. We report on the first phase of the project, in which we studied a slightly simplified version of an existing state-of-the-art image processing pipeline at Océ implemented as an embedded system.

* Research carried out in the context of the Octopus project, with partial support of the Netherlands Ministry of Economic Affairs under the Senter TS program.

1.1 The Case Study

The industrial partner in the Octopus project, Océ Technologies, is a designer and manufacturer of systems that perform a variety of image processing functions on digital documents in addition to scanning, copying and printing. In addition to locally using the system for scanning and copying, users can also remotely use the system for image processing and printing. A generic architecture of an Océ system used in this project is shown in Figure 1. [2]

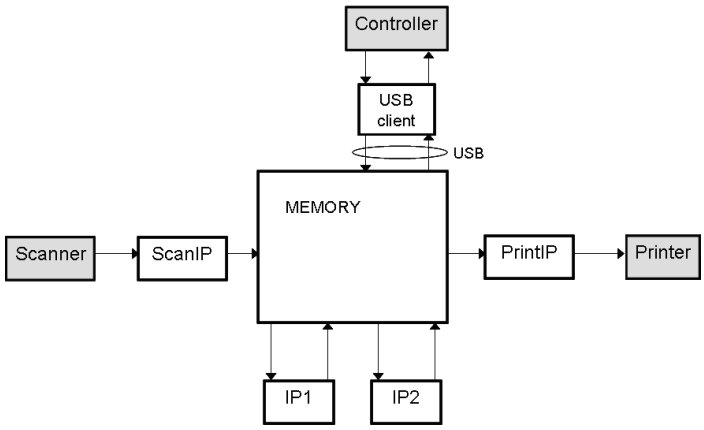


Fig. 1: Architecture of Océ system.

As shown in Figure 1, the system has two input ports: Scanner and Controller. Users locally come to the system to submit jobs at the Scanner and remote jobs enter the system via the Controller. These jobs use the image processing (IP) components (ScanIP, IP1, IP2, PrintIP), system resources such as the memory, and USB bandwidth for the executing the jobs. Finally, there are two output ports where the jobs leave the system: Printer and Controller. Jobs that require printed outputs use the Printer and those that are to be stored in a storage device or sent to a remote user are sent via the Controller.

All the components mentioned above (Scanner, ScanIP, IP1, IP2, PrintIP) can be used in different combinations depending on how a document of a certain job is requested to be processed by the user. Hence this gives rise to different *use-cases* of the system i.e. each job could use the system in a different way. The list of components used by a job defines the *data path* for that job. Some possible data paths for jobs are listed and explained below:

- **DirectCopy**: Scanner $\rightsquigarrow$ ScanIP $\rightsquigarrow$ IP1 $\rightsquigarrow$ IP2 $\rightsquigarrow$ USBClient, PrintIP
- **ScanToStore**: Scanner $\rightsquigarrow$ ScanIP $\rightsquigarrow$ IP1 $\rightsquigarrow$ USBClient
- **ScanToEmail**: Scanner $\rightsquigarrow$ ScanIP $\rightsquigarrow$ IP1 $\rightsquigarrow$ IP2 $\rightsquigarrow$ USBClient
- **ProcessFromStore**: USBClient $\rightsquigarrow$ IP1 $\rightsquigarrow$ IP2 $\rightsquigarrow$ USBClient
- **SimplePrint**: USBClient $\rightsquigarrow$ PrintIP
- **PrintWithProcessing**: USBClient $\rightsquigarrow$ IP2 $\rightsquigarrow$ PrintIP

The data path listed for *DirectCopy* means that the job is processed in order by the components Scanner, ScanIP, IP1, IP2 and then simultaneously sent to

the Controller via the USBClient and also for printing through PrintIP. In the case of the *ProcessFromStore* data path, a job is remotely sent via the Controller and USBClient for processing by IP1 and IP2 after which the result is sent back to the remote user via the USBClient and the Controller. The interpretation for the rest of the data paths is similar.

Furthermore, there are additional constraints possible on the dependency of the processing of a job by different components in the data path. It is not mandatory that the components in the data path should process the job sequentially, as the design of the Océ system allows for a certain degree of parallelism. Some instances of this are shown in Figure 2.

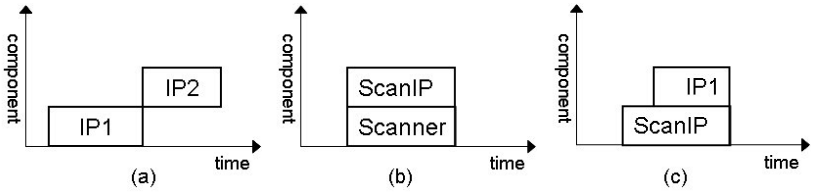


Fig. 2: Dependency between components processing a job.

According to the Océ system design, IP2 can start processing a page in a document only after IP1 has completed processing that page. This is due to the nature of the image processing function that IP1 performs. Hence as shown in Figure 2(a) IP1 and IP2 process a page in a document in sequence. Considering Scanner and ScanIP, they can process a page in parallel as shown in Figure 2(b). This is because ScanIP works full streaming and has the same throughput as the Scanner. The dependency between ScanIP and IP1 is shown in Figure 2(c) and in this case IP1 works streaming and has a higher throughput than ScanIP. Hence IP1 can start processing the page as ScanIP is processing it, with a certain delay due to the higher throughput of IP1.

In addition to using the different components of the system for executing jobs, there are other system resources that are needed to process jobs. The two key system resources addressed currently in this project are the memory and the USB bandwidth. Regarding the memory, a job is allowed to enter the system only if the entire memory required for completion of the job is available before its execution commences. If the memory is available, then it is allocated and the job is available for execution. Each component requires a certain amount of memory for its processing and releases this memory once it completes processing. Hence utilization of memory is a critical factor in determining the throughput and efficiency of the system. Another critical resource is the USB. The USB has a limited bandwidth and it serves as the bridge between the USBClient and the memory. Whenever the USBClient writes/reads data to/from the memory, it has to be transmitted via the available USB. Since this bandwidth is limited, it can

be allocated only to a limited number of jobs at a time. This determines how fast the jobs can be transferred from the memory to the Controller or vice versa.

The overview of the system just given illustrates the complexity of the Oc system. The characteristics of critical system resources such as memory and USB bandwidth, and the components determine the overall performance. Moreover, resource conflicts need to be resolved to ensure a high performance and throughput. The resource conflicts include competition for system components, memory availability, and USB bandwidth.

1.2 The Approach

In our approach, colored Petri nets (CPN) are used to model the Oc system. The CPN modeling strategy [3] is aimed at providing flexibility for design space exploration of the system using the model. Hence, design of reusable building blocks is vital during the modeling process. Simulation of the model is used for performance analysis to identify bottleneck resources, utilization of components, decisions during design space exploration and design of heuristic scheduling rules (in the future). *CPN Tools* is used for modeling, simulation and performance analysis of the system. Additionally, *ProM*, a versatile process mining tool, is used to provide further insights into the simulation results and also present these results to the domain user in different forms. Interestingly, ProM can be used to monitor both the simulated and the real system, thus facilitating easy comparison.

2 Modeling Using CPN

The modeling approach takes an architecture oriented perspective to model the Océ system. The model, in addition to the system characteristics, includes the scheduling rules (First Come First Served) and is used to study the performance of the system through simulation. Each component in the system is modeled as a subnet. Since the processing time for all the components, except the USB, can be calculated before they start processing a job, the subnet for these components looks like the one shown in Figure 3. The transitions *start* and *end* model the beginning and completion of processing a job, while the places *free* and *do* reflect the occupancy of the component. In addition, there are two places that characterize the subnet to each component: *compInfo* and *paperInfo*. The place *compInfo* contains a token with information about the component, namely the component ID, processing speed and the recovery time required by the component before starting the next job. The place *paperInfo* contains information on the number of bytes the particular component processes for a specific paper size. The values of the tokens at places *compInfo* and *paperInfo* remain constant after initialization and govern the behavior of the component. Since the behavior of the USB is different from the other components, its model is different from the other components and is shown separately. The color sets for *paperInfo* and *compInfo* used in the CPN Tools model are listed below.

```
colset PAPERINFO=record paper:STRING*inputSize:INT;
colset COMPINFO=record compID:STRING*speed:INT*recovery:INT;
```

In the color set *PAPERINFO*, the record-element *paper* contains the information on the size of the paper, such as A4 or A3, and element *inputSize* denotes the memory required for this size of paper. In the color set *COMPINFO*, the element *compID* is used to name the component (scanner, scanIP, etc.), speed denotes the processing *speed* of the component and *recovery* contains the information about the recovery time needed by the component between processing two jobs.

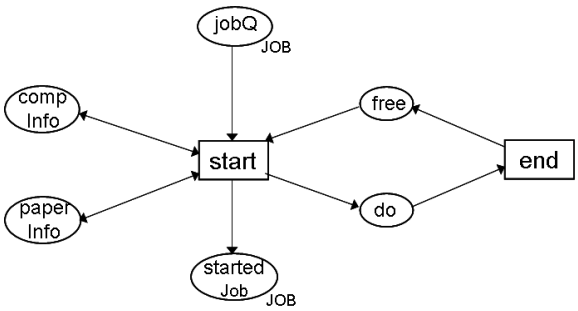


Fig. 3: Hierarchical subnet for each component

In Figure 3, the place *jobQ* contains tokens for the jobs that are available for the components to process at any instance of time. The color of a token of type *Job* contains information about the job ID, the use case and paper size of the job. Hence, the component can calculate the time required to process this job from the information available in the *Job* token, and the tokens at the places *compInfo* and *paperInfo*. Once the processing is completed, transition *end* places a token in place *free* with a certain delay, governed by the recovery time specific to each component, thus determining when the component can begin processing the next available job. The color set for the type *Job* is as follows,

```
colset JOB=record
jobID:STRING*
jobType:STRING*
inputPaper:STRING*
from:STRING*
to:STRING*
startTime:INT*
endTime:INT timed;
```

The record element *jobID* is used to store a unique identifier for each job, *jobType* contains the use-case of the job (DirectCopy or ScanToEmail, etc.), and

the element *inputPaper* specifies what paper size is used in this job. The elements *from* and *to* are used for the source and destination component IDs respectively, as the job is being processed by one component after another according to the data path. The *startTime* and *endTime* are used by each component to contain the timestamps of start and estimated end of processing the job.

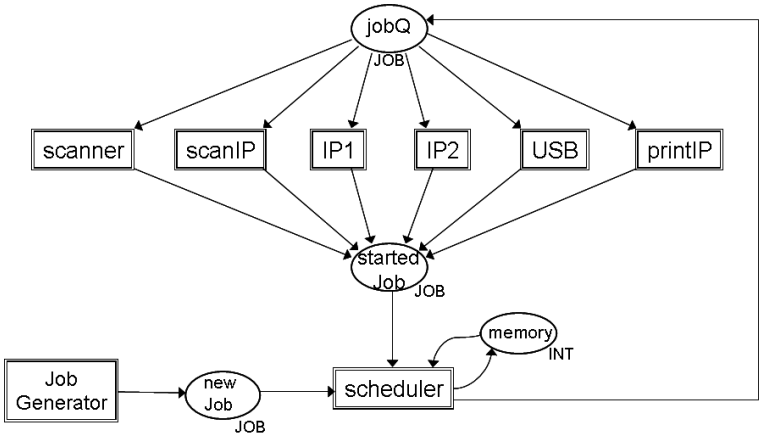


Fig. 4: Architectural view of the CPN model.

Figure 4 shows an abstract view of the model. New jobs for the system can be created using the *Job Generator* subnet, which are placed as input to the *Scheduler* subnet at the place *newJob*. The *Scheduler* subnet is the heart of the system that models the concepts including the scheduling rules, memory management rules and routing each job step-by-step from one component to the next depending on the data path of the use-case to which the job belongs. From this it can be observed that the scheduling rules are modeled as being global to system and not local to any of the components or distributed.

Vital to the Scheduler's task of routing jobs from one component to the next is the information about the use-cases and the data paths. From the information on data paths in Section 1.1, it can be inferred that each data path is a partial order. Hence, a list of list (of color STRING) is used to represent the partial order. An example of a data path represented in the *Scheduler* component is shown here.

```
ucID="DirectCopy",
dataPath= [ ["scanner","scanIP"],
["scanIP","IP1"],
["IP1","IP2"],
["IP2","printIP","USBup"],
["USBup"],["printIP"]
]
```

The data path of the use-case *DirectCopy* is explained in Section 1.1. In this example, each sublist inside the data path list contains two parts: the first element being the source component and the remaining being the destination(s). Hence, `["scanIP","IP1"]` indicates that in the *DirectCopy* use-case, a job processed by *scanIP* will be processed by *IP1* next. Similarly, `["IP2","printIP","USBup"]` denotes that a job processed by *IP2* will be processed simultaneously by *printIP* and *USBupload* in the next step.

The *Scheduler* picks a new job that enters the system from the place *newJob* and estimates the amount of total memory required for executing this job. If enough memory is available, the memory is allocated (the memory resource is modeled as an integer token in the place memory) and the job is scheduled for the first component in the data path of this job by placing a token of type Job in the place *jobQ*, which will be consumed by the corresponding component for processing. When a component starts processing a job, it immediately places a token in the *startedJob* place indicating this event. The *Scheduler* consumes this token to schedule the job to the next component in its data path, adding a delay that depends on the component that just started, the next component in the data path, and the dependency explained and shown in Figure 2 (a), (b) and (c). Thus the logic in the *Scheduler* includes scheduling new jobs entering the system (from place *newJob*) and routing the existing jobs through the components according to the corresponding data paths.

As mentioned above, the *Scheduler* subnet also handles the memory management. This includes memory allocation and release for jobs that are executed. When a new job enters the system, the *Scheduler* schedules it only if the complete memory required for the job is available (checked against the token in the place memory). During execution, part of the memory allocated may be released when a component completes processing a job. This memory release operation is also performed by the *Scheduler* subnet.

Modeling the USB component is different from the other components and cannot be models using the "pattern" shown in Figure 5. As described earlier, for the USB, the time required to transmit a job (upstream or downstream) is not constant and is governed by other jobs that might be transmitted at the same time. This necessitates making the real-time behavior of the USB bus dependent of multiple jobs at the same time. It is to be noted that if only one job is being transmitted over the USB then a *high* MBps transmission rate is used, and when more than one job is being transmitted at the same time then a lower *low* MBps transmission rate is used.

The model of the USB as shown in Figure 5 works primarily by monitoring two events observable in the USB when one or more jobs are being transmitted: (1) the event of a new job joining the transmission, and (2) the event of completion of transmission of a job. Both these events govern the transmission rates for the other jobs on the USB and hence determine the transmission times

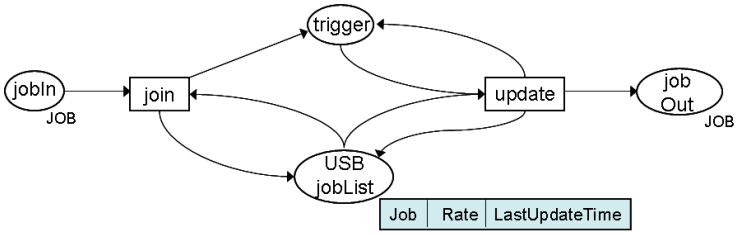


Fig. 5: CPN model for the USB.

for the jobs. In the model shown in Figure 5, there are two transitions *join* and *update*, and two places *trigger* and *USBjobList*. The place *USBjobList* contains the list of jobs that are currently being transmitted over the USB. Apart from containing information about each job, it also contains the transmission rate assigned, the number of bytes remaining to be transmitted and the last time of update for each job. Transition *join* adds a new job waiting at place in that requests use of the USB (if it can be accommodated) to the *USBjobList*, and places a token in place *trigger*. This enables transition *update* that checks the list of jobs at place *USBjobList* and reassigns the transmission rates for all the jobs according to the number of jobs transmitted over the USB. The *update* transition also recalculates the number of bytes remaining to be transmitted for each job since the last update time, estimates the job that will finish next and places a timed token at *trigger*, so that the transition *update* can remove the jobs whose transmissions have completed. The jobs whose transmission over the USB is complete are placed in place *out*. Thus transition *join* catches the event of new jobs joining the USB and the transition *update* catches the event of jobs leaving the USB, which are critical in determining the transmission time for a single job.

3 Simulation and Analysis

This section presents some analysis methods used to study the results from the simulation of the model. Section 3.1 presents the information collected in CPN Tools through monitors and how it is used to measure relevant performance metrics. Section 3.2 presents the use of the process mining tool *ProM* for an alternative presentation and analysis of the simulation results. ProM uses event logs, which are recorded by CPN Tools. The event log contains details about the events (i.e., transition firings) that take place in the simulation.

We are unable to share detailed data about the Océ system because this information is highly confidential. Hence, the actual parameters and simulation results should be seen as potential settings and outcomes.

For the simulation experiment to illustrate possible results obtained by CPN Tools and ProM, 150 jobs are generated by the *Job Generator* component of the model in Figure 4 in each run. These jobs are created by picking a random number of jobs from the six use-cases listed in Section 1.1. The arrival times of jobs are distributed negative exponentially with an inter-arrival time of 2 seconds.

3.1 Simulation Results

When performing simulation in CPN Tools, the different categories of monitors available can be used to collect the simulation results in different ways [1]. Here, two examples of how different types of monitors are used to aggregate the simulation results to performance analysis metrics are presented.

Table 1 presents the statistics produced by the data collection monitor that was used to aggregate the waiting times of jobs before their execution starts at each component. The averages provided by CPN Tools in the performance report can be obtained by replicating the simulation for multiple runs. The waiting times of jobs thus obtained through monitors during simulations can be used to identify the components that are probable bottleneck resources in the system. Similarly, using the data collection monitor, the utilization times for each component can be obtained to determine the under- and over-utilized components in the system.

Name	**Avrg**	**90% Half Length**	**95% Half Length**	**99% Half Length**
		IP1		
count_iid	100.119400	0.134347	0.160568	0.212527
max_iid	3007.696600	4.862893	5.812036	7.692745
min_iid	0.000000	0.000000	0.000000	0.000000
avrg_iid	34.302562	1.301284	1.555269	2.058537
		IP2		
count_iid	100.048200	0.133754	0.159861	0.211590
max_iid	2860.038400	37.247604	44.517618	58.923016
min_iid	0.000000	0.000000	0.000000	0.000000
avrg_iid	48.990676	0.935130	1.117649	1.479308
		USB		
count_iid	174.983400	0.105168	0.125695	0.166368
max_iid	242724.770400	535.206794	639.668843	846.658458
min_iid	0.000000	0.000000	0.000000	0.000000
avrg_iid	23679.481434	143.889599	171.974075	227.622944
		printIP		
count_iid	74.900800	0.144126	0.172257	0.227998
max_iid	96590.504600	524.005807	626.281639	828.939306
min_iid	0.000000	0.000000	0.000000	0.000000
avrg_iid	13155.451373	126.373949	151.039708	199.914452
		scanner		
count_iid	75.136000	0.141720	0.169381	0.224191
max_iid	735681.475800	532.367990	636.275959	842.167675
min_iid	5406.491400	866.457382	1035.573160	1370.672942
avrg_iid	341606.033984	696.226511	832.116504	1101.380010

Table 1: Waiting times of jobs at the different components

From Table 1, it can be observed that the average waiting time for jobs in front of components *Scanner* and *USB* is higher than for the rest of the components. For example, with 90confidence, the *USB* is seen to have an average waiting time of 23680 seconds, with a half length of 144 seconds, for jobs in the queue in front of it. This is attributed to the scheduling rule that jobs have to wait for memory allocation before entering the system for processing through the *Scanner* or the *USBdown*. The simulation experiment here was conducted with minimal memory availability, and hence the longer queues. Also, the average waiting time in front of the *printIP* is also higher as it is the slowest component in the system according to the design specifications.

The second example presented here uses the write-in-file monitor to log the events when memory is allocated or released by the *Scheduler* component. Using this log of the time stamps and the amount of memory available, a simple tool can be used to plot the chart shown in Figure 6. The chart depicts the amount of memory available in the system at each instant of time. Information about the utilization characteristics of the memory resource is a key input in designing the memory architecture, designing scheduling rules for optimal memory utilization with high system throughput and analyzing the waiting times in front of each component in the system.

Fig. 6: Memory Utilization chart

The above simulation results are typical for simulation tools, i.e., like most tools, CPN Tools focuses on measuring key performance indicators such as utilization, throughput times, service levels, etc. Note that the BRITNeY Suite animation tool [5] can be used to add animations to CPN simulations. Moreover, it allows for dedicated interactive simulations. This facilitates the interaction with end users and domain experts (i.e., non-IT specialists).

3.2 Using ProM

ProM is a process mining tool, i.e., it is used to investigate real-life processes by analyzing footprints of processes in the form of event logs, audit trails, database

entries, message exchanges, translation logs, etc. ProM offers a wide variety of analysis techniques. Because simulation can be seen as imitating real-life, it is interesting to see what additional insights process mining techniques can provide. This section presents some of the plug-ins of ProM that have been explored in the context of Océ's systems. The plug-ins of ProM use event logs, which is list of events recording when each component starts and completes processing a job. These event logs have been generated using the approach described in [6].

Fuzzy Miner The fuzzy miner plug-in along with the animation part of it provides a visualization of the simulation. The event log is used to replay the simulation experiment on the fuzzy model of the system. Figure 7 shows a snapshot during the animation. During the animation, jobs flow between components in the fuzzy model in accordance with the events during simulation. It provides a view of the queues in front of each component, which serves as an easy means to identify key components, bottleneck resources and the utilization of components in the system. For example, from Figure 7 it can be observed that during this simulation run, the queue in front of *printIP* was longer, which can be attributed to it being the slowest component in the system. More importantly, the fuzzy miner animation provides live insight into the simulation run and is an easier means of communication with the domain users, which is significant in the context of the Octopus project.

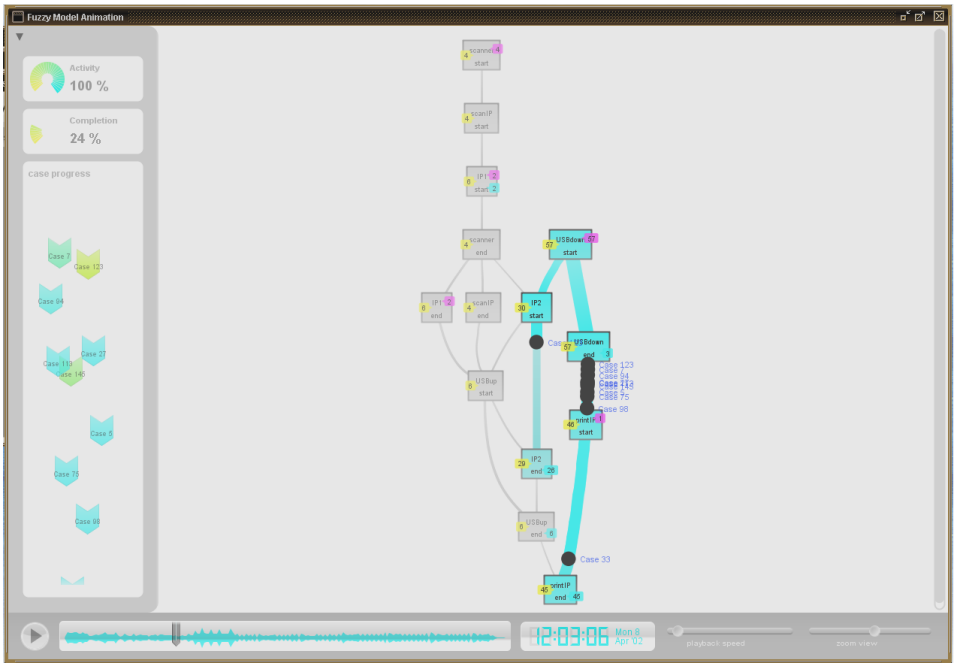


Fig. 7: Fuzzy Miner Animation

Dotted Chart Analysis This plug-in uses the event log to create a dotted chart with each dot referring to an event in the log. The chart can be viewed using different perspectives. The x-axis always shows the time (can be absolute or relative) and the y-axis shows a particular perspective. If the "instance perspective" is selected, then each job is represented by a horizontal dotted line showing the events that took place for this job. If the "originator perspective" is selected, each use-case is represented by a horizontal dotted line. Figure 8 shows the dotted chart from the "task perspective" (i.e., the components in the system). Hence, each pair of dots represents the start and end of processing a job by that component. The plug-in can provide an overview of the dynamics of the execution of jobs and also the system load.

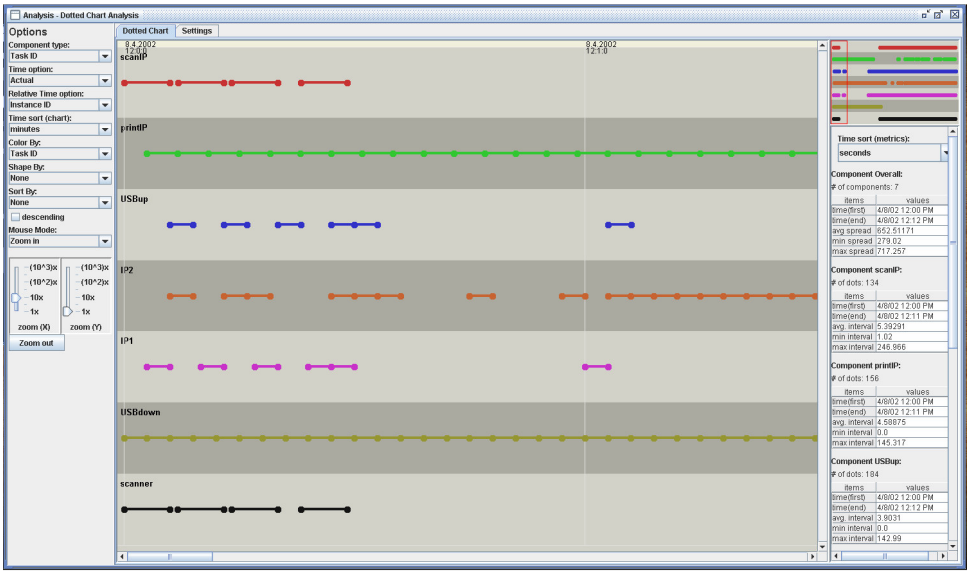


Fig. 8: Dotted Chart Analysis

For instance, the distribution of the dots along the timeline for each component gives an insight into the utilization characteristics of the component, which helps to identify the under- and overutilized components. For example, from this chart, it was observed that IP2 is a component with high utilization rate throughout this simulation experiment. Also, the dotted chart provides details about the distribution of the types of jobs (use-cases) over the simulation. In this case, it can be observed from Figure 8 that the remote jobs (use-cases that originate at the USBdown) are generated in a burst at the start of the simulation, whereas the number of local jobs submitted at the scanner is fewer during the same interval. Thus this chart gives detailed insight into the steps of simulation and hence can provide input for a better design of the simulation environment setup.

Performance Sequence Diagram Analysis The performance sequence diagram plug-in provides a means to assess the performance of the system. The plug-in can provide information about behaviors that are common from the event log. These patterns can be visualized from different perspectives such as the components of the system and the data paths in the system. Figure 9 shows a screenshot of the pattern diagram generated from the view of the components. In this context, the patterns depicted correspond to the different data paths listed in Section 1.1. Also, statistics about the throughput times for each pattern are presented, which can be used to determine the patterns that seem to be common behavior, those that are rare and those that result in high throughput times.

On the other hand, this plug-in can be used to analyze an event log from the Océ system to identify the data paths available thus assisting in identifying the architecture and behavior of the system and also in the modeling process.

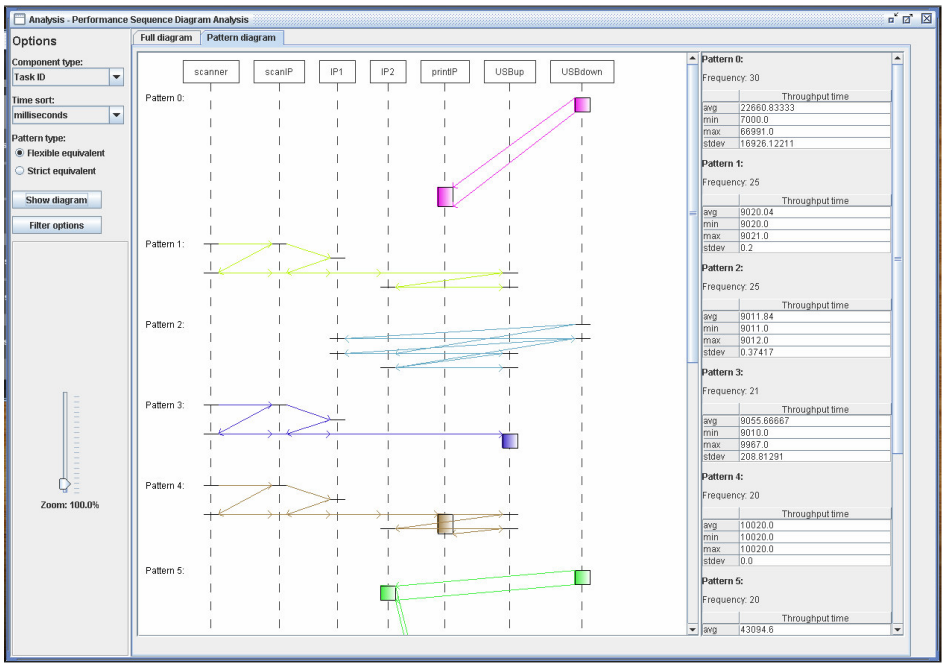


Fig. 9: Pattern diagram - Performance Sequence Diagram Analysis

Trace Clustering Figure 9 shows the frequent patterns in the event log as sequence diagrams. In the context of process and data mining many clustering algorithms are available. ProM supports various types of trace clustering. In Figure 10 the result of applying the K-means clustering algorithm with a particular distance metric is shown, where six clusters are identified. These correspond to

the different usecases or datapaths. For each cluster, the corresponding process model can be derived. Figure 10 shows two Petri nets. These nets have been discovered by applying the alpha algorithm [7] to two of the cluster. These diagrams nicely show how the dependencies depicted in Figure 2 can be discovered. For this particular setting of the clustering algorithm, the basic use-cases are discovered. However, other types of clustering and distance metrics can be used to provide insights into the different data-paths.

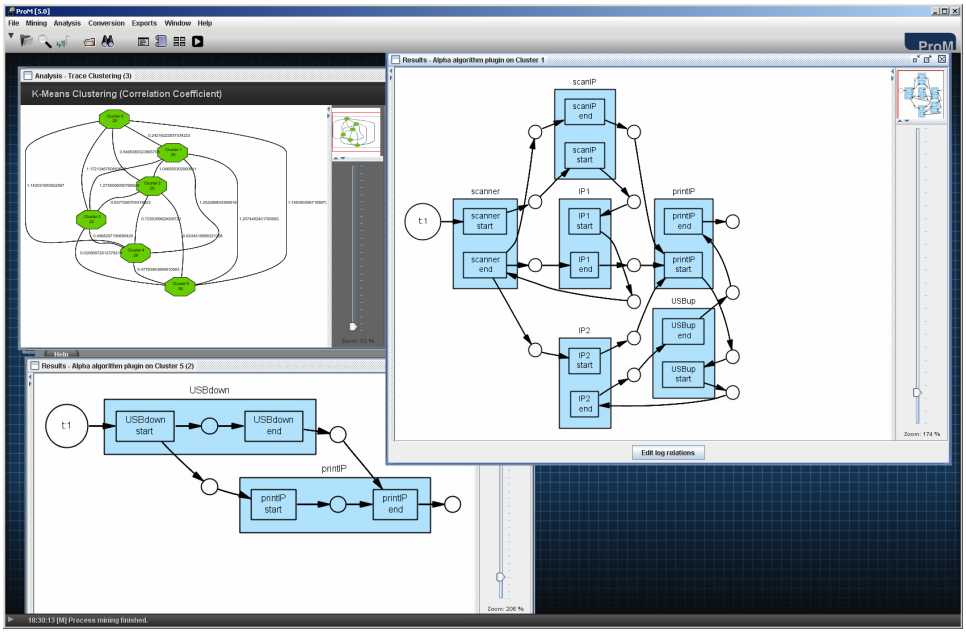


Fig. 10: Using Trace Clustering the Different Use Cases can be Identified and the Corresponding Detailed Process Models can be Discovered

Performance Analysis Figure 11 shows a detailed performance analysis of one of the use-cases using the Performance Analysis with Petri net plug-in. The focus of the plug-in is to provide key performance indicators, which can be summoned in an intuitive way. For this, the event logs of the selected cluster are replayed in the Petri net model of the use-case generated using the alpha algorithm. From this simulation of a single use-case, performance indicators including average throughput time, minimum and maximum values, and standard deviation for the use-case throughput are derived. These provide a detailed insight into parts of the system during the simulation experiment, in this case the six use-cases of the system.

Additionally, the color of the places in the Petri net indicates where in the process (datapath in this case) the jobs of this use-case spend most time. For example, we can observe and verify, based on the prior system knowledge, that since the PrintIP is the slowest component, jobs spend most time waiting in its queue.

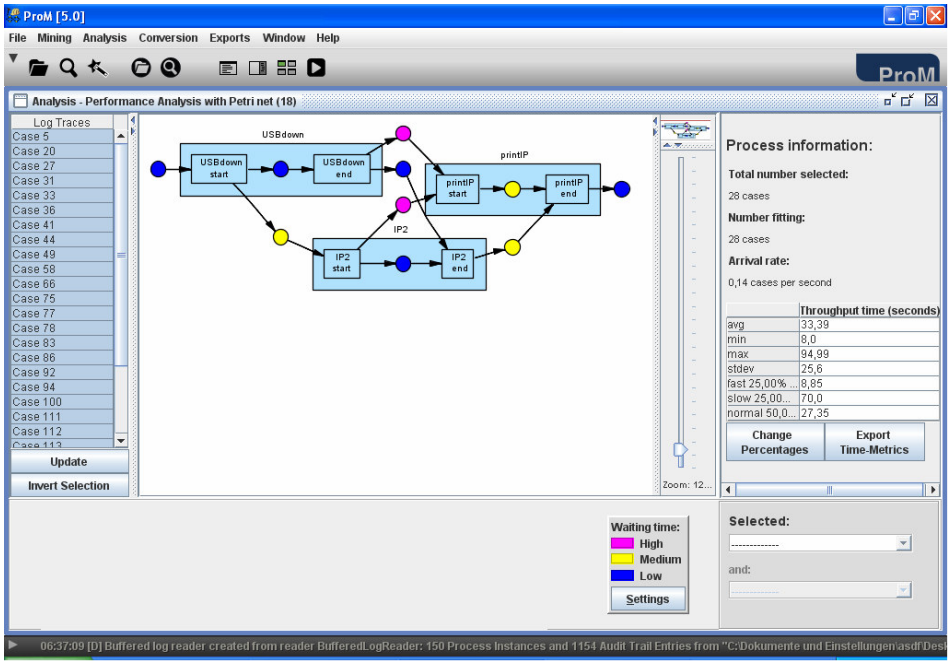


Fig. 11: A Detailed Performance Analysis Is Performed For One of the Clusters Discovered

Social Network Analysis Figure 12 shows the result of using Social Network Analysis (SNA) on the event log. This plug-in is typically used to quantify and analyze social interaction between people in business process environment. However, by mapping the roles played by people to components in this context, the analysis provides information about interaction statistics among the components.

The analysis plug-in uses the SNA matrix generated by the social network miner plug-in, which uses the data on causal dependency in hand over of work among components, derived from the event log. As a result it is possible to show the flow of work between the various components. The shape and size of the nodes give a direct indication of the utilization of the component. The height of the node is directly proportional to the amount of work flowing into the component and the width to the amount flowing out. The arc weights are an

indicator of the amount of work flowing between the components. This provides a quantification to analyze the interaction among the components.

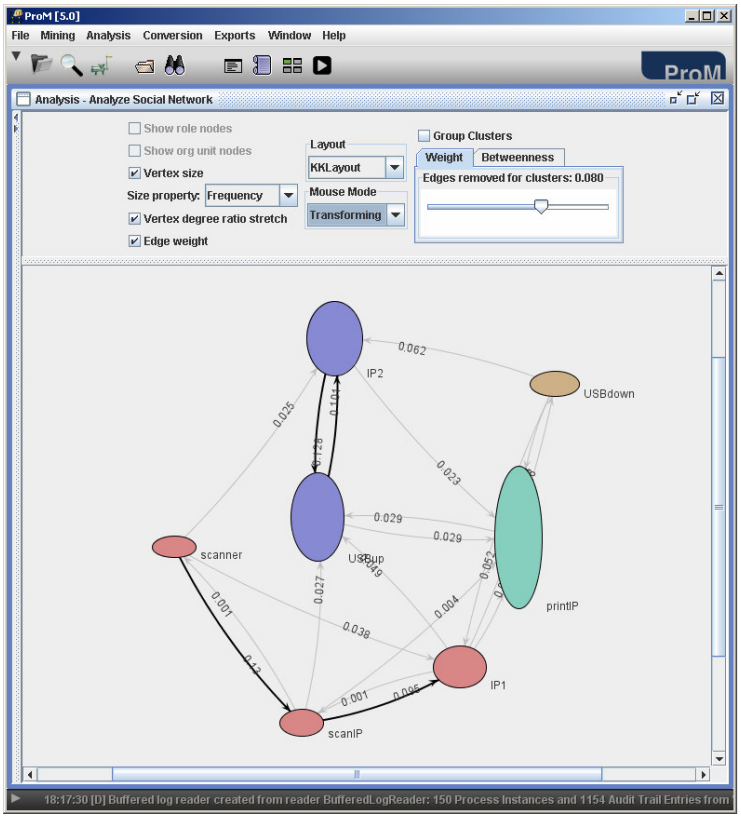


Fig. 12: Social Network Analysis Applied to the Components of Océ's System

3.3 Comparison and Discussion

Section 3.1 showed the classical simulation results obtained from monitors in CPN Tools. Parameters such as waiting times of jobs and utilization rates help in identifying the critical resources and to study the system performance and behavior. The averages and standard deviations of such parameters are helpful in analyzing the overall performance of the system over the entire simulation. However, such classical simulation results typically do not present the dynamics and detailed behavior of the system during the simulation.

On the other hand, Section 3.2 looks into some of the plug-ins available in the process mining tool ProM and illustrates their application to event logs of a CPN simulation. They provide the advantage to observe the dynamics and details of the system behavior and performance during the simulation experiment.

For instance, the fuzzy miner and the dotted chart plug-ins can show views of utilization characteristics of components in the system from different perspectives. Also, the performance sequence diagram analysis presents patterns and their statistics (such as throughput times) helping in studying their occurrences and impact on the system performance. Clustering techniques can be used to group jobs and analyze each group individually. Hence, even though the classical simulation results provide an overall view of the system performance and characteristics, ProM provides some added advantages in presenting the detailed view of the simulation process with insights into the dynamics of the system's behavior and simulation.

Another important observation is that process mining tools ProM can be used to observe and analyze real-world process and simulated processes. Currently, system analysts tend to use different tools for monitoring real systems and simulated systems. This is odd, since often the ultimate goal is to compare the real system with the simulated system. (Recall that simulation is used to understand and improve real systems!)

4 Related Work

The use of CPN Tools as a simulation tool is described in [1]. In this paper, the monitor concept is described in detail. The BRITNeY Suite animation tool [5] extends the visualization and interaction functionality of CPN Tools. The animation tool can be connected to the running simulation engine and used to present the simulated behavior in a domain specific and interactive manner. ProM is described in [8]. The current release of ProM contains more than 230 plug-ins. In the paper, we could only show a few and we refer to www.processmining.org for details.

In [2] we modeled the basic components of Océ's copiers using different formalisms. In [9] the authors present the modeling of the features of a mobile phone. The work also involves identification and analysis of the interaction among features, helping in identifying faults in specification and improvement of the architecture. In [10] the practical use of colored Petri nets is demonstrated by an industrial case study involving a flowmeter instrument that consists of hardware components performing estimation and calculation functions by interacting with each other.

5 Conclusions and Future Work

In this paper, initial experiences with using colored Petri nets in Octopus project have been presented. Petri nets allow for modeling all the details and dynamics of the embedded system used in this case study. This permits providing practical inputs and solutions to real-world problems. A slightly simplified version of a currently existing Océ system was used as the case study. In the modeling process the goal was to identify building blocks to allow re-use of components in the model. Also modeling the dynamic behavior of the USB is a significant solution

to future problems such as modeling memory bus and processors. CPN Tools and ProM prove to be effective tools in analyzing and studying the performance of the system. They provided insights into identifying the bottleneck resources, utilization of resources and system dynamics during execution of jobs. The pros and cons of the classical presentation of simulation results and the application of ProM in analyzing the results are also studied.

From the modeling perspective, the next steps are to model the complete copier systems at Océ, as opposed to the slightly simplified case studied here. Hence, it is essential to identify patterns and design re-usable building blocks in the CPN model. This will allow flexibility in exploring different system architectures and design decisions through the model. In addition, the analysis of simulation results using CPN Tools and ProM will be used to further explore the design space and build heuristic scheduling rules in the next steps of the project. We feel that it is important to use the same tools to monitor and analyze the real system and its simulated counterparts. This will allow for a better comparison and a more detailed analysis as shown in this paper.

References

1. K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for modeling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT).*, Volume 9, Numbers 3-4, June 2007.
2. G. Igna, V. Kannan, Y. Yang, T. Basten, M. Geilen, F. Vandraager, M. Voorhoeve, S. de Smet, and L. Somers. Formal Modeling and Scheduling of Data Paths of Digital Document Printers. *6th International Conference FORMATS 2008*, Proceesings, September 15-17 2008.
3. K. Jensen. Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. *EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, 1992.
4. W.M.P. van der Aalst, J. Nakatumba, A. Rozinat, and N. Russell. Business Process Simulation: How to get it right? *BPM-08-07, Eindhoven*, BPMcenter.org, 25pp.
5. M. Westergaard, K.B. Lassen. The BRITNeY Suite Animation Tool. *Proceedings of the 27th International Conference on Application Theory of Petri Nets and Other Models of Concurrency (ICATPN 2006)*, Lecture Notes in Computer Science 4024, Springer, pages 431-440, 2006.
6. A.K. Alves de Medeiros, and C.W. Günther. Scheduling with timed automata. *Theor. Comput. Sci.*, 354(2):272–300, 2006.
7. W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128-1142, 2004.
8. W.M.P. van der Aalst, B.F. van Dongen, C.W. Gnther, R.S. Mans, A.K. Alves de Medeiros, A. Rozinat, V. Rubin, M. Song, H.M.W. Verbeek, and A.J.M.M. Weijters. ProM 4.0: Comprehensive Support for Real Process Analysis. *In J. Kleijn and A. Yakovlev, editors, Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2007)*, volume 4546 of Lecture Notes in Computer Science, pages 484-494. Springer-Verlag, Berlin, 2007.
9. L. Lorenstsen, A.-P. Touvinene, J. Xu. Modelling Feature Interaction Patterns in Nokia Mobile Phones using Coloured Petri Nets and Design/CPN. *In K. Jensen*

(ed.) Proceedings of the Third Workshop and Tutorial on Practical Use of Coloured Petri Nets and CPN Tools, 2001.

10. L. Lorentsen. Modelling and Analysis of a Flowmeter System. *Proceedings of Workshop and Tutorial on Practical Use of Coloured Petri Nets and Design/CPN*, 1999.

Application of Coloured Petri Net for Agent Control and Communication in the ABAsim Architecture

Antonín Kavička[1] and Michal Žarnay[2]

[1] Faculty of Electrical Engineering and Informatics, University of Pardubice,
Nám. Čs.legií 565, CZ-532 10 Pardubice, The Czech Republic
Antonin.Kavicka@upce.cz

[2] Faculty of Management Science and Informatics, University of Žilina,
Univerzitná 8215/1, SK-01026 Žilina, The Slovak Republic
Michal.Zarnay@fri.uniza.sk

Abstract. Petri nets represent a convenient formalism for description of the operational logic of internal agent components within agent-based architectures of simulation models using message-oriented communication paradigm. The approach supports higher flexibility of simulation models as well as formal analysis related to relevant parts of the models. The ABAsim architecture (as an example) already utilizes place/transition Petri nets for description of internal components of agents. Presented modified approach pays attention to an application of non-hierarchical coloured Petri nets (describing behavioural rules of autonomous agents) instead of place/transition ones because of higher modelling capabilities.

Keywords: Coloured Petri net, agent-based simulation, message-oriented communication.

1 Introduction

During the last decade a significant progress was achieved in the area of micro-simulation models reflecting transportation logistic systems [1, 2]. The models are *flexible*, which means that they are namely: (i) composed of reusable components and sub-models, (ii) configurable on a conceptual level and (iii) ready for changes of granularity related to sub-models. A proprietary agent-based architecture (called *ABAsim A*gent-*B*ased *A*rchitecture of *sim*ulation models) has been developed and it has been successfully applied within many simulation studies reflecting transportation and logistic systems (an example is represented in [4]). The architecture utilizes simulation model decomposition into individual agents (concentrated on distinct tasks), which are organized within hierarchical structure and communicate by means of sending messages. Each agent is composed of internal components (potentially using internal communication), whereas their logic can be described either

imperatively (with the help of programme routines) or declaratively (e.g. applying formalism of Petri nets) – the respective specification is activated by means of system interpreter during the run of simulation programme.

Up to now, the ABAsim architecture has utilized a subclass of *place*/*transition Petri net* (P/T PN), called *ABA-graph* [3] for description of control agent components denoted as managers. Presently is the attention paid to an application of a subclass of coloured Petri net, named *ABA-CPN*, which brings more flexible approach in defining the component descriptions. It includes, for example, easier and more natural construction of conditional branching and differentiation of various message instances.

The paper is organised as follows: sections two and three shortly describe background of the paper: autonomous agents and ABAsim architecture. Section four explains the ABA-CPN definition in detail, section five provides illustratory example for it and section six discusses conventions for notation of elements in the ABA-CPN followed by conclusion of the paper.

Since it has been too challenging to combine all goals in one example, while keeping its size in reasonable limits, there are two examples. The first accompanies explanations of the definition – it includes all properties from definition, it uses only symbolic names and it has no connection to examples from real world. The second example is built on a simple real case from an ABAsim simulation model of service system, including meaningful labels – it illustrates usage of notation conventions.

2 Paradigm of autonomous agents

Let us remind the generally-respected agent definition [7]: *Agent is an encapsulated computer system situated in some environment and capable of flexible, autonomous action in that environment in order to meet its design objectives*, where the agent features are as follows:

- *Autonomy* – i.e. the agent is able to work autonomously without exogenous interventions, entirely able to control its activities and inner status.
- *Social behaviour* – is based on the agent's interaction with other agents (or with human beings) by means of some communication mechanism or language.
- *Re-activeness* – the agent responds to external influences from its surrounding.
- *Pro-activeness* – the agent acts with initiative and goal-orientation.

The agent realizes, according to its mission, its own life-cycle: *sensing – decision making – acting* (within its life space) using the support of *solving* (focused on making solution proposals) and *communicating* with other agents (eventually with human operators). If the agent detects a problem or a situation beyond its delegated competence, it informs other agents about the need for a corresponding solution.

3 Brief summary of the ABAsim architecture

The agent-based architecture of simulation models of *ABAsim* was mainly developed for simulation of queuing, transportation and logistic systems. Those systems can be considered (from the viewpoint of order processing) strictly *hierarchical*. The order (the customer) entering the system initiates a recursive sequence of suborders, according to the rules of competence redistribution.

The entities (orders/customers and resources) within the frame of the ABAsim architecture are divided into specialized classes with defined behavioural rules. This means that the responsibility for the behaviour of these entities is taken over by their superior subjects (agents). It is necessary in most cases to transfer service resources to the customer (or vice versa), in order to realize the service activity, i.e. frequent and complex transposition processes are typical within such systems.

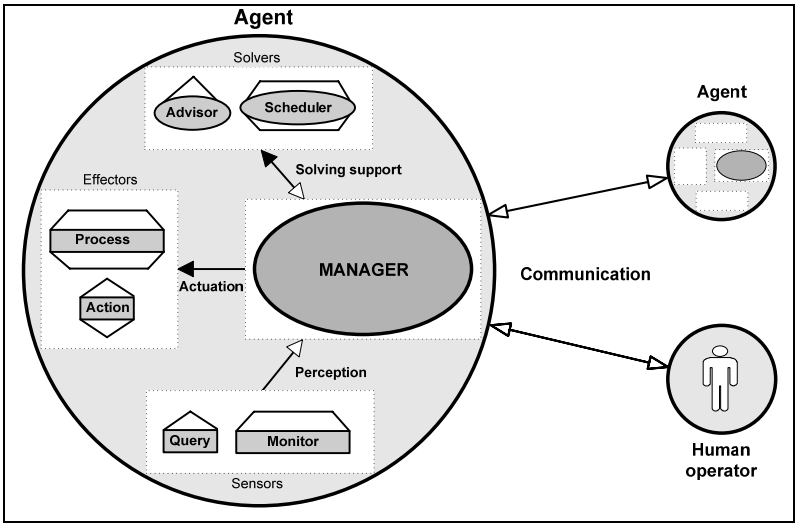


Fig. 1. Agent's decomposition

3.1 Agent components

Each agent can be decomposed into the following groups of *internal components* (presented on fig. 1):

a) The first, control and decision making component (called the manager) is responsible for making decisions and for inter-agent communication. In addition, the manager represents the central agent component because it initiates the work of other internal components and can also communicate with all of them.

b) The group of *sensors* is specialized for mining information from the system's state space. This group is composed of two kinds of components - the *query*

delivers the required forms of information instantly, and the *monitor* scans the state space in some time intervals and continuously brings important information to the manager.

c) The next group, called *solvers*, provides solutions for problems to the manager, which can accept them or ask for alternative ones. The *advisor* represents a passive component, which is able to react only to the manager's requests for delivery of proposals for problem-solving. A typical advisor can be represented e.g. by an optimization algorithm, an artificial neural network, a fuzzy regulator or a human operator. On the other hand, the *scheduler* (focused on a restricted scope of problems) works continuously for the manager, on the basis of either a priori rosters or schedules, which have been created (e.g. related to the allocation of resources), or by making its own dynamic forecast for a defined time interval.

d) The last component group includes *effectors* (actuators), which make changes to system status after receiving corresponding instructions from the manager. No other agent components are allowed to make these changes. An *action*-component makes instant state changes (e.g. switch traffic lights, close a train's doors), while a *process*-component (e.g. a crane's movement) makes them continuously until its task is finished.

The effectors, sensors, and solvers are, for brevity, given the umbrella term of manager's *assistants*, and can be further distinguished as:

- *Continual assistants*, the activities of which fill up some interval in the simulation time (processes, monitors, and schedulers).
- *Instant assistants*, which are active only in discrete instants of simulation time (actions, queries and advisors).

The question arises, how to realize the internal agent components appropriately - they can be described alternatively either

- using *imperative approach* (implementation of program routines constructed in a given source code), or
- by means of *declarative forms* (connected with some kind of symbolic formalism), which are reflected by structured input data and read by a corresponding interpreter; e.g. Petri nets represent effective formalisms appropriate for describing agent internal components.

3.2 Community of agents and its structure

Simulation models for simple real systems could be composed of only one agent; however, the simulation of complex service systems is obviously connected with a *multi-agent approach* using the agents within some organizational structure. Let us remark that the philosophy of the ABAsim architecture was also partly inspired by the paradigm of reactive agents, which is based on a society of reactive rather than proactive agents. The intelligence of such society *emerges* when one observes the whole community and not its separate members (individually of relatively low intelligence).

To summarize the philosophy of the ABAsim operation: The control role is played by mutually communicating managers (supported by sensors and solvers), which initiate the activities of effectors at the correct time instants and under particular conditions.

3.3 Communication mechanism

One way to realize inter-agent communication is to use standard communication languages (e.g. KQML or FIPA-ACL). Another approach is to implement a customized communication mechanism able to reflect, in the best way, the features of the respective architecture.

Communication within the ABAsim architecture is based on a simple, original mechanism applied to *inter-agent* and also *intra-agent* communications. As was already mentioned, inter-agent communication is made by manager components, and intra-agent communications are realized between the managers and their assistants. Both kinds of the ABAsim-communications utilize exclusively the paradigm of sending messages (from this viewpoint, the ABAsim represents *message-oriented architecture*).

The following description characterizes selected kinds of messages used within the ABAsim architecture in a simple way. *Notice*–messages contain some information for the addressee without expecting any answer, *Request*–messages carry specific demands, which are expected to be satisfied or supplied by means of corresponding *Response*–messages. Continual assistants are initiated by *Start*–messages (sent by superior managers), whereas *Finish*–messages (sent by continual assistants) delivered to corresponding managers, indicate completion of an activity related to relevant continual assistant. In addition, managers exploit *Execute*–messages in order to obtain promptly required results from their inferior instant assistants. Finally, *Hold*–messages exclusively mediate the augmentation of simulation time. They involve so-called *time stamps*, which define duration of their deliveries (equal or greater than the current simulation time). The attributes *sender* and *addressee* contain the same values – i.e. the continual assistants send those messages to themselves with some time delay. Thus, after sending *Hold*–message, the continual assistant remains idle and resumes its activity after the message returns. We have to emphasize that the augmentation of the simulation time is realized exclusively by continual assistants, i.e. synchronization of simulation time is based on synchronization of these components.

3.4 ABAsim versus other agent-based architectures

Seeing that general paradigm of autonomous agents influenced the design and development of the ABAsim architecture, it is only natural that the architecture shares some common principles with other agent-based simulation architectures that were inspired by the same paradigm.

Among many, it can be mentioned for example Cougaar architecture [8] (with similar hierarchical organization of agent communities or agent decomposition to

simpler executive units) or architecture HIDES [9], which shares the same view on importance of hierarchical structure of agents reflecting modelled system and supports forming of agent communities responsible for specific tasks. Since its beginning, ABAsim architecture has been oriented to creation of simulation models of complex large-scale service systems, mainly transportation systems, with emphasis on flexibility for simulation model designers, programmers as well as for end-users of simulation models. Since comparison of the ABAsim architecture with outlined or other simulation architectures and detailed explanation of benefits that it introduces is out of scope of this paper, we recommend the reader to pay attention to the following papers [10,11,12].

4 Specification of ABA-CPN

Coloured Petri net (CPN) describing the logic of an agent component can be defined within an environment of a specific software tool, where an analysis of the net is supposed to be carried out. Analysed nets are consequently made available (via respective data files) to a simulation engine of the ABAsim architecture. A relevant interpreter maintains then the evolution of the nets during simulation.

For the needs of simulation models based on the ABAsim architecture, modelling capabilities of the non-hierarchical coloured Petri net can be restricted. This results in a specific subclass of coloured Petri net, called *ABA-CPN*, partially inspired by the mentioned ABA-graph.

The following definition builds upon CPN definitions from [5]. Since it is quite complex, we divide it into four parts that are interleaved with comments and illustrations on a small example depicted on the figure 2 and built just for that purpose.

Definition 1:
ABA-CPN represents a subclass of *CPN* = ($\boldsymbol{\Sigma}$, $\boldsymbol{P}$, $\boldsymbol{T}$, $\boldsymbol{A}$, $\boldsymbol{N}$, $\boldsymbol{C}$, $\boldsymbol{G}$, $\boldsymbol{E}$, $\boldsymbol{I}$) that satisfies the following:

(i) $\boldsymbol{\Sigma}$ is a finite set of non-empty types, called *colour sets*.

(ii) The finite set of *places* $\boldsymbol{P} = \{\boldsymbol{p_{in}}\} \cup \{\boldsymbol{p_{out}}\} \cup \boldsymbol{P_S}$, where $\boldsymbol{p_{in}}$ is called *input place*, $\boldsymbol{p_{out}}$ *output place* and $\boldsymbol{P_S}$ is composed of *internal places* and the three sets are mutually disjoint.

(iii) The finite set of *transitions* $\boldsymbol{T} = \boldsymbol{T_D} \cup \boldsymbol{T_A} \cup \boldsymbol{T_S} \cup \boldsymbol{T_B}$, $\boldsymbol{T} \neq \emptyset$, where elements of $\boldsymbol{T_D}$ are denoted as *decision transitions*, elements of $\boldsymbol{T_A}$ as *assistant transitions*, elements of $\boldsymbol{T_S}$ as *sending transitions* and elements of $\boldsymbol{T_B}$ as *standard transitions*, the four sets are all mutually disjoint and $\boldsymbol{T} \cap \boldsymbol{P} = \emptyset$.

(iv) $\boldsymbol{A}$ is a finite set of *arcs* such that $\boldsymbol{P} \cap \boldsymbol{A} = \boldsymbol{T} \cap \boldsymbol{A} = \emptyset$.

(v) $\boldsymbol{N}$ is a *node* function defined from $\boldsymbol{A}$ into $(\boldsymbol{P} \times \boldsymbol{T}) \cup (\boldsymbol{T} \times \boldsymbol{P})$.

(vi) $\boldsymbol{C}$ is a *colour function* defined from $\boldsymbol{P}$ into $\boldsymbol{\Sigma}$.

(ii) The set of places is divided to subsets in order to distinguish specific kinds of places. A token in the input place $\boldsymbol{p_{in}}$ corresponds to an input message and a token in the output place $\boldsymbol{p_{out}}$ corresponds to an output message associated with a relevant

agent component. In the illustration net on the figure 2, the input place $p_{in} = p_1$, the output place $p_{out} = p_9$ and there are seven intern. places: $P_S = \{p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$.

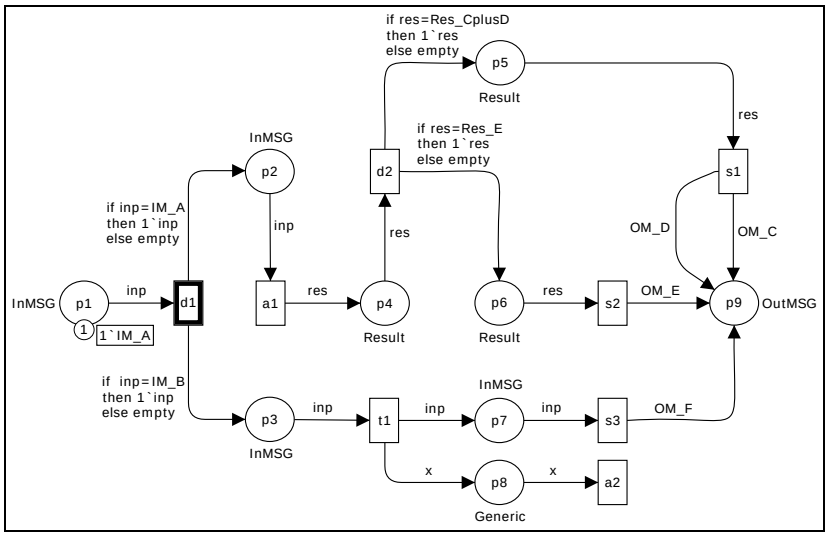


Fig. 2. Illustration net related to definition of ABA-CPN

(iii) The set of transitions is divided to four subsets in order to distinguish various actions in the ABA-CPN application. Decision transitions (involved in T_D) represent points of variant conditional branching (transitions d_1 and d_2 within the illustration net), assistant transitions (folded in T_A) reflect actions of corresponding instant assistants (a_1, a_2), sending transitions (elements of T_S) reflect sending messages to other model components (s_1, s_2, s_3), while only standard transitions (contained in T_B) have no specific meaning in the ABA-CPN application (t_1).

(i) + (vi) In the illustration net, there are four colour sets, i.e. $\Sigma = \{$*InMSG*, *Result*, *Generic*, *OutMSG*$\}$. Examples of colour function are: $C(p_1)$ = *InMSG* and $C(p_6)$ = *Result*. The *InMSG* colour set consists of two values *IM_A* and *IM_B*, the *OutMSG* colour set can have four different values: *OM_C*, *OM_D*, *OM_E* and *OM_F*. Colour set *Result* contains two values *Res_CplusD* and *Res_E*, and finally colour set *Generic* has a single value *e*.

Definition 1 cont'd:

(vii) E is an *arc expression function* defined from A into *arc expressions* (specified in [5]).

(viii) For arcs, there are the following specifications:

a) There is no such pair of arcs $a_1, a_2, a_1 \neq a_2$ with $p(a_1) = p(a_2) \wedge t(a_1) = t(a_2) \wedge ((N(a_1) \in T \times P \wedge N(a_2) \in P \times T)$, where $p(a)$ is the place and $t(a)$ is the transition of $N(a)$,

b) Every arc $a \in A$ belongs to one of the following categories:

- If $t(a) \in T_D \wedge N(a) \in T \times P$, a is called *decision arc* and its arc expression $E(a)$ contains a condition expression for variant branching,

- For all $t(a) \notin T_D$ and all $t(a) \in T_D$: $N(a) \in P \times T$, a is one of the following: *constant arc*, if $E(a)$ is composed of a single constant only, or *elementary variable arc*, if $E(a)$ consists of one variable only.

In the section (viii), the former specification states that self-loops are not admissible, so ABA-CPN represents a *pure net*. The latter specification introduces allowable arc expressions that divide arcs into categories. Arc expressions of all arcs going out of decision transitions (named *decision arcs*) have a condition expression. In the illustration net, it is the case of arcs (d_1, p_2), (d_1, p_3), (d_2, p_5) and (d_2, p_6). Arc expressions of arcs starting from any other transition can contain only one constant (*constant arcs*) or one variable (*elementary variable arcs*). In the illustration net, (p_1, d_1), (p_2, a_1), (a_1, p_4), (p_4, d_2), (p_5, s_1), (p_6, s_2), (p_3, t_1), (t_1, p_7), (p_7, s_3), (t_1, p_8), (p_8, a_2) are elementary variable arcs. Remaining arcs are classified as constant arcs.

Definition 1 cont'd:

(ix) G is a *guard function* defined from T into *guard expressions* (specif. in [5]).

(x) Elements from the sets P and T have the following additional properties (related to use of the formalism):

a) $(\forall\, t \in T: \neg\exists\, a \in A: N(a) = (t, p_{in})) \wedge (\exists_1\, a \in A: N(a) = (p_{in}, t), t \in T)$,

b) $(\exists\, a \in A: N(a) = (t, p_{out}), t \in T) \wedge (\forall\, t \in T: \neg\exists\, a \in A: N(a) = (p_{out}, t))$,

c) $\forall\, p \in P_S: (\exists\, a \in A: N(a) = (t, p), t \in T) \wedge (\exists_1\, a \in A: N(a) = (p, t), t \in T)$,

d) $\forall\, t \in T_B: \exists\, a \in A: N(a) = (p, t), p \in P$,

e) $\forall\, t \in T \setminus T_B: \exists_1\, a \in A: N(a) = (p, t), p \in P$,

f) $\forall\, t \in T_D \cup T_S : \exists\, a \in A: N(a) = (t, p), p \in P$,

g) $\exists_1\, t \in T_D: \exists_1\, a \in A: N(a) = (p_{in}, t)$, where t is denoted as *input transition*,

h) $t \in T: (\exists\, a \in A: N(a) = (t, p_{out})) \vee (\neg\exists\, a \in A: N(a) = (t, p), p \in P)$ is called *output transition*,

i) $t \in T: (\neg\exists\, a \in A: N(a) = (p_{in}, t)) \wedge (\neg\exists\, a \in A: N(a) = (t, p_{out})) \wedge (\exists\, a \in A: N(a) = (t, p), p \in P)$ is named as *internal transition*,

j) $\forall\, t \in T: G(t) = \varnothing$.

In the section (x), properties a), b) and c) deal with places: the *input place* has no incoming and only one outgoing arc (place *p1* in the illustration net); the *output place* can have only incoming arcs (place p_9) and all the other places have at least one incoming and just one outgoing arc.

Properties d) to j) deal with transitions. All transitions have at least one input arc. *Decision*, *assistant* and *sending transitions* have just one incoming arc, *standard transitions* may have more incoming arcs. As for outgoing arcs, they must be present by *decision* and *standard transitions* (case of transitions d_1, d_2, s_1, s_2 and s_3 in the illustration net), while not necessarily by the other transition types (e.g. transition a_2 has no outgoing arc). *Input transition* is just one and it is the one that follows the *input place* (transition d_1 after input place p_1). *Output transitions* are those having at least one outgoing arc to *output place* or not having any outgoing arc (transitions s_1, s_2, s_3 and a_2). All the other transitions are named *internal* (transitions d_2, a_1 and t_1). No transition disposes of a guard.

Definition 1 cont'd:

(xi) Petri net built from all elements of $\boldsymbol{P}$, $\boldsymbol{T}$, and $\boldsymbol{A}$ represents an *acyclic* net structure.

(xii) $\boldsymbol{I}$ is an *initialization function* defined from $\boldsymbol{P}$ into *closed expressions* (specified in [5]).

(xiii) The set of *admissible initial markings* $\boldsymbol{M_0} \subset \boldsymbol{I(P)}$ is defined as follows: $\boldsymbol{M_0} = \{{}^{j}\boldsymbol{M_0} \mid \boldsymbol{j} = 1,2, \ldots, |\boldsymbol{C}(\boldsymbol{p_{in}})|\}$, where ${}^{j}\boldsymbol{M_0}$ denotes j-th initial marking in the set, for which holds: $\forall\ \boldsymbol{p} \in \boldsymbol{P}$: $|{}^{j}\boldsymbol{M_0}(\boldsymbol{p})| = 1$, if $\boldsymbol{p} = \boldsymbol{p_{in}}$, and $|{}^{j}\boldsymbol{M_0}(\boldsymbol{p})| = 0$, if $\boldsymbol{p} \neq \boldsymbol{p_{in}}$, and for $i \neq j$, ${}^{i}\boldsymbol{M_0}(\boldsymbol{p}) \neq {}^{j}\boldsymbol{M_0}(\boldsymbol{p})$.

(xiii) The tokens of different colours in ABA-CPN reflect differently filled message forms, which are in the ABAsim architecture utilised for communication purposes. An admissible initial marking of ABA-CPN allows an occurrence of just one token in the whole net which is located in the input place $\boldsymbol{p_{in}}$; all other places dispose of no tokens. This represents a state, where an input message waits in the input place to be processed and there are no other messages being processed. Since the ABA-CPN is constructed to process all relevant input messages for the given manager separately and the input messages are represented by tokens from the colour set of the input place $\boldsymbol{C}(\boldsymbol{p_{in}})$, the set of admissible initial markings $\boldsymbol{M_0}$ contains as many markings as is the number of input messages, i.e. $|\boldsymbol{C}(\boldsymbol{p_{in}})|$, and ${}^{j}\boldsymbol{M_0}$ denotes the j-th initial marking from the set. In the illustration net on the fig. 2, an admissible initial marking is displayed: the input place $\boldsymbol{p_1}$ contains one token of colour *IM_A*, while all other places are empty. This represents a situation of input message *IM_A* coming to the agent to be processed.

5 Example of ABA-CPN

Let us illustrate application of ABA-CPN to a description of manager component involved within an agent named *Resource controller*. The agent represents a part of a model (based on the ABAsim architecture) reflecting simple service system (fig. 3).

The system's customers, coming into the system from its surroundings, are supposed to be consecutively involved in two kinds of serving activities and after their finishing they leave the system. While the first kind of services is carried out (within the process *Service A*) with the help of an immovable service resource (i.e. the customer has to come up to its place), the second one is made by a mobile resource (exploiting the process *Service B*), which is able to move (using the process *Resource transfer*) directly to the customer's spot.

A simulator of the mentioned system is composed of three agents: the *Surroundings* agent, the *Service controller* agent and the *Resource controller* agent. The first one is responsible for a connection between the system and its surroundings (deals with arrivals and departures of individual customers), the second one organizes all concurrently running service activities and finally the third one is competent to assign/release service resources and to make disposals for their transpositions. Such designed simulator can be specified in the form of the *ABAsim model*, the simplified form of which (excluding instant assistants for the sake of simplification) is depicted

on fig. 3. The mentioned picture shows all essential model components and their communication links.

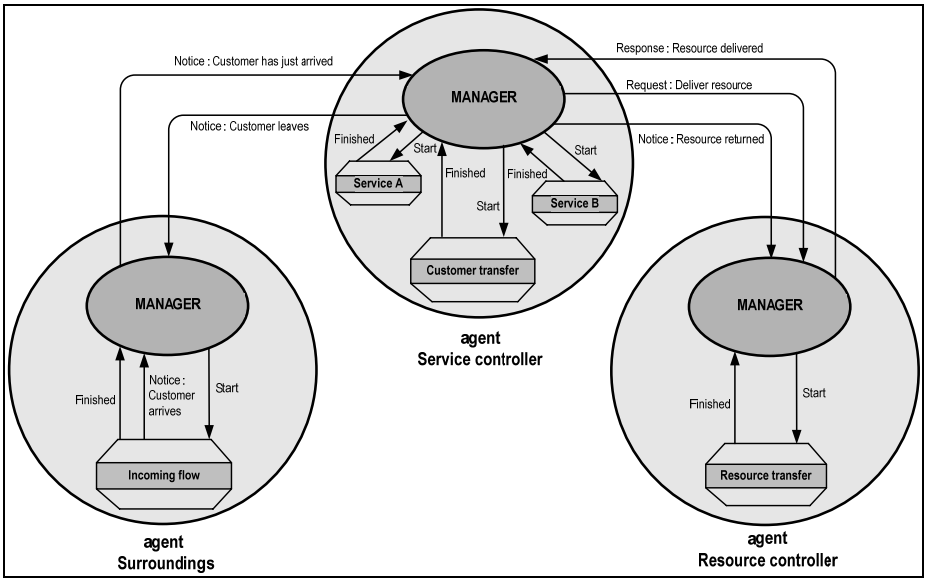


Fig. 3. Simplified ABAsim model (with commun. links) of the elementary service system

The ABA-CPN of the manager component (encapsulated within the *Resource controller* agent) is shown on the fig. 4. It disposes of the following characteristics (according to the ABA-CPN definition):

- Set of places $\boldsymbol{P} = \{\boldsymbol{p_i} \mid \boldsymbol{i} = 1,\ldots,16\}$, where: $\boldsymbol{p_{in}} = \boldsymbol{p_1}$, $\boldsymbol{p_{out}} = \boldsymbol{p_{16}}$, $\boldsymbol{P_S} = \{\boldsymbol{p_i} \mid \boldsymbol{i} = 2,\ldots, 15\}$.
- Set of transitions $\boldsymbol{T} = \boldsymbol{T_D} \cup \boldsymbol{T_A} \cup \boldsymbol{T_S} \cup \boldsymbol{T_B}$, where $\boldsymbol{T_D} = \{\boldsymbol{d_i} \mid \boldsymbol{i} = 1,\ldots, 4\}$, $\boldsymbol{T_A} = \{\boldsymbol{a_i} \mid \boldsymbol{i} = 1,\ldots, 8\}$ (elements of $\boldsymbol{T_A}$ are commented in the table 1), $\boldsymbol{T_S} = \{\boldsymbol{s_i} \mid \boldsymbol{i} = 1,2\}$ and $\boldsymbol{T_B} = \{\boldsymbol{t_1}\}$; the input transition is represented by $\boldsymbol{d_1}$ and output transitions by $\boldsymbol{s_1}$, $\boldsymbol{s_2}$, $\boldsymbol{a_4}$, $\boldsymbol{a_8}$.

Table 1. Characteristics of assistant transitions involved in ABA-CPN presented within fig. 4

Transition	reflects component	Transition	reflects component
$\boldsymbol{a_1}$	instant assistant - advisor *Selection from avail. resources*	$\boldsymbol{a_5}$	instant assistant - action *Resource assign. to applicant*
$\boldsymbol{a_2}$	instant assistant - action *Resource release*	$\boldsymbol{a_6}$	instant assistant - action *Dequeue applicant*
$\boldsymbol{a_3}$	instant assistant - query *Queue for released resource?*	$\boldsymbol{a_7}$	instant assistant - query *Resource transfer to applicant?*
$\boldsymbol{a_4}$	instant assistant - action *Enqueue applicant*	$\boldsymbol{a_8}$	instant assistant - action *Update transfer statistics*

- The set of admissible initial markings $\boldsymbol{M_0} = \{{}^1\boldsymbol{M_0}, {}^2\boldsymbol{M_0}, {}^3\boldsymbol{M_0}\}$ represents three different input messages that can reach the manager component of the *Resource_controller* agent and is defined as follows:
  ${}^1\boldsymbol{M_0}(\boldsymbol{p_1}) = \{REQ\_Deliver\_resource\}$ and ${}^1\boldsymbol{M_0}(\boldsymbol{p_j}) = \emptyset$, $\boldsymbol{j} = 2, \ldots, 16$;
  ${}^2\boldsymbol{M_0}(\boldsymbol{p_1}) = \{NTC\_Resource\_returned\}$ and ${}^2\boldsymbol{M_0}(\boldsymbol{p_j}) = \emptyset$, $\boldsymbol{j} = 2, \ldots, 16$;
  ${}^3\boldsymbol{M_0}(\boldsymbol{p_1}) = \{FIN\_Transfer\}$ and ${}^3\boldsymbol{M_0}(\boldsymbol{p_j}) = \emptyset$, $\boldsymbol{j} = 2, \ldots, 16$.

  Occurrence graph of the illustrative ABA-CPN for the initial marking ${}^1M_0(p_1)$ is presented on fig. 5 (using the *CPN Tools* software). It shows that processing of input message *REQ_Deliver_resource* can be ceased in three terminal states: either produce one output message (state space nodes 13 and 14) or no output message (node 7). The output message can be one of these two: *START_Transfer* (node 14) or *RESP_Resource_delivered* (node 13).

6 Conventions for the notation of ABA-CPN

In order to carry out unambiguous transformation of the ABA-CPN into relevant data structures of simulation model program and to enable its correct evolution using a specialized interpreter, there are certain conventions for the notation of the ABA-CPN elements. ABA-CPN is constructed within the environment of the *CPN Tools* software (developed at the University of Aarhus, Denmark [6]). Notation conventions are as follows (see an example on fig. 4):

a) places are denoted as $\boldsymbol{p_i}$ for $\boldsymbol{i} = 1..\boldsymbol{n}$, where $\boldsymbol{n} = |\boldsymbol{P}|$; $\boldsymbol{p_1}$ denotes *input place* and $\boldsymbol{p_n}$ corresponds to *output place* of the net,
b) *decision transitions* are denoted as $\boldsymbol{d_i}$ for $\boldsymbol{i} = 1..\boldsymbol{m}$, $\boldsymbol{m} = |\boldsymbol{T_D}|$, (as $\boldsymbol{d_1}$ we denote *input transition*), *standard transitions* are denoted as $\boldsymbol{t_i}$ for $\boldsymbol{i} = 1..\boldsymbol{k}$, $\boldsymbol{k} = |\boldsymbol{T_B}|$, *assistant transitions* dispose of description $\boldsymbol{a_i}$ for $\boldsymbol{i} = 1..\boldsymbol{l}$, $\boldsymbol{l} = |\boldsymbol{T_A}|$ and *sending transitions* are associated with notation $\boldsymbol{s_i}$ for $\boldsymbol{i} = 1..\boldsymbol{r}$, $\boldsymbol{r} = |\boldsymbol{T_S}|$,
c) descriptions of *constant arcs* or *elementary variable arcs* use declared constants, elements of colour sets and variables, each arc with one symbol only,
d) *decision arcs* typically dispose of an expression in the form 'if *var* = *const* then *case1* else *case2*', where *var* or *const* represent relevant variable or constant, *case1* or *case2* reflect elementary notations composed of one variable or one constant or the key word *empty*.

Conventions for denoting variables and constants in arc expressions follow the goal that the designer should be able to control reactions to relevant "content" of a token (in this case content of a relevant message form). In addition, using the conventions enables to manage "movements" of token instances within the net.

For purpose of the following explanation, let us consider $\boldsymbol{s}$ as a string, whereas $\boldsymbol{i}$-th character of the string is denoted as $\boldsymbol{s^i}$.

Proposal of conventions related to denoting variables and constants in arc expressions of arcs adjacent with a transition $\boldsymbol{t} \in \boldsymbol{T}$, is as follows:

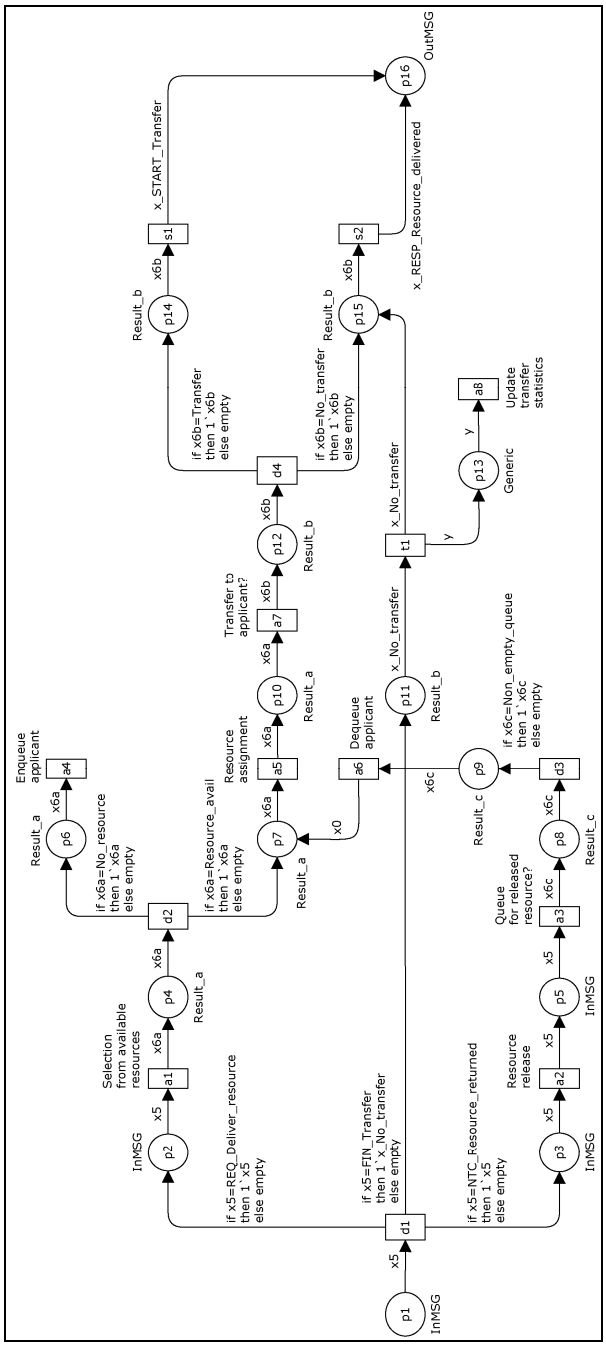


Fig. 4. Coloured Petri net reflecting a manager related to Resource controller agent

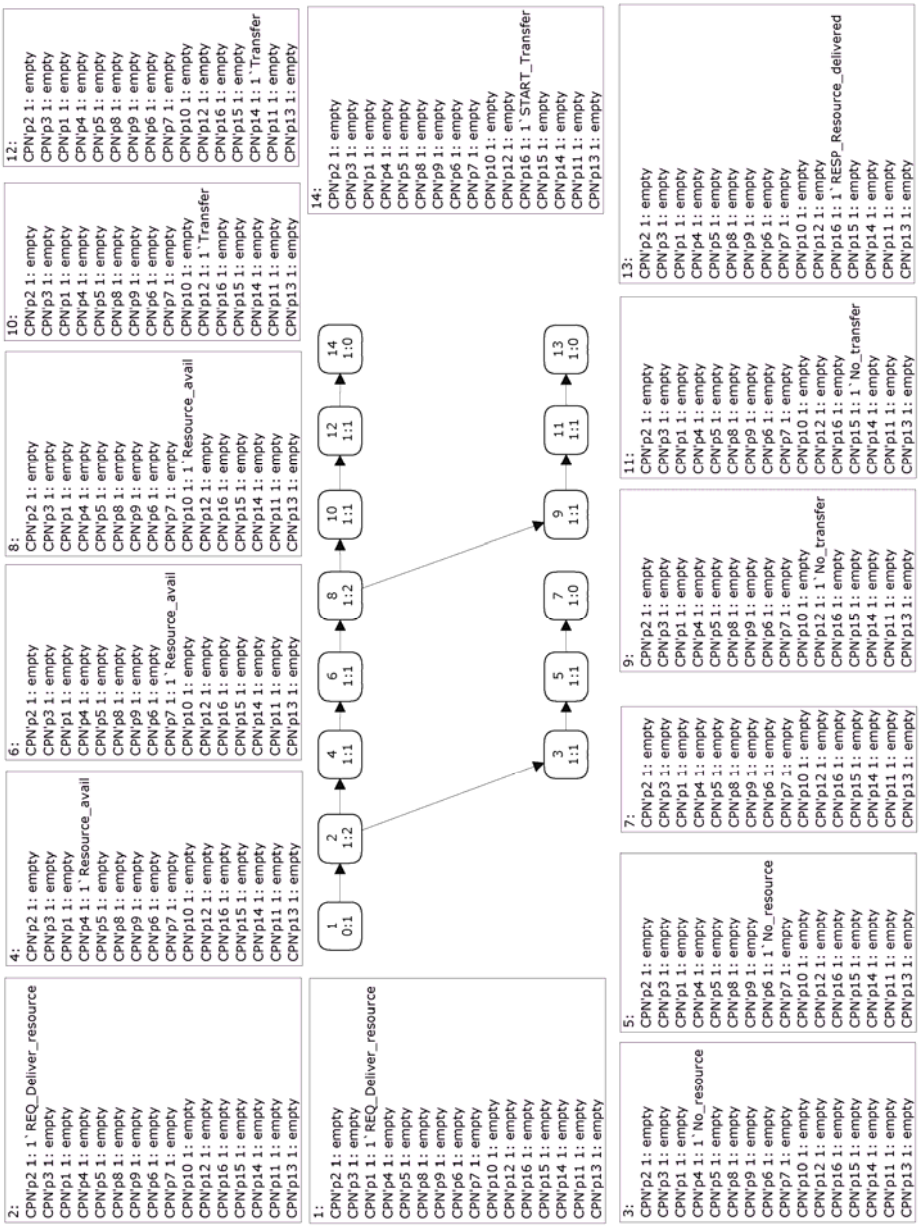


Fig. 5. Occurrence graph for $M_0(p_1)$=REQ_Deliver_resource, $M_0(p_i)$=∅, i = 2,…,14

a) For firing transition $\boldsymbol{t} \in \boldsymbol{T}$, it holds that token instance removed from the place $\boldsymbol{p} \in \boldsymbol{P}$: $\boldsymbol{N}(\boldsymbol{a}) = (\boldsymbol{p}, \boldsymbol{t})$, $\boldsymbol{a} \in \boldsymbol{A}$ is consequently placed (itself or its identical copy) to a place $\boldsymbol{q} \in \boldsymbol{P}$: $\boldsymbol{N}(\boldsymbol{a}) = (\boldsymbol{t}, \boldsymbol{q})$, $\boldsymbol{a} \in \boldsymbol{A}$ under following conditions:

- Token instance from the place $\boldsymbol{p}$ is removed by means of a constant or a variable (let's call it ***input*** for the sake of this explanation) contained within expression of the arc ($\boldsymbol{p}$, $\boldsymbol{t}$).
- Positioning token instance to the place $\boldsymbol{q}$ is mediated by a constant or a variable (let's call it ***output***) encapsulated within expression of the arc ($\boldsymbol{t}$, $\boldsymbol{q}$).
- For the mentioned identifiers of constants or variables, it holds: $\boldsymbol{input}^{\mathbf{1}} = \boldsymbol{output}^{\mathbf{1}}$, i.e. the first character of both identifiers is the same.

The presented convention enables to affect the concrete "path" of a token instance within the net. It means in fact that designer of a corresponding net (included within the frame of a simulation model based on the ABAsim architecture) can determine a particular passing of a message instance (carrying specific data) through the net. Such a feature extends behavioural rules of classical coloured Petri nets: instead of disappearing of "old" and appearing of "new" tokens during firing of transitions the tokens representing messages can be understood as preserved.

b) The second character within string identifier (if it has more than one character) of a variable (not a constant) involved in a relevant arc expression is utilized for determining of data that is contained within $\boldsymbol{j}$-th item of a given token $\boldsymbol{c}$. Firing of decision transition $\boldsymbol{t} \in \boldsymbol{T_D}$ follows the next stages:
   - Token instance $\boldsymbol{c}$ from the place $\boldsymbol{p} \in \boldsymbol{P}$: $\boldsymbol{N}(\boldsymbol{a}) = (\boldsymbol{p}, \boldsymbol{t})$, $\boldsymbol{a} \in \boldsymbol{A}$ is removed by means of a variable (with string identifier ***input***) contained within expression of ($\boldsymbol{p}$, $\boldsymbol{t}$).
   - The second character of the ***input*** string is identified as $\boldsymbol{j} = \boldsymbol{int}(\boldsymbol{input}^{\mathbf{2}})$, where function ***int*** transforms character $\boldsymbol{input}^{\mathbf{2}} \in \boldsymbol{M}$ (set of characters '1',…,'9') to a corresponding result from interval of integers $\langle 1, \ldots, 9 \rangle$.
   - The corresponding data content of the $\boldsymbol{j}$-th item of $\boldsymbol{c}$-token is obtained – let us denote it as $\boldsymbol{val}^{c,j}$ – it is consequently elaborated within expressions of decision arcs, which go out of decision transition $\boldsymbol{t}$.

   In case that the second character of the string identifier $\boldsymbol{s}^{\mathbf{2}}$ is equal to '0', it indicates that all data items associated with an instance of relevant token in the simulation program are during the operation represented by the adjacent transition reset to initialisation values.

c) Places are denoted as $\boldsymbol{p_i}$ for $\boldsymbol{i} = 1..\boldsymbol{n}$, where $\boldsymbol{n}=|\boldsymbol{P}|$; $\boldsymbol{p_1}$ denotes *input place* and $\boldsymbol{p_n}$ corresponds to *output place* of the net.

d) The third character within the string identifier $\boldsymbol{s}^{\mathbf{3}}$ of variables is utilized only in the cases that there is more than one identifier in one net with the first two characters being the same, while they are related to tokens of different colours.

e) Naming convention for constant identifiers for their second and every additional character differs from variable identifiers. There is no further rule for that, which means that constant identifiers follow only the convention in the point a).

Let us demonstrate proposed conventions (using the ABA-CPN from fig. 4) for construction of identifiers related to constants and variables on the cases of two transitions.

- For the input decision transition $\boldsymbol{d_1}$, it holds that arc expressions associated with all adjacent arcs dispose of identifiers related to constants and variables, the first

characters of which are equal to 'x' ('$x5$','$x\_No\_transfer$'). It means that the message represented by a token instance removed from the place p_1 is consequently bound to a new token positioned to a relevant place q. The second character of the variable identifier associated with elementary variable arc (p_1, d_1) is equal to '5', i.e. interpreter of ABA-CPN decides about variant branching according to the current content of the 5-th data item of the message encapsulated by a token c that is being currently removed from the place p_1. For example, in the case of $val^{c.5} = REQ\_Deliver\_resource$ the message is bound to a new token that is placed to p_2.

- On the other hand, situation on the transition t_1 is different: firing it does not use only the message instance in the token removed from p_{11} (subsequently placed to p_{15}), but creates another (new) instance as a copy of it and puts it to p_{13}. This behaviour is based on the fact that the first character of the constant identifier '$x\_No\_transfer$' (associated with arc (p_{11}, t_1)) is not equal to the first character of the variable identifier 'y' linked to arc (t_1, p_{13}).

7 Conclusions

In this paper, we described the ABA-CPN, subclass of coloured Petri nets, used for formalization of control and communication of agents within the ABAsim architecture. The ABA-CPN is *structurally bounded*, not *reversible* and not *live* Petri net from definition, since it is acyclic (point (xi) of Def. 1) and there is no source transition allowed (points (x) d) and e) of Def. 1).

State space of the ABA-CPN usually contains at least one dead marking, which can be of two types. The first type is caused by existence of output place p_{out} with no outgoing arc (point (x) b) of Def. 1). In this type of dead marking, only the place p_{out} contains one or more tokens and all the other places are empty. The second type is caused by existence of standard transition $t \in T_B$ or assistant transition $t \in T_A$ with no outgoing arcs, what may lead potentially to a dead marking with no tokens in the net. All dead markings represent admissible end states of processing of initial message (initial marking M_0) in the ABA-CPN. The former type of dead marking symbolises sending of message (result of processing) from current agent to another agent, the latter type stands for consumption of the message form and no need of further communication.

As for liveness of individual transitions, the only transition occurring in all sequences is the input transition $t \in T_D$. If the net is designed correctly, it contains no transition that would be dead in all occurrence sequences for all admissible initial markings. Analysis of the liveness property is the principal benefit from use of the ABA-CPN. It is used for checking of correct construction of a concrete ABA-CPN. If the occurrence graph contains dead markings related to non-admissible states, it indicates a mistake in structure of the constructed net.

Current development concentrated on descriptions of agent components within the ABAsim architecture of simulation models prefers utilizing a specific subclass of coloured Petri net (ABA-CPN) to a subclass of P/T Petri net due to higher modelling capabilities of CPN. For example, construction of conditional branching and

differentiation of various instances of messages is more natural and feasible using formalism of CPN than formalism of P/T PN.

At the present time, the development of a software application (interpreter of ABA-CPN) is carried out. Within the ABAsim simulation kernel, it is supposed to maintain the evolution of the mentioned ABA-CPN. Constructed and analyzed ABA-CPN (within the *CPN Tools* environment) is at interpreter's disposal using XML-formatted file. The interpreter of the ABA-CPN is currently intensively tested.

Acknowledgments. This work has been supported by the Czech National research program under project MSM 0021627505 "Theory of transportation systems" and by the grant of the Scientific Grant Agency VEGA 1/4057/07 in the Slovak Republic.

References

1. Kavička, A., Klima, V., Adamko, N.: Simulations of transportation logistic systems utilizing agent-based architecture, International Journal of Simulation Modelling, DAAAM International, Vienna, 1 (2007) 13-24
2. Adamko, N., Kavička, A., Klima, V.: Agent based simulation of transportation logistic systems, DAAAM International Scientific Book 2007, Chapter 36, B. Katalinic (Ed.), DAAAM International, Vienna (2007) 407- 422
3. Kavička, A.: Petri net with decision transitions applied within ABAsim architecture of simulation models. In MOSIS'03 – Proceedings of the 37th conference Modelling and simulation of systems, MARQ, Ostrava (2006) 373-380
4. Kavička, A., Klima, V., Adamko, N.: Analysis and optimization of railway nodes using simulation techniques, In COMPRAIL 2006 – Proceedings of 10th Computer system design and operation in the railway and other transit system, WIT-Press, Southampton (2006) 663-672
5. Jensen, K.: Coloured Petri nets – basic concepts. Springer Verlag, Berlin (1997)
6. CPN Tools home page. [online]. [cited on 29 February 2008] Available at: <http://www.daimi.au.dk/ CPNTools/>
7. Jennings, R.: An agent-based approach for building complex software systems, *Communications of the ACM*, Vol. 44 (2001) 35-41
8. Helsinger, A., Thome, M., Wright, T.: Cougaar: A Scalable, Distributed Multi-Agent Architecture, *Proceedings of Systems, Man and Cybernetics, IEEE International Conference*, Cambridge (2004) 1910-1917
9. Henoch, J., Ulrich, H.: HIDES: Towards an Agent-Based Simulator, *Proceedings of the Workshop on Agent Based Simulation,* SCS European Publishing House, [cited on 24 September 2008] Available at: *<http://www.ifor.math.ethz.ch/publications/oldpublications/ 2000_towardsagentbasedsimulator.pdf >*
10. Daniel Moldt, D., Wienberg,F.: Multi-Agent-Systems Based on Coloured Petri Nets, Proceedings of the 18th International Conference on Application and Theory of Petri Nets (1997) 82-101
11. Fernandes, J., M., Bello, O.: Modeling of Multi-agent System Activities through Colored Petri Nets: an Industrial Production System Case Study. In Proc. of the16 Int. Conf. on Applied Informatics, Anaheim, CA, (1998) 17-20.
12. Weyns, D., Holvoet, T.: A colored Petri-net for a multi-agent application, Proceedings of MOCA'02 (Moldt, D., ed.), vol 561, DAIMI PB (2002) 121-141

The ComBack Method Revisited: Caching Strategies and Extension with Delayed Duplicate Detection*

Sami Evangelista, Michael Westergaard and Lars Michael Kristensen

DAIMI, University of Aarhus, Denmark
{evangeli,mw,kris}@cs.au.dk

Abstract. The ComBack method is a memory reduction technique for explicit state space search algorithms. It enhances hash compaction with state reconstruction to resolve hash conflicts on-the-fly thereby ensuring full coverage of the state space. In this paper we provide two means to lower the run-time penalty induced by state reconstructions: a set of strategies to implement the caching method proposed in [18], and an extension through delayed duplicate detection that allows to group reconstructions together to save redundant work.

1 Introduction

Model checking is a formal method used to detect defects in system designs. It consists of a systematic exploration of the reachable states of the system whose behavior can be formally represented as a directed graph whose nodes are states and arcs are possible transitions from one state to another. This principle is simple, can be easily automated, and, in case of errors, a counter-example can be provided to the user.

However, even simple systems may have an astronomical or even infinite number of states. This state explosion problem is a severe obstacle to the application of model checking to industrial size systems. Numerous possibilities are available to alleviate, or at least delay, this phenomenon. One can for example exploit the redundancies in the system description that often induce symmetries [3], exploit the independence of some transitions to reduce the exploration of redundant interleavings [6], or encode the state graph using compact data structures such as binary decision diagrams [1].

Hash compaction [15,19] is a graph storage technique that reduces the amount of memory used to store states. It uses a hash function h to map each encountered state s into a fixed-size bit-vector $h(s)$ called the *compressed state descriptor* which is stored in memory as a representation of the state. The *full state descriptor* is not stored in memory. Thus, each discovered state is represented compactly using typically 32 or 64 bits. The disadvantage of hash compaction is that two different states may be mapped to the same compressed state descriptor

* Supported by the Danish Research Council for Technology and Production.

which implies that the hash compaction method may not explore all reachable states. The probability of *hash collisions* can be reduced by using multiple hash functions [10,15], but the method still cannot guarantee full coverage of the state space. This is acceptable if the intent is to find errors, but not sufficient if the goal is to prove the correctness of a system specification.

The ComBack method [18] extends hash compaction with a backtracking mechanism that allows reconstruction of full state descriptors from compressed ones and thus resolve conflicts on-the-fly to guarantee full coverage of the state space. Its underlying principle is to store for any state a sequence of events that generated this state. Thus, when the search algorithm checks if it already visited a state s, it can reconstruct states mapped to the same hash value as s and compare them to it. Only if none of the states reconstructed is equal to s can the algorithm consider it as a new state.

This storage technique stores a small amount of information per state, typically between 16 and 24 bytes depending on the system being analyzed. Thus it is especially suited to industrial case studies for which the full state descriptor stored by a classical search algorithm can be very large (from 100 bytes to 10 kilo-bytes). This important reduction, however, has a time cost: a ComBack based algorithm will explore many more arcs in order to reconstruct states. As the graph is given implicitly, visiting an arc consists of applying a successor function that can be arbitrarily complex, especially for high-level languages such as Promela [8] or Colored Petri nets [9]. Experiments made in [18] report an increase in run-time ranging from 50% for the simplest examples to more than 600% for real-life protocols.

The goal of the work presented in this paper is to propose solutions to tackle this problem. Starting from the proposal of [18] to use a cache of full state descriptors to shorten sequences, we first propose different caching strategies. We also extend the ComBack method with delayed duplicate detection, a technique widely used by disk-based model checkers [16]. The principle is to delay the instant we check if a state has already been visited from the instant of its generation. Any state reached is put in a set of candidates and only occasionally is this set compared to the set of already visited states in order to identify new ones. The underlying idea of this operation is that comparing these two sets may be much cheaper than checking separately if each candidate has already been visited. Applied to the ComBack method, this results in saving the visit of transitions that are shared by different sequences. For instance if sequences $a.b.c$ and $a.b.d$ reconstruct respectively states s and s' we may group the reconstructions of s and s' in order to execute sequence $a.b$ only once instead of twice. This will result in the execution of 4 events instead of 6 events.

This article has the following structure. The basic elements of labeled transition systems and the ComBack method are recalled in Section 2. In Section 3, different caching strategies are proposed. An algorithm that combines the ComBack method with delayed duplicate detection is presented in Section 4. Section 5 reports on experiments made with the ASAP tool [12] which implements the techniques proposed in this paper. Finally, Section 6 concludes this paper.

2 Background

We give in this section the basic ingredients that are required for understanding the rest of this paper and provide a brief overview of the ComBack method [18].

2.1 Transition systems

As the methods proposed in this work are not linked to a specific formalism they will be developed in the framework of labeled transition systems that are the most low-level representation of concurrent systems.

Definition 1 (Labeled Transition System). *A labeled transition system is a tuple $\mathcal{S} = (S, E, T, s_0)$, where S is a finite set of* ***states****, E is a finite set of* ***events****, $T \subseteq S \times E \times S$ is the* ***transition relation****, and $s_0 \in S$ is the* ***initial state****.*

In the rest of this paper we assume that we are given a labeled transition system $\mathcal{S} = (S, E, T, s_0)$. Let $s, s' \in S$ be two states and $e \in E$ an event. If $(s, e, s') \in T$, then e is said to be *enabled* in s and the *occurrence* (execution) of e in s leads to the state s'. This is also written $s \xrightarrow{e} s'$. An *occurrence sequence* is an alternating sequence of states s_i and events e_i written $s_1 \xrightarrow{e_1} s_2 \cdots s_{n-1} \xrightarrow{e_{n-1}} s_n$ and satisfying $s_i \xrightarrow{e_i} s_{i+1}$ for $1 \leq i \leq n-1$. For the sake of simplicity, we assume that events are *deterministic*[1], i.e., if $s \xrightarrow{e} s'$ and $s \xrightarrow{e} s''$ then $s' = s''$.

We use $\rightarrow^*$ to denote the transitive and reflexive closure of T, i.e., $s \rightarrow^* s'$ if and only if there exists an occurrence sequence $s_1 \xrightarrow{e_1} s_2 \cdots s_{n-1} \xrightarrow{e_{n-1}} s_n$, $n \geq 1$, with $s = s_1$ and $s' = s_n$. A state s' is *reachable* from s if and only if $s \rightarrow^* s'$. The *state space* of a system is the directed graph (V, E) where $V = \{\, s' \in S \mid s_0 \rightarrow^* s' \,\}$ is the set of nodes and $E = \{(s, e, s') \in T \mid s, s' \in V\}$ is the set of edges.

2.2 The ComBack method

A classical state space search algorithm (Algorithm 1) operates on a set of visited states $\mathcal{V}$ and a queue of states to visit $\mathcal{Q}$. An iteration of the algorithm (lines 4–7) consists of removing a states from the queue, generating its successors and inserting the successor states that have not been visited so far both in the visited set and in the queue for a later exploration[2].

Using hash compaction [19], items stored in the visited set are not actual state descriptors but compressed descriptors, typically a 32-bit integer, obtained through a hash function h. Algorithm 2 uses this technique. The few differences with Algorithm 1 have been underlined. This storage scheme is motivated by the observation that full state descriptors are often large for realistic systems,

[1] For an extension of the ComBack method to non-deterministic transition systems the reader may consult Section 5 of [18].

[2] We will use the term of *state expansion* to refer to this process.

Algorithm 1 A classical search algorithm.

```
1: 𝒱 ← empty ; 𝒱.insert (s_0)
2: 𝒬 ← empty ; 𝒬.enqueue (s_0)
3: while 𝒬 ≠ empty do
4:     s ← 𝒬.dequeue ()
5:     for e, s' | (s, e, s') ∈ T do
6:         if s' ∉ 𝒱 then
7:             𝒱.insert (s') ; 𝒬.insert (s')
```

Algorithm 2 A search algorithm based on hash compaction.

```
1: 𝒱 ← empty ; 𝒱.insert (h(s_0))
2: 𝒬 ← empty ; 𝒬.enqueue (s_0)
3: while 𝒬 ≠ empty do
4:     s ← 𝒬.dequeue ()
5:     for e, s' | (s, e, s') ∈ T do
6:         if h(s') ∉ 𝒱 then
7:             𝒱.insert (h(s')) ; 𝒬.insert (s')
```

i.e., typically between 100 bytes and 10 kilo-bytes, which drastically limits the size of state spaces that can be explored. Though hash compaction considerably reduces memory requirements, it comes at the cost of possibly missing some parts of the state space and potentially some errors. Indeed, as h may not be injective, two different states may erroneously be considered the same if they are mapped to the same hash value. Hence, hash compaction is preferably used at early stages of the development process for its ability to quickly discover errors rather than proving the correctness of the system.

The ComBack method extends hash compaction with a backtracking mechanism that allows it to retrieve actual states from compressed descriptors in order to resolve hash collisions on-the-fly and guarantee full coverage of the state space. This is achieved by modifying the hash compaction algorithm as follows:

1. A *state number* (integer), or identifier, is assigned to each visited state s.
2. A *state table* stores for each compressed state descriptor a *collision list* of state numbers for visited states mapped to this compressed state descriptor.
3. A *backedge table* is maintained which for each state number of a visited state s stores a *backedge* consisting of an event e and a state number of a visited predecessor s' such that $s' \xrightarrow{e} s$.

The key algorithm of the ComBack method is the insertion procedure that checks whether a state s is already in the visited set and inserts it into it if needed. Its principle can be illustrated with the help of Figure 1 which depicts a simple state space. Each ellipse represents a state. The hash value of each state is written in the right part of the ellipse. The state and backedge tables used to resolve hash conflicts have been depicted to the right of the figure for two different steps of the search. For the sake of clarity, we have also depicted on the state space the identifier of each state (the square next to the ellipse) and highlighted (using

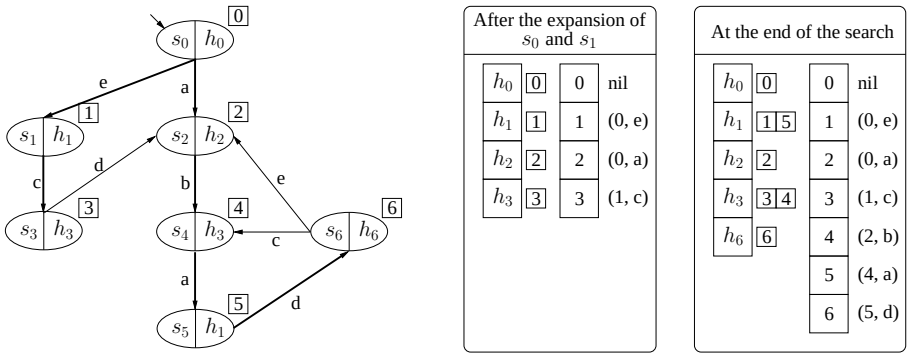


Fig. 1. A state space and the state and backedge tables at two stages.

thick arcs) the transitions that are used to backtrack to the initial state, i.e., the edges constituting the backedge table. Note that these identifiers also coincide with the expansion order of states.

After the expansion of s_0 and s_1, the set of visited states is $\{s_0, s_1, s_2, s_3\}$. As no hash conflict was detected, a single state is associated in the state table (the left table of the first rounded box) with each hash value. In the backedge table (the right table of the first rounded box) a nil value is associated with state 0 (the initial state) as any backtracking will stop here. The table also indicates that the actual value of state 1 (s_1) is retrieved by executing event e on state 0 and so on for the other entries of the table. After the execution of event b on state s_2 we reach s_4. Algorithm 2 would claim that s_4 has already been visited — since $h(s_3) = h(s_4)$ — and stop the search at this point, missing states s_5 and s_6. Using the two tables the hash conflict between s_3 and s_4 can be handled as follows. The insertion procedure first looks in the state table if any state has already been mapped to $h(s_4) = h_3$ and finds out value 3. The comparison of state 3 (of which we do not have the actual state descriptor) to s_4 is first done by recursively following the pointers of the backedge table until the initial state is reached, i.e., 3 then 1 and then 0. Then the sequence of events associated with the entries of the table that have been met during the backtrack, i.e., $e.c$, is executed on the initial state[3]. Finally, a comparison between s_3 and s_4 indicates that s_4 is new. We therefore assign to s_4 a new identifier (4) insert it in the collision list of hash value h_3 and insert the entry $4 \rightarrow (2, b)$ in the backedge table.

This storage scheme is especially suited to systems exhibiting large state vectors as it allows to represent each state in the visited set with only a few bytes. The only elements of the state and backedge tables that are still dependent of

[3] We will use the term of *state reconstruction* (or more simply *reconstruction*) to refer to this process, i.e., backtracking to the initial state and then executing a sequence of events to retrieve a full state descriptor. Sequence $e.c$ will be called the *reconstructing sequence* of state 3.

the underlying model are the events stored to reconstruct states. In the case of Colored Petri Nets, this comprises a transition identifier and some instantiation values for its variables while for some modeling languages it may be sufficient to identify an event with a process identifier and the line of the executed statement. Still, a state rarely exceeds 16–24 bytes.

However, the ComBack method is penalized by an (important) run-time increase due to the reconstruction mechanism. After a state s has been reached it will be reconstructed once for each following incoming arc, hence $in(s) - 1$ times where $in(s)$ denotes the in-degree of s. If we denote by $d(s)$ the length of the shortest path from s_0 to s, the number of event executions due to state reconstructions is lower bounded by:

$$\sum_{s \in S} (in(s) - 1) \cdot d(s)$$

Note that in Breadth-First Search (BFS) each sequence executed to reconstruct a state s is exactly of length $d(s)$ while it may be much longer in Depth-First Search (DFS). This is evidenced by some data of Table 1 in [18] showing that the ComBack method combined with DFS is in some cases much slower than with BFS while the converse is not true.

In addition, the time spent in reconstructing states depends, to a large extent, on the complexity of executing an event that ranges from trivial (e.g., for PT-nets) to high, e.g., for Promela or Colored Petri Nets for which executing an event may include the execution of embedded code.

3 Caching strategies

A cache mapping state identifiers to full descriptors is a good way to reduce the cost of state reconstructions. The purpose of such a cache is twofold. Firstly, the reconstruction of a state identified by i may be avoided if i is cached. Secondly, if a state has to be reconstructed we may stop backtracking as soon as we encounter a state belonging to the cache and thus execute a shorter reconstruction sequence from this state. As an example, consider the configuration of Fig. 1. Caching the mapping $1 \rightarrow s_1$ may be useful in two ways.

- To avoid the reconstruction of state 1. A lookup in the cache directly returns state s_1, which saves the backtrack to s_0 and the execution of event e.
- For the reconstruction of state 3. During the backtrack to s_0 the algorithm finds out that state 1 is cached, retrieves its descriptor and only executes event c from s_1 to obtain s_3, once again saving the execution of event e.

We now propose four strategies to implement this cache. We focus on strategies based on BFS as the traversal order it induces enables to take advantage of some typical characteristics of state spaces [13].

Random cache The simplest and easiest way is to implement a randomized cache. This gives us the first following strategy.

> **Strategy R**: *When a new state is put in the visited set, it is inserted in the cache with probability p (1 if the cache is not full) and the state to replace (if needed) is randomly chosen.*

Fifo cache A common characteristics of state spaces is the high proportion of forward transitions[4], typically around 80%. This has a significant consequence in BFS in which levels are processed one by one: most of the transitions outgoing from a state will lead to a new state or to a state that has been recently generated from the same level. Hence, a good strategy in BFS seems to be to use a fifo cache since when a new state at level $l+1$ is reached from level l it is likely that one of the following states of level l will also reach it. If the cache is large enough to contain any level of the graph, only backward transitions will generate reconstructions as forward transitions will always result in a cache hit. This strategy can be implemented as follows.

> **Strategy F**: *When a new state is put in the visited set, insert it unconditionally into the cache. If needed, remove the oldest state from the cache.*

Heuristic based cache Obviously, the benefit we can obtain from caching a state may largely differ from one state to another. For instance, it is pointless to cache a state s that does not have any successor state pointing to it in the backedge table as it will not shorten any reconstruction sequence, but only avoid the reconstruction of s.

To evaluate the interest of caching some state s we propose to use the following caching heuristic H.

$$H(s) = d(s) \cdot p(s) \text{ with } p(s) = \frac{r(s)}{L(d(s))}$$

where

- $d(s)$ is the distance of s to the initial state in the backedge table
- $r(s)$ is the number of states that reference s in the backedge table
- $L(n)$ is the number of states at level n, i.e., with a distance of n from the initial state

A cache hit is more interesting if it occurs early during the backtrack as it will shorten the sequence executed. Thus the benefit of caching a state s increases with its distance $d(s)$. Through rate $p(s)$ we evaluate the probability that s

[4] If we define *level* l as the set of states that are reachable from s_0 in l steps (and not less), a transition that has its source in level l and its target in level $l+1$ is called a *forward transition*. Any other transition is called a *backward transition*.

belongs to some reconstructing sequence. This one increases if many states point to s in the backedge table and decreases with the number of states on the same level as s. The distance of s could also be considered in the computation of $p(s)$ as s cannot appear in a reconstructing sequence of a length less than $d(s)$. Our choice is based on another typical characteristic of state spaces [17]: backward transitions are usually short in the sense that the levels of its destination and source are often close. Thus, in BFS, if a state has to be reconstructed, it is likely that the length of its reconstructing sequence is close to the current depth which is an upper bound of the length of a reconstructing sequence. Hence, assuming that the state space has this characteristic, the distance slightly impacts on $p(s)$.

Our third strategy is based on this heuristic.

> **Strategy H**: *After all outgoing transitions of state s have been visited compute $H(s)$. Let s' be the state that minimizes H in the cache. If $H(s') < H(s)$ replace s' by s in the cache.*

Note that after the visit of s, all necessary information to compute $H(s)$ is available since all its successors have been generated and the BFS search order implies that $L(d(s))$ is known.

Other possibilities are available. In [5] a reduction technique also based on state reconstruction is proposed. The algorithm is parametrized by an integer k and only caches states at levels $0, k, 2 \cdot k, 3 \cdot k \ldots$. The motivation of this strategy is to bound the length of reconstructing sequences to $k - 1$. As presented, the strategy in [5] does not bound the size of the cache but k could be dynamically increased to solve this problem.

Different strategies may also be combined. We can for example cache recently inserted states following strategy F and when a state leaves this cache it can be inserted into a second level cache maintained with strategy H. Thus we will keep some recently visited states in the cache and some old strategic states.

4 Combination with delayed duplicate detection

Duplicate detection consists of checking the presence of a newly generated state in the set of visited states. If the state has not been visited so far, it must be included in the set and later expanded. With delayed duplicate detection (DDD), this check is delayed from the instant of state generation by putting the state reached in a *candidate set* that contains potentially new states. In this scheme, duplicate detection consists of comparing the visited and candidate sets to identify new states. This is motivated by the fact that this comparison may be much cheaper than checking individually for the presence of each candidate in the visited set.

Algorithm 3 is a generic algorithm based on DDD. Besides the usual data structures we find a candidate set $\mathcal{C}$ filled with states reached through event execution (lines 7–8). An iteration of the algorithm (lines 4–9) consists of expanding all queued states and inserting their successors in the candidate set.

Algorithm 3 A generic search algorithm using delayed duplicate detection

```
1:  V ← empty ; V.insert (s_0)
2:  Q ← empty ; Q.enqueue (s_0)
3:  while Q ≠ empty do
4:      C ← empty
5:      while Q ≠ empty do
6:          s ← Q.dequeue ()
7:          for e, s' | (s, e, s') ∈ T do
8:              C.insert (s')
9:      duplicateDetection ()
10: proc duplicateDetection () is
11:      new ← C \ V
12:      for s ∈ new do
13:          V.insert (s)
14:          Q.enqueue (s)
```

Once the queue is empty duplicate detection starts. We identify new states by removing visited states from candidate states (line 11). States remaining after this procedure are then put in the visited set and in the queue (lines 12–14).

The key point of this algorithm is the way the comparison at line 11 is conducted. In the disk-based algorithm of [16], the candidate set is kept in a memory hash table and visited states are stored sequentially in a file. New states are detected by reading states one by one from the file and deleting them from the table implementing the candidate set. States remaining in the table at the end of this process are therefore new. Hence, in this context, DDD replaces a large number of individual disk look-ups — that each would likely require to read a disk block — by a single file scan. It should be noted that duplicate detection may also be performed if the candidate set fills up, i.e., before an iteration (lines 4–9) of the algorithm has been completed.

4.1 Principle of the combination

The underlying idea of using DDD in the ComBack method is to group state reconstructions together to save the redundant execution of some events shared by different reconstruction sequences. This is illustrated by Fig. 2. The search algorithm first visits states s_0, s_1, s_2, s_3 and s_4 each mapped to a different compressed state descriptor. Later, state s is processed. It has two successors: s_4 (already met) and s_5 mapped to h_3 which is also the compressed state descriptor of s_3. With the basic reconstruction mechanism we would have to first backtrack to s_0, execute sequence $a.b.d$ to reconstruct s_4 and find out that e does not, from s, generate a new state, and then execute $a.b.c$ from s_0 to discover a conflict between s_5 and s_3 and hence that f generates a new state. Nevertheless, we observe some redundancies in these two reconstructions: as sequences $a.b.c$ and $a.b.d$ share a common prefix $a.b$, we could group the two reconstructions together so that $a.b$ is executed once for both s_3 and s_4. This is where DDD can help us. As we visit s, we notice that its successors s_4 and s_5 are mapped to hash values already met. Hence, we put those in a candidate set and mark the identifiers of states that we have to reconstruct in order to check whether s_4 and s_5 are new or not, i.e., 3 and 4. Duplicate detection then consists of reconstructing marked states and to delete them from the

candidate set. This can be done by conducting a DFS starting from the initial state in search of marked states. However, as we do not want to reconstruct the whole search tree, we have to keep track of the subtree that we are interested in. Thus, we additionally store for each identifier the list of its successors in the backedge table that have to be visited. The DFS then prunes the tree by only visiting successors included in this list. On our example this will result in the following traversal order: s_0, s_1, s_2, s_3 and finally s_4.

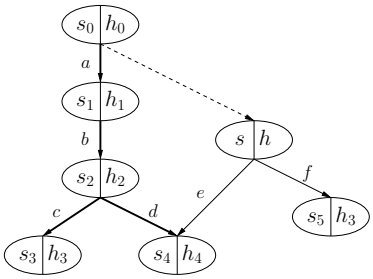


Fig. 2. The prefix $a.b$ of the reconstructing sequences of s_3 and s_4 can be shared.

4.2 The algorithm

We now propose Algorithm 4 that combines the ComBack method with DDD. As it is straightforward to extend the algorithm with a full state descriptor cache as discussed in Section 3 we only focus here on this combination.

The two main data structures in the algorithm are the queue $\mathcal{Q}$ containing full descriptors of states to visit together with their identifiers and the visited set $\mathcal{V}$. The latter comprises three structures.

- As in the basic ComBack method, the *stateTable* maps compressed states to state identifiers. It is implemented as a set of pairs (h, id) where h is a hash signature and id is the identifier of a state mapped to h.
- *backedgeTable* maps each identifier id to a tuple $(id_{pred}, e, check, succs)$ where
  - id_{pred} and e are the identifier of the predecessor and the reconstructing event as in the basic ComBack method;
  - *check* is a boolean specifying if the duplicate detection procedure must verify whether or not the state is in the candidate set;
  - *succs* is the identifier list of its successors which must be generated during the next duplicate detection as previously explained.
- *candidates* is a set of triples (s, id_{pred}, e) where s is the full descriptor of a candidate state. In case duplicate detection reveals that s does not belong to the visited set, id_{pred} and e are the reconstruction information that will be associated with the state in *backedgeTable*.

The main procedure (lines 1–10) works basically as Algorithm 3. A notable difference is that procedure *insert* (see below) may return a two-valued answer:

`NEW` - if the state is surely new. In this case, the identifier assigned to the inserted state is also returned by the procedure. The state can be unconditionally inserted in the queue for a later expansion.
`MAYBE` - if we can not answer without performing duplicate detection.

Algorithm 4 The ComBack method extended with delayed duplicate detection

```
 1: V ← empty ; Q ← empty
 2: n ← 0 ; id ← newState (s_0, nil, nil) ; Q.enqueue (s_0, id)
 3: while Q ≠ empty do
 4:     V.candidates ← empty
 5:     while Q ≠ empty do
 6:         (s, s_id) ← Q.dequeue ()
 7:         for e, s' | (s, e, s') ∈ T do
 8:             if insert (s', s_id, e)= NEW(s'_id) then Q.enqueue (s', s'_id)
 9:         if V.candidates.isFull () then duplicateDetection ()
10:     duplicateDetection ()
-----------------------------------------------------------------------
11: proc newState (s, id_pred, e) is
12:     id ← n ; n ← n + 1
13:     V.stateTable.insert (id, h(s))
14:     V.backedgeTable.insert (id → (id_pred, e, false, []))
15:     return id
-----------------------------------------------------------------------
16: proc insert (s, id_pred, e) is
17:     ids ← {id | (h(s), id) ∈ V.stateTable}
18:     if ids = ∅ then
19:         id ← newState (s, id_pred, e)
20:         return NEW(id)
21:     else
22:         V.candidates.insert (s, id_pred, e)
23:         for id in ids do
24:             V.backedgeTable.setCheckBit (id)
25:             backtrack (id)
26:         return MAYBE
-----------------------------------------------------------------------
27: proc backtrack (id) is
28:     id_pred ← V.backedgeTable.getPredecessorId (id)
29:     if id_pred ≠ nil then
30:         if id ∉ V.backedgeTable.getSuccessorList (id_pred) then
31:             V.backedgeTable.addSuccessor (id_pred, id)
32:             backtrack (id_pred)
-----------------------------------------------------------------------
33: proc duplicateDetection () is
34:     dfs (s_0, 0)
35:     for (s, id_pred, e) in V.candidates do
36:         id ← newState (s, id_pred, e)
37:         Q.enqueue (s, id)
38:     V.candidates ← empty
-----------------------------------------------------------------------
39: proc dfs (s, id) is
40:     check ← V.backedgeTable.getCheckBit (id)
41:     if check then V.candidates.delete (s)
42:     for succ in V.backedgeTable.getSuccessorList (id) do
43:         e ← V.backedgeTable.getReconstructingEvent (succ)
44:         dfs (s.exec (e), succ)
45:     V.backedgeTable.unsetCheckBit (id)
46:     V.backedgeTable.clearSuccessorList (id)
```

Procedure *newState* inserts a new state to the visited set together with its reconstruction informations. It computes a new identifier for s, a state to insert, and update the *stateTable* and *backedgeTable* structures.

Procedure *insert* receives a state s, the identifier id_{pred} of one of its predecessors s' and the event used to generate s from s'. It first performs a lookup in the *stateTable* for identifiers of states mapped to the same hash value as s (line 17). If this search is unsuccessful (lines 18–20), this means that s has definitely not been visited before. It is unconditionally inserted in $\mathcal{V}$, and its identifier is returned by the procedure. Otherwise (lines 21–26), the answer requires the reconstruction of states whose identifiers belong to set *ids*. We thus save s in the candidate set for a later duplicate detection, set the check bit of all identifiers in *ids* to true so that the corresponding states will be checked against candidate states during the next duplicate detection and backtrack from these states.

The purpose of the *backtrack* procedure is, for a given state s with identifier id, to update the successor list of all the states on the path from s_0 to s in the backedge table so that s will be visited by the DFS performed during the next duplicate detection. The procedure stops as soon as a state with no predecessor is found, i.e., s_0, or if id is already in the successor list of its predecessor, in which case this also holds for all its ancestors.

To illustrate this process, we have depicted in Fig. 3 the evolution of (a part of) the backedge table for the graph of Fig. 2. The four values specified for each state are respectively the identifier of the predecessor, the event used to reconstruct the state, the check bit (set to False or True), and the successor list. After the execution of event e from s we reach a state mapped to hash value h_4 already associated with state 4. We thus set the check bit of state 4 to true, backtrack from it and update the successor list of its ancestors 0, 1 and 2. The same treatment is performed for state 3 after the execution of f from s since the state thus reached and state 3 are mapped to the same hash value. The backtrack stops as we reach state 2 since it already belongs to the successor list of state 1.

Duplicate detection (lines 33–38) is conducted each time the candidate set is full (line 9), i.e., it reaches a certain peak size, or the queue is empty (line 10).

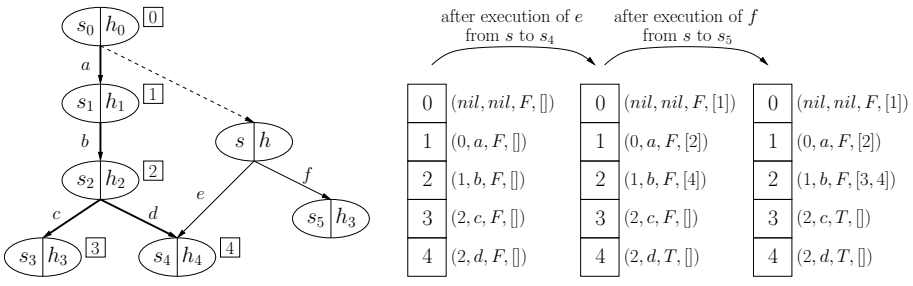


Fig. 3. Evolution of the backedge table after the execution of e and f from s

Using the successor lists constructed by the backtrack procedure, we initiate a depth-first search from s_0 (see procedure *dfs*). Each time a state with its check bit set to true is found (line 41) we delete it from the candidate set if needed. When a state leaves the stack we set its check bit to false and clear its successor list (lines 45–46). Once the search finishes (lines 35–37) any state remaining in the candidate set is new and can be inserted into the queue and the visited set.

4.3 Additional comments

We discuss several issues regarding the algorithm proposed in this section.

Memory issues Our algorithm requires the storage of some additional information used to keep track of states that must be checked against the candidate set during duplicate detection. This comprises for each state a boolean value (the *check* bit) and a list of successors that must be visited. As any state may belong to the successor list of its predecessor in the backedge table, the memory overhead is theoretically one bit plus one integer per state. However, our experiments reveal (see Section 5) that even very small candidate sets show good performance. Therefore, successor lists are usually short and the extra memory consumption low. We did not find any model for which the algorithm of [18] terminated whereas ours did not due to a lack of memory.

Grouping reconstructions of queued states In [18] the possibility to reduce memory usage by storing identifiers instead of full state descriptors in the queue (Variant 4 in Section 5) was mentioned. This comes at the cost of an additional reconstruction per state required to get a description of the state that can be used to generate its successors. The principle of grouping state reconstructions can also be applied to the states waiting in the queue. The idea is to dequeue blocks of identifiers from the queue instead of individual ones and reconstruct those in a single step using a procedure similar to *dfs* given in Algorithm 4.

Compatibility with depth-first search A nice characteristic of the basic ComBack method is its total decoupling from the search algorithm thereby making it fully compatible with, e.g., LTL model checking [2,7]. Delaying detection from state generation makes an algorithm implicitly incompatible with a depth-first traversal where the state processed is always the most recent state generated. At first glance, the algorithm proposed in this section also belongs to that category. However, we can exploit the fact that the insertion procedure can decide if a state is new without actually putting it in the candidate set (if the hash value of the state has never been met before). The idea is that the search can progress as long as new states are met. If some state is then put in the candidate set the algorithm puts a marker on the stack to remember that a potentially new state lies here. Finally, when a state is popped from the stack, duplicate detection is performed if markers are present on top of the stack. If we

find out that some of the candidate states are new, the search can continue from these ones. This makes delayed detection compatible with depth-first search at the cost of performing additional detections, during the backtrack phase of the algorithm.

5 Experimental results

We report in this section the data we collected during several experiments with the proposed techniques. We used the ASAP verification tool [12] where we have implemented the algorithms described in this article. A nice characteristic of ASAP is its independence from the description language of the model. This allowed us to perform experimentations on DVE models taken from the BEEM database [14] and on CPN models taken from our own collection.

Experimenting with caching strategies In this first experiment we evaluated the different strategies proposed in Section 3. We picked out 102 instances of the BEEM database having from 100,000 to 10,000,000 states and run the ComBack algorithm of [18] using BFS with 6 caching strategies and 3 sizes of cache (100, 1,000 and 10,000 states). Out of these 6 strategies 3 are simple: R (Random, with a replacement probability $p = 0.5$), F (Fifo), H (Heuristic); and 3 are combinations[5] of the first ones: F(20)-H(80), F(50)-H(50) and F(80)-H(20). We measured after each run the number of event executions that were due to state reconstructions. The results are summarized in table 1.

Strategy F performs well compared to R but it seems that its performance degrades (in comparison) as we allocate more states to the cache. This is also confirmed by the fact that the combination F-H seems to perform better for a large cache when the proportion of states allocated to the fifo sub-cache is low. Apparently with this strategy we quickly reach a limit where all (or most of) the forward transitions lead to a cached (or new) state and most backward transitions lead to a non cached state. Such a cache failure always implies backtracking to the initial state (the fifo strategy implies that if a state is not cached none of its ancestors in the backedge table is cached) which can be quite costly. Beyond this point, allocating more states to the cache is almost useless.

The performance of strategy H is poor for small caches but progresses well compared to strategy F. With this strategy, most transitions will be followed by a state reconstruction. However, our heuristic works rather well and reconstructing sequences are usually much shorter than with strategy F. Still, strategy H is usually outperformed by strategy F due to a high presence of forward transitions in state spaces [13]. To sum up, strategy F implies few reconstructions but long sequences and strategy H has the opposite characteristics.

From these observations it is not surprising to see that the best strategy is to maintain a small fraction of the cache with strategy F and the remainder with

[5] F(X)-H(Y) denotes the combination where X% of the cache is allocated to a fifo sub-cache and Y% is allocated to a heuristic based sub-cache.

Table 1. Evaluation of caching strategies on 102 DVE instances.

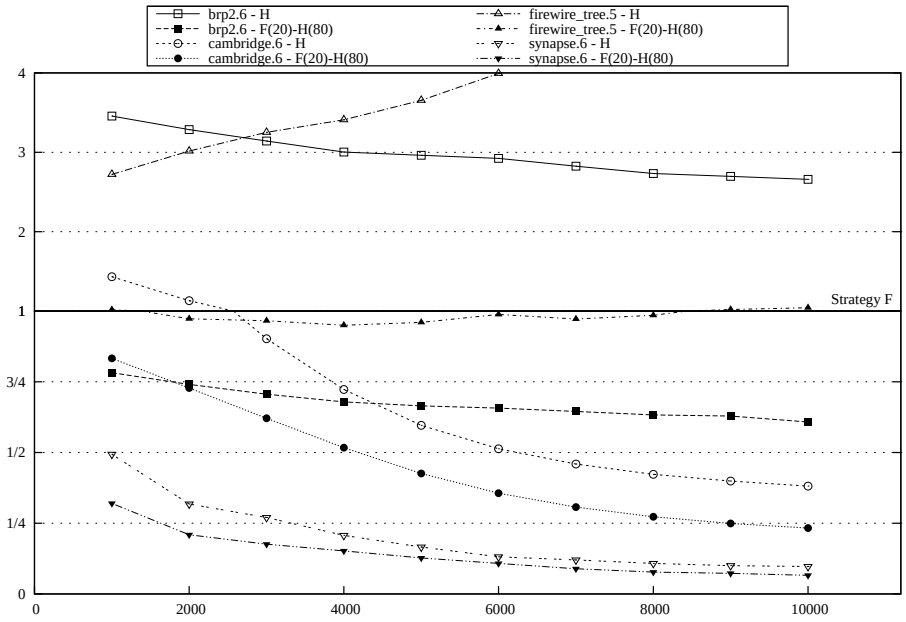
Cache size	Strategy R	Strategy F	Strategy H	Strategy F(20)-H(80)	Strategy F(50)-H(50)	Strategy F(80)-H(20)
10^2	1.0	0.429	1.128	0.436	0.397	**0.390**
10^3	1.0	0.437	1.178	0.364	**0.347**	0.355
10^4	1.0	0.488	0.742	**0.255**	0.262	0.302
10^2	0	7	2	21	34	**42**
10^3	0	3	1	**51**	38	17
10^4	0	7	3	**80**	14	9

Top rows: Average on all instances of the number of event executions due to state reconstruction with this strategy reported to the same number obtained with strategy R. Bottom rows: Number of instances for which this strategy performed best.

strategy H, that is to keep a small number of recently visited states and many strategic states from previous levels that will help us shorten reconstructing sequences.

Out of these 102 instances we selected 4 instances that have some specific characteristics (`brp2.6`, `cambridge.6`, `firewire_tree.5` and `synapse.6`) and evaluated strategies F, H and F(20)-H(80) with different sizes of cache ranging from 1,000 to 10,000. Data collected are plotted on figure 4. On the x-axis are the different cache sizes used. For each run we recorded the number of event executions due to reconstructions and reported it to the same number obtained with strategy F. For instance, with `brp2.6` and a cache of 4,000 states, reconstructions generated approximately three times more event executions with strategy H than with strategy F. We also provide the characteristics of these graphs in terms of number of states and transitions, average degree, number of levels and number of forward transitions as a proportion of the overwhole number of transitions.

The graph of `firewire_tree.5` only has forward transitions, which is common for leader election protocols. Therefore, a sufficiently large fifo cache is the best solution. This is one of the few instances where increasing the cache size benefits strategy F more than H. Moreover its average degree is high, which leads to a huge number of reconstructions with strategy H. On the opposite side the graph of `cambridge.6` has a relatively large number of backward transitions. Increasing the fifo cache did not bring any substantial improvement: from 262,260,647 executions with a cache size of 1,000 it went down to 260,459,235 executions with a cache size of 10,000. Strategy H is especially interesting for `synapse.6` as its graph has a rather unusual property: a low fraction of its states have a high number of successors (from 13 to 18). These states are thus shared by many reconstructing sequences and, using our heuristic, they are systematically kept in the cache. Thus, strategy H always outperforms strategy F even for small caches. The out-degree distribution of the graph of `brp2.6` has the opposite characteristics: 49% of its states have 1 successor, 44% have 2 successors and the other states have 0 or 3 successors. Therefore, there is no state that is



Instance	States	Transitions	Deg.	Levels	Forward tr.
`brp2.6`	5,742,313	9,058,624	1.57	571	92.6%
`cambridge.6`	3,354,295	9,483,191	2.83	259	66.8%
`firewire_tree.5`	3,807,023	18,225,703	4.79	202	100%
`synapse.6`	625,175	1,190,486	1.90	70	74.9%

Fig. 4. Evolution of strategies F, H and F(20)-H(80) on some selected instances.

really interesting to keep in the cache. This is evidenced by the fact that the relative progressions of heuristic based strategies are not so good. It goes from 3.456 to 2.660 for strategy H and from 0.782 to 0.608 for strategy F(20)-H(80).

Experimenting with delayed duplicate detection (DDD) To experiment with delayed detection we picked out 94 DVE instances from the BEEM database (all instances having between 500,000 and 50,000,000 states) and 2 CPN instances from our own database. The ComBack method was especially helpful for nets `dymo` and `erdp` that model two industrial protocols — a routing protocol [4] and an edge router discovery protocol [11] — and have rather large descriptors (1,000 - 5,000 bytes).

Table 2 summarizes our observations. Due to a lack of space we only report the data for some DVE instances but still provide the average on all instances. We used caching strategy F(20)-H(80), as it is apparently the best we proposed, with a cache size of 10,000. For each instance we performed 4 tests: one with a standard storage method, i.e., full state descriptors are kept in the visited

Table 2. Evaluation of delayed duplicate detection on DVE and CPN instances.

Std. T	Type of items in the queue	ComBack					
		No DDD		DDD(10^2)		DDD(10^3)	
		T↑	E↑	T↑	E↑	T↑	E↑
DVE instances							
`brp.5`*		17,740,267 states			36,903,290 transitions		
23.1	SD	11.02	21.19	8.08	7.56	9.01	7.10
	ID	35.03	69.12	14.57	17.48	15.74	15.37
`cambridge.7`		11,465,015 states			54,850,496 transitions		
195	SD	14.12	35.88	2.63	4.40	2.30	3.49
	ID	18.54	44.50	3.22	5.92	2.96	5.03
`iprotocol.5`*		31,071,582 states			104,572,634 transitions		
63.1	SD	8.74	11.71	5.07	2.97	4.93	2.65
	ID	21.12	30.81	6.72	5.44	6.63	4.91
`pgm_protocol.8`		3,069,390 states			7,125,121 transitions		
18.45	SD	2.07	2.64	1.65	1.48	1.59	1.35
	ID	16.87	42.04	4.60	8.05	4.42	7.31
`rether.6`		5,919,694 states			7,822,384 transitions		
13.0	SD	3.86	7.23	3.16	3.89	3.17	3.64
	ID	24.38	75.16	9.11	19.99	9.33	19.18
`synapse.6`		625,175 states			1,190,486 transitions		
1.4	SD	2.42	1.74	2.50	1.45	2.42	1.43
	ID	3.50	3.54	3.57	3.01	3.57	3.01
Average on 94 instances							
	SD	4.92	7.06	3.92	3.44	4.02	3.06
	ID	10.11	18.69	5.49	6.31	5.61	5.69
CPN instances							
`dymo.6`*		1,256,773 states			7,377,095 transitions		
2,115	SD	2.97	2.88	1.65	1.42	1.72	1.37
	ID	4.12	4.39	1.93	1.94	1.93	1.85
`erdp.3`*		2,344,208 states			18,739,842 transitions		
5,425	SD	3.99	6.17	2.19	2.69	2.10	2.28
	ID	4.37	7.50	2.15	3.15	1.92	2.70
Average on 2 instances							
	SD	3.48	4.52	1.92	2.05	1.91	1.82
	ID	4.24	5.94	2.04	2.54	1.92	2.27

$\star$: standard search out of memory. Time in column Std. is approximated.
Type of items in the queue: SD (full state descriptor) or ID (state identifier).

set, (column Std.), one with the ComBack method without delaying detection (column ComBack - No DDD) and two with delayed detection enabled with a candidate set of size 100 and 1,000 (columns ComBack - DDD(10^2) and ComBack - DDD(10^3)). Each test using the ComBack method actually comprises two runs: one keeping full state descriptors in the queue (line SD) and one keeping only identifiers in the queue (line ID) — as described in [18], Variant 4 of Section 5. For this second run, we used the optimization described in Section 4.3 that consists of grouping the reconstruction of queued identifiers. Each block of identifiers dequeued to be reconstructed had the same size as the candidate set. Hence, when DDD was not used this optimization was turned off. In column Std. - T we provide the execution time in seconds using a standard search algorithm. In columns T↑ we measure the run-time increase (compared to the standard search) as the ratio $\frac{\text{execution time of this run}}{\text{T (with standard search)}}$ and in column E↑ the increase of the number of event executions as the ratio $\frac{\text{event executions during this run}}{\text{transitions of the graph}}$. Hence, a value of 1 in this column means that we executed exactly the same number of events as the basic algorithm and that no state reconstruction occurred. Some runs using standard storage ran of out memory. This is indicated by a $\star$. For these, we provide the time obtained with hash compaction as a lower approximation.

We first observe that DDD is indeed useful to save the redundant exploration of transitions during reconstruction even with small candidate sets. Typically we can reduce the number of events executed by a factor of 3 or even more if we group the reconstruction of queued identifiers. It seems that, using BFS, states generated successively are "not so far" in the graph so their reconstructing sequences are quite similar, which allows many sharings.

However, this reduction does not always impact on the time saved as we could expect. Indeed DDD is much more interesting for CPN models than DVE models. If we consider for example the average made on the 94 DVE instances with our optimization disabled we divided the number of events executed by more than 2 (7.06 → 3.44) whereas the average time slightly decreased (4.92 → 3.92). The reason is that executing an event is much faster for DVE models than for CPN models. Events are typically really simple in the DVE language, e.g., a variable incrementation, whereas they can be quite complex with CPNs and include the execution of some embedded code. Therefore, the only fact of maintaining the candidate set or successors lists has a non negligible impact for DVE models which means that DDD reduces time only if the number of executions decreases in a significant way, e.g., instance `cambridge.7`.

Grouping the reconstruction of queued states can save a lot of executions, especially for long graphs like those of `brp.5` and `cambridge.7`. It should be noted that by storing identifiers in the queue we obtain an algorithm that bounds the number of full state descriptors kept in memory. Hence, we can theoretically consume less memory compared to an algorithm based on hash compaction which has to store full descriptors in the queue. This was indeed the case for nets `dymo.6` and `erdp.3`. Both have rather wide graphs: their largest levels contains approximately 10% of the state space and so contained the queue as it reached its peek size.

6 Conclusion

The ComBack method has been designed to explicitly store large state spaces of models with large state descriptors. The important reduction factor it may provide is however counterbalanced by an increase in run-time due to the on-the-fly reconstructions of states. We proposed in this work two ways to tackle this problem. First, some strategies have been devised in order to efficiently maintain a full state descriptor cache, used to perform less reconstructions and shorten the length of reconstructing sequences. Second, we combined the method with delayed duplicate detection that allows to group reconstructions and save the execution of events that are shared by multiple sequences. We have implemented these two extensions in ASAP and performed an extensive experimentation on both DVE models from the BEEM database and CPN models from our own collection. These experiments validated our proposals on many models. Compared to a random replacement strategy, a combination of our strategies could, on an average made on a hundred of DVE instances, decrease the number of transitions visited by a factor of four. We also saw that delaying duplicate detection is efficient even with very small candidate sets. In the best cases, we could even approach the execution time of a hash compaction based algorithm. Moreover, by storing identifiers instead of full descriptors in the queue we bound the number of full state descriptors that reside in memory. Hence, our data structures can theoretically consume less memory during the search than hash compaction structures. We experienced this situation on several instances.

In this work, we mainly focused on caching strategies for breadth-first search. BFS is helpful to find short error-traces for safety properties, but not if we are interested in the verification of linear time properties that is inherently based on depth-first search. The design of strategies for other types of search is thus a future research topic. In addition, the combination with delayed duplicate detection opens the way to an efficient multi-threaded algorithm based on the ComBack method. The underlying principle would be to have some threads exploring the state space and visiting states while others are responsible for performing duplicate detection. We are currently working on such an algorithm.

References

1. J.R. Burch, E.M. Clarke, D.L. Dill, L.J. Hwang, and K. McMillan. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of Logic In Computer Science*, pages 428–439, 1990.
2. J.-M. Couvreur. On-the-fly verification of linear temporal logic. In *Proceedings of Formal Methods*, Volume 1708 of *Lecture Notes in Computer Science*, pages 253–271. Springer-Verlag, 1999.
3. E.A. Emerson and A.P. Sistla. Symmetry and model checking. *Formal Methods in Systems Design*, 9(1-2):105–131, 1996.
4. K.L. Espensen, M.K. Kjeldsen, and L.M. Kristensen. Modelling and initial validation of the dymo routing protocol for mobile ad-hoc networks. In *Proceedings of Application and Theory of Petri Nets*, Volume 5062 of *Lecture Notes in Computer Science*, pages 152–170. Springer-Verlag, 2008.

5. S. Evangelista and J.-F. Pradat-Peyre. Memory efficient state space storage in explicit software model checking. In *Proceedings of SPIN – Software Model Checking*, Volume 3639 of *Lecture Notes in Computer Science*, pages 43–57. Springer-Verlag, 2005.
6. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, Volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
7. P. Godefroid and G.J. Holzmann. On the verification of temporal properties. In *Proceedings of Protocol Specification, Testing and Verification*, Volume C-16 of *IFIP Transactions*, pages 109–124. North-Holland, 1993.
8. G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
9. K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. Vol. 1-3*. Springer-Verlag, 1992-1997.
10. W. Knottenbelt, M. Mestern, P. Harrison, and P. Kritzinger. Probability, parallelism and the state space exploration problem. In *Proceedings of Modelling, Techniques and Tools*, Volume 1469 of *Lecture Notes in Computer Science*, pages 165–179. Springer-Verlag, 1998.
11. L.M. Kristensen and K. Jensen. Specification and validation of an edge router discovery protocol for mobile ad-hoc networks. In *Proceedings of Integration of Software Specification Techniques for Applications in Engineering*, Volume 3147 of *Lecture Notes in Computer Science*, pages 248–269. Springer-Verlag, 2004.
12. L.M. Kristensen and M. Westergaard. The ASCoVeCo state space analysis platform. In *Proceedings of Practical Use of Coloured Petri Nets and the CPN Tools*, Volume 584 of *DAIMI-PB*, pages 1–6, 2007. Available at: `http://www.daimi.au.dk/~ascoveco/asap.html`.
13. R. Pelánek. Typical structural properties of state spaces. In *Proceedings of SPIN – Software Model Checking*, Volume 2989 of *Lecture Notes in Computer Science*, pages 5–22. Springer-Verlag, 2004.
14. R. Pelánek. BEEM: Benchmarks for explicit model checkers. In *Proceedings of SPIN – Software Model Checking*, Volume 4595 of *Lecture Notes in Computer Science*, pages 263–267. Springer-Verlag, 2007.
15. U. Stern and D.L. Dill. Improved probabilistic verification by hash compaction. In *Proceedings of Correct Hardware Design and Verification Methods*, Volume 987 of *Lecture Notes in Computer Science*, pages 206–224. Springer-Verlag, 1995.
16. U. Stern and D.L. Dill. Using magnetic disk instead of main memory in the Murϕ verifier. In *Proceedings of Computer Aided Verification*, Volume 1427 of *Lecture Notes in Computer Science*, pages 172–183. Springer-Verlag, 1998.
17. E. Tronci, G. Della Penna, B. Intrigila, and M. Venturini Zilli. Exploiting transition locality in automatic verification. In *Proceedings of Correct Hardware Design and Verification Methods*, Volume 2144 of *Lecture Notes in Computer Science*, pages 259–274. Springer-Verlag, 2001.
18. M. Westergaard, L.M. Kristensen, G.S. Brodal, and L. Arge. The ComBack method - extending hash compaction with backtracking. In *Proceedings of Application and Theory of Petri Nets*, Volume 4546 of *Lecture Notes in Computer Science*, pages 445–464. Springer-Verlag, 2007.
19. P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Proceedings of Computer Aided Verification*, Volume 697 of *Lecture Notes in Computer Science*, pages 59–70. Springer-Verlag, 1993.

Two Interfaces to the CPN Tools Simulator

Michael Westergaard and Lars Michael Kristensen

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

Abstract. Coloured Petri nets (CP-nets or CPNs) is a useful modeling formalism for formally describing concurrent systems, and CPN Tools provides a mature environment for constructing, simulating, and performing simple analysis of CPN models. Sometimes, this does not suffice, however. For example, if one wishes to extend the analysis capabilities or to integrate CPN models into other programs. In this paper we present two new interfaces which facilitate this. One is written in Standard ML and is very close to the simulator component of CPN Tools, providing a solid foundation for developing advanced analysis tools. The other interface is written in Java and provides an object-oriented representation of CPN models as well as a means to load models created using CPN Tools. Furthermore, the Java interface provides a high-level interface to the simulator component facilitating integration of simulation of CPN models into other programs. We illustrate the interfaces by providing the complete implementation of a command-line state space exploration tool. The interfaces are available to interested parties.

1 Introduction

Coloured Petri nets (CP-nets or CPNs) provide a useful modeling formalism for formally describing concurrent systems, such as network protocols [12] or work-flows in companies [8]. CPN Tools [14] provides a mature environment for editing and simulating CPN models, and to a limited degree also for formally verifying that a given model is correct using state space analysis.

Sometimes, this is not enough, however. The basic problem is that CPN Tools is inherently graphical and cannot be controlled by outside applications. This makes it difficult to use CPN Tools in settings that are outside its scope of interactive use by one user. Such examples include repeated simulation on multiple servers in a grid, which is a useful analysis technique for models that are too large for exhaustive analysis techniques like state space analysis, to describe a complex decision procedure in a parametrised manner for use in a regular application, and allowing users to set parameters of a model using a custom user interface and just present the end-result of a simulation. It is also difficult to implement new analysis techniques such as new more efficient state space methods or completely different analysis methods (e.g., coverability graphs, bounded model-checking techniques, or invariant analysis), especially if we intend to also build a user-friendly interface for the new methods.

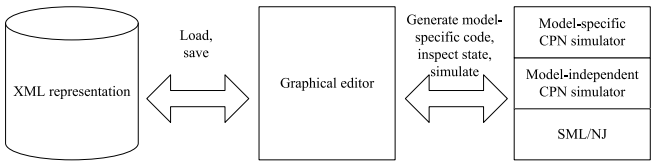


Fig. 1: Architecture of CPN Tools.

CPN Tools basically consists of two components (see Fig. 1), a graphical editor (middle) and a simulator daemon (right). The graphical editor allows the user to interactively construct a CPN model. The model is transmitted to the simulator daemon, which checks it for syntactical errors and generates model-specific code to simulate the model. The graphical editor can invoke the generated simulator code and present the results graphically. The graphical editor can also load and save models using an XML format (left in Fig. 1). The graphical editor imposes most of the previously mentioned restrictions; the simulator daemon is basically a generic Standard ML/New Jersey (SML/NJ or SML) [15] run-time environment and compiler with functions for syntax checking CPN models. It is obvious that by replacing the graphical editor with our application, we can alleviate most of the limitations imposed by the graphical editor, and this has indeed also been done in different settings [10,16]. The CPN simulator, however, suffers from two problems making such a replacement difficult. Firstly, the protocol used for communication between the graphical editor and the simulator is rather low-level and complex to implement. Secondly, the CPN simulator is optimised for simulation and incremental code generation making it difficult to use for other purposes as the model-specific code is difficult to use.

In this paper we propose two new interfaces to the CPN simulator[1]. Neither aim to replace CPN Tools as editor for CPN models, but rather to allow people to make experiments with the formalism. Both of the interfaces have been developed as part of the ASCoVeCo [1] project and the ASAP model checking platform [11], but are believed to be useful in other settings as well. Neither of the interfaces are intended for end-users; both of the interfaces provide rather low-level simulation primitives, which can be used by programmers to build new generic tools. We present the interfaces in the context of formal verification because that has been our motivation for developing the interfaces, but numerous applications can build upon the foundation to allow more high-level use of the CPN simulator. One of the interfaces is written in Java and the other in SML. In Fig. 2 we see how the new interfaces augment and replace parts of CPN Tools. The Java interface (middle) consists of an object-oriented representation of a CPN model, the ability to transmit this representation to the simulator and to programmatically perform simulation and inspection of the current state in the simulator. Furthermore, it includes an importer module which can import models created using CPN Tools. In effect, this allows programmers to load a model created using CPN Tools (left), instantiate a simulator for this model and perform simulation of the

[1] The interfaces are available to interested parties; send an email to `ascoveco@cs.au.dk` for more information.

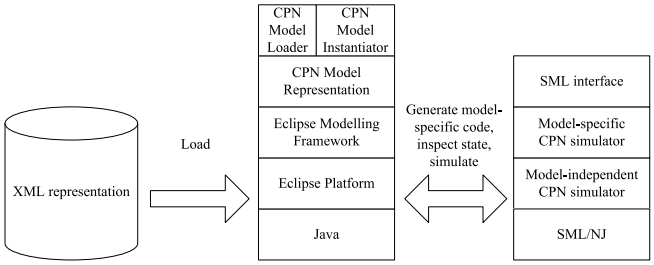


Fig. 2: Architecture of new interfaces.

model in their own applications, which can be anything from a simple command-line utility to a full-fledged CPN editor. The SML interface (right in Fig. 2) encapsulates the complex data-structures used in the simulator, and provides a simple frozen interface to the state of a CPN model which facilitates very fast simulation. This is in particular useful for efficient analysis, e.g., by means of state spaces, but applicable for any application that requires fast execution of transitions with little or no user-interaction.

The rest of this paper is structured as follows: In the next section, we introduce a simple example, that is used throughout this paper. In Sect. 3, we describe the SML interface to the simulator, and in Sect. 4, we describe the Java interface to the simulator. These two sections are independent of each other. In Sect. 5 we use the two interfaces to create a simple command-line tool for state space analysis of CPN models. Finally, in Sect. 6, we sum up our conclusions and provide directions for future work.

2 Example CPN Model

Throughout this paper we will use a CPN model of a simple stop-and-wait protocol with one sender and two receivers. The top module of the model can be seen in Fig. 3, where we have a substitution transition for the sender, the network, and one for each receiver. The network has a maximum capacity modeled by the Limit place. If the network still has available capacity, the sender (Fig. 4 (left)) transmits packets onto the A place. The place Send contains the packets to send. The network (Fig. 4 (middle)) then transmits the packet to B1 and B2, optionally dropping one or both of the packets. The receivers (Fig. 4 (right)) receive the packets on Received and transmit back acknowledgements onto C1 or C2, which the network transmits to D, optionally dropping one or both. When the sender receives acknowledgements from both receives, the NextSend counter is updated and the cycle restarts. We observe that the model consists of four different modules: Top, Sender, Network, and Receiver. The Receiver module is instantiated twice as Receiver 1 and Receiver 2 in the module Top.

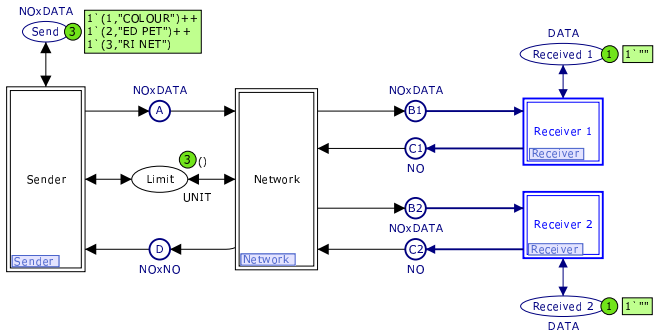


Fig. 3: Top page of a simple stop-and-wait protocol model with two receivers.

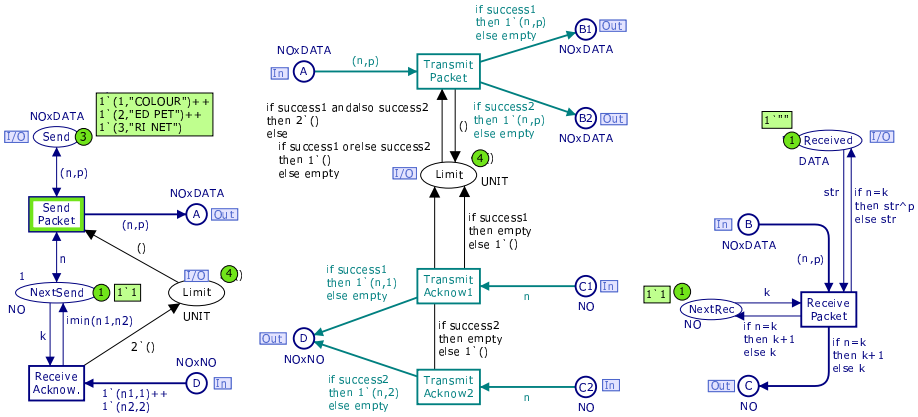


Fig. 4: Sender (left), network (middle), and receiver (right) modules of the protocol.

3 The SML CPN Model Interface

In this section we present the old SML interface to the simulator and some of its shortcomings. We also present our new interface and explain why it is superior. The aim of the SML model interface is to provide efficient access to the CPN simulator, in particular with the purpose of implementing efficient analysis methods. To support this, the SML interface provides an interface to the state of a CPN model and to execute enabled transitions. For performance reasons, this interface is written in the same language as the CPN simulator itself, namely SML/NJ [15]. We suggest that all applications that are algorithmic in nature use the SML interface described in this section. Using SML as implementation may seem a bit strange as it is not as well-known as, e.g., Java. The choice makes sense, however, both because this interface is the fastest as it is written in the same language as the simulator itself and because SML is a useful language for declaratively implementing complex algorithms due to its functional paradigm.

3.1 The Old SML Interface

In Listing 1.1 we see part of the current interface for the model in Figs. 3 and 4. In lines 1–10 we see the definition of the place NextRec in the module Receiver. We first notice that the relationship to the place and module is not immediately visible, as the place is only referred to by a generated identifier (CPN'placeid168). All places reside at the top level, so the modularity of the model is not visible in the interface. The functions get and set (ll. 7–8) take as parameter an instance number, which is the internal number of the instance of the place. This number is not immediately derivable from the model (we have, e.g., no guarantee that the instance corresponding to Receiver 1 has number 1). The ims.cs ms type is a multi-set over the type of the place, in this case NO.

The rest of Listing 1.1 shows representations of three different transitions, Send Packet from Sender (ll. 11–15), Transmit Acknow1 from Network (ll. 16–21), and Transmit Packet from Network (ll. 22–29). Like places, all transitions are referred to by a generated identifier rather that their user-recognisable name. Transitions, like places, live at the top-level, and the CPN'occfuns (ll. 12, 17–18, and 23–25) take an internal instance number as the first parameter. The last parameter given to CPN'occfun is a boolean indicating whether the step-counter should be incremented. This is used internally by the simulator for handling monitors, and during normal simulation should always be set to true. The middle parameter to a CPN'occfun describes the binding of the variables of the transition. For Send Packet, this consists of a record containing all variables. The two

Listing 1.1: Current interface.

```
1  structure CPN'placeid168: sig
2    structure ims: sig
3      structure cs: COLORSET
4      type cs = cs.cs
5      ... (* 1 type definition and 22 functions *)
6    end
7    val get: int -> ims.cs ms
8    val set: int -> ims.cs ms -> unit
9    ... (* 2 constants and 8 functions *)
10 end
11 structure CPN'transitionID1264271480: sig
12   val CPN'occfun: int * {n:NO, p:DATA} * bool -> CPN'Sim.result * string list
13   val CPN'bindings: int -> {n:NO, p:DATA} list
14   ... (* 5 constants, 3 variables, and 6 functions *)
15 end
16 structure CPN'transitionID1264276591: sig
17   val CPN'occfun:
18     int * ({n:NO} * BOOL) * bool -> CPN'Sim.result * string list
19   val CPN'bindings: int -> ({n:NO} * BOOL) list
20   ... (* 5 constants, 3 variables, and 6 functions *)
21 end
22 structure CPN'transitionID1264276586: sig
23   val CPN'occfun:
24     int * ({n:NO, p:DATA} * {success1:BOOL, success2:BOOL}) * bool
25     -> CPN'Sim.result * string list
26   val CPN'bindings:
27     int -> ({n:NO, p:DATA} * {success1:BOOL, success2:BOOL}) list
28   ... (* 5 constants, 3 variables, and 6 functions *)
29 end
```

transmit transitions are more complex. The technical reason is that, in the case of the Transmit Acknow1, the variables n and success1 are not correlated in any way, and can be bound independently, so by separating them it is possible to find legal bindings for the transition more efficiently. The CPN'occfun for Transmit Packet is just a more complex example of this. The result of CPN'occfun is a result from the simulator, indicating whether the transition was successfully executed, whether the transition was disabled, or whether the transition was not enabled a the current time stamp (for timed models). Additionally, a list of descriptive error messages may be returned. All transitions also have a function, CPN'bindings (ll. 13, 19, and 26–27), which given an instance number returns a list of all enabled bindings using the same grouping of variables as CPN'occfun.

This interface is well-suited for high-performance simulation and incremental code generation. By distributing the state to multiple structures, it is possible to update only markings of places affected by the execution of a given binding element (transition with associated binding of all variables), making the execution independent of the size of model. This also makes the enabling calculation more efficient, as the enabling is only affected for transitions connected to modified places (and we can even exploit monotonicity of enabling to further improve the enabling calculation). Furthermore, as all places and transitions are represented as separate structures, incremental code generation is independent of the size of the model. Adding a place or transition simply means we have to add a new structure. Modifying a transition only requires the regeneration of a single structure, and modifying a place only requires that we regenerate the structure corresponding to the place and all structures corresponding to transitions connected to the place, which is in practise a low number. Finally, during simulation, we are just interested in whether a transition is enabled, and, if so, to execute one enabled binding element. This is greatly facilitated by grouping the variables of transitions, as there is no reason to calculate all binding elements, which can be found as elements of the Cartesian product of elements of each group.

The properties of the interface facilitate an editor with incremental syntax check and efficient simulation of CPN models, but the requirements for a state space tool are different as we are dealing with many states (as opposed to just one during simulation), requiring that it is possible to represent more than one state. Also, we need to obtain all enabled binding elements in a given state. As the state is distributed across multiple structures in the old interface, it is difficult to represent more than one state at a time, as we would need to traverse all structures to read the marking of each place. As the enabling calculation of transitions is distributed across many structures, gathering all enabled transitions requires checking enabledness of transitions individually. Finally, the old interface is not very user-friendly, as we refer to all nodes using internal generated names and instance numbers not easily obtainable by the user.

3.2 The New SML Interface

Instead, we define a completely new interface to CPN models. The interface is designed with state space analysis in mind, but can of course be used for other

Listing 1.2: Model interface.

```
1  signature MODEL = sig
2    eqtype state
3    eqtype event

5    exception EventNotEnabled

7    (* Get the initial states and enabled events in each state              *)
8    val getInitialStates: unit -> (state * event list) list

10   (* Get the successor states and enabled events in each successor state  *)
11   val nextStates: state * event -> (state * event list) list

13   (* Execute event sequence, return resulting states and enabled events   *)
14   val executeSequence: state * event list -> (state * event list) list

16   (* String representations of states and events                          *)
17   val stateToString: state -> string
18   val eventToString: event -> string
19 end
```

purposes. The interface is designed to be independent of the actual formalism at the most abstract level, which allows us to build tools that are formalism-independent. The entire interface can be seen in Listing 1.2. The interface defines the concepts of states and events (ll. 2–3). The most important functions are getInitialStates (l. 8) and nextStates (l. 11). getInitialStates returns the list of initial states. The reason that this is a list and not just a singleton state is to support non-deterministic formalisms. In addition to the state, we also return a list of enabled events for each initial state. The reason for this is that it makes it possible to optimize enabling calculation during depth-first traversal. nextStates takes as argument a state and an event and returns the successors using the same format as getInitialStates. If the given event is not enabled, the exception EventNotEnabled (l. 5) is raised. Additionally, the interface has a function for executing a sequence of events, executeSequence (l. 14), which works like nextStates, except it can execute zero, one, or more events rather than just one. Finally, the interface contains two functions, stateToString and eventToString (ll. 17–18) for converting states and events to a user-readable string.

State Representation. The interface in Listing 1.2 is formalism-independent. In order to instantiate the interface for CPN models, we need to define the types state and event, and define the functions in the interface.

As mentioned earlier, we need to be able to represent multiple states in a state space tool. To increase familiarity for previous users of the state space tool of CPN Tools [14], we define a structure Mark with data types and functions for manipulating states. We do not want to distinguish between the type used internally and the type manipulated by users in order to alleviate the need for translating between different representations, so the type should closely reflect the underlying CPN model. In Listing 1.3, we see (most of) the Mark structure for the model in Figs. 3 and 4. The type of the state is defined inductively in the hierarchy of the model. For each page, we define a record, which contains

Listing 1.3: New state representation.

```
1  structure Mark : sig
2    type Sender = {NextSend: NO ms}
3    type Network = {}
4    type Receiver = {NextRec: NO ms}
5    type Top = {A: NOxDATA ms, B1: NOxDATA ms, B2: NOxDATA ms, C1: NO ms,
6                C2: NO ms, D: NOxNO ms, Limit: UNIT ms, Received_1: DATA ms,
7                Received_2: DATA ms, Send: NOxDATA ms, Network: Network,
8                Receiver_1: Receiver, Receiver_2: Receiver, Sender: Sender}
9    type state = {Top: Top, time: time}
10   val get'Top'Receiver_1'NextRec : state -> NO ms
11   val set'Top'Receiver_1'NextRec : state -> NO ms -> state
12   val get'Top'Receiver_2'NextRec : state -> NO ms
13   val set'Top'Receiver_2'NextRec : state -> NO ms -> state
14   val get'Top'Receiver_1'B : state -> NOxDATA ms
15   val set'Top'Receiver_1'B : state -> NOxDATA ms -> state
16   ... (* several more accessor functions *)
17 end
```

entries for all places and sub-pages of the page. For example, in Listing 1.3 l. 2 we see the record defined for the Sender page in Fig. 4 (left). We see that we have only included "real" places, i.e., the four port places are not included so only the NextSend place is present. The type uses the names used in the model, and NextSend is thus represented using the record entry NextSend. The type of the NextSend is NO ms, i.e., multi-sets over the color NO of the place NO. The multi-set type is the same as used by CPN Tools. Similarly, types are defined for Network (l. 3), which contains no non-port places, and Receiver (l. 4), which contains one non-port place. The Top page is more complex (ll. 5–8), but uses the same structure. It contains entries for all non-port (i.e., all) places (ll. 5–6), but also entries for all sub-pages (ll. 6–8). The entries for sub-pages are named after the substitution transition and the type is that of the sub-page. For example, we see that the sub-page defined by the substitution transition Receiver 1 is represented by the entry Receiver_1 of type Receiver. Finally, at the top-level, we define the type of the state itself. As it is possible for a model to contain more than one top page, we define a new top level (l. 9), which contains all top pages (in this case just one entry Top of type Top). The state type also contains an entry for all reference declarations (in this model there are none) and the model time. As an example, we see the initial state of the network protocol in Listing 1.4.

State records, like the one in Listing 1.4, can be used as is, i.e., by using SML pattern matching or built-in accessor functions to pull values out of the record, or by building new structures with the correct names. For the user convenience,

Listing 1.4: Initial state of network protocol.

```
1  val initial = { Top = {
2    A = empty, B1 = empty, B2 = empty, C1 = empty, C2 = empty, D = empty,
3    Limit = 3`(), Received_1 = 1`"", Received_2 = 1`"", Send = 1`(1,"COLOUR")++
4    1`(2,"ED PET")++1`(3,"RI NET"), Network = {}, Receiver_1 = {NextRec = 1`1},
5    Receiver_2 = {NextRec = 1`1}, Sender = {NextSend = 1`1} }, time = 0 }
```

we have also created set- and get-functions to access all pages and places of the structure. These functions all use the same naming convention, which is the function name (get or set) followed by a quote ('). Then comes the complete path to the place or page we wish to access, separated by quotes. The functions take a complete state as argument. Getter functions return either a multi-set of the appropriate type or a record describing the selected page. Setter functions instead take an additional parameter of the correct multi-set or record type and returns a new state, which is identical to the one given as the first parameter, except that the selected place/page marking has been replaced. Examples of setter and getter functions can be seen in Listing 1.3 in ll. 10–15. In addition to providing accessor functions for the "real" places represented in the state record, we also provide accessors which provide access to port and fusion places, so it is possible to use, e.g., get'Top'Receiver_1'B, to get the marking of the port place B in the receiver module. This function looks up the value on the corresponding socket place. This function is identical to get'Top'B1.

Event Representation. For events, we must make a choice between ease of use and compositionality. We first outline the obvious hierarchical approach to events and some of the problems of that. Then we describe our current implementation, which is not hierarchical (and thus does not as easily support compositionality).

The hierarchical event representation (Listing 1.5) is the natural companion to the state representation. Instead of types and records, we use structures and data types. For each page, we have a structure defining a data-type with a constructor for each transition and substitution transition. The type of each

Listing 1.5: Hierarchical representation of events.

```
1  structure Bind : sig
2    structure Top : sig
3      structure Sender : sig
4        datatype event = Send_Packet of {n: INT, p: STRING}
5                       | Receive_Acknow of {k: INT, n1: INT, n2: INT}
6      end
7      structure Network : sig
8        datatype event =
9          Transmit_Packet of
10           {n: INT, p: STRING, success1: BOOL, success2: BOOL}
11       | Transmit_Acknow1 of {n: INT, success1: BOOL}
12       | Transmit_Acknow2 of {n: INT, success2: BOOL}
13     end
14     structure Receiver : sig
15       datatype event =
16         Receive_Packet of {k: INT, n: INT, p: STRING, str: STRING}
17     end
18     datatype event = Sender of Sender.event
19                    | Network of Network.event
20                    | Receiver_1 of Receiver.event
21                    | Receiver_2 of Receiver.event
22   end
23   datatype event = Top of Top.event
24 end
```

constructor contains either a record with all variables (for normal transitions) or a reference to a previously defined data-type (for substitution transitions).

While this type definition is nice and natural, it has the major deficit that it is very cumbersome to use. The problem is that while data-type constructors are scoped, they are not context-sensitive. Thus, to refer to the transition Receive Acknow on the Sender page, we would need to write `Bind.Top.Sender Bind.Top.Sender.Receive_Acknow {k, n1, n2}`, and the verbosity and redundancy only gets worse if we have deeper hierarchies. We cannot solve this problem by opening all structures unless we require that all transitions, globally in the model, have unique names, and this is against the locality inherent in Petri nets.

Instead, we define a data-type as in Listing 1.6. We define a constructor for each transition named after the page it resides on and the name of the transition. The type of each constructor is a pair of an instance number and a record containing all variables associated with the transition. This definition is not as natural as the hierarchical one, and it re-introduces the "magic" instance numbers. To alleviate the introduction of instance numbers, we also define symbolic constants (ll. 10–14) for the path to each page instance. Using this, we can refer to the Receive_Acknow transition on Sender as `Bind.Sender'Receive_Acknow (Bind.Top.Sender, {k, n1, n2})`, where only Bind and Sender are repeated, and the latter only because the substitution transition has the same name as the page.

A final way to represent events is to create a data-type with a constructor for each transition instance, named after the path leading to the transition instance. While this is nice to use at first sight, it is even less compositional than both of the previous representations, and has the problem of making two instances of the same transition have completely different constructors.

3.3 Optimizations

A thing to notice about the representation of the state in Listing 1.3 is that it is immutable, i.e., that it is impossible to change markings of individual places

Listing 1.6: New representation of events.

```
1  structure Bind : sig
2    datatype event =
3      Network'Transmit_Acknow1 of int * {n: INT, success1: BOOL}
4    | Network'Transmit_Acknow2 of int * {n: INT, success2: BOOL}
5    | Network'Transmit_Packet of
6        int * {n: INT, p: STRING, success1: BOOL, success2: BOOL}
7    | Receiver'Receive_Packet of int * {k: INT, n: INT, p: STRING, str: STRING}
8    | Sender'Receive_Acknow of int * {k: INT, n1: INT, n2: INT}
9    | Sender'Send_Packet of int * {n: INT, p: STRING}
10   val Top : int
11   val Top'Network : int
12   val Top'Receiver_1 : int
13   val Top'Receiver_2 : int
14   val Top'Sender : int
15 end
```

in a state without creating a completely new state. This is a nice property we can use to make several optimisations. Immutability allows us to use the same representation internally as we expose to the user, as the user is not able to modify the representation. This has the great advantage that we do not need to translate between different representations in a state space tool (as happens in CPN Tools, where the exposed representation of a state is a Node, which is really an integer pointing into a mutable tree). Having the same representation internally and externally also lowers the barrier for users to become developers and experiment with more advanced aspects of state space reduction methods.

The implementation of the most interesting function from the interface in Listing 1.2, nextStates is implemented as in Listing 1.7. The setState function (not shown) basically copies the state record into the simulator. execute contains a large switch, which calls the correct CPN'occfun with the right parameters, and getState (not shown) reads the simulator representation and constructs a state record. The implementation is in fact slightly more intelligent. setState and getState keep track of the latest state record copied to/from the simulator. This improves performance a lot, in particular when doing depth-first traversal, as we will, most of the time, want to compute successors of a successor of the state currently stored in the simulator. As we have already calculated successors of this state and do not change it, the simulator is able to use locality to more efficiently calculate the desired successors. By exploiting immutability of the state record we can re-use parts of it to do even better by combining it with locality to implement BDD-like data-structure, which is essentially a faster but less memory efficient implementation of the tree-based storage of CPN Tools [2]. Assume we are given a state-record, e.g., the initial state from Listing 1.4. When we execute the Send Packet transition on Sender, we know (statically), that we can only change A and Limit on Top. We can thus re-use the representation of all other places at the top level and the representation of all sub-pages by making getState used in Listing 1.7 dependent on the event. This not only alleviates the need to transfer state from the simulator to the new state records, it also makes equality tests faster by reducing to pointer comparison for sub-pages and unchanged places. Furthermore, re-using old representations conserve memory. This does not ensure that we only store the multi-set 1‘1 once (and is hence not as memory efficient as the representation of CPN Tools), but on the other hand does not spend any time trying to unify multi-sets that are almost the same. This can also be exploited in the other direction. When asked to compute successors for a certain state, we only need to transfer pages and places that have actually changed (by changing the implementation of setState used in Listing 1.7). All of this can be done completely independently of the interface, without making explicit whether the interface is implemented in the most naive way or whether locality-optimisations take place (except for faster execution in the latter case).

Listing 1.7: Implementation of nextStates function.

```
1 fun nextStates (state, event) =
2   (setState state; execute event; getState())
```

3.4 Auxiliary Functions

In order to provide the interface in Listing 1.2, we need to generate model-specific functions; basically the getState, execute, and setState functions used in Listing 1.7. Furthermore, we need to generate the Mark and Bind structures. The CPN simulator contains a set of tables, which can be used to inspect the model, but these tables are optimized for incremental syntax-check and fast simulation, and are therefore not very easy or fast to traverse. We have therefore developed an interface to the static part of the model, i.e., the pages with places, transitions, arcs, and all annotations of each. This interface can also be used for other purposes. We have already used it to generate model-specific hash-functions, marshaling of states and events, and ordering of states and events.

The generated hash-functions calculate hash values inductively in the structure of the model. We build "strings" on several levels, from multi-sets as strings of tokens (which may again be strings of simpler values), over pages as strings of places (multi-sets), to models as strings of pages. Using a simple combinator function which can calculate the hash value of a string given the hash values of each of it elements and hash-functions for all simple types, we can calculate a hash value for an arbitrary CPN model in a very efficient way. Furthermore, by using different combinator functions, we can efficiently generate multiple linearly independent hash functions. Such hash functions are useful for many things, such as putting states into hash tables (implementing full state space traversal), storing only a hash-value for each state (implementing hash compaction), or using the hash-value to set a bit in a bit-array (implementing bit-state hashing).

Marshaling is implemented using a strategy similar to the hash function. If we know how to store each character of a string, we can store the entire string by writing the length of the string and each character. Marshaling is useful for storing states to disk (implementing various disk-based state space traversal algorithms), or for transmitting states over a network (implementing distributed state space traversal).

Ordering is also implemented using the same strategy, by basically inductively defining a lexicographical order. Orders are useful for storing states to disk, as it is often useful to sort states when storing them on disk. It is also useful for storing states in search trees, which is used by many algorithms built into Standard ML, such as algorithms for calculating strongly connected components of graphs, which is useful for determining certain liveness properties of CPN models.

4 The Java CPN Model Interface

As mentioned in the introduction, many applications can benefit from tight integration with CPN models and the CPN simulator. If such applications are algorithmic in nature, we suggest using the SML interface described in the previous section, as it does not have the overhead of communication via TCP/IP. For most other applications, we propose that the Java interface described in this

section is used as the overhead is irrelevant for many applications. The Java interface provides a high-level object-oriented representation of CPN models as well as an implementation of the protocol used by the CPN Tools graphical editor to communicate with the CPN simulator. As we furthermore provide an importer package that is able to read models created with CPN Tools, this interface makes it possible to create tools that load, manipulate, and simulate CPN models. Applications with these purposes often need to provide a user-friendly user interface or integrate with other applications. For these reasons, we have decided to create this interface in Java, which is widely used and provides many frameworks and tools for creating user-friendly applications.

4.1 Object Model

The CPN object model is a cleaned-up re-implementation of the model of the BRITNeY Suite [16], created for the ASAP model checking platform [11]. ASAP builds on the Eclipse platform [4], and so it is natural to use Eclipse frameworks for the implementation of the Java interface. In order to improve interoperability with other tools, we also support the ISO/IEC 15909-2 transfer format standardisation effort [7].

Our object model builds on version 1.1.5 of ISO/IEC 15909-2, in particular the *PNML Core Model* (Fig. 2 in [7]) and the *High-Level Core Structure* (Fig. 8 in [7]). In addition, we have added some extensions for CPN Tools specific features (to support CPN Tools' concept of time and code segments for transitions). In order to not pollute the basic model, we have basically implemented the PNML Core Model, and added features from the High-Level Core Structure and the CPN Tools specific extensions as add-ins. We have also extended the PNML Core Model with a simplified version of *Modular PNML* [9] to support hierarchical nets. The resulting object model can be seen in Fig. 5. Basically, we have a PetriNet at the top left corner. A Petri net can contain one or more Pages (middle left), which can contain any number of Arcs and Objects (middle). Objects are basically Places and Transitions (bottom). Additionally, objects can be Instances, which basically correspond to substitution transitions in CPN Tools. Objects can have any number of Labels (middle top), which are annotations, that correspond to initial markings, place types, arc inscriptions, names, guards (or conditions), code segments, and time inscriptions (middle from left to right). Places, transitions, and arcs each have one or more add-ins (classes with dark gray background), which basically allows them to have typed access to their annotations. Annotations also have an add-in, which makes it possible to store a structured version of the annotation as well as a plain text version. The Annotations package with the light gray background at the top right is basically an implementation of the High-Level Core Structure except that we have added Time and Code annotations. The white classes outside of this package basically implements the PNML Core Model. The Instance and ParameterAssignement are simplified versions of ModInstance and ParamAssign (renamed to remove abbreviations). The change is that where Modular PNML introduces a concept of modules and import nodes, we just use the already defined concepts of page and

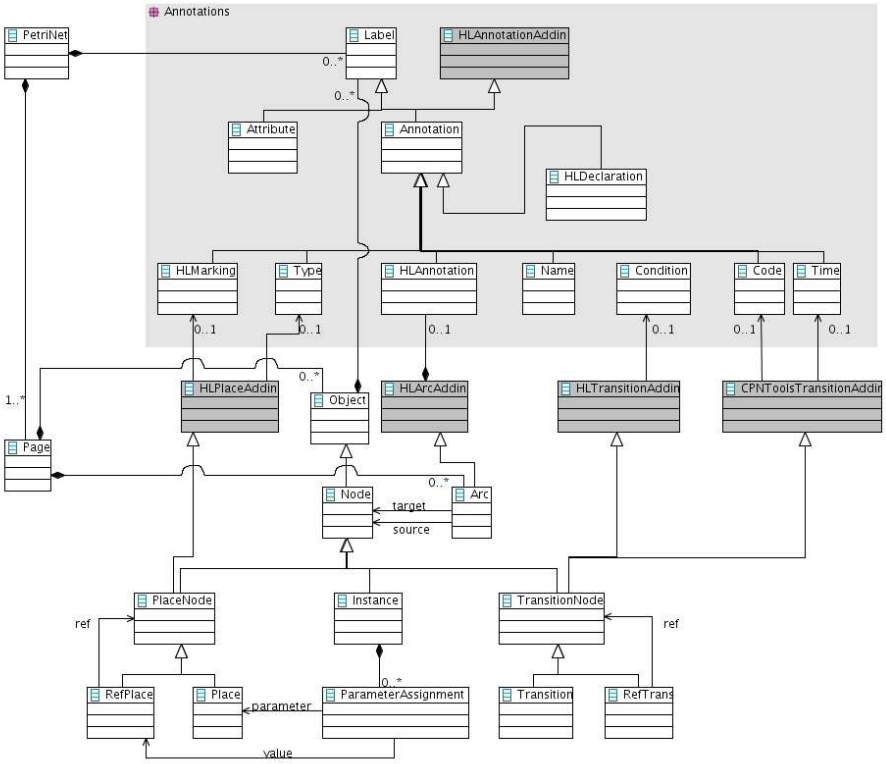


Fig. 5: Object model for CP-nets in the Java interface

place (as we only allow place-bordered modules). Furthermore, our Instance class is a Node and not just an object as CPN Tools allows arcs to and from substitution transitions. Finally, the place and transition add-ins do not contain their annotations (as they do in High-Level Core Structure), but just refer to them, as objects already contain labels and the add-ins merely provide typed access to these. We also have a few add-ins not shown in the figure. One adds an identifier to pages, arcs, labels and objects, and another adds names to pages and objects. Finally, we have an add-in for tool-specific information to Petri nets, objects, and labels.

The actual implementation of the object model is done using the Eclipse Modeling Framework (EMF) [5], which is a framework for implementing object models. EMF can generate implementation code from Java interfaces or from an UML diagram [13]. EMF is furthermore able to generate Java interfaces and UML diagrams from the model as well. In our case, we have described the model using Java interfaces, and the UML diagram in Fig. 5 is automatically generated from the model. In addition to automatic implementation, EMF also provides some nice features, such as automatic generation of XML marshaling

and unmarshaling as well as an adapter functionality which is an extension of an observer architecture [6, Chap. 5]. This makes it possible to observe the object model for changes which is useful for editors, and to attach adapters adding new functionality to the classes.

CPN Tools Importer. Instances of the object model in Fig. 5 can be generated programmatically. It is of course desirable to create such models using a graphical user interface instead. For this reason we have created an importer, which allows programmers and users to import models created with CPN Tools.

The importer only imports the net-structure of the model but is prepared to support the graphical information as well, as we have made a preliminary implementation of the *Graphical Information* (Fig. 3 in [7]). All labels except for HLDeclarations are loaded as flat text; HLDeclarations use a structure similar to the *TermsUserDeclarations* (Fig. 17 in [7]), but the details are not shown here.

4.2 Protocol Implementation

The CPN Tools GUI communicates with the simulator process using a custom protocol. The protocol is an implementation of a remote procedure call (RPC) system [3, Chap. 5.3]. The protocol sends packets over a TCP/IP stream. Packets are transmitted in the custom BIS (boolean, integer, string) format, which is a binary packet format that basically takes care of marshaling of simple data types. Packets have an opcode which indicates the type of packet. CPN Tools primarily uses two opcodes, namely 1 (evaluate SML code) and 9 (RPC request). Packets with opcode 1 just contain a string to be sent for evaluation. Packets with opcode 9 have an additional integer to indicate which command to execute and sometimes another integer to determine a sub-command. Such commands must be combined in the correct way to syntax check an entire CPN model and generate simulator code for it.

In order to implement this protocol, one must implement the BIS packet format as well as high-level constructs translating to the lower-level command and sub-command integers, which is a tedious and error-prone job. Finally, we need to construct a component that can take a CPN object model and correctly send it to the simulator for syntax check and simulation. In Fig. 6 we see how this has been implemented in the Java interface. We see five packages. cpn.model represents the object model from Fig. 5, and cpn.model.importer is a package implementing an importer able to load a file created using CPN Tools. The class Job, which is outside of any of the packages, is part of Eclipse. The remaining three packages implement the protocol used to communicate with the CPN simulator. The classes are listed with the most high-level at the left. Only the classes at the top are meant to be used by most implementers. At the bottom-right, we have Packet, which implements the BIS package format. Such packets can be sent to a Simulator. The Simulator uses a delegate DaemonSimulator to communicate with the simulator via TCP/IP in the same way as CPN Tools. The Simulator

class provides communication at the level of packets. The HighLevelSimulator provides stubs for all the calls supported by the simulator, and it is thus possible to communicate with named methods. It uses a PacketGenerator factory to actually create the packets it needs. The Checker class ties this to the object model hierarchy, and makes it possible to perform higher-level operations, such as syntax checking all declarations of a model. CheckerJob further lifts this and makes it possible to syntax check an entire net using a single call. The checker job integrates with the Eclipse platform and can provide feedback to the user. If this is not desired, one can use the simpler Checker class, which can be used independently of the platform used. For operations other than checking (such as simulation), one must go to the HighLevelSimulator. One will very rarely need to consider the Simulator, PacketGenerator, and their underlying classes.

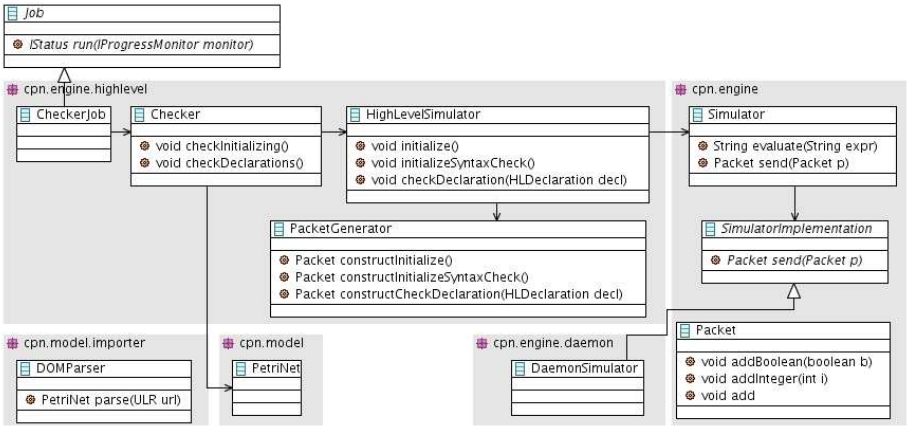


Fig. 6: Implementation of the protocol used to communicate with the simulator

5 Examples

In this section we show how to use the aforementioned interfaces by implementing a simple state space exploration tool that can check a model for dead-locks from the command-line. We first show the SML code implementing the traversal algorithm using the SML interface from Sect. 3, and then turn to the Java code for the command-line application loading a model and launching the exploration.

5.1 State-space Exploration

The implementation of the state space exploration algorithm can be seen in Listing 1.8. We actually implement an algorithm parametrised with a state property, so it is possible to check for other properties than dead-locks. The algorithm basically performs a recursive depth-first traversal of the state space and stores

already expanded states in a hash-table. If a state not satisfying the property is found an exception is raised. The code starts (l. 1) by defining an exception to raise if a violating state is found. Then the built-in parametrised hash-function is instantiated. Then follows the implementation of the actual algorithm (ll. 6–35), which takes a predicate to apply to each state and a list of states from which to start the exploration. The function first defines the storage using SML's built-in HashTable (ll. 8). Then two mutually recursive functions dfs' and dfs'' are defined. dfs' (ll.20–31) traverses a list of states. It starts by checking if we have already traversed the state (l. 22), and, if so, continues with the next state (l. 23). If the state is new, it is stored (l. 25) and the predicate is checked (l. 26). If the predicated is violated, the exception is raised (l. 29). Otherwise we call dfs'' with the state before continuing with the rest of the states. dfs'' takes care of exploring successors resulting from executing all enabled events for a given state. It basically calculates successor states for each event (l. 14), and explores them using dfs' (l. 15) before traversing the rest of the events (l. 17). The entire function just calls dfs' with the given state(s). If no exception is raised, we return that no

Listing 1.8: Implementation of a simple state space exploration algorithm.

```
1  exception Violating of CPNToolsModel.state
2
3  fun combinator (h2, h1) = Word.<<(h1, 0w2) + h1 + h2 + 0w17
4  val hash = CPNToolsHashFunction combinator
5
6  fun dfs predicate states =
7  let
8    fun equals (a, b) = a = b
9    val storage = HashTable.mkTable (hash, equals) (1000, LibBase.NotFound)
10
11   fun dfs'' state [] = ()
12     | dfs'' state (event::events) =
13       let
14         val successors = CPNToolsModel.nextStates (state, event)
15         val _ = dfs' successors
16       in
17         dfs'' state events
18       end
19
20   and dfs' [] = ()
21     | dfs' ((state, events)::rest) =
22       if Option.isSome (HashTable.find storage state)
23       then dfs' rest
24       else let
25              val _ = HashTable.insert storage (state, ())
26              val violates = predicate (state, events)
27            in
28              if violates
29              then raise Violating state
30              else (dfs'' state events; dfs' rest)
31            end
32 in
33   (dfs' states; (NONE, storage))
34   handle Violating state => (SOME state, storage)
35 end
36
37 fun none _ = false
38 fun dead (_, events) = List.null events
```

state violating the property was found, and the storage (l. 33). If an exception is raised, we also return the state violating the property. The last part of the listing contains a predicate that is never satisfied (l. 37) and one that checks for dead-locks (l. 38). The first is useful for performance testing, as it forces a full generation.

We have tested this implementation against the one built into CPN Tools. By varying the number of packets to transmit in the CPN model in Figs. 3 and 4 (altering the marking of the Send place) from two and upwards, we see that this implementation is 50-290 times faster (for 4-19 packets), discovers the same number of states as CPN Tools, and is able to explore larger state spaces than CPN Tools ($3.0 \cdot 10^6$ states when transmitting 25 packets compared to CPN Tools' $1.7 \cdot 10^6$ states when transmitting 19 packets).

5.2 Command-line State-space Analyser

To keep the example short, we use a simple implementation strategy. We load the model given as the first parameter, load the SML code shown in the previous example, which we assume is stored in a file simple-dfs.sml. Finally, we perform the exploration and show the result to the user. The implementation can be seen in Listing 1.9. We start by importing some classes needed (ll. 1–9). The rest of the code is the class implementing our state space tool. The class starts by obtaining the name of the file to analyse (l. 13). The file is loaded as a Petri net (l. 14), and we create a HighLevelSimulator. As we are running this outside of an Eclipse run-time environment, we need to supply a simulator manually. The simulator requires a delegate, which requires information about which host and port to connect to as well as the name of the run-time system to load. All of this takes place in ll. 16–18. If we are using the interface as part of an Eclipse application, we can just use the simplified version in l. 15, which obtains all parameters from a preference pane exposed to the user. We then create a new CheckerJob (l. 20), which requires a name (we just give it the name of the file), a Petri net, and a high-level simulator. We start (schedule) the job and wait for it to terminate (ll. 21–22). We then load the state-space algorithm developed previously (l. 23), and launch an exploration (ll. 24–30). We process the result of the exploration so the result we show the user is the violating state (if any) and the number of nodes explored. When we are done, we destroy the simulator (l. 32). This is needed as the simulator starts an external application, which should be shut down as well as a couple of Java threads for communication. By destroying the simulator we make sure to clean this up. If we quit the application (such as pressing the cross in a graphical application), this is performed automatically, but for this command-line application do this manually in order to terminate the program when the exploration is done.

The command-line tool can be executed as java StateSpaceTool protocol.cpn, and shows the first encountered dead-lock if there is one as well as the number of states stored.

Listing 1.9: Implementation of a command-line state space exploration tool.

```
import java.io.File;
import java.net.InetAddress;
import java.net.URL;
import dk.au.daimi.ascoveco.cpn.engine.Simulator;
import dk.au.daimi.ascoveco.cpn.engine.daemon.DaemonSimulator;
import dk.au.daimi.ascoveco.cpn.engine.highlevel.HighLevelSimulator;
import dk.au.daimi.ascoveco.cpn.engine.highlevel.checker.CheckerJob;
import dk.au.daimi.ascoveco.cpn.model.PetriNet;
import dk.au.daimi.ascoveco.cpn.model.importer.DOMParser;

public class StateSpaceTool {
  public static void main(String[] args) throws Exception {
    String file = args[0];
    PetriNet petriNet = DOMParser.parse(new URL("file://" + file));
//  HighLevelSimulator s = HighLevelSimulator.getHighLevelSimulator();
    HighLevelSimulator s = HighLevelSimulator.getHighLevelSimulator(
        new Simulator(new DaemonSimulator(
          InetAddress.getLocalHost(), 23456, new File("cpn.ML"))));
    try {
      CheckerJob checkerJob = new CheckerJob(file, petriNet, s);
      checkerJob.schedule();
      checkerJob.join();
      s.evaluate("use \"simple-dfs.sml\"");
      System.out.println(s.evaluate(
        "let " +
        "  val (state, storage) = " +
        "    dfs dead (CPNToolsModel.getInitialStates()) " +
        "in " +
        "  (state, HashTable.numItems storage) " +
        "end"));
    } finally {
      s.destroy();
    }
  }
}
```

6 Conclusion and Future Work

In this paper we have described two interfaces to the CPN Tools simulator. One is very close to the simulator and written in Standard ML, and provides fast access to the simulator. The interface is useful for analysis methods and other algorithmic applications requiring little user-interaction. The other interface is written in Java and provides an object-oriented representation of CPN models, a means to import models created using CPN Tools, and high-level abstractions of the communication with the CPN Tools simulator, making it possible to integrate CPN simulation into Java applications, ranging from simple command-line applications to full-fledged graphical applications. Both of the interfaces are available to interested parties. Send an email to `ascoveco@cs.au.dk` for more information.

Future work includes replacing the current event implementation with the indicated hierarchical implementation from Listing 1.5. We can alleviate the syntactical problems by observing that while names of transitions may overlap, they rarely do in practise, so by just opening all structures, we can refer to the transition Receive Acknow. on the Sender page as `Top Sender Receive_Acknow {k, n1, n2}`. For transitions with overlapping names, we still need to use the

very verbose naming, but we find that this is a reasonable price to pay for the more convenient representation.

The current Java interface only supports loading CPN models and syntax-checking them in one action. It would be useful to integrate the incremental syntax-checking capabilities of the simulator with the adapter functionality of the object model, so that whenever the object model is altered, it is automatically syntax-checked, independently of how the model is altered. This would be useful for editors, but also for applications generating models, as they are automatically checked for correctness and ready to be simulated.

References

1. ASCoVeCo Project webpage. Online: `www.daimi.au.dk/~ascoveco/`.
2. S. Christensen and L. M. Kristensen. State Space Analysis of Hierarchical Coloured Petri Nets. *Petri Net Approaches for Modelling and Validation, Lincom Studies in Computer Science 01*, pages 1–16, 2003.
3. G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems - Concepts and Design*. Addison-Weslay, 3rd edition, 2001.
4. Eclipse webpage. Online: `www.eclipse.org/`.
5. Eclipse Modelling Framework (EMF). `www.eclipse.org/modeling/emf/`.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
7. ISO/JTC1/SC7/WG19. Software and System Engineering—High-level Petri nets—Part 2: Transfer Format, version 1.1.5.
8. J.B. Jørgensen and K.B. Lassen. Aligning Work Processes and the Adviser Portal Bank System. In *REBNITA'05*, 2005.
9. E. Kindler and M. Weber. A Universal Module Concept for Petri Nets—an implementation-oriented approach. *Informatik-Berichte*, (150), June 2001.
10. L.M. Kristensen, P. Mechlenborg, L. Zhang, B. Mitchell, and G.E. Gallasch. Model-based Development of a Course of Action Scheduling Tool. *STTT*, 10(1):5–14, 2007.
11. L.M. Kristensen and M. Westergaard. The ASCoVeCo State Space Analysis Platform: Next Generation Tool Support for State Space Analysis. In *Proc. of 8th CPN Workshop*, volume 584 of *DAIMI-PB*, pages 1–6, 2007.
12. L.M. Kristensen, M. Westergaard, and P.C. Nørgaard. Model-based Prototyping of an Interoperability Protocol for Mobile Ad-hoc Networks. In *Proc. of IFM'05*, volume 3771 of *LNCS*, pages 266–286. Springer-Verlag, 2005.
13. Object Management Group. Unified Modeling Language (UML), Version 2.1.1. Online: `www.omg.org/technology/documents/formal/uml.htm`, 2007.
14. A.V. Ratzer, L. Wells, H.M. Lassen, M. Laursen, J.F. Qvortrup, M.S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In *Proc. of ATPN'03*, volume 2679 of *LNCS*, pages 450–462. Springer-Verlag, 2003.
15. J.D. Ullman. *Elements of ML Programming*. Prentice-Hall, 1998.
16. M. Westergaard and K.B. Lassen. The BRITNeY Suite Animation Tool. In *Proc. of ATPN'06*, volume 4024 of *LNCS*, pages 431–440. Springer-Verlag, 2006.

Modeling Bus Communication Protocols Using Timed Colored Petri Nets — The Controller Area Network Example

Marko Bago[1], Nedjeljko Perić[1], and Siniša Marijan[2]

[1] University of Zagreb,
Faculty of Electrical Engineering and Computing,
Unska 3, 10000 Zagreb, Croatia
{marko.bago,nedjeljko.peric}@fer.hr
http://www.fer.hr
[2] Končar - Electrical Engineering Institute,
Fallerovo šetalište 22, 10000 Zagreb, Croatia
sinisa.marijan@koncar-institut.hr
http://www.koncar-institut.hr

Abstract. Engineers in industry usually design new systems based on earlier experience and available development tools. Unfortunately, software tools that enable industrial users insight into systems' inner behavior before the production, are still not in everyday use. However, such tools, when used properly, can save both time and money.
In this paper a system based on Controller Area Network is modeled using timed colored Petri nets. This system is verified for the desired properties and then it is validated. Validation is done using two real-life vehicle control units used in light rail applications. The results, as well as possible future use of the model, are presented.

Key words: Controller Area Network, timed colored Petri net, modeling, simulation, verification, validation

1 Introduction

Distributed systems are based on communication networks. Different communicating entities (nodes) interact with each other. The interaction becomes more complex as the system grows. Developing a distributed system also means developing an adequate communication network, one that is able to support all the requirements set by each individual node.

During recent light rail vehicle development CAN was used as a fieldbus, [1, 2]. Testing the functionality of such a system required *Software Simulation Tools* (*SSTs*) and, in the end, real-life equipment. SSTs allow for easier and faster development of a system than real-life equipment. Unfortunately, communication SSTs are usually based on a single communication protocol. This raises at least two problems. First, if the system uses more than one communication protocol, it is necessary to use more than one SST. The interaction between the tools is

either difficult or impossible. Second, using more than one SST increases the cost, and the development engineers have to be familiar with several different development environments.

One solution to these problems would be a single SST with support for multiple communication protocols, [3, 4]. In order to achieve this, a development environment for the SST has to be defined. This environment should fulfill the following requirements: it should be flexible, user friendly, and it should support verification and validation of communication system models.

"Whereas verification checks a model against a given specification, validation checks a model against the modeled system.", [6].

Verification of a system can only be performed if the system is modeled using some formal method.

In [5] it is correctly observed that *"A model is always a reduced rendering of the system that it represents."*. Five key characteristics for a *Model-Driven Development* (*MDD*) of software are given: abstraction, understandability, accuracy, predictiveness and inexpensiveness. These five characteristics are universal to all modeling and can be recognized in the requirements for the SST, e.g. understandability can be understood as a part of user friendly requirement, accuracy and predictiveness are in fact support for verification and validation, etc.

Petri nets fulfill all of these requirements. The graphical representation of the net gives the user an easier understanding of the modeled process, and system models can be verified because Petri nets are a formal method. For a correct validation, a real-life system should be modeled. Results obtained from the simulation model and from the real-life system should be compared. For communication protocols two factors are important for the validation: (i) order of messages gaining bus access must be identical and (ii) message propagation time has to be equal to that of the real-life system.

This paper presents a way to model *Controller Area Network* (*CAN*) communication protocol using timed *colored Petri nets* (*CPN*), [7–9]. A short introduction to the CAN bus is presented in Sect. 2. In Sect. 3 a short introduction to hardware and software tools used to create, verify and validate the model, is given. Section 4 presents the CPN model of the CAN bus created using *CPN Tools*. All modules used to create the CAN bus network are explained. Section 5 presents verification and validation methods used on the model and explains how and why they are used. This section also presents the results of the verification and validation processes. Verification was achieved using state space queries within CPN Tools. Validation of the model was done using two real-life *vehicle control units* (*VCUs*) used in light rail applications. Section 6 concludes the paper and gives a brief overview of the future work.

2 CAN bus communication

CAN was originally developed for automotive applications in the early 1980's by Robert Bosch GmbH. The CAN protocol was internationally standardized by

ISO (International Organization for Standardization) and SAE (Society of Automotive Engineers). Today, CAN is used in many markets, like motor vehicles, industrial automation, medical equipment etc.

CAN bus is an event-triggered, multi-master, bus communication system with priority-based access control and automatic retransmission of corrupted frames, [7, 8]. Frames are labeled with identifiers (11-bit or 29-bit) that are used to determine both the priority and the content of a frame. Frame identifiers (ID) are transmitted together with useful user data. There are no node addresses. Two bit levels exist on the bus, dominant and recessive. The dominant bit level (logical 0) overwrites the recessive bit level (logical 1). The CAN bus bit rate can be up to 1 Mbit/s.

CAN protocol uses four different frame formats for the communication:

1. *Data frame* - carries data from the transmitter to all receivers on the bus. Data is labeled with a message ID.
2. *Remote frame* - used to request transmission of a data frame with identical message ID.
3. *Error frame* - transmitted by any node (transmitter or receiver) in case of a bus error detection.
4. *Overload frame* - used for providing an extra delay between the preceding and succeeding data or remote frames.

Data frames and remote frames are separated from the preceding frames by an interframe space. There are two parts of interframe space, intermission and bus idle. Intermission is inserted after data or remote frame and is 3 bits (recessive) long. During intermission no node is allowed to send either data or remote frames. Only sending of an overload frame is allowed. Bus idle may be of arbitrary length and only recessive values are on the bus.

CAN frames. A CAN frame is simultaneously accepted either by all nodes or by none.
Data frames are composed of seven different bit fields, as in Fig. 1.a and Fig. 1.c: *SOF (Start Of Frame)* - 1 bit, *arbitration field* - 12 bits (11-bit ID) or 32 bits (29-bit ID), *control field* - 6 bits, *data field* - [0-8] bytes, i.e. [0-64] bits, *CRC field (Cyclic Redundancy Check)* - 16 bits, *ACK field (Acknowledge)* - 2 bits and *EOF (End Of Frame)* - 7 bits.
Remote frames are composed of six different bit fields, as in Fig. 1.b and Fig. 1.d: *SOF* - 1 bit, *arbitration field* - 12 bits (11-bit ID) or 32 bits (29-bit ID), *control field* - 6 bits, *CRC field* - 16 bits, *ACK field* - 2 bits and *EOF* - 7 bits.
Error frame is composed of two fields, as in Fig. 1.e and Fig. 1.f: *error flag field* - [6-12] bits and *error delimiter* - 8 bits.
Overload frame is composed of two fields, as in Fig. 1.e and Fig. 1.f: *overload flag field* - [6-12] bits and *overload delimiter* - 8 bits.

Frame coding. The following bit sequences (fields) will be coded by the bit stuffing method: SOF, arbitration field, control field, data field and CRC sequence. Whenever a transmitter detects five consecutive bits (including stuff

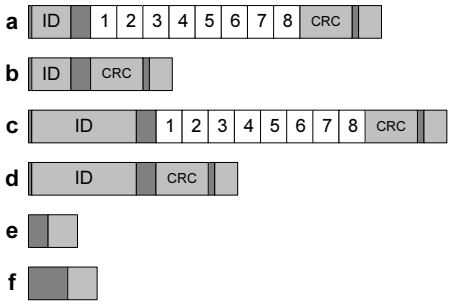


Fig. 1. Following frames are presented: **a)** 11-bit ID, 8 byte Data frame; **b)** 11-bit ID remote frame; **c)** 29-bit ID, 8 byte Data frame; **d)** 29-bit ID Remote frame; **e)** 6-bit flag field error/overload frame; **f)** 12-bit flag field error/overload frame. The data fields are colored white.

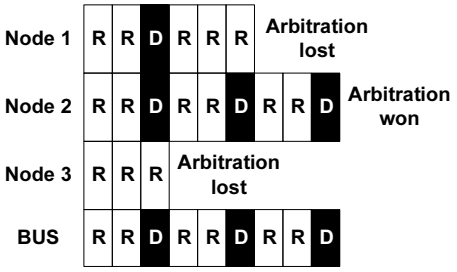


Fig. 2. An example of an arbitration process on the CAN bus. The arbitration is among three competing nodes. *Node 3* is the first to lose the arbitration, while the *Node 2* won the arbitration. *BUS* shows values present on the CAN bus.

bits) of identical value in the bit stream, it automatically inserts a complementary bit into the stream. The receiver automatically destuffs this bit from the received bit stream.

Bus access. Every node has the right to access the idle bus at any moment in time. SOF (data or remote frame) marks the beginning of bus access and contention-based arbitration takes place. Every transmitter compares the bit being transmitted with the bit on the bus. If a recessive bit is being sent, but a dominant bit is detected, the node loses arbitration and will not send any more bits, Fig. 2. The node that lost the arbitration becomes a receiver. If a data and a remote frame are sent to the CAN bus at the same time, and if the two frames have identical identifiers, then the data frame will win the arbitration process, i.e. it will gain access to the CAN bus first.

Error detection and handling. The following mechanisms are provided for error detection: *monitoring*, *stuff rule check*, *frame check*, *15-bit CRC*, *ACK check*.
Monitoring: A node sending a bit on the bus monitors the bus at the same time. If the value sent is different from the value detected then an error is detected. Exceptions happen during arbitration, during an ACK slot and while sending an error passive flag.
Stuff rule check: A stuff error is detected during the sixth consecutive bit of equal level in the frame field coded by the bit stuffing method (SOF, arbitration field, control field, data field and CRC sequence).
Frame check: This error is detected if one or more illegal bit values is detected in a bit field, e.g. during EOF the node detects a dominant bit and only 6 recessive

bits instead of 7.
15-bit CRC: CRC is calculated by the receiver in the same way it is calculated by the transmitter. If the values do not match then an error is detected.
ACK check: An ACK error is detected by the transmitter whenever a dominant bit is not detected during the ACK slot.
Errors are registered and error frames are automatically retransmitted.

In this paper 11-bit message IDs and 500 kbit/s were used for CAN communication.

3 Tools

Creation, verification and validation of a simulation model requires different tools. The tools can be divided into two groups, hardware and software tools, i.e. hardware equipment and a computer with appropriate programs. In order to create, verify and validate a timed colored Petri net model, CPN Tools is used, [10]. Validation also requires a real-life system. The real-life system was composed of two vehicle control units (VCUs) that are used in light rail applications, [2].

3.1 Hardware tools

Hardware used for validation purposes consists of three CAN communication nodes. Two CAN communication nodes are VCUs used for control of TMK2200 trams operating in the city of Zagreb, Croatia, Fig. 3, [1]. The third node is a general purpose CAN communication node connected via a USB cable to a laptop PC. It is used to start the message exchange between VCUs and to log all data traffic on the CAN bus. All logged messages are time stamped with 1 μs resolution. The hardware validation test system is given in Fig. 3.

Fig. 3. The assembled test system consisting of a VCU and a PC on top of it.

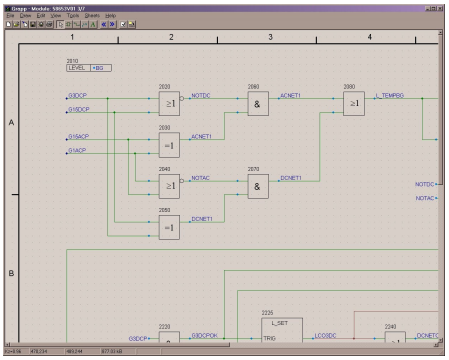


Fig. 4. The integrated development environment for the programming of the VCU.

Vehicle Control Unit. VCU is a twin-channel, multiprocessor system that supports sequencing, protection, regulation, diagnostic and communication functions, [2].

Integrated Development Environment (IDE). User programs are developed by the engineers that are application oriented. To support them, an IDE based on block diagrams was developed, Fig. 4. This IDE was used to program the validation system.

3.2 Software tools

CPN Tools software was used to create, verify and validate timed colored Petri net models. CPN Tools is a program developed and maintained by the CPN group from University of Aarhus, Denmark. The program is capable of creating hierarchical timed colored Petri net models, [10]. Petri nets are represented by the graphical layout, while additional information and interaction of the model comes from the CPN ML programming language. CPN ML is based on the Standard ML programming language, [11, 12]. Using CPN ML, it is possible to define complex data structures and functions to handle these structures.

CPN Tools is based on the formally defined syntax of colored Petri nets. The semantics, i.e. the behavior of the net, is also defined. The defined semantics enable simulation of the model. Simulation can be interactive, i.e. with user intervention, or automatic. Automatic simulation enables validation of the system. Since both the syntax and the semantics of the CPN models created using CPN Tools are defined, it is possible to generate the full state space of the models. The state space can be queried. Queries have to be written using CPN ML programming language. Queries enable verification of the desired properties of the system.

4 Model

The concept of the bus communication system is given in Fig. 5. The nodes have an identical structure. This means that it is possible to create one node structure, and reuse it to model multiple nodes. The modular approach is also used for message generators, i.e. message formatting.

Actual data to be transferred is not modeled. It is possible to abstract the real data from the model since, in this case, the data is not used for any sort of control of the system. A model of a higher level protocol based on CAN, e.g. CANopen, would require the actual data. The final CANopen model would look a bit different, but it could be based on the CAN model presented in this paper.

Since time is used for modeling, it is necessary to define how much real time is represented in a single simulation time unit. In this paper the single simulation time unit represents 10 ns of real time.

The top level of the model, given in Fig. 6, contains: two nodes (substitution transitions *Node_1* and *Node_2*); three places (*Node2Bus*, *BusFree* and

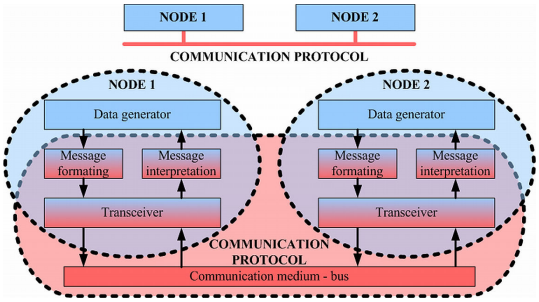


Fig. 5. The concept of the bus communication systems.

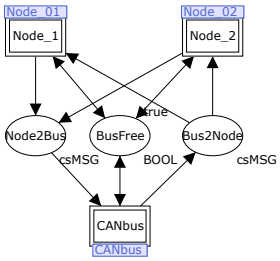


Fig. 6. The top level of the CAN bus communication model.

Bus2Node) that represent the state of the CAN bus; and a substitution transition (*CANbus*) that handles messages on the CAN bus.

The first node is configured as given in Fig. 7. The node sends 9 messages. All messages are to be sent with a 10 ms period. The length of messages is defined by the parameter DLC (number of data bytes), while ID defines message identifier.

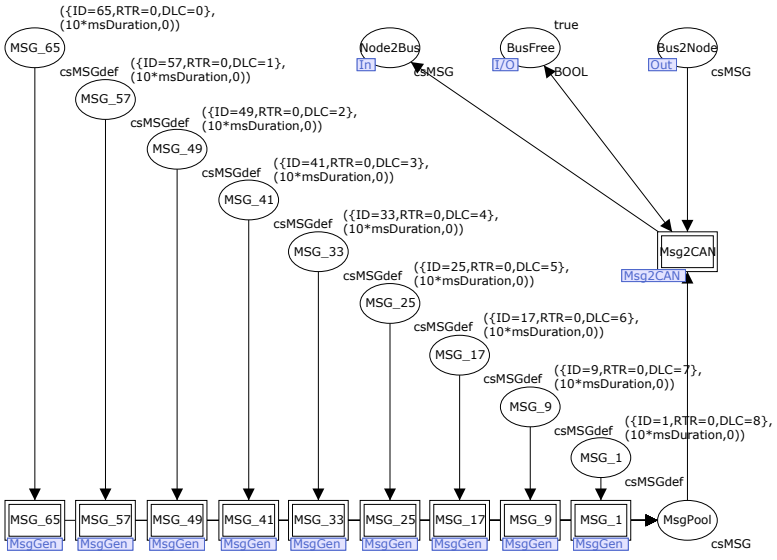


Fig. 7. The CAN node, substitution transition label *Node_01*, Fig. 6.

The second node is configured as given in Fig. 8. The node sends 9 messages. All messages are to be sent with a 10 ms period. The length of messages is

defined by the parameter DLC (number of data bytes), while ID defines message identifier.

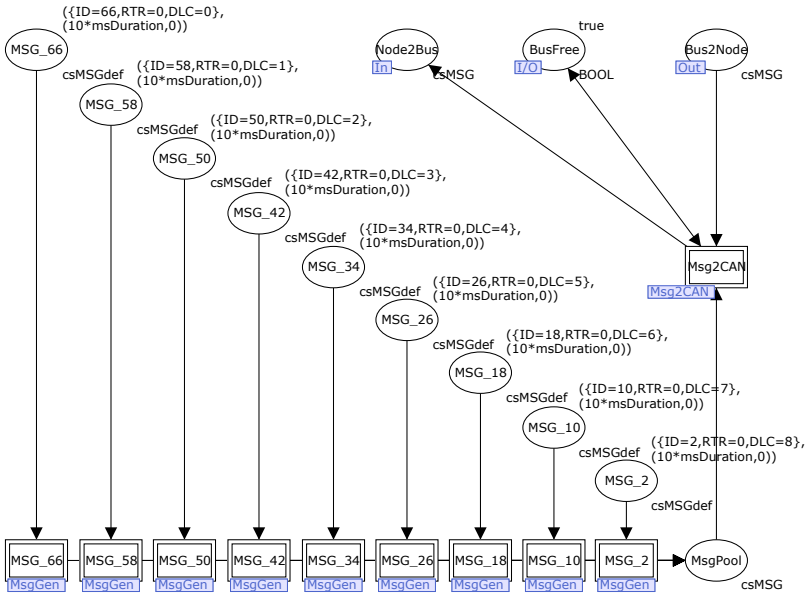


Fig. 8. The CAN node, substitution transition label *Node_02*, Fig. 6.

Each node uses nine message generators, modelled by substitution transition *MsgGen* in Fig. 7 and Fig. 8. Besides message generators, each node uses a substitution transition that forwards messages from the node to the CAN bus (substitution transition label *Msg2CAN*).

The message generator, given in Fig. 9, creates a single shot or periodic messages for the node to transfer. It is important to be able to create both types of messages, since start-up procedures and similar actions use one-time messages, while the system in operation usually uses periodic messages. Alarms or error situations in the system can be communicated by the one-time messages too.

Two functions are present in the arc inscriptions originating from the transition *ReadMsg*, Fig. 9. The first one is *abs(int)*. This built-in function returns the absolute value of an integer *int*. The second function is *fGenFrame(ID, RTR, DLC)*. This is a user defined function and it creates a CAN message frame based on the identifier (*ID*), type of frame (data or remote, *RTR*), and number of data bytes (*DLC*).

The system used to transfer generated messages to the CAN bus is given in Fig. 10. It is composed of two substitution transitions. The first one, *MSG2RAM*, takes generated messages and places them in the input FIFO memory buffer, just like a regular communication processor. The second one, *Transceiver*, takes

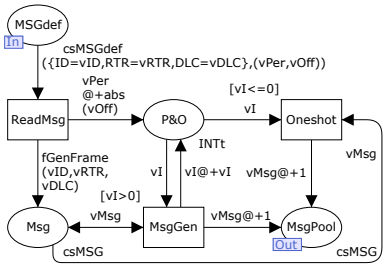


Fig. 9. The CAN message generator, substitution transition label *MsgGen*, Fig. 7 and Fig. 8.

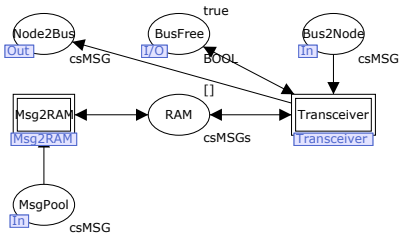


Fig. 10. The system used to transfer messages from the node to the CAN bus, substitution transition label *Msg2CAN.*, Fig. 7 and Fig. 8.

the first message from the input FIFO buffer and sends it over the CAN bus. *MSG2CAN* is structured that way in order to enable easier simulation of different message sorting mechanisms.

In *MSG2RAM*, simultaneously generated messages, e.g. the messages generated in the same interrupt routine, are sorted according to their priority, Fig. 11. The highest priority message (lowest ID) is on the top of the list, while the lowest priority message (highest ID) is on the bottom of the list. The entire sorted list is appended at the end of the input FIFO memory buffer. Thus the input FIFO buffer is not sorted, i.e. it can happen that a higher priority message is behind a lower priority message. The real system behavior is simulated that way. Other system behaviors can be simulated too, e.g. a system with a priority sorted input FIFO buffer.

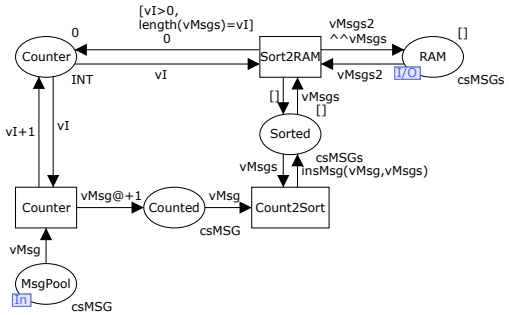


Fig. 11. The system used to transfer messages from RAM to the CAN transceiver, substitution transition label *Msg2RAM*, Fig. 10.

Substitution transition *MSG2RAM* has two functions on the arc inscriptions originating from transitions *Sort2RAM* and *Count2Sort*, Fig. 11. The first one is

a built-in function *length(lst)*. It returns the length of the list *lst*, i.e. the number of elements on the list. The second function is a user defined *insMsg(msg, lst)*. This function inserts a message *msg* to the priority sorted list of messages *lst*.

Figure 12 shows the message transceiver. It takes only one message at a time. The message is taken from the top of the input FIFO buffer. The transceiver sends the message to the CAN bus (place *Node2Bus*) only if the bus is available. The state of the CAN bus is defined in the place *BusFree*. The CAN bus can be either available (*true*) or occupied (*false*).

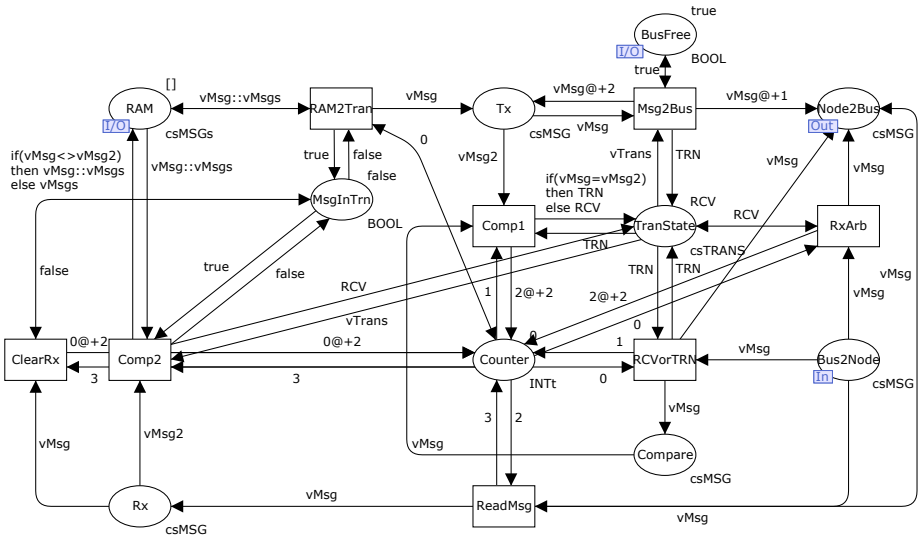


Fig. 12. The model of the CAN transceiver, substitution transition label *Transceiver*, Fig. 10.

When the node gains access to the bus, it sends the message from the transmit register (place *Tx*). If the message has the highest priority, compared to all the other messages sent by competing nodes, then the transceiver remains a transmitter. If the message has a lower priority, then the transceiver turns into a receiver. The message is handled by the *CANbus* module and the end result is passed to transceivers through the place *Bus2Node*.

The *CANbus*, given in Fig. 13 handles messages sent by the nodes. The maximum number of messages in the place *Node2Bus* must be equal or smaller than the number of connected nodes. Once the messages are received, the bus changes the state of the *BusFree* place. Next, the messages are sorted according to their priority. The function *insMsg(msg, lst)* adds message *msg* to the sorted list *lst*.

The highest priority message is sent to the place *Bus2Node*, Fig. 13. This way all nodes become aware of the highest priority message and can change

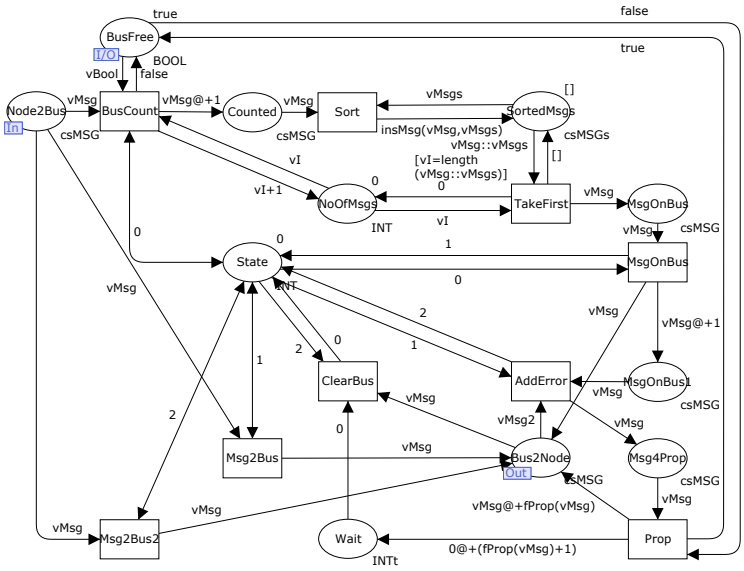


Fig. 13. The model of the CAN bus, substitution transition label *CANbus*, Fig. 6.

their respective state (transmitter or receiver). In order to synchronize the state change of all the nodes it is necessary to add a time delay on the arc between the transition *MsgOnBus* and the place *MsgOnBus1*. The message is then processed by the *AddError* transition. Here complex error behavior of the bus can be modeled. The message, modified or not, is sent to the place *Msg4Prop*. In this paper no errors on the bus were assumed.

Message propagation time is calculated by the transition *Prop*, Fig. 13. Depending on the size of the message, and other parameters influenced by the *AddError* transition, it is calculated how long the message actually occupied the bus. The message is sent to the *Bus2Node* place, where it is read by all the nodes. If there was an error during the transmission, the transmitting node would automatically try to retransmit the message.

The substitution transition *CANbus* uses two user defined functions, Fig. 13. First one is *insMsg(msg, lst)*. This function inserts a message to the priority sorted list. The second function, *fProp(msg)*, is used to calculate the message propagation time. It uses total number of bits in the message frame to calculate the duration on the CAN bus. All delays added in the *CANbus* are compensated for.

The place *State* in Fig. 13 defines the state of the CAN bus as follows: *0* - no message present on the bus; *1* - a single message present on the bus, which is used to inform the nodes about the message that won the arbitration process; *2* - a single message present on the bus which is used to transfer the message to all the nodes.

4.1 Model restrictions

The state machine of the CAN node, that depends on the error counters, was not implemented. Modeling the error counter would be a major difficulty for the verification. TRN - transmit error counter has 256 states. RCV - receive error counter has 128 states. This alone gives $128 * 256 = 32768$ states per node.

If there are occasional errors in the communication system, and we model them, then the model will behave in an identical way as the real-life system. In the case of a heavily disturbed communication (system error) the CAN nodes will eventually go into "bus off" state, [7, 8]. Heavy disturbances are usually caused by physical defects, i.e. short-circuit, wave reflection due to poor termination, disconnected wires etc. This is out of scope of our model.

5 Verification and validation results

5.1 Verification

Verification is used to test the model for desired (or undesired) properties, [18–20]. Since verification of a large system tends to get extremely difficult (due to the state explosion problem), it is possible to use modular analysis of the system, [13–17]. There are different possible approaches to the modular analysis.

It is possible to analyze every module as a separate entity. The boundary conditions should, in such cases, be identical to the ones when the module is part of the system. It is possible to define such conditions and mimic them. Thus, all possible local states of the module are checked.

Three types of modules can be used in a modular analysis approach: a source module, a transport module, and a sink module. A source module generates tokens without any external influence. A transport module transfers the tokens from the input place(s) to the output place(s). It does not generate any tokens on its own. The sink module consumes all tokens it receives from the input(s). The most critical modules are the transport modules. It is necessary for these modules to have independent input interfaces, i.e. if more than one input is present, then the inputs have to cause independent actions within the module.

There are three separate modules for the state space analysis in this paper: (i) message generator - generates messages for the node to send to the CAN bus, (ii) the node - gets messages from message generators and sends them via the CAN bus, and (iii) CAN bus - handles messages that the nodes want to send. In this paper message generator is the source module, CAN bus is the transport module, while the node is the sink module. For the state space analysis of all modules, the following branching condition has been used:

```
fn (n:Node) =>
    if (n=hd(sort INT.lt (EqualsUntimed(n))))
    then
        true
    else
        false
```

This condition limits the generation of the state space. States (nodes in the state space) are compared to each other, but without time marks. If there are two identical states with different time marks, only one will be processed. This is a perfectly legal condition since there are no time controlled events except the generation of the periodic messages. Properties of these periodic messages do not change with time. This means that a message generated at time $T1$ will cause identical system behavior as a message generated at time $T2$, where $T1 \neq T2$, if the state of the system is otherwise identical (time invariant system).

Message generator module. The input place to message generator module is *MSGdef*. The output place is *MsgPool*. The module has no other interaction with its environment, Fig. 14. The CAN message generator can generate two types of messages: (i) single message and (ii) periodically repeating message. The module has to be verified for both types.

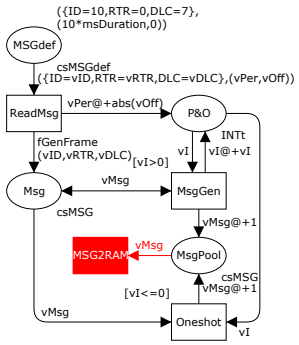


Fig. 14. The model used for verification of the message generator module. Filled transition is used to mimic the behavior of the environment.

The single message generator should have the following properties: (i) only one type of message is generated and (ii) if the module deadlocks, it terminates properly. Both properties have been verified by the state space analysis. The state space has 4 nodes and 3 arcs. There is one dead marking and there is one dead transition *MsgGen* (this transition creates periodic messages).

The periodic message generator should have the following properties: (i) only one type of message is generated, (ii) module does not deadlock, and (iii) module is in livelock. All properties have been verified by the state space analysis. State space has 4 nodes and 3 arcs. It is not the full state space because time changes, so no markings are identical. There is no dead marking and there is one dead transition *Oneshot* (this transition creates the single message).

Node module. The input places to the node module are *Node_1*, *BusFree* and *Bus2Node*. The output places are *Node2Bus* and *BusFree*. The module has no

other interaction with its environment, Fig. 15. The CAN node can get two types of messages from the message generator: (i) single message and (ii) periodically repeating message. The module has to be verified for both types of messages.

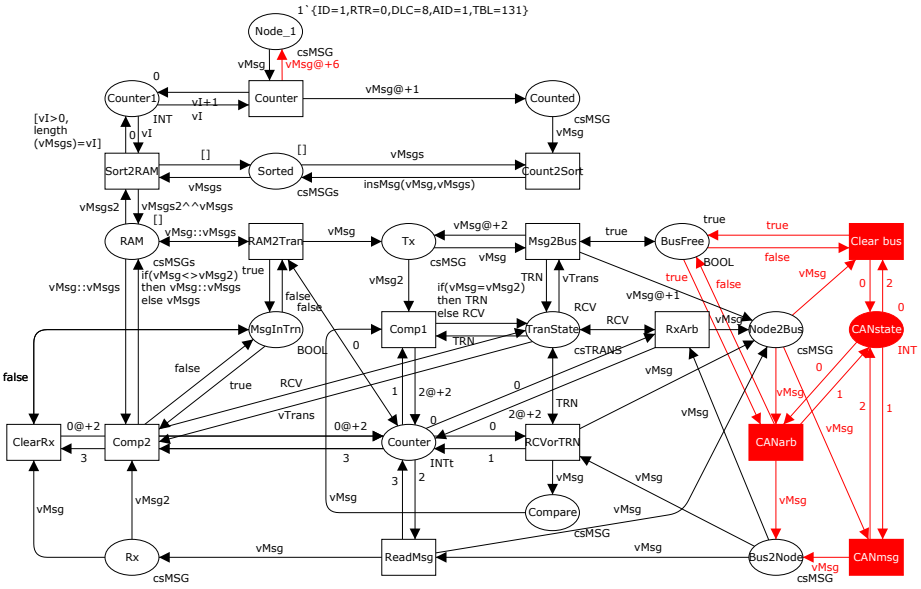


Fig. 15. The model used for verification of the node module. Filled places and transitions are used to mimic the behavior of the environment.

With the single message the node should have the following properties: (i) only one type of message is handled and (ii) module deadlocks. Both properties have been verified by the state space analysis. The state space has 14 nodes and 14 arcs. There is one dead marking and there are two dead transitions *ClearRx* and *RxArb* (these transitions are used when the node is receiver). The correct behavior of two nodes, each sending a single message (with different IDs), has also been tested. In such a case there is no dead transition. This test also verified the automatic retransmission capability.

With the periodic message the node should have the following properties: (i) only one type of message handled, (ii) module does not deadlock, (iii) module is in livelock, and (iv) only one message at the time sent to the bus. All properties have been verified by the state space analysis. The state space has 14 nodes and 14 arcs. It is not full state space because time changes so no markings are identical. There is no dead marking and there are two dead transitions *ClearRx* and *RxArb*. The correct behavior of two nodes, each sending a periodic message (with different IDs), has also been tested. In such a case there is no dead transition. This test also verified the automatic retransmission capability.

Creation of multiple messages and handling of these messages has only been simulated. There is no need for the verification of this property because messages with periods lower and higher than message propagation time have been generated for the state space analysis.

In the case of the periods lower than message propagation time (input arc to the place *Node_1*, Fig. 15), messages stack up in the node but the node sends only one message at a time. The state space analysis run terminates when the deadline, i.e. threshold time, is reached. In Fig. 15 the initial value of the place *Node_1* is set to the maximum message length, and a message period significantly lower than the propagation time. In the case of the period higher or equal to the message propagation time, all messages are sent and the buffer is empty. The analysis terminates properly, i.e. as expected.

CAN bus module. The input places to the CAN bus module are *BusFree* and *Node2Bus*. The output places are *Bus2Node* and *BusFree*. The module has no other interaction with its environment, Fig. 16. The CAN bus module can get two types of messages from the node: (i) a single message and (ii) a periodically repeating message. For the verification of the module this is not important. It is important that, while one message is being processed, another one does not access the bus. CAN bus was modeled with 5 different non-periodic messages, representing 5 different nodes.

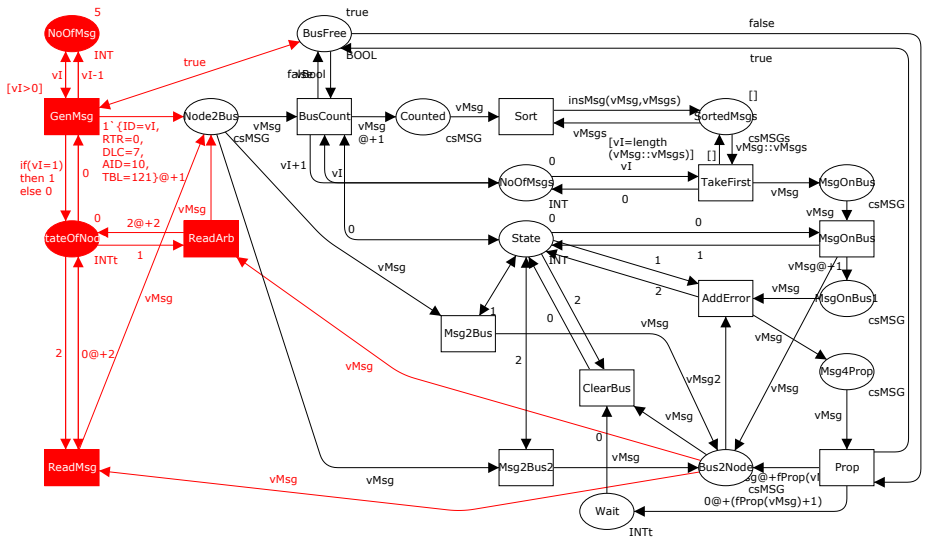


Fig. 16. The model used for verification of the CAN bus module. Filled places and transitions are used to mimic the behavior of the environment.

The CAN bus module should have the following properties: (i) multiple messages can access the *Node2Bus* place (max. number of messages equals max. number of connected nodes), (ii) only one message should be handled, (iii) message of highest priority should be handled, and (iv) deadlock can occur only in case there are no more messages to handle. All properties have been verified by the state space analysis. State space has 77 nodes and 174 arcs. There is one dead marking and no dead transition. *Node2Bus* holds at most 5 messages. Place *Bus2Node* contains the message of highest priority.

5.2 Validation

The CAN communication system consists of two nodes with the following properties: bit rate of 500 kbit/s, 11-bit message ID, each node sends 9 different messages (18 messages in total), all messages have different IDs, message lengths are in range from 0 to 8 bytes, no errors on the bus. The only thing to consider when developing the test system is that the propagation time of all messages has to be lower than half the round period. In this case total propagation time is cca. 3 ms while the round period is 10 ms, i.e. 3 ms $<$ 5 ms.

Two tests were conducted. The first test, Test1, considered messages with data that produced the highest number of bits (due to bit stuffing). The second test, Test2, considered messages with data that produced the lowest number of bits. Nodes were programmed as shown in Tab. 1 and Tab. 2.

Table 1. Message settings for two tests of Node 1.

Node 1			
ID	DLC	Test1	Test2
1	8	0x3C	0xAA
9	7	0xC3	0xAA
17	6	0x0F	0xAA
25	5	0xF0	0xAA
33	4	0x1E	0xAA
41	3	0xE1	0xAA
49	2	0xF0	0xAA
57	1	0xE1	0xAA
65	0	no data	no data

Table 2. Message settings for two tests of Node 2.

Node 2			
ID	DLC	Test1	Test2
2	8	0x3C	0xAA
10	7	0xC3	0xAA
18	6	0x0F	0xAA
26	5	0xF0	0xAA
34	4	0x1E	0xAA
42	3	0xE1	0xAA
50	2	0xF0	0xAA
58	1	0xE1	0xAA
66	0	no data	no data

Both nodes had to start sending messages at the same time. All messages are sent only once. The main reason is that the simulation model has the knowledge of global time, the simulation time, while the real-life equipment does not.

Real-life equipment has inherent drifts and needs synchronization in order to have a notion of global time. This was not implemented in the real-life system, so only one round of messages was allowed. The authors did actually test the real-life system allowing it to continue operation without synchronization.

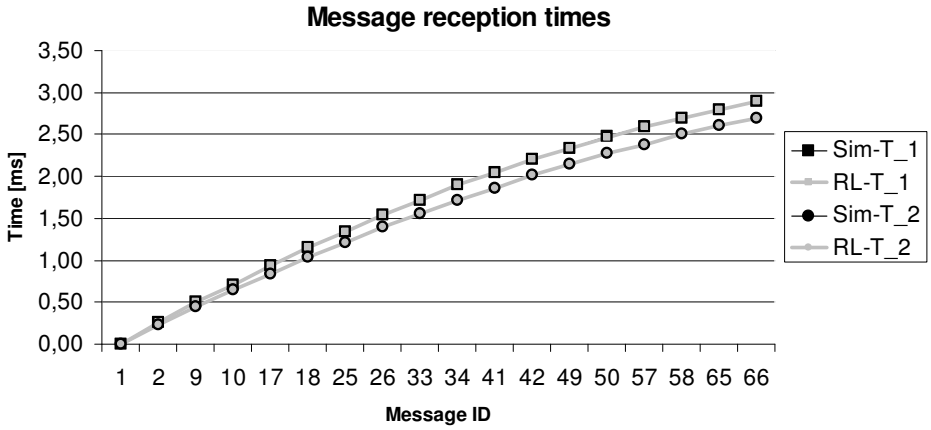


Fig. 17. The results of the validation tests. *Sim-T_x* stands for simulation test x. *RL-T_x* stands for real-life test x.

An internal test report [21] shows a significant drift (up to 10.5 ms per hour) between two identical oscillators. The same oscillators were used in the microelectronic boards for this paper. The 10.5 ms per hour drift translates to $0.0105/(60 * 60) = 2.917 * 10^{-6} = 2.917$ ppm.

Since a single bit lasts $1/500000 = 2 * 10^{-6}$ sec it can be seen that an offset of a single bit between the two boards will be within $2 * 10^{-6}/2.917 * 10^{-6} = 0.686$ sec. This is roughly 68 ms. This gives about 6 rounds of messages, at 10 ms period, without distortion. This calculation does not include other factors, such as interrupt latencies, which can cause drifting and significantly lower the number of synchronized rounds.

The difference was observed, in average, after the third round. Then a message sent by one of the nodes was sent later then planed, but still within the block of messages. After about 1-2 min the messages form the nodes were not interleaved any more but rather separate blocks of messages were observed on the bus. These results were much better at a lower communication speed (100 kbit/s) and with a lower number of messages (3 per node). Now the drifting became apparent after about 13 rounds (cca. 1.3 sec). The separate blocks of messages were observed after 7 min.

The real-life system had a starting message. It was not considered to be the starting point of time measurement, because of the message processing overhead, which is not modeled. The moment the first message (ID 1) was received, is considered the starting point, i.e. time 0. All other messages were timed according to the first message. The obtained results are given in Fig. 17.

The time difference, between the model and the real-life system, was in both tests less than 1 μs, so it is not visible in Fig. 17.

6 Conclusion and future work

It has been shown that the proposed concept for communication system modeling is adequate. The verification and validation results show that the simulation model can be used during the development of new CAN based systems. In future work a model of 16 (+4 optional) nodes will be created. This model will be used to simulate and analyze the CAN bus system in a future city train. Due to the high number of nodes, and even higher number of messages, a modular verification approach is necessary. Future work will address a more formal description of necessary and sufficient module boundary conditions.

In the future, gateway systems, based on real-life equipment, between the CAN and the WTB buses will also be developed.

Acknowledgments. This work was supported by Končar - Electrical Engineering Institute, Croatia, and the Ministry of Science, Education and Sports of the Republic of Croatia.

References

1. Marijan, S.: Control Electronics of TMK2200 Type Tramcar for the City of Zagreb. International Symposium on Industrial Electronics - ISIE, volume IV, Dubrovnik (2005) 1617–1622
2. Marijan, S.: Vehicle control unit for the light rail applications. International Conference on Electrical Drives and Power Electronics - EDPE, Dubrovnik (2005)
3. Bago, M., Marijan, S., Perić, N.: Modeling Controller Area Network Communication. International Conference on Industrial Informatics - INDIN, Vienna (2007) 485–490
4. Bago, M., Perić, N., Marijan, S.: Modeling Wire Train Bus Communication Using Timed Colored Petri Nets. International Conference on Instrumentation, Control and Information Technology - SICE, Tokyo (2008) 2905–2910
5. Selic, B.: The Pragmatics of Model-Driven Development. IEEE Software, vol. 20, No. 5 (2003) 19–25
6. Desel, J., Juhas, G.: What is a Petri Net?. Unifying Petri Nets, Springer-Verlag (2001) 1–25
7. ISO: Road vehicles - Controller area network (CAN) - Part 1: Data link layer and physical signalling. ISO 11898-1:2003, The International Organization for Standardization (2003)
8. ISO: Road vehicles - Controller area network (CAN) - Part 2: High-speed medium access unit. ISO 11898-2:2003, The International Organization for Standardization (2003)
9. Jensen, K.: An Introduction to the Theoretical Aspects of Coloured Petri Nets. A Decade of Concurrency, Lecture Notes in Computer Science, vol. 803, Springer-Verlag (1989) 230–272
10. Jensen, K., Kristensen, M.L., Wells, L.: Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. International Journal on Software Tools for Technology Transfer, vol. 9, No. 3-4 (2007) 213–254
11. Standard ML of New Jersey. http://www.smlnj.org

12. Ullman, J.D.: Elements of ML Programming. Prentice-Hall, Englewood Cliffs (1998)
13. Valmari, A.: The State Explosion Problem. Lectures on Petri Nets I: Basic Models, Lecture Notes in Computer Science, vol. 1491, Springer-Verlag (1998) 429–528
14. Valmari, A.: State of the Art Report: STUBBORN SETS. Petri Net Newsletter, No. 46 (1994) 6–14
15. Christensen, S., Petrucci, L.: Modular Analysis of Petri Nets. The Computer Journal, vol. 43, No. 3 (2000) 224–242
16. Lakos, C., Petrucci, L.: Modular Analysis of Systems Composed of Semiautonomous Subsystems. Fourth International Conference on Application of Concurrency to System Design - ACSD, (2004) 185–195
17. Lakos, C., Petrucci, L.: Modular state space exploration for timed petri nets. International Journal on Software Tools for Technology Transfer, vol. 9, No. 3-4 (2007) 393–411
18. Toussaint, J., Philippe, C., Simonot-Lion, F.: A Model of CAN-based Applications for the Verification of Temporal Properties. 3rd IFAC Symposium on Intelligent Components and Instruments for Control Applications - SICICA, Annecy, (1997)
19. Krakora, J., Hanzalek, Z.: Timed Automata Approach to CAN Verification. 11th IFAC Symposium on Information Control Problems in Manufacturing - INCOM, vol. 1 (2004)
20. Liu, L., Billington, J.: Verification of the Capability Exchange Signalling protocol. International Journal on Software Tools for Technology Transfer, vol. 9, No. 3-4 (2007) 305–326
21. Bago, M.: Test report - oscillator drift. Končar - Electrical Engineering Institute (2005) 1–12

Banker's Algorithm Implementation in CPN Tools

Michal Žarnay[1]

University of Žilina, Univerzitná 8215/1, SK-01026 Žilina, Slovak Republic.
michal.zarnay@fri.uniza.sk

Abstract. When constructing discrete simulation models of complex transportation systems, their designers face problems of deadlock states occurring in the course of simulation. When analyzing it, the issue was transformed to a problem of solving deadlock states in resource allocation systems (RAS) with non-sequential processes with flexible routing and use of resources of multiple types at once. As a suitable deadlock-avoidance policy, the banker's algorithm (BA) has been chosen. The task was to modify the basic version of the BA and to test the developed algorithm on a sample transportation system with the outlined properties. As a suitable environment for this, the CPN Tools were chosen, what led to an implementation of the modified version of the BA in the CPN ML, language used by the CPN Tools. The paper explains modifications of the algorithm, describes an implementation of it in the CPN ML and shows its use on a coloured Petri net model of a small example from the outlined category of the RAS.

1 Introduction

Motivation for this work came from the field of detailed computer simulation of complex transportation systems, such as railway marshalling yard processing a few thousands of wagons per hour in trains of various technological processing descriptions with help of over one hundred resources (individual tracks, locomotives, members of personnel). From experience with real projects, main technological processes in complex systems are usually clearly defined, however, there are often little details complicating the models and causing that designers of models face problems of deadlock states occurring in the course of simulation.

Deadlock state is a state of a system, where two or more system processes are blocked in their execution because they wait for two or more resources, and the awaited resources are at the same time occupied by the processes included in the waiting list. The waiting processes thus block and are blocked. Unblocking this state is possible only by an exceptional operation.

When analyzing the issue, we learnt that it is similar to solving of deadlock states in other fields like flexible manufacturing systems, and that it has been tackled in literature for many years. However, none of the proposed solutions seemed to be adequate for this problem. Further analysis in [1] transformed the issue to a problem of solving deadlock states in resource allocation systems

(RAS) with non-sequential processes with flexible routing and use of resources of multiple types at once.

As a suitable deadlock-avoidance policy for such a system, the banker's algorithm (BA) has been chosen [1]. The task was to modify the basic version of the BA and to test the developed algorithm on a sample transportation system with the outlined properties. From our literature review, we are not aware of any use of the BA in a RAS combining flexible routing with concurrent processing. As a suitable environment for this, the CPN Tools were chosen: to construct a CPN model for the sample system and to implement an adjusted version of the BA. Reasons of this decision lie in the abilities of Petri nets and CPN Tools to construct a model of a RAS quicker compared to other means, e.g. high-level programming languages, and to facilitate a qualitative analysis of the model with and without the BA for testing its effectiveness in deadlock avoidance. In the end, the algorithm itself has been implemented in the CPN ML, language used by the CPN Tools.

From the complex work, this paper focuses on explaining basic modifications of the BA, describing an implementation of it in the CPN ML and showing its successful use on a coloured Petri net model. For the illustrative model, however, only a theoretical example from the outlined category of the RAS has been included, since the model of a sample transportation system developed in [1] and briefly described in [2] is too complex to serve as a basis for showing the algorithm.

The paper is organised as follows. In the section 2, we introduce background for the paper about resource allocation systems and their modelling in Petri nets, and the banker's algorithm and our modifications for the studied system. The section 3 describes the implementation of the modified algorithm in the CPN ML, its integration with the CPN model and its analysis results. That is followed by discussing main contributions and issues of the work in the conclusion.

2 Starting Point

2.1 Resource Allocation System in Petri Net

Resource allocation system (RAS) is a system consisting of concurrently running processes that in certain stages, in order to get successfully completed, require an exclusive use of certain number of system resources [3]. Resources are limited and re-usable as their allocation and de-allocation changes neither their character nor quantity. Based on its character, a process in the RAS is *sequential* or *non-sequential*, i.e. some stages of the process run concurrently. The resulting system is then either sequential (if all involved processes are sequential) or non-sequential (if at least one process in the system is non-sequential). Furthermore a process may contain *flexible routing*, which means that in a certain moment, a process execution continues in one of available options and if correctly defined, taking any of them brings the process to the same final state. Finally, the number and the type of resources allocated at the same time distinguish between

a *single-unit* RAS (every process is allowed to have only one resource unit allocated at a time, i.e. before allocation of the next resource, the previous must be returned), a *single-type* RAS (at least one process has two or more units of the same resource type at a time) or a *multiple-type* RAS (at least one process has two or more units from two or more resource types at a time). The outlined attributes create categories of the RAS with varied complexity.

The system used in this paper is a non-sequential multiple-type RAS with flexible routing. It means that at least one process in the system combines flexible routing with concurrent execution. To distinguish modelling elements of both properties, we use the following naming convention in the paper:

- Variant – one of possible routes of a process execution (sequential or non-sequential) that is chosen by flexible routing
- Branch – a part of a process that is executed concurrently with another part (branch) of the same process

For the modelling of a RAS, the coloured Petri net (CPN) is used. In the net, subnets of two types are found: a *process subnet* and a *resource subnet*. A *process subnet* consists of places, transitions and arcs in a structure starting by an initial transition and ending with a final transition and describing causal relations between stages of a process. A stage (a task) in the process corresponds to a place and events of beginning and ending of tasks correspond to transitions. Together with a place for idle processes (let's denote it $P0$), which connects the final transition with the initial transition of the process description, it makes a strongly connected component. The variants of the flexible routing in the process description are created, when at least one place has at least two output transitions (conflict in Petri net, like in state machines) and another place has at least two input transitions, while all possible routes in the process subnet contain the place $P0$. The branches of the concurrent processing are created, when a transition has at least two output places and another transition has at least two input places (synchronization in Petri net, like in marked graphs), while the place $P0$ is in a part of the subnet with no more than one branch. A *resource subnet* consists of one place and adjacent arcs. Content of the place represents actually free resources and arcs express their allocation and release to and from stages of processes. Typically, there are several process subnets, one for each modelled process, and one resource subnet in a RAS model.

For a description of the system's dynamic behaviour, we'll distinguish between process types and process instances. The *process type* is an abstract description of a process. The *process instance* is a concrete occurrence of a process according to a process type. In the CPN, the process type is modelled by the process subnet and the process instance by one or more tokens of one colour. A position of a token in a place of the process subnet represents a *stage* of the process and a combination of places occupied by tokens of the same value constitutes the actual *state* of the process instance. Similarly, there are resource types and resource instances. The former modelled by colours of a colour set for all resources (one resource type = one colour) and the latter by individual coloured

tokens (number of tokens of a colour = number of resources of the respective resource type).

The figure 1 depicts a CPN model of an illustrative RAS that we use in this paper. The system contains one non-sequential process type modelled by the process subnet. The places $P1, \ldots, P9$ represent individual activities (tasks) in the process. The place $P0$ represents the outside environment, where the finished process instances (tokens coming to the place $P0$) get replaced by new ones before their processing (tokens leaving the place $P0$). Transitions $T1, \ldots, T9$ interlink the activities to give them a logical structure.

An execution of the process type is divided into two branches from the initial transition $T1$, which continue until the final transition $T9$. The left-hand branch is further divided into two variants between the places $P1$ and $P8$. The left-hand variant of these has further 2 branches between transitions $T2$ and $T8$. The place $P0$ contains two tokens of different colours from the integer colour set $cProcessID$ modelling two process instances currently outside of the system. When an instance is in the system, it is represented by two or three tokens of the same colour, depending on the number of branches visited in the actual state. Movements of the coloured tokens are facilitated by the variable *proc* present on all the arcs in the process subnet.

The place *Resources* with adjacent arcs represents the resource subnet. The arcs define relations of allocation/release of resources to/from process activities. Nine tokens in the place *Resources* of 3 colours $R1, R2$ and $R3$ from the colour set $cResources$ represent nine available resource instances, three of each resource type.

2.2 Banker's Algorithm

The *banker's algorithm*, first introduced in [4], uses information about a current system state to decide, whether an allocation request of a process instance can be fulfilled. It is called every time, when an allocation request is made. It tries to find, whether the allocation leads to a *safe state*, i.e. a state, from which all running processes can finish their execution. If yes, the request can be fulfilled, otherwise, the requesting process instance must wait until another process instance returns resources. In order to decide, whether the state is safe, the BA tries to order all running process instances in such a sequence, so that each of them can be finished with resources that it has currently allocated or that are currently available in the system or that are returned from finished process instances already in the sequence prior to the tested process. If it succeeds in finding such a sequence, we say that the state is *ordered*, and since every ordered state is safe [5], the state is also safe. If it fails to find the sequence, we say that the state is *unordered*, which does not mean that the state is unsafe. However, the allocation request cannot be fulfilled. This is due to the suboptimality of the BA, while finding an optimal algorithm for solving the question about state safeness is a NP-hard problem.

The basic principle of the BA is described in a pseudo-code based on the Algorithm 1 from [5]. It uses the following data structures:

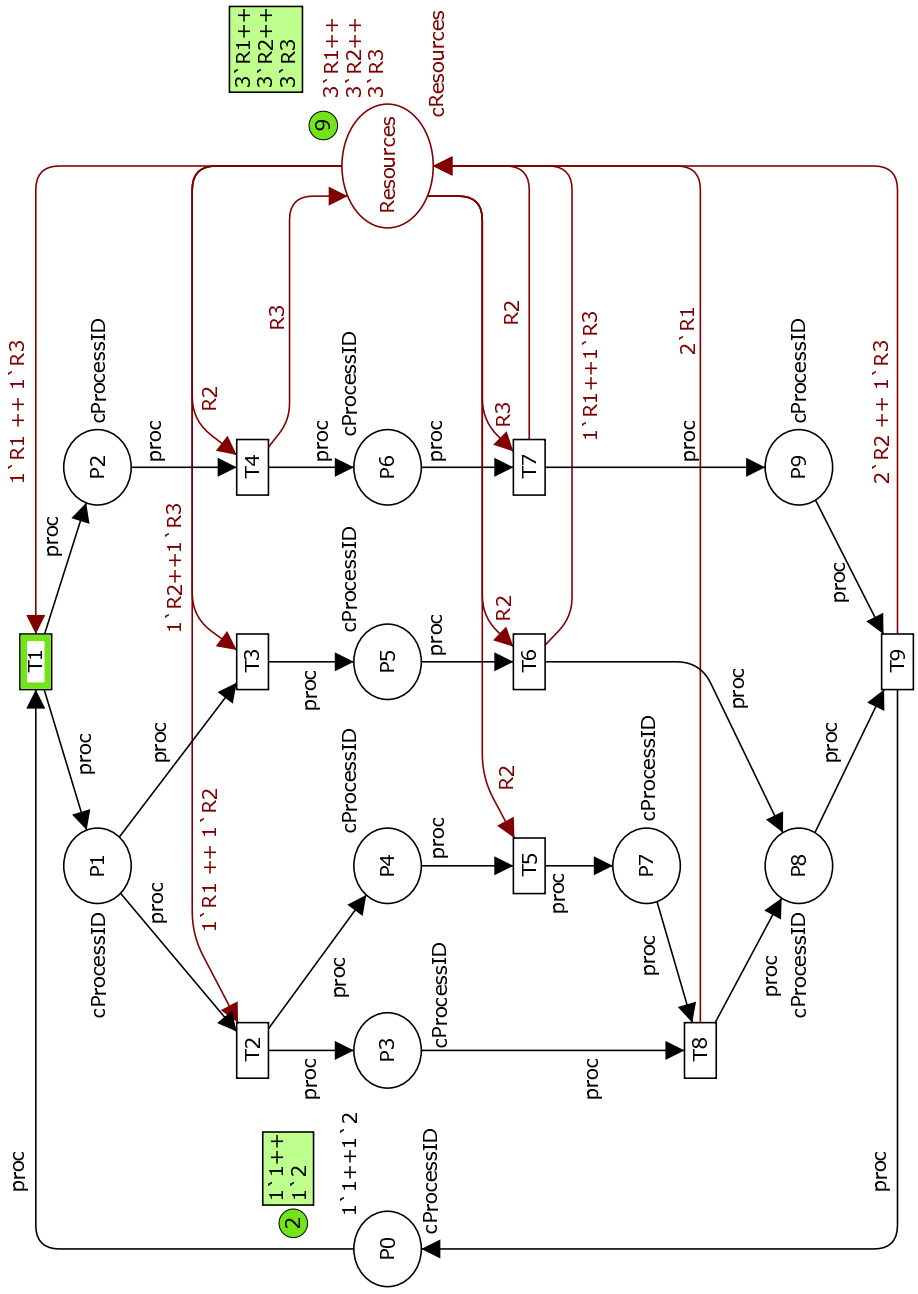


Fig. 1. CPN model of illustration RAS.

- *Allocated* – a matrix of allocated resource units to process instances
- *Allocated*$[i][j]$ – a number of resources of the kind j allocated to the process instance i
- $Needed$ – a matrix of needed resource units to finish execution of process instances
- $Needed[i][j]$ – a number of resources of the kind j needed by the process instance i
- *Available* – a vector of available resource units in the system
- *Available*$[j]$ – a number of available resources of the kind j
- S – a set of all process states currently being executed by process instances
- Π – a set of identification numbers of process stages in S
- R – a set of all resource types
- $p(i)$ – the i-th item in the vector of all running process instances p

When called, the algorithm uses data about a current state of the system in the data structures *Allocated*, $Needed$ and *Available* as input and provides an answer to the question *Is the given state ordered?* as output values *Admit* or $Reject$:

```
begin
            // Set of currently running process instances. //
   Π = {1, 2, ..., | S |};
   loop
            // If all instances have been ordered, admit the state. //
      if Π = ∅ then return Admit
            // Otherwise find a p(i) that can be added to the order. //
      else find i ∈ Π such that
         Needed[i][j] ≤ Allocated[i][j] + Available[j];
      if no such i exists, return Reject;
            // Otherwise, add p(i) to the order. //
      for j = 1 to | R | do begin
         Available[j] = Available[j] + Allocated[i][j];
         Allocated[i][j] = 0;
         Needed[i][j] = 0;
      end;
      Π = Π \ i
   end loop
end
```

The outlined algorithm has been tested for a sequential single-unit resource allocation system (RAS) with flexible routing [5].

The algorithm assumes that as each process instance enters the system, it declares the maximum number of units of each resource type needed for its execution. In a more elaborated version, supportive routines update the number of remaining needed resources of each type of the system processes based on descriptions of their execution and actual positions of process instances [5] [6] and [7].

Applications of the banker's algorithm described in the mentioned literature sources as well as in [8], [9] and [10] have been connected with the sequential RAS-s only. On the other side, authors in [11] consider a banker's-like deadlock avoidance policy for a RAS with non-sequential processes without flexible routing. Our literature review did not bring to our attention any application of the banker's algorithm or banker's-like deadlock avoidance policies that would treat a RAS combining both concurrent processing as well as flexible routing in a process.

2.3 Modifications of Banker's Algorithm

Data structures (the matrices *Allocated*, $Needed$ and the vector *Available*) used by the original version of the BA stay the same in our implementation except renaming the $Needed$ matrix to $RemainingNeeded$ to express more precisely that a content of the data structure is modified during a process instance execution. Resources that will not be requested any more, are subtracted from the vector after their allocation. Only those resources will stay recorded that will be requested in the remaining part of execution. This is based on the assumption that we know process descriptions and we know states, when values of the $RemainingNeeded$ vector are changed.

In [7], it is proposed to assign values of the $RemainingNeeded$ vector to individual process stages of a process on its route to the end of execution. This proposal is suitable for flexible routing, but not for concurrent processing of non-sequential processes. In such processes, the current process instance state may be represented by tokens in more than one place, where the places are in different branches. Since tokens in branches move concurrently, the order of their movements is non-deterministic and it is not possible to know the number of currently allocated and the number of remaining needed resources for each place (for the time, when a place is occupied by a token) generally for all possible executions. That's why we introduce vectors of *relative* changes (called $Change$) which modify the $RemainingNeeded$ vector for the current process instance according to the change in the allocation of resources in the related stage. If resources are being allocated, the $Change$ vector will contain a positive number of the allocated units, if released, the number will be negative. The $Change$ vector must be defined for each process stage with any allocation or release of resources. In addition, it is necessary to take into account, whether resource units will be allocated to the same process instance repeatedly, i.e. allocations and releases of a unit of the same resource type occur in disjunctive time periods at least two times till the end of the process instance execution. This is recorded in the $RemainingNeeded$ vector through values of the $Change$ vector distinguishing two groups of bits in the integer used by every *release of resource units*. An item of the $Change$ vector expresses a number of units of the relevant resource type that are to be allocated/released. When a value of the $Change$ vector contains a non-zero number encoded in the lower half of the bits (the bits 0-3 by 8-bit numbers), the corresponding units will be used again. When a non-zero number

is encoded in the upper half (the bits 4-7), the units won't be used again. For instance, if there are 2 units released without a planned re-allocation, the number will be -32. If they were both to be re-allocated later, the number will be -2, if only one should be re-allocated, the correct number will be -17. The outlined mechanism is relevant only for releases of resources. By allocations of resource units, only the lower half of bits is considered.

As for the division of bits, it is not necessary that the available bits are divided into two halves. It is important that the smallest used part of bits is enough to record the highest number of resource units allocated or released at once in the modelled system. The chosen bits are manipulated with help of operations divide ($/$) and modulo ($\backslash$) and of a relevant constant called $ByteDiv$ (for the discussed division to upper and lower bits of an 8-bit value, $ByteDiv = 16$).

The routine updating the three main data structures then looks as follows:

for $j = 1$ to $\mid R \mid$ do
  if $Change[j] \neq 0$ then do begin
    if $Change[j] < 0$
    then $ChangeValue = -((-Change[j])/ByteDiv$
      $+ (-Change[j]) \setminus ByteDiv)$
    else $ChangeValue = Change[j]/ByteDiv + Change[j] \setminus ByteDiv$;
    $Available[j] = Available[j] - ChangeValue$;
    $Allocated[i][j] = Allocated[i][j] + ChangeValue$;
    if $Change[j] \leq 0$
    then $ChangeValue = -(-Change[j]) \setminus ByteDiv$
    else $ChangeValue = Change[j] \setminus ByteDiv$;
    $RemainingNeeded[i][j] = RemainingNeeded[i][j] - ChangeValue$
  end;

The vectors of relative changes are used not only for (de)allocation of resources, but also for flexible routing. The $RemainingNeeded$ vectors may be different for individual variants of a process description, while every process instance must have one of them in its initial state. That is why we propose two steps to carry out:

1. To order all available variants of the process description and to set the first of them as *primary* – its $RemainingNeeded$ vector will be initial for every process.
2. To define a differential vector between an old and a new variant for every point in the net, where a switch of the two variants is realized.

In the illustrative example, the switch from the primary to the secondary variant is necessary on realizing the transition $T3$ providing the variant with the transitions $T2$, $T5$ and $T8$ is primary and variant with $T3$ and $T6$ is secondary.

In summary, our modifications to the banker's algorithm consist of renaming the $Needed$ data structure to $RemainingNeeded$ (otherwise the original algorithm in the section 2.2 is the same), specifying details of the routine updating the BA data structures and defining related change vectors for every stage of

processes with a (de)allocation in the underlying system. The changes were motivated by the fact that the original algorithm has been constructed for a simpler class of RAS than our application RAS.

3 Banker's Algorithm in CPN

3.1 Construction in CPN ML

Apart from the BA main logic (finding, if the new state is ordered), the implementation needs data structures and routines for recording and managing information about a current state of running processes in the system. Both parts are implemented in user-defined functions of the language CPN ML. In the following sections, we describe colour sets, variables, constant values and functions used. They are fully cited in the appendix.

Data Structures The principal data structure is defined by the colour set $cBankerAlgData$, which is a product of three components corresponding with the above mentioned matrices *Allocated*, $RemainingNeeded$ and the vector *Available*. The first two components are of the colour set $cAllProcessesWRes$, which is a list of items as products $cProcessID * cResNumbersList$. Each product represents a process ($cProcessID$) and a list of numbers of resources ($cResNumbersList$), where each list item corresponds to one resource type. The third component of the $cBankerAlgData$ product is of the latter colour set. An example of use of the $cBankerAlgData$ structure is seen in the constant value $vInitBAData$.

Content of the value $vInitBAData$ corresponds to the above discussed illustrative RAS in its initial state. The model has two process instances, which in the beginning contain no resources, hence the *Allocated* part of the $vInitBAData$ contains the following list of values: $[(1, [0, 0, 0]), (2, [0, 0, 0])]$. Needs of the process instances (initial value of the $RemainingNeeded$ matrix) are in the second row of the $vInitBAData$ value definition and express that both process instances have the same needs expressed by another constant value:
$[(1, vInitMaxNeedPrimary), (2, vInitMaxNeedPrimary)]$. The value $vInitMaxNeedPrimary$ corresponds to maximal needed numbers of resources for one execution of the primary description of the process type, and that is 2 units of the resource type R1, 3 units of R2 and 1 unit of R3 (it can be verified in the model at fig. 1).

$vMaxNeedPrimarySecondaryDiff$ is the difference vector between the primary and the secondary variants of the example's process type description. This means, that the secondary variant has the maximal needs $[1, 3, 2]$. $vByteDiv$ is the factor for the division of bits in a byte to two halves.

The final group of constant values defines the change vectors (implemented as CPN ML lists) that modify the BA data structure by firing of individual transitions $T1$ to $T9$. Values in vectors correspond to the explanation in the section 2.3 and are connected to the original model on the fig. 1. For instance,

the list [∼16,1,∼16] of the $vChangeT6$ constant value corresponds to the change on the $T6$ transition in the model: an allocation of one unit of the resource $R2$ and a release of the resources $R1$ and $R3$ (one unit of each), while both units will not be requested again in the process type description.

Functions Hierarchy Relations among functions in the CPN implementation are depicted at the figure 2. Arcs represent relations of calling – from a superior function to a subordinate function in the direction of their arc. The functions are divided to three groups. The *Main Algorithm* group corresponds to the slightly modified pseudo-code from the section 2.2. The *Data Structure Manipulation* functions implement the manipulation with data structures introduced in the section 2.3. Finally, there are *General* functions that are not directly attributed to the banker's algorithm. They work with lists of integers – they use a recursion to look through the lists and produce their results.

General Functions The $ModifyList$ function modifies the list pA with the list pB returning a new list in which for every item: $pA{+}pOper{*}pB$ (relevant items in the given lists). It is assumed that both initial lists contain integer values and the parameter $pOper$ determines, whether the operation is an addition ($pOper = 1$) or a subtraction ($pOper = \sim 1$). The $IsIn$ function checks, if all items of the list pA are less than or equal to equivalent items of the list pB (i.e. pA "is in" pB). The function is widely used to compare vectors of resources required and available, and to check, if the request can be covered.

The $ULBits$ and $LowerBits$ functions look at given numbers in the $pList$ list parameter as two-part numbers: upper and lower bits, where the edge is defined by the $vByteDiv$ value (bits manipulation is discussed in the section 2.3). The former function sums up numbers encoded in the upper and the lower bits of values in the given $pList$, e.g. the given list [∼32,18,∼7] is changed to [∼2,3,∼7]. The latter function retrieves only numbers encoded in lower bits of values in the given $pList$, in the example it returns [0,2,∼7].

Data Structure Manipulation Functions Most of the CPN ML functions for the banker's algorithm work with lists, thus use a recursion to traverse them.

A set of functions uses an abbreviation PRL that stands for a list of process instances with a list of resources adjacent to it, shorter *process-resources-list*. The functions are used to manipulate with data in the *Allocated* and the $RemainingNeeded$ lists. The $LocateListInPRL$ function locates the PRL of the process of $pKey$ in the $pPRL$ list and returns its list of resource numbers. The $ModifyPRL_List$ function modifies the given PRL-type list: it selects the item with the $pKey$ and modifies its resource number values according to the $pOper$ operation (addition or subtraction) and returns the updated PRL list. The $RemoveItemFromPRL_List$ function removes the item with $pKey$ from $pPRL_List$ and returns the updated PRL list.

The simple function $ChangeMaxNeed$ only updates BA data according to the $pChange$ item. It is used for switching from an old to a new variant of

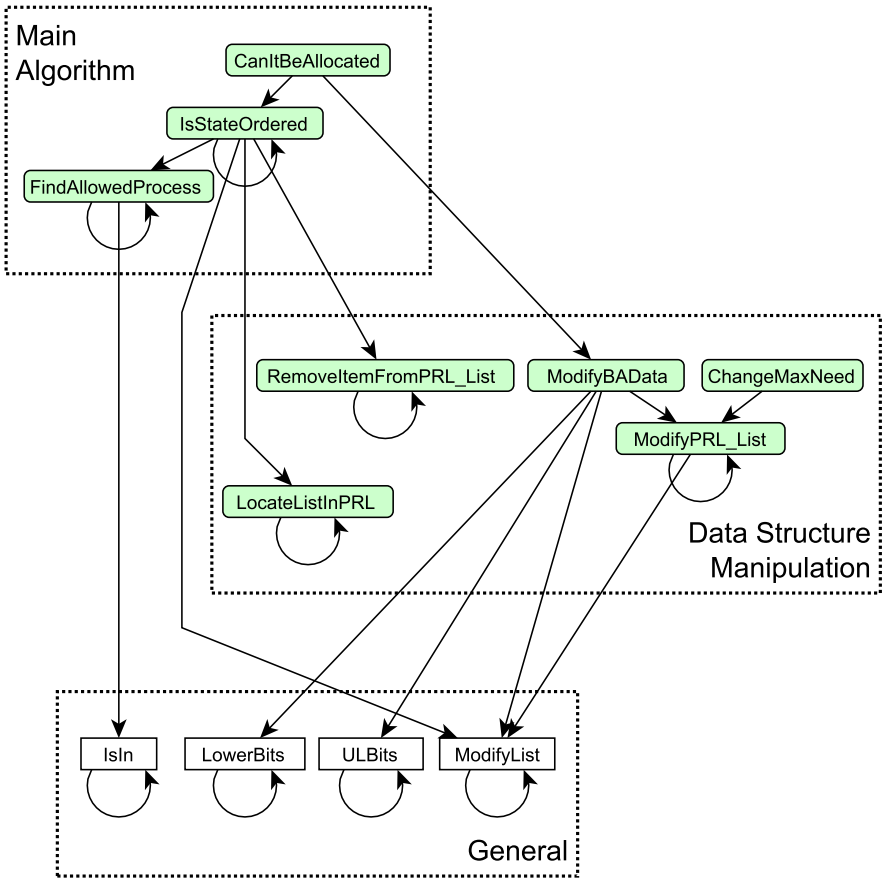


Fig. 2. Diagram of functions in the CPN ML implementation.

process description by flexible routing. The *pChange* argument contains the ID of the process instance and the difference vector between the switched variants. It affects only the middle part of the BA data structure related to the *RemainingNeeded* matrix and only its item related to the given process instance ID.

The $ModifyBAData$ function modifies data for the banker's algorithm given in the *pBAData* parameter by data from the *pChange* parameter, which identifies the respective process instance and contains the change vector with information encoded in its lower and upper bits as explained above. All resources stated in the change vector (i.e. upper and lower bits) are added to the allocated resources of the given process and subtracted from the list of all available resources in the system. However, only the resources encoded in lower bits affect the remaining needed resources of the process (see the 2^{nd} statement in the function).

Main Algorithm Functions The $FindAllowedProcess$ function looks for an item in the list of running process instances (*pRemainNeed*), remaining needed resources of which can be covered by the list of available resources (*Avail*), i.e. a process that can be finished with current available processes. If no process is found, it returns ∼1, otherwise the ID of the process found.

The $IsStateOrdered$ function is the principal function of the banker's Algorithm. It tries to find an ordered sequence of process instances that can be finished in the given conditions. If it is successful, it returns the ordered process sequence. If not, it returns a list with 1 item: [∼1]. [∼2] serves only for recognizing the bottom of the recursion – when all processes were chosen to the order.

Finally, the top function *CanItBeAllocated* in the hierarchy answers the question, if the process instance with its resource request can be allocated the requested resources. It checks, if the state after the allocation will be ordered – then it returns true, otherwise false.

3.2 Adding Banker's Algorithm to CPN Model

In this phase, there are two tasks to fulfill:

- To construct the required data structure in the CPN and to connect it to the underlying CPN model of RAS,
- To interlink all points of allocation in the model with calls of the BA.

The BA data structure is represented by one token of the *cBankerAlgData* colour set in a dedicated place called $Banker'sAlgData$ (see fig. 3). Its initial value is equal to the constant value $vInitBAData$ (see section 3.1). The connection of the new place to the underlying model is made via pairs of arcs with all transitions, at which contents of the BA data structure are to be changed, i.e. all transitions, where at least one resource allocation or release is modelled. One of the arcs in a pair leads from the place $Banker'sAlgData$ to the transition and

brings the BA data structure token through the variable $BAData$ in the arc inscription to make it available for an execution of the banker's algorithm and for an update of data in case the transition fires. The other arc leads in the opposite direction and contains a call to the function $ModifyBAData$ in order to update the data structure after the transition has been fired. As arguments, the function needs the process ID, the respective change vector in resource allocation and the BA data structure itself, for example:

```
ModifyBAData ((proc, vChangeT5), BAData)
```

where $vChangeT5$ is a constant value containing the change vector for firing the transition $T5$ (*proc* and $BAData$ are variables bringing the other needed arguments in).

The BA is called in guards of all transitions where a resource allocation request is made. In our example, it is at all transitions except $T7$ and $T8$. The top BA function $CanItBeAllocated$ is called with the same arguments as the $ModifyBAData$ function, for example:

```
[CanItBeAllocated((proc, vChangeT5), BAData)]
```

Being in the transition guard, the algorithm has direct effect on whether the transition is enabled or not. It serves as the last condition for enabling the firing after all basic conditions secured by the structure of RAS (a process instance is in the appropriate state, resources to be allocated are available) are fulfilled. If the state to come after firing is safe according to the algorithm, the $CanItBeAllocated$ function will allow the transition firing. In case that not, the transition will not be enabled. It can be, however, enabled in the future – when the state of the RAS model changes and the transitions in the CPN affected by the change will be re-calculated, the result of the function (while not changing the basic conditions for the transition) may become positive and enable the transition.

When a process instance chooses another variant of execution by flexible routing, the guard of the relevant transition contains a modification of the maximal needed resources for the given process instance via the $ChangeMaxNeed$ function. In the illustrative model, it happens on the transition T3 and its guard is the following:

```
[BADataAmended =
ChangeMaxNeed ((proc, vMaxNeedPrimarySecondaryDiff), BAData),
CanItBeAllocated ((proc, vChangeT3), BADataAmended)]
```

The amended data structure in $BADataAmended$ is then used in the inscription of the arc from the transition to the $Banker'sAlgData$ place instead of $BAData$. For the transition T3 in the example, it is:

```
ModifyBAData ((proc, vChangeT3), BADataAmended)
```

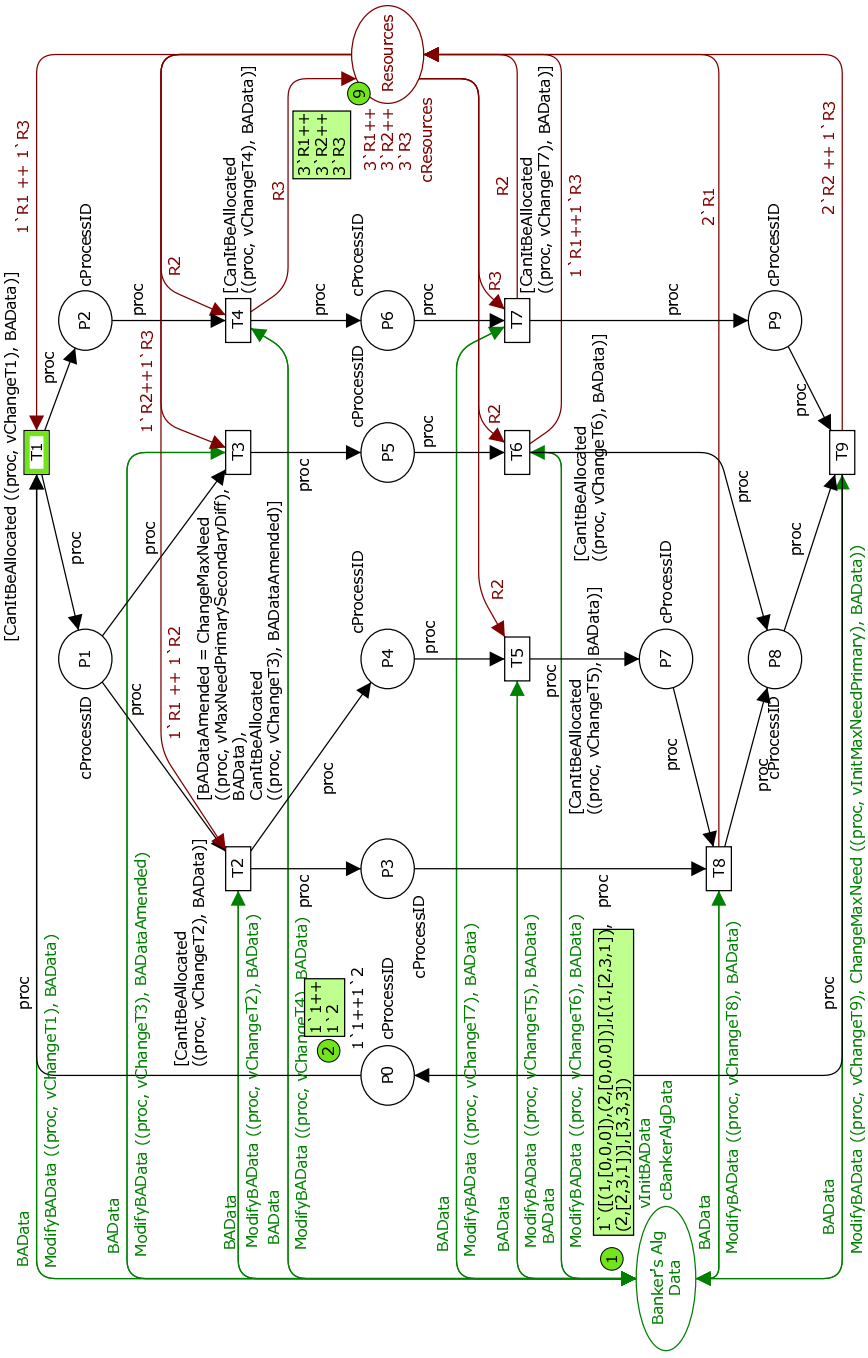


Fig. 3. CPN model of illustration RAS with banker's algorithm.

After its execution, a finished process instance is replaced by a new process instance in the place $P0$. In the BA control subsystem, it is reflected by a use of the $ChangeMaxNeed$ function in the inscription of the arc from the last transition of the process type description to the $Banker'sAlgData$ place – to update maximal needed resources for the new process instance in the BA data structure to the initial $vInitMaxNeedPrimary$ value. Inscription of the arc (in the illustration model from transition T9) looks like this:

```
ModifyBAData ((proc, vChangeT9),
  ChangeMaxNeed ((proc, vInitMaxNeedPrimary), BAData))
```

3.3 Analysis Results

The effect of the banker's algorithm to avoid deadlock states in a modelled system is measured by a number of deadlock states in the occurrence graphs of two CPN models. One is the RAS CPN model without the BA implementation and the other one with it. It is expected that in the first case, the number of deadlock states is not zero, i.e. there are deadlock states in the original RAS to be eliminated. Then, if the deadlock avoidance of the BA is effective, the number of deadlock states in the second model goes down to zero.

In the validation and verification phases, we also further used liveness and home properties of the available analysis tool to check, if the existence of the BA control does not restrict the behaviour of the model in an unappropriate manner, i.e. if all the transitions in the model can occur and whether all reachable markings form a home space. The state space report proved that it was correct.

Furthermore, it is interesting to see, how the algorithm restricts the state space of the original CPN model, since it avoids not only the deadlock states, but also unsafe states leading to deadlock states and also some safe states that are however not accepted by the algorithm due to its suboptimal calculation (see section 2.2).

Tests of the outlined algorithm with two RAS CPN models always showed that it avoids all deadlock states as expected. In the illustrative example, the CPN model of the RAS without the BA contains 12 deadlock states, while the CPN model with the BA contains no deadlock states. As for restriction of their state space, the occurrence graph of the original model has 152 nodes and 378 arcs, while the BA-controlled model has only 113 nodes and 260 arcs.

4 Conclusion

In this paper, we focused on an implementation of the banker's algorithm (BA) for deadlock avoidance in a resource allocation system (RAS) with non-sequential processes with flexible routing and a use of resources of multiple types at once, all modelled as a coloured Petri net in the environment of the CPN Tools. The original version of the BA has been slightly modified in the direction of its application for the outlined system, requiring the introduction of vectors of relative

changes necessary for processes combining all three outlined properties: concurrent processing, flexible routing and use of multiple resources of multiple types at once.

The algorithm has been verified to be effective on such a system. This result we consider as the major contribution of this work, since the application of the banker-like algorithms to such a class of RAS has not been apparently discussed in the literature so far.

Selection of the CPN Tools environment was mainly motivated by the abilities of the fast construction of the underlying RAS model in the CPN and of the available analysis of occurrence graphs on the presence of deadlock states. Both proved to be beneficial. Especially the analysis ability is a very strong tool – it enables to further study behaviour of the BA and its versions on a toy example via detailed analysis of its state space.

However, the implementation of the basic version of the BA in 12 CPN ML functions looks rather complicated compared to sequential programming. Also maintaining information about the global state of a RAS modelled in CPN is rather complicated – it is concentrated to one place, which is connected to many transitions, where allocation requests occur. This makes the CPN more difficult to read, especially for large RAS models. Solution to this is using hierarchy for the CPN: dividing process subnet(s) in RAS to CPN subpages, each of them containing only a few transitions. In the illustrative example in the paper, the process subnet could be split e.g. to three subpages. However, in order to show the whole approach, the hierarchy was not used for the model in this paper.

Further issues are connected to the use of the outlined results in the field, where the motivation comes from – for deadlock avoidance in computer simulation of complex transportation systems.

First, the complex models produce very large state spaces that cannot be fully verified in a reasonable time like the presented simple example. We believe that once the BA has been verified on a small scale example preserving the outlined properties of the complex transportation system, it will be effective also on complex models with tens of process types and process instances and hundreds of resource types as well as resource units. The research will be rather focused on making the run of the BA more effective for the large system.

Secondly, the BA needs to be adjusted to a more complicated way of allocation and release of resources that is currently present in the original simulation models. The first version of the BA with this property has been already made [1], however, it requires further research and testing.

Thirdly, apart from the BA's basic version, we have implemented two more versions of the BA according to [5] – for partially-ordered and V1-ordered states in the CPN Tools [1]. Their explanation requires however more space and is thus outside of scope of this paper. The open question here is, if implementation of other versions, for Vn-ordered states (according to the mentioned paper) is effective and beneficial in our application field.

All the briefly outlined issues frame our future work in this context.

Acknowledgements. This paper has been supported by the grant of the Scientific Grant Agency VEGA 1/4057/07 in the Slovak Republic and the research project MŠM 0021627505 – Theory of transport systems in the Czech Republic.

References

1. Žarnay, M.: Systém na podporu rozhodovania pre riadenie dopravných procesov [Decision-support system for transportation systems control]. PhD thesis, Faculty of Management Science and Informatics, University of Žilina (January 2007) in Slovak.
2. Žarnay, M.: Solving deadlock states in model of railway station operation using coloured petri nets. In Tarnai, G., Schnieder, E., eds.: Formal Methods for Automation and Safety in Railway and Automotive Systems. (2008) to appear
3. Peterson, J.L.: Operating System Concepts. Addison-Wesley (1981)
4. Dijkstra, E.W.: Co-operating sequential processes. In Genuys, F., ed.: Programming Languages, New York, Academic Press (1968) 43112 Reprinted from: Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.
5. Lawley, M.A., Reveliotis, S.A., Ferreira, P.M.: The application and evaluation of banker's algorithm for deadlock-free buffer space allocation in flexible manufacturing systems. International Journal of Flexible Manufacturing Systems **10** (1998) 73–100
6. Tricas, F., Colom, J.M., Ezpeleta, J.: Some improvements to the banker's algorithm based on the process structure. Proceedings of IEEE International Conference on Robotics and Automation **3** (2000) 2853–2858 San Francisco, CA, USA.
7. Tricas, F.: Deadlock Analysis, Prevention and Avoidance in Sequential Resource Allocation Systems. PhD thesis, Departamento de Informática e Ingeniería de Sistemas, Universidad de Zaragoza (May 2003)
8. Ezpeleta, J., Tricas, F., García-Vallés, F., Colom, J.M.: A banker's solution for deadlock avoidance in fms with flexible routing and multiresource states. IEEE Transactions on Robotics and Automation **18**(4) (August 2002) 621–625
9. Lang, S.D.: An extended banker's algorithm for deadlock avoidance. IEEE Transactions on software engineering **25**(3) (1999) 428–432
10. Reveliotis, S.A.: Conflict resolution in agv systems. IIE Transactions **32**(7) (2000) 647–659
11. Ezpeleta, J., Valk, R.: A polynomial deadlock avoidance method for a class of nonsequential resource allocation systems. IEEE Transactions on Systems, Man and Cybernetics, Part A **36**(6) (2006) 1234–1243

Appendix

A complete listing of colour sets, variables, constant values and functions used for the implementation of the BA in the CPN ML is available here. It complements the description of the BA implementation in the CPN ML in the section 3.1 Concrete values are related to the presented illustrative CPN model.

Data Structures

```
colset cProcessID = INT;
colset cResources = with R1 | R2 | R3;
colset cResNumbersList = list INT;
colset cProcessList = list cProcessID;
colset cResources4Process = product cProcessID * cResNumbersList;
colset cAllProcessesWRes = list cResources4Process;
colset cBankerAlgData = product cAllProcessesWRes *
  cAllProcessesWRes * cResNumbersList;

var proc: cProcessID;
var BAData, BADataAmended: cBankerAlgData;

val vInitMaxNeedPrimary = [2,3,1]
val vMaxNeedPrimarySecondaryDiff = [~1,0,1];
val vByteDiv = 16;
val vInitBAData =
([(1,[0,0,0]), (2,[0,0,0])],
[(1,vInitMaxNeedPrimary), (2,vInitMaxNeedPrimary)],
[3,3,3]);

val vChangeT1 = [1,0,1];
val vChangeT2 = [1,1,0];
val vChangeT3 = [0,1,1];
val vChangeT4 = [0,1,~1];
val vChangeT5 = [0,1,0];
val vChangeT6 = [~16,1,~16];
val vChangeT7 = [0,~16,1];
val vChangeT8 = [~32,0,0];
val vChangeT9 = [0,~32,~16];
```

General Functions

```
fun ModifyList (pA, _, []) = pA
| ModifyList ([], _, pB) = pB
| ModifyList (pA, pOper, pB) =
  (hd pA + pOper * hd pB) :: ModifyList (tl pA, pOper, tl pB);

fun IsIn ([],[]) = true
| IsIn ([], _) = false
| IsIn (_, []) = false
| IsIn (pA, pB) =
  if hd pA <= hd pB andalso IsIn (tl pA, tl pB)
  then true else false;
```

```
fun ULBits ([]) = []
| ULBits (pList) =
  if hd pList < 0 then
      ~(~(hd pList) div vByteDiv + ~(hd pList) mod vByteDiv) ::
      ULBits (tl pList)
    else ((hd pList) div vByteDiv + (hd pList) mod vByteDiv) ::
      ULBits (tl pList);

fun LowerBits ([]) = []
| LowerBits (pList) =
  if hd pList < 0 then
      ~(~(hd pList) mod vByteDiv) :: LowerBits (tl pList)
    else ((hd pList) mod vByteDiv) :: LowerBits (tl pList);
```

Data Structure Manipulation Functions

```
fun LocateListInPRL ([], _) = []
| LocateListInPRL (pPRL: cAllProcessesWRes, pKey) =
  if #1 (hd pPRL) = pKey then #2 (hd pPRL)
  else LocateListInPRL (tl pPRL, pKey);

fun ModifyPRL_List ([], _, _) = []
| ModifyPRL_List (pPRLList: cAllProcessesWRes,
pOper, pPRLItem: cResources4Process) =
  if #1 (hd pPRLList) <> #1 (pPRLItem)
  then (hd pPRLList) ::
        ModifyPRL_List (tl pPRLList, pOper, pPRLItem)
  else (#1 (hd pPRLList), ModifyList (#2 (hd pPRLList),
    pOper, #2 pPRLItem)) :: (tl pPRLList);

fun RemoveItemFromPRL_List ([], _) = []
| RemoveItemFromPRL_List (pPRL_List: cAllProcessesWRes, pKey) =
  if #1 (hd pPRL_List) = pKey then tl pPRL_List
  else hd pPRL_List ::
    RemoveItemFromPRL_List (tl pPRL_List, pKey);

fun ChangeMaxNeed (pChange, pBAData: cBankerAlgData) =
(#1 pBAData,
 ModifyPRL_List (#2 pBAData, 1, pChange),
 #3 pBAData);

fun ModifyBAData (pChange: cResources4Process,
pBAData: cBankerAlgData): cBankerAlgData =
(
  ModifyPRL_List (#1 pBAData, 1,
    (#1 pChange, ULBits(#2 pChange))),
```

```
  ModifyPRL_List (#2 pBAData, ~1,
    (#1 pChange, LowerBits(#2 pChange))),
  ModifyList (#3 pBAData, ~1, ULBits(#2 pChange))
);
```

Main Algorithm Functions

```
fun FindAllowedProcess ([], _): cProcessID = ~1
| FindAllowedProcess (pRemainNeed: cAllProcessesWRes,
pAvail: cResNumbersList): cProcessID =
  if IsIn (#2 (hd pRemainNeed), pAvail) then #1 (hd pRemainNeed)
  else FindAllowedProcess (tl pRemainNeed, pAvail);

fun IsStateOrdered (_, [], _) = [~2] (* recursion at the bottom *)
| IsStateOrdered (Alloc, RemainNeed: cAllProcessesWRes,
  Avail: cResNumbersList): cProcessList =
  let
    (* looking for a process that can be chosen to the order *)
    val proc = FindAllowedProcess (RemainNeed, Avail)
  in
    (* if unsuccessful, state is not ordered and return [~1] *)
    if proc = ~1 then [~1]
    else
      (* if process found, continue to the next round *)
      let val result = IsStateOrdered
          (RemoveItemFromPRL_List (Alloc, proc),
          RemoveItemFromPRL_List (RemainNeed, proc),
          ModifyList (LocateListInPRL (Alloc, proc), 1, Avail))
      in
          case result of (* of previous round of recursion *)
            [~1] => [~1] (* was unsuccessful, pass it further *)
          | [~2] => [proc] (* returned from end of recursion,
                start to build up the ordered process sequence *)
          | _ => proc :: result  (* was successful:
                building up the process sequence *)
        end
  end;

fun CanItBeAllocated (pRequest: cResources4Process,
pBAData: cBankerAlgData): BOOL =
  if IsStateOrdered (ModifyBAData (pRequest, pBAData)) = [~1]
  then false
  else true;
```

Augmenting a Workflow Management System with Planning Facilities using Colored Petri Nets

R.S. Mans[1,2], N.C. Russell[1], W.M.P. van der Aalst[1], A.J. Moleman[2], P.J.M. Bakker[2]

[1] Department of Information Systems, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands. {r.s.mans,n.c.russell,w.m.p.v.d.aalst}@tue.nl
[2] Academic Medical Center, University of Amsterdam, Department of Quality Assurance and Process Innovation, Amsterdam, The Netherlands. {p.j.bakker,a.j.moleman}@amc.uva.nl

Abstract. Traditional workflow management systems distribute workitems to users via a worklist and leave the actual timing of workitem execution to the individual resource(s) performing the task. In work environments in which resources are scarce, expensive and multiple resources are necessary to undertake the workitem, often an appointment-based approach is utilized in order to maximize resource utilization. To this end, we propose the extension of a workflow management system with planning and monitoring facilities in order to guarantee effective resource utilization and minimize dead-time for resources as a result of canceled appointments. This paper describes the approach taken in which first a conceptual model for these extensions has been developed which is based on Colored Petri Nets. Second, based on the conceptual model, a prototype has been developed using YAWL and the collaborative software product Microsoft Exchange Server 2007. The applicability of the approach for the development of large scale systems will be demonstrated by elaborating on the conceptual model and the experiences that have been gained. Finally, the operation of the system is demonstrated in the context of a real-life healthcare scenario.

1 Introduction

Nowadays, hospitals are focusing on improving the quality of care and the service that is delivered to a patient. Typically, when making appointments for a patient, patient preferences are taken into account. Examples of this include considering whether appointments can all be scheduled on one day, and querying the availability of the patient. Similarly, constraints imposed by doctors, nurses, rooms, and medical equipment also need to be considered.

Usually, the scheduling of these appointments is done on a manual basis and does not take into account which preceding tasks are necessary and whether they have been performed. For example, in order to perform surgery it is important that the patient is first seen by an anaesthetist in order to determine the anaesthetics that are needed. Moreover, prior to surgery a last check is performed before proceeding with the final preparations for the operation. If these tasks are not performed in time, this can lead to a delay in performing the operation. Another example occurs where during a regular meeting to discuss the status of patients and plan subsequent treatment, a doctor finds out that not all required information is available. The above mentioned examples lead to the inefficient use of scarce, expensive resources, and necessitate the rescheduling of appointments.

Workflow management systems support process execution by managing the flow of work such that individual workitems are done at the right time by the proper person [1]. The benefits being that processes can be executed faster, more efficiently, and their progress can be monitored. Based on business process definitions, which define the ordering between the

tasks which need to be performed, so called *workitems* are distributed to resources (typically people) for execution. A workitem is an indivisible task of work and corresponds to a task which needs to be performed in the context of a given case. An example of a workitem is the performance of a "CT-scan" for patient "Rose". Typically, the user sees available workitems via a so called *worklist*, which can be seen as a to-do list in which people can view the various workitems that they need to perform. At an arbitrary point in time, a user can pick a workitem and perform the task associated with it.

In healthcare the actual execution of a workitem is often linked to an appointment in which several people can be involved. In other words, an appointment-based approach is often utilized for scheduling workitem execution due to the scarce and limited availability of resources that are involved. However, this schedule-based way of working is not supported by the worklist approach offered by current workflow management systems. Moreover, patient preferences need to be taken into account when making these appointments. Consequently, we need to *extend workflow management systems with planning facilities*. Furthermore, planned appointments need to be monitored to ensure that preceding tasks, in the corresponding process definition, are performed on time. If limited time is left to complete them, this needs to be signalled. If they can not be performed on-time, the appointment and possibly subsequent appointments will need to be rescheduled, which is highly undesirable. So, in addition to planning facilities there is the need to incorporate monitoring facilities as well. Note that the focus is on how workflow management can be *integrated* with scheduling and monitoring facilities instead of extending the functionalities of a workflow management system or a planning system.

In this paper, we present the approach taken to *design* and *implement* a workflow system offering (re)scheduling and monitoring facilities. Moreover, the appointments made can also be shown to the people involved. Figure 1 sketches the approach that has been used.

First of all, we started by augmenting a workflow language with planning functionality. Then, we created a conceptual model of a workflow management system augmented with planning and monitoring facilities. The conceptual model is based on Colored Petri Nets (CPNs) [9] thus providing a complete and formal specification of the system to be implemented. The complete specification of the system in CPNs consists of 27 pages, 377 transitions, 169 places, and over 1000 lines of ML code. The construction of the whole model was undertaken by one person, with advanced knowledge about CPNs and CPN Tools, in about 3 months. This, together with the size of the model, illustrates the overall complexity of the system. Finally, we build a prototype of the system. For this prototype we used the open-source, service-oriented architecture of YAWL [2] and the Microsoft Exchange Server 2007 together with several Outlook 2003 clients. The implementation of the system was done by one person, having already built several software tools, in around three months.

The paper proceeds as follows: Section 2 introduces the research approach that was followed. Section 3 describes how a workflow language can be augmented with planning functionality, followed by a description of the conceptual model, constructed in CPNs, in

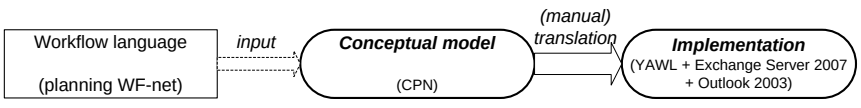


Fig. 1. Overall approach. The workflow language serves as input for the conceptual model. The conceptual model is used as design model for the implementation of the system.

Section 4. Section 5 elaborates on the implementation of the system and outlines a concrete application of the realized system. Section 6 presents related work. Finally, Section 7 discusses the experiences of following the aforementioned approach and concludes the paper.

2 Approach

In this section, we elaborate on the approach that has been followed, as shown in Figure 1, to provide a concrete implementation of a workflow management system augmented with planning and monitoring facilities. As can be seen in the figure, a model-based approach has been used, in which intermediate models are used in order to obtain the final implementation.

The first step was to get insight into how a workflow language could be augmented with planning functionality. As Petri nets are extensively used in Workflow (WF) modeling, primarily because of their mathematical definition and graphical representation, WF-nets were chosen as the basis for these extensions. The main advantage of this choice was that it assisted in the formalization of the augmented workflow language. Note that in principle these extensions can be applied to any workflow language.

Next, we constructed the conceptual model of the system to be realized. A conceptual model serves in understanding a problem domain and identifying how functionality can be added which should collaborate with already existing functionality. The conceptual model that has been constructed is based on CPNs [9]. CPNs provide a well-established and well-proven language suitable for describing the behavior of systems involving characteristics such as concurrency, resource sharing, and synchronization. In this way, they are an excellent candidate for the formalization of such a system.

The CPN language is supported by the CPN Tools offering [9] which we used for creating, simulating and analyzing the model being constructed. This setting allows for *experimentation*, during which deep insights and a good understanding of the design and behavior of the system can be gained. Additionally, it allows for *rapid prototyping*. A complete model of the system can be executed, simulated and analyzed. Flaws in the design can be detected and fixed, leading to a more complete specification. Finally, the system has been implemented using YAWL, Microsoft Exchange Server 2007 and several Outlook 2003 clients by (manually) translating the conceptual model into a working system.

It is important to note that the workflow language is at a different level of abstraction to the conceptual model and the implementation. The workflow language is used as input for the conceptual model. However, the conceptual model and the implementation operate at a similar level. The conceptual model is in such a level of detail that it completely specifies the behavior of the system to be implemented. So, on the basis of the conceptual model, we immediately implement the desired functionality and no other graphical models have been used other than the conceptual model and the implementation. The difference between them is that the conceptual model abstracts from implementation details and language specific issues. The advantage of this is that for the conceptual model we only need to consider the functionality that will be provided by the system and that for the implementation we only need to focus on realizing a working system.

3 Workflow Language

In order to extend workflow systems with planning functionality some new terminology and concepts need to be introduced. We will use a running example for this purpose. It is

assumed that the reader is familiar with basic workflow management concepts, like "case", "role", and so on [1].

3.1 Flow and Schedule tasks

Figure 2 outlines a hospital process in which a patient suspected to be suffering from a lung disease is diagnosed.

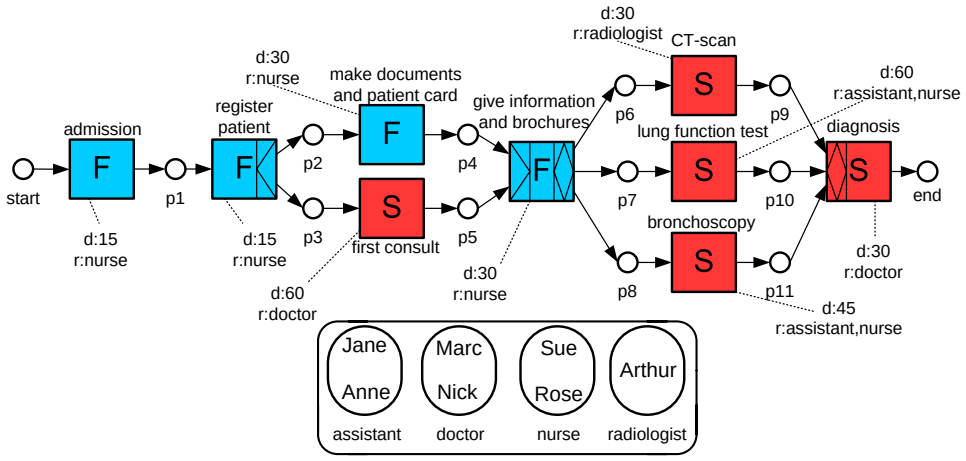


Fig. 2. WF-net for the running example showing schedule (S) and flow (F) tasks. The prefix "d:" indicates the average time needed for performing the task and prefix "r:" indicates which roles are necessary to perform the task. From each associated role, exactly one person needs to be assigned to the task. Note that the "register patient" and "give information and brochures" tasks have XOR split and join semantics associated with them. Moreover, the "give information and brochures" and "diagnosis" tasks have OR split and join semantics. Furthermore, for all of the schedule tasks, the patient is required to be present.

As can be seen in the figure, first the admission is done by a nurse, i.e., some patient related data is recorded and an appointment is made for the first visit of the patient (task "admission"). The next step is that the patient arrives at the outpatient clinic for the first appointment with the doctor (task "register patient"), followed by the first appointment with the doctor (task "first consult"). In this step, a decision is made about the tests to be performed before the second visit of the patient. In parallel, a nurse prepares the documents and the patient card (task "make documents and patient card"). Afterwards, a nurse provides information and brochures to the patient (task "give information and brochures"). Next, the diagnostic tests are performed which are needed for the diagnosis of the patient which is performed by a doctor (task "diagnosis"). For these diagnostic tests a choice can be made from the following tests: CT-scan (task "CT-scan"), lung function test (task "lung function test"), or bronchoscopy (task "bronchoscopy").

From this example, it can be seen that two kinds of tasks can be distinguished: *flow* tasks and *schedule* tasks. In the figure, a flow task is labeled by an "F" and a schedule task is labeled by an "S".

Tasks with an "F" annotation should be performed as soon as a resource is able to undertake them. For example, task "make documents and patient card" can be performed by any nurse when task "register patient" is finished. Basically, a flow task can be performed at *an arbitrary point in time when a resource becomes available and there is no reason to postpone it to a specific point in time*. These tasks can be presented in an ordinary worklist where a given resource can start working on the task when it becomes available. Therefore, as only a single resource is needed to perform the task, it is sufficient to define only one role for them. For example, for the flow task "make documents and patient card" only a single nurse is needed which explains why the "nurse" role has been defined.

The tasks annotated by an "S" in the figure correspond to schedule tasks. For these tasks typically multiple resources are required, with different capabilities. A schedule task needs to be performed by *one or more resources at a specified time*. As multiple resources can be involved in the actual performance of a schedule task, at least one role needs to be defined for each of them. For each role specified, only *one* resource may be involved in the actual performance of the task. For example, for task "lung function test" an appointment is needed in which one assistant and one nurse are involved which explains why the "assistant" and "nurse" roles are defined. Note that for the schedule tasks the patient may also be involved which means that the patient is also a required resource for these tasks. The patient is not involved in the actual execution of a task but is only a passive resource who needs to be present. For that reason, the patient is not added to any of the roles for the task, nor are they defined in terms of a separate role. Instead, it is necessary to identify which schedule tasks the patient needs to be present for.

Flow tasks are presented in an ordinary worklist. However, schedule tasks are presented in a calendar as for each of them specific appointments need to be made involving multiple resources. Each resource has its own specific calendar in which the appointments made for schedule tasks can be seen. In this way, a single appointment made for a schedule task can appear in multiple calendars but only in the calendars of the resources which are involved in the actual performance of the task even though a workitem does not yet exist. When the workitem becomes available, the schedule task can be performed. Note that an appointment in a calendar may also refer to an activity which is not workflow related.

For the booking of appointments in a calendar, it is important to mention that a calendar consists of *blocks* of equal length. So, all blocks represent the same timeperiod. So, a block may either represent one hour but also one minute. Depending on the length of an appointment and the timeperiod of a block, an appointment may occupy several blocks. For example, the "first consult" task has a duration of 60 blocks if a block represents one minute.

To be more precise, for the correct scheduling of appointments for schedule tasks it must be known at runtime what the estimated duration is of the appointment and what the earliest time is that the appointment may be booked. Therefore, for every task in the process model an average duration needs to be specified. As we use the notion of blocks in calendars, we specify the duration of tasks in terms of blocks. In Figure 2 for each task this is indicated by prefix "d:". For example, one blocks takes 1 minute and task "make documents and patient card" requires 30 blocks which means that the task takes 30 minutes on average to complete.

For reasons of simplicity we only include the average task duration for a task to complete. Ideally, more information on the probability distribution could be used, e.g., the standard deviation.

3.2 Formalization

In this section, a formalization of the augmented workflow language will be presented. A WF-net is a Petri net with one initial and one final place such that every place or transition is on a directed path from the initial to the final place [1]. The execution of a case is represented as a firing sequence that starts in the initial marking, consisting of a single token in the initial place. The token in the final place with no tokens left in the other places indicates proper termination of case execution. A model is called sound if every reachable marking can terminate properly.

WF-net extended with the schedule and flow tasks is called a *planning WF-net* (pWF-net).

A pWF-net is a tuple $N = (P, T_f, T_s, F, CR, Res, Role, R, Rtf, Rts, D)$, where

- P is a non-empty finite set of *places*;
- T_f is a finite set of *flow tasks*;
- T_s is a finite set of *schedule tasks*;
- $T_f \cup T_s = T$ and $T_f \cap T_s = \emptyset$ and $T_f \cup T_s \neq \emptyset$, i.e., T_s and T_f partition T. So, a task is either a flow task or a schedule task, but not both. Moreover, the set T may not be empty;
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation);
- $CR \subseteq T_s$ is a set of schedule tasks for which the human resource for whom the case is being performed is also required to be present. For our healthcare example this means the schedule tasks for which the patient is required to be present during their execution.
- Res is a non-empty finite set of *resources*;
- $Role$ is a non-empty finite set of *roles*;
- $R\colon Res \rightarrow \mathcal{P}(Role)$ is a function which maps resources onto sets of roles;
- $Rtf\colon T_f \nrightarrow Role$ is a partial function which maps flow tasks onto roles;
- $Rts\colon T_s \rightarrow \mathcal{P}(Role)\backslash\{\emptyset\}$ is a function which maps schedule tasks onto at least one role;
- $D\colon T \rightarrow \mathbb{N}$ is a function which maps tasks onto a the number of blocks that are needed for the execution of the task. This value indicates the average time it takes to execute the task. One block corresponds to a specific actual duration, e.g. a block can be half an hour or one minute[3].

The running example in Figure 2 can be easily mapped onto this formalization. For example, the "lung function test" task belongs to T_s, and where $Rts(lung\_function\_test) = \{assistant, nurse\}$ and $D(lung\_function\_test) = 60$.

4 Conceptual Model in Colored Petri Nets

The conceptual model which defines the precise behavior of a workflow management system augmented with planning facilities is defined in terms of a CPN model. The complete CPN model consists of a series of CPNs in which several layers can be distinguished. Figure 3 shows the hierarchy of CPNs in the CPN model, together with the relationships between them. In total, there are 27 distinct CPNs. An indication of the complexity of each net is expressed by the p and t value included for each them, showing the number of places and transitions they contain. It is not possible to discuss all the nets in details in this paper. Only the blocks in Figure 3 which are colored grey will be discussed. However, this is sufficient to give an overview of the operation of the model. At the end of the section, Section 4.4, we will focus on the analysis of the conceptual model.

[3] Currently, we only use the average value for (re)planning. However, in the future we plan to utilize more information (variance etc).

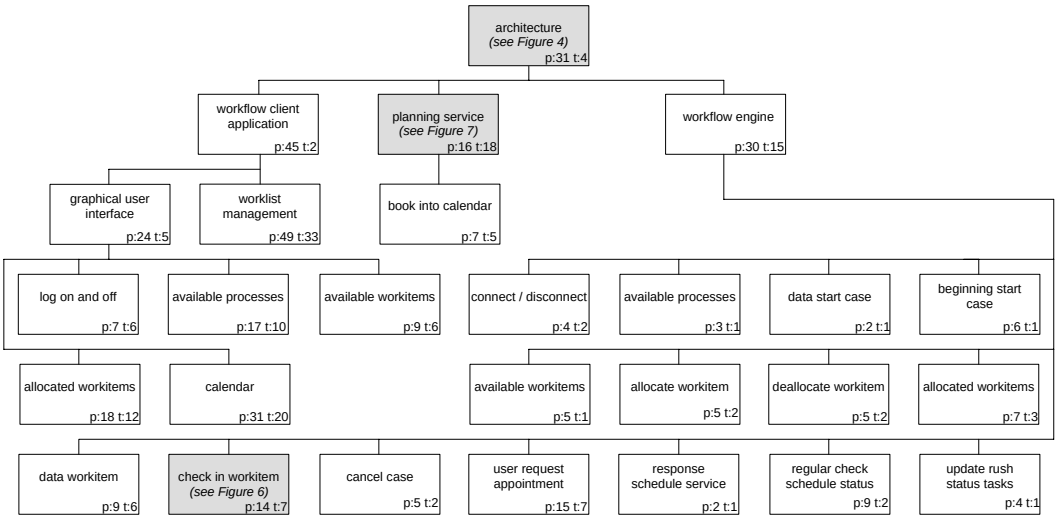


Fig. 3. CPN hierarchy of the conceptual model: each square represents a (sub)net containing places and transitions.

4.1 Overview

Figure 4 shows the topmost net in the CPN model and gives an idea of the main components in the system and the interfaces between them. It can be seen in the figure that there are three substitution transitions. They represent the major functional units in the system, namely: workflow engine, workflow client application and planning service. Each place which is connecting two components forms part of the interface between the two components, except for the place "calendars users" which stores the calendars for each user. The components of the system are set-up in a service-oriented way such that the workflow client application and planning service can interchange data with the workflow engine on a loosely coupled basis. In order to guarantee this, the interface, which defines how two components should interact, should be as minimal as possible. However, this has the advantage that the components can easily be coupled with any other workflow engine component.

The conceptual model consists of three main components.

- The **workflow engine** is the most important component of the workflow system as it is the heart of the system. Based on the business process definition, the engine routes cases through the organization and ensures that the tasks of which they are comprised are carried out in the right order and by the right people. Next to this, the engine takes care of offering workitems to users, once they become available for execution.
- The **workflow client application** communicates the distributed workitems to the users so that they can select and perform them. In our case, workitems that correspond to flow tasks are advertised via the worktray. The appointments that are created for schedule tasks are advertised via a calendar. Once a workitem becomes available for such an appointment, the work can be performed. However, where appointments have been made, users can express their dissatisfaction with the nominated scheduling by requesting: (1) the rescheduling of the appointment, (2) the rescheduling of the appointment to a specified data and time, or (3) the reassignment of the appointment to another

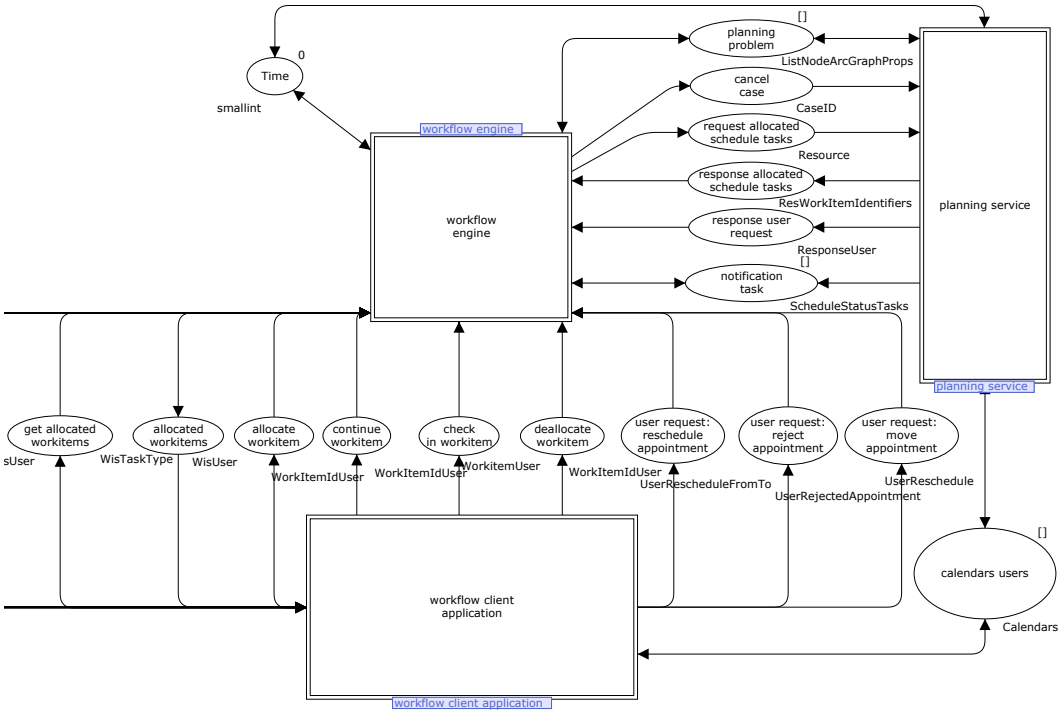


Fig. 4. Overview of the conceptual model.

employee. In addition, the workflow client also indicates whether limited time is left in which to undertake workitems related to preceding tasks for an appointment.
The users who utilize the workflow system interact with it via the workflow client application. All allowed user actions are modeled in subnets of the "graphical user interface" net, which in its turn is a subnet of the "workflow client application" net (see Figure 3).

– The **planning service** component provides the planning capabilities needed by the system. The planning service behaves in a passive way and its operation must be explicitly triggered. Its operation is initiated by the engine which sends a planning problem for a specific case. This planning problem is represented as a graph indicating the planning constraints which hold between the tasks in the corresponding process definition for the case, e.g. the ordering between tasks. Based on this graph, the planning service is responsible for determining whether appointments need to be (re)scheduled. Moreover, the planning service identifies whether limited time is left for the completion of workitems for preceding tasks of an appointment. For such workitems, a warning is forwarded to the users via the workflow engine.

An example of a planning problem that is sent from the workflow engine to the planning service can be found in Figure 5. As can be seen in the figure, the planning graph is formulated as a graph having nodes and directed arcs between the nodes. Additionally, the graph, the nodes and the arcs may have properties. These properties are represented as name-value attributes. In this way, we can add additional constraints to the graph which are relevant for the planning activity. For correct planning of a case, the ordering of tasks in a given process definition is relevant. Therefore, for the corresponding process definition of a case,

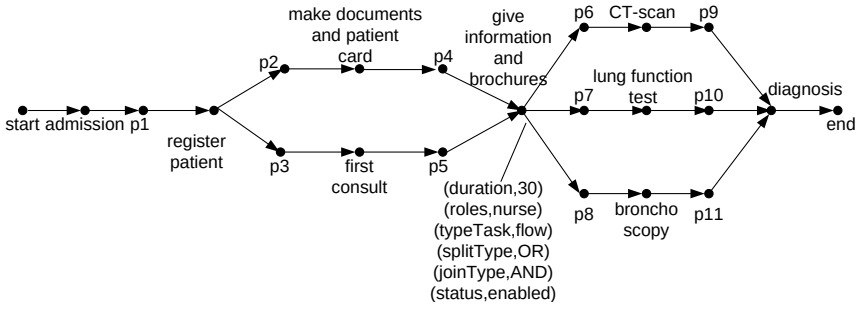


Fig. 5. Planning graph for the running example in Figure 2. The "give information and brochures" task is currently enabled.

we map all nodes and arcs of the process definition to the graph. If the human resource for which the case is being performed is also required in order to perform some of the schedule tasks, then the name of the calendar for this resource is included together with the names of the relevant schedule tasks. If a workitem exists for a certain node, this is also included in the graph as only this task or subsequent tasks need to be (re)scheduled. Additionally, the following information for a task is included: split and join semantics, whether the node represents a schedule, flow, or dummy (i.e. routing) task, and the roles which are involved in performing the task. So, in Figure 5, we can see how the graph of Figure 2 is mapped to a planning graph. For the "give information and brochures" task it is shown that the average duration is 30 minutes, only a single nurse is needed to execute the task, it is a flow task exhibiting OR split and AND join semantics, and a workitem exists which is in the enabled state. In order to simplify the graph, the properties of the other nodes have not been shown.

4.2 Workflow Engine

Figure 3 shows that the workflow engine comprises of 15 distinct CPNs. In total, they consist of 125 places and 54 transitions thus illustrating that the engine demonstrates fairly complex behavior. The naming of the different subnets in the workflow engine CPN gives a good overview of the functionality that is provided by the workflow engine as they all appear as substitution transition on the workflow engine subpage. It is not possible to describe all aspects of these subnets in detail. Hence, we focus on a specific subnet, the checking in of workitems (substitution transition "check in workitem"), which will be discussed in detail.

Before discussing the operation of this subnet, it is important to mention that a workitem passes through a series of states during execution. We make a distinction between the following three states: *enter*, *execution*, and *completion*. A workitem is in the *entered* state when it may be executed, but it is not yet been allocated to a resource. A workitem is the *execution* state, when it has been allocated. A workitem is in the *completed* state, when it has been checked back into the engine, indicating that its execution is completed.

The process associated with the checking in of a workitem is depicted in Figure 6. The thick black lines in the figure shows the paths that can be followed when a request for checking in a workitem arrives at the "check in workitem" place for a given case. Starting from this place it is checked whether: (1) there is a corresponding workitem with the same id in the *executing state* (place "state cases"), (2) the case is active (place "active cases"), and (3) the case is not blocked (place "blocked cases"). If one of the first two prerequisites are

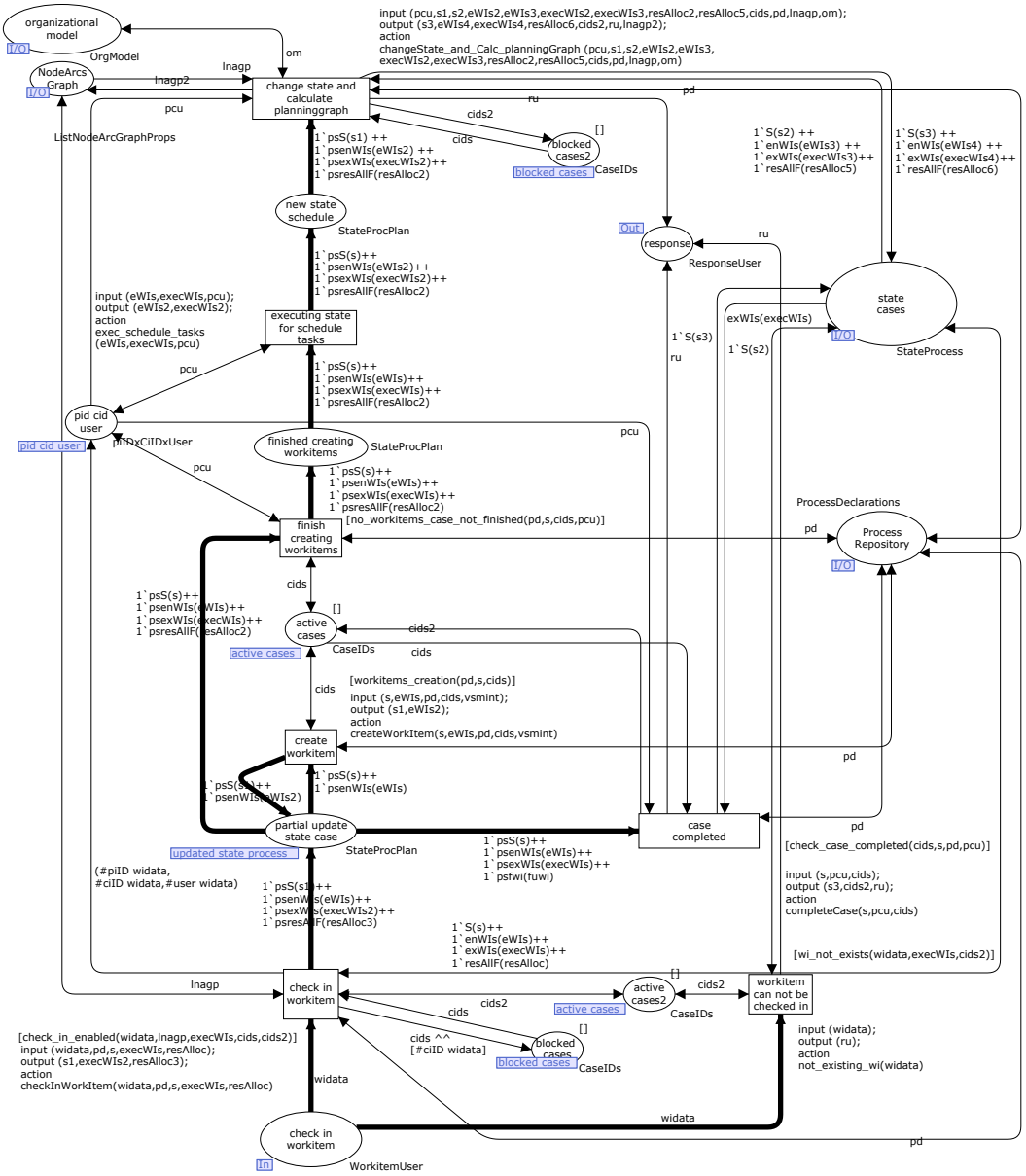


Fig. 6. Checking in of a workitem.

not fulfilled, the “workitem can not be checked in” transition will be fired which informs the requester that checking the workitem into the engine was not successful (place “response”). If the prerequisites are fulfilled, the “check in workitem” transition fires and the following actions are taken:

1. the case is blocked (place “blocked cases”).
2. the new state of the case is calculated.

3. the resource allocation information for the workitem, being checked into the engine is removed from the state information in the "state cases" place.
4. four tokens are produced in the "partial update state case" place containing the following information: the state of the case, the workitems in *entered* or *executing* state, and the resource allocations for the flow workitems in state *executing*.
5. a token is produced in the "pid cid user" case which contains the *ProcessID*, *CaseID* and the user id of the requester.

Several things can happen now. The "create workitem" transition fires when a workitem can be created for a case. When it fires, the following actions are taken:

1. a workitem is created for the task which will be in the *entered* state.
2. the state of the case, in the "state cases" place, will be updated.

If no workitems can be created for the case, the "finish creating workitems" transition will fire which moves the token to place "finished creating workitems". However, it could also happen that no new workitems can be created because the case is complete. In that situation the "case completed" transition fires and the following actions are taken:

1. all case related information is removed from the "state cases" place.
2. the case is deactivated by removing the case id from the "active cases" place.
3. the requester is informed about case completion by putting a token in the "response" place.

After the "finish creating workitems" transition has fired, two steps remain. The first step relates to the "executing state for schedule tasks" transition. This transition changes the state of the schedule workitems, which just have been created, from *entered* into *executing*. The resource allocation for them is done by the planning service.

The second step relates to the "change state and calculate planning graph" transition. When fired, the following actions are taken:

1. a planning problem is formulated and sent to the planning service via place "NodeArcsGraph".
2. the updated state of the case and the updated resource allocation for the flow workitems is saved in place "state cases".
3. the case is unblocked ("blocked cases2" place).
4. the requester is informed about the successful completion of the workitem by putting a token in place "response".

4.3 Planning Service

Figure 7 shows the uppermost model of the planning service. Looking back at Figure 3, we can see that this model consists of 21 places and 13 transitions. However, modeling all the required behavior necessitated writing hundreds of lines of ML code which indicates that this component is fairly complex in its behavior.

Three different parts can be distinguished in the model shown in Figure 7. First, at the top, there is the part which is responsible for receiving a planning problem, (re)scheduling tasks if needed, and generating warnings that limited time is left for performing tasks preceding a schedule task. Second, the "cancel case" substitution transition is responsible for removing all appointments for a case. Third, the "get appointments for resource" substitution transition is responsible for finding all appointments for a specific resource.

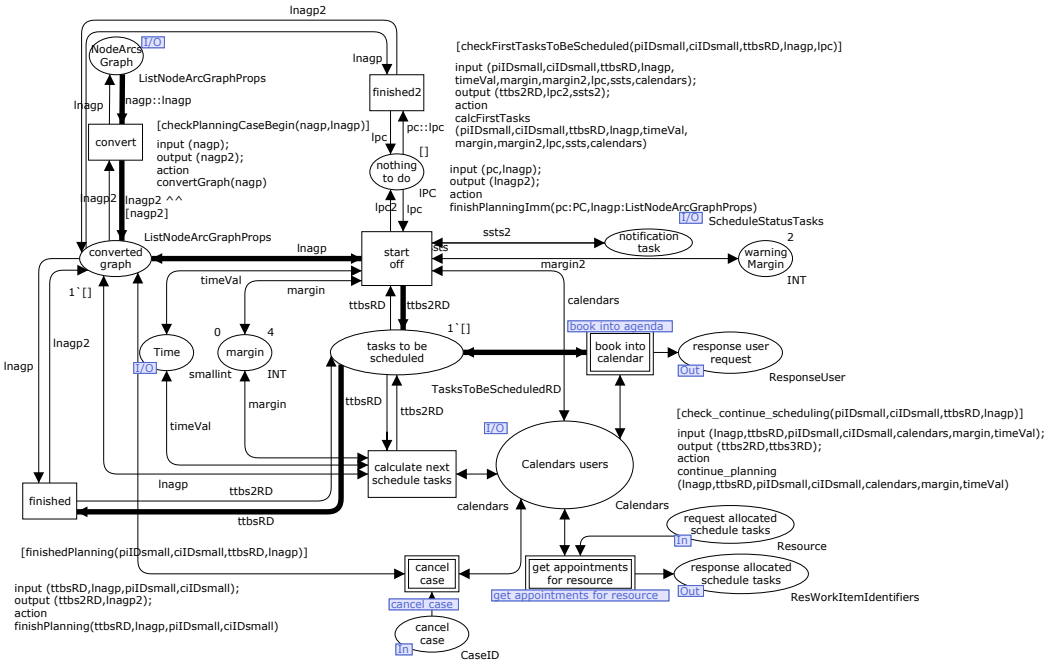


Fig. 7. Top level model of the planning service.

For the remaining part of this section we restrict our discussion to the process of receiving a planning problem from the engine and the steps that are taken afterwards. The sequence of these steps are indicated by a path of thick black lines starting from the "NodesArcGraph" place. When a planning problem is sent to the planning service, the required data for the planning problem is added to the "NodesArcGraph" place. The planning problem is represented by a graph containing the planning constraints which hold between the tasks in the corresponding process definition for the case, e.g. the ordering between the tasks. Once this has occurred, the "convert" transition can fire if the planning service is not busy handling another planning problem for the same case. When it fires, nodes are removed from the graph, which represent a task that has already been performed for the case. Also nodes are removed which represent tasks which we are not sure they will ultimately be executed. So, no optimistic planning takes place. The first nodes in the graph, which do not have an incoming arc, represent tasks in the case for which a workitem exists. Note that our algorithm does not take into account any constraints which may hold between tasks that already have been performed and succeeding tasks, which justifies that the nodes for already performed tasks are removed from the graph. However, for more advanced algorithms it might be the case that this removal step is not allowed.

When the "convert" transition fires, a token containing the converted graph is put into the "converted graph" place. Next, the "start off" transition can fire and the following actions are taken:

1. determine whether the first schedule tasks in the graph, viewing it from the start, need to be (re)scheduled or if a warning should be generated. For a user request, the task

which is selected for rescheduling is considered to be the first schedule task as only this task needs to rescheduled and possibly subsequent schedule tasks.
2. for the schedule tasks which need to be (re)scheduled, the earliest time is calculated at which they may be executed. This is dependent on any preceding tasks which need to be completed.
3. other relevant information for scheduling the task is determined, such as the defined roles and the duration of the task.
4. for every first schedule task, that is a schedule task in the graph for which no preceding schedule task exists, which needs to be (re)scheduled a token containing the information mentioned above is put in the "tasks to be scheduled" place. An example of such a first schedule task is the "first consult" node in Figure 5.
5. for the first schedule tasks in the graph, that are the schedule tasks in the graph for which no preceding schedule task exists, it is determined whether a warning needs to generated because (too) little time is left for performing preceding tasks. If a warning is needed, a notification is sent to the engine via the "notification task" place. The value for deciding how early such a warning needs to be generated, is stored in the "warning margin" place.
6. for each task that needs to be rescheduled, a notification is sent to the engine via the "notification task" place.

The first tasks which need to be (re)scheduled are added to the "tasks to be scheduled" place, and the substitution transition "book into calendar" is responsible for the actual (re)scheduling. The (re)scheduling is done automatically, which means that there is no user involvement. It should be noted that multiple roles can be specified for a schedule task and that for each role specified only *one* resource may be involved in the actual performance of the task. In the "book into calendar" substitution transition a search is started for the first opportunity that for one resource for every required role an appointment can be booked for the respective workitem. If found, an appointment is booked in the calendars of these resources. If the patient for which the case is performed also needs to be present at the appointment, then this is also taken into account.

However, it can also be that no tasks needs to be (re)scheduled at all. This is determined by the "start off" transition which then puts a token into the " nothing to do" place. Afterwards, the "finished2" transition fires removing the planning problem from the "converted graph" place, indicating that the planning problem has been dealt with.

If all schedule tasks for a case that are present in the "tasks to be scheduled" place are (re)scheduled, then it is checked by the guard of the "calculate next schedule tasks" transition whether succeeding schedule tasks in the planning problem graph need to be (re)scheduled. If the transition fires, the following actions are taken:

1. it is determined which subsequent schedule tasks need to be (re)scheduled.
2. for the schedule tasks which need to be (re)scheduled, the earliest time is determined at which they may be executed.
3. the same relevant information for scheduling the task is determined as when the "start off" transition happens.

For each schedule task which needs to be (re)scheduled, a token containing the information described above is put in the "tasks to be scheduled" place triggering another cycle of (re)scheduling and checking. When no subsequent schedule tasks need to be (re)scheduled, transition "finished" fires. If this transition fires, the planning problem present in the "converted graph" place is removed, indicating that the planning problem has been dealt with.

4.4 Analysis

A serious drawback that we faced was that no meaningful verification of the CPN model was possible due to its size and complexity. Even more, as an unlimited number of business process models and users can be represented, state space analysis would be impossible. Therefore, we have tested the model by manually simulating a well-chosen set of scenarios. Although this approach revealed several errors, it does not guarantee that the final model is indeed error-free. An example of such a test scenario is that a case is executed from begin to end during which some appointments are rescheduled as consequence of a user action.

5 Implementation

In this section, we will elaborate on the development of a concrete implementation of a workflow management system augmented with planning facilities. First, we elaborate on the architecture of the implemented system, followed by a discussion of its application to a realistic healthcare scenario.

5.1 Architecture

Figure 8 shows the architecture of the system that has been realized. We can see the components and services that are used, and the means by which they interact with each other. The open-source workflow system YAWL has been chosen as the workflow engine [2]. For storing the calendars of users, we selected Microsoft Exchange Server 2007 which offers several interfaces for viewing and manipulating these calendars. Together with this system we could easily use Microsoft Outlook 2003 clients for obtaining a view on an individual users calendar. Moreover, these clients are configured in such a way that they can interact with the YAWL system via an adaptor. By doing so, an Outlook client can act as a full workflow client application. Finally, the planning service is implemented as a Java service which communicates with both YAWL and the Microsoft Exchange Server 2007[4].

The dashed rectangles around the components in Figure 8 indicate how each substitution transition in Figure 4 has been realized. For example, the "workflow engine" substitution transition of Figure 4 has been realized using the YAWL workflow engine and an adaptor which communicates with the workflow client application and the planning service. For implementing the system, we clearly benefitted from the knowledge contained in the CPN model. As the model is a complete specification of the system that needs to be implemented, while abstracting from implementation details, we could immediately start coding from it. Particularly, given the ML-code and the logic, e.g. ordering of transitions, in the CPN model, the code has directly been written. However, if existing third party software could provide the desired functionality of a (part of a) substitution transition, then of course this software is chosen. For example, YAWL has been chosen as workflow engine as it provides the majority of the functionality that needs to be provided by the "workflow engine" substitution transition. On the other hand, the "planning service" substitution transition has been implemented completely in Java code.

[4] Of course one could argue that for the implementation of the planning service the corresponding part of the CPN model itself could be used. However, pursuing this approach introduces other complex issues like opening and starting a CPN model without opening CPN Tools, communication with external systems, and so on.

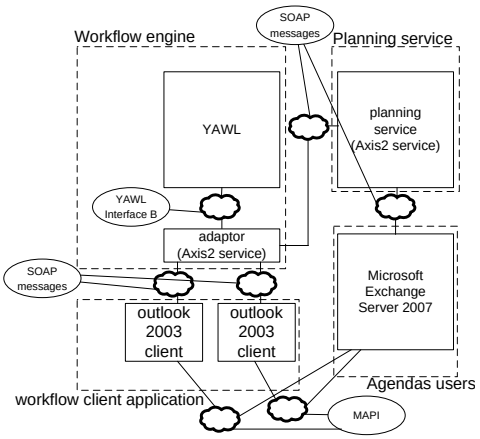


Fig. 8. Architecture of the implemented system. The dashed rectangles indicate how each substitution transition of Figure 4 has been realized. For example, the workflow engine substitution transition has been realized using the YAWL workflow engine together with custom written adaptor.

.

In total, it took around three months for a single person to implement the whole system which involved both component selection and coding. As part of this effort, over 8000 lines of code was written.

5.2 Application

In the remainder of this section, we demonstrate the operation of the system that we realized in the context of a real-life healthcare scenario. As a candidate care process, we have taken the diagnostic process of patients visiting the gynecological oncology outpatient clinic at the AMC hospital, a large academic hospital in the Netherlands. The healthcare process under consideration is a large process consisting of around 325 activities. This healthcare process deals with the diagnostic process that is followed by a patient who is referred to the AMC hospital for treatment, up to the point where the patient is diagnosed. For our scenario we will only focus on the initial stages of the process shown in Figure 9.

At the beginning of the process, a doctor in a referring hospital calls a nurse or doctor at the AMC hospital resulting in an appointment being made for the first visit of the patient. Several administrative tasks need to be finished before the first visit of the patient (task "first consultation doctor"). For example, the referring hospital needs to be asked to send the radiology data to the AMC (task "call for radiology data"). When the patient visits the outpatient clinic for the first time, the doctor decides whether an "MRI", "CT" or "pre-assessment" or a combination of these tasks is necessary. After performing these diagnostic tests, the results will be discussed during the next visit of the patient (task "consultation doctor"). Note that for the MRI, CT and pre-assessment tasks we do not show the preceding tasks at the respective departments that need to performed in order to simplify the presented model.

In this scenario, we assume that the task "additional information and brochures" has been performed in which a nurse provides the patient with information and brochures prior to the execution of the diagnostic tests. Furthermore, it has also been confirmed that the

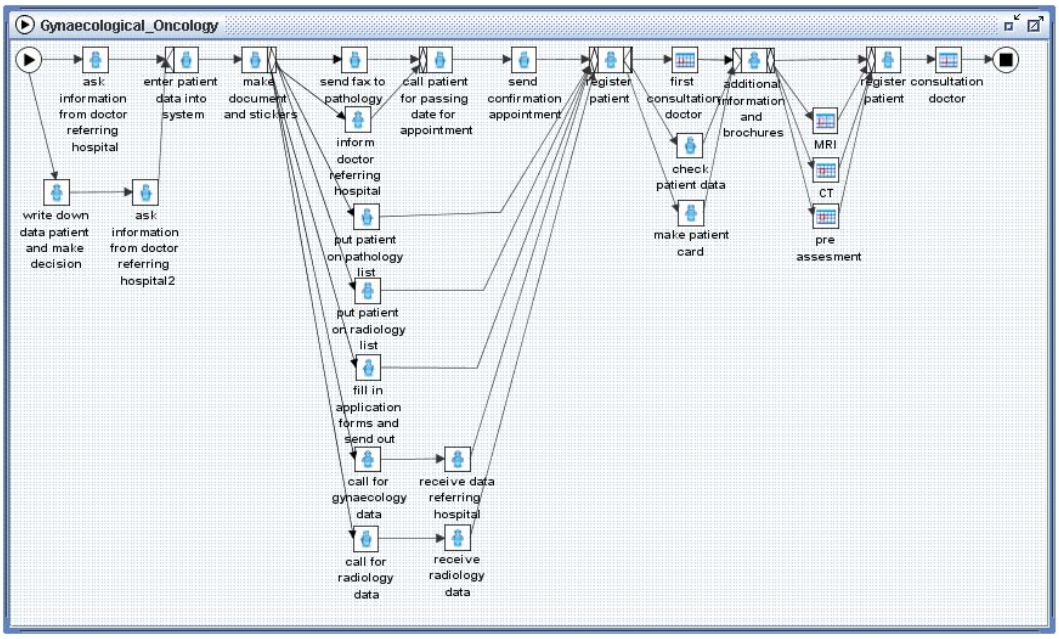


Fig. 9. Screenshot of the YAWL editor showing the initial stages of the gynaecological oncology healthcare process. The flow tasks are indicated by a person icon and the schedule tasks are indicated by a calendar icon. For all schedule tasks, the patient is required to be present.

.

doctor requires an MRI and a pre-assessment for the patient. So, by looking at the process model it becomes clear that the tasks “MRI”, “pre-assessment” and “consultation doctor” need to be scheduled. The result of the scheduling performed by the system for these tasks is shown in Figure 10. Note that our case has “Oncology” as a process identifier and has “126” as case identifier. Moreover, for the “consultation doctor”, “pre assessment”, and “MRI” examination, a doctor, an anaesthetist, and MRI machine are needed respectively. Consequently, these tasks have role “doctor”, “anaesthetist”, and “MRI” respectively. Moreover, the patient is also required to be present.

In Figure 10, going from left to right, we can see that the “MRI” has been scheduled for 8:00 till 8:45, the consultation with the doctor has been scheduled for 13:00 till 13:30 in the calendar of doctor “Nick” and that the pre-assessment has been scheduled for 11:00 till 11:30 in the calendar of anaesthetist “Jules”. At the far right, we can see the agenda of patient Fred who also needs to be present during these appointments. All the previously mentioned appointments are also present in his agenda. Moreover, it is important that the “consultation doctor” is scheduled after the “MRI” and “pre-assessment” task, which is also consistent with the corresponding process definition, where the “consultation doctor” task also occurs after them.

However, a problem is now identified: the patient now has been scheduled for an MRI in which they have to lie in a tube. Unfortunately, the patient is suffering from claustrophobia which means that the patient can only be scanned in an MRI having an open system design, a so-called “open-MRI”. As the role “MRI” includes both the open and closed MRI, we need to reschedule the patient to use the open-MRI by rejecting the current appointment for the

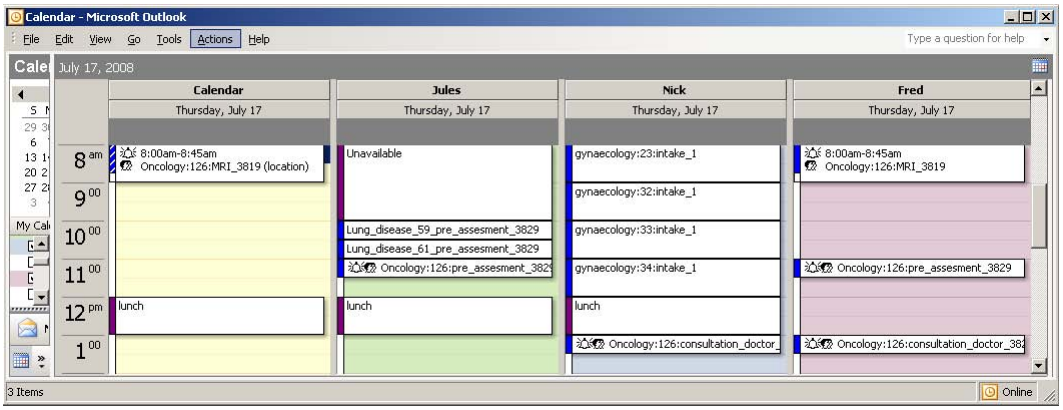


Fig. 10. Screenshot of the calendars for the MRI, consultation with doctor "Nick", and the pre-assessment done by anaesthetist "Jules".

.

(closed) MRI. Rejecting the appointment means that the appointment must be rescheduled and that the resource who rejected the appointment may not be involved anymore. The effect of this specific rescheduling request can be seen in Figure 11.

In this figure, the messagebox indicates that the MRI has been successfully rescheduled. Moreover, the calendar of the open MRI is now shown on the right hand side. It can be seen, that an appointment for the open MRI has been made, taking place from 14:00 to 14:45. However, as can be seen in the second column of the calendar, which shows the calendar of doctor "Nick", it was also necessary to reschedule the appointment with the doctor which will now occur from 15:00 to 15:30. These changes are also reflected in the agenda of patient "Fred", which is shown in the third column. As can be seen in Figure 9, this rescheduling step is necessary as the task "consultation doctor" occurs after the "MRI" task and the "register patient" task falls in between these two tasks and takes 15 minutes. Moreover, in the left top of the figure, we see the form that is generated automatically for performing the "MRI" task.

6 Related Work

A review of relevant literature shows that extensive research has been done into the problem of appointment scheduling in healthcare, e.g. in areas such as appointment scheduling for outpatient service services [6], operating room scheduling [5] and diagnostic resources. However, these approaches tend to focus on specific facilities and not on the complete careflow process. Another approach is described in [22] in which the online problem of scheduling multiple appointments on a single day is considered. In our approach, the whole careflow is taken into account and appointments are scheduled when it is clear that they need to be executed.

Much effort has been put into experimenting and developing workflow management systems so that they can be applied in the healthcare domain [18, 14]. These efforts vary in the sense that they support evidence-based medical procedures, therapies and hospital administrations [16, 15, 21, 4]. One of the most important challenges that needs to be addressed is flexibility support [20]. Unfortunately, current workflow management systems are falling

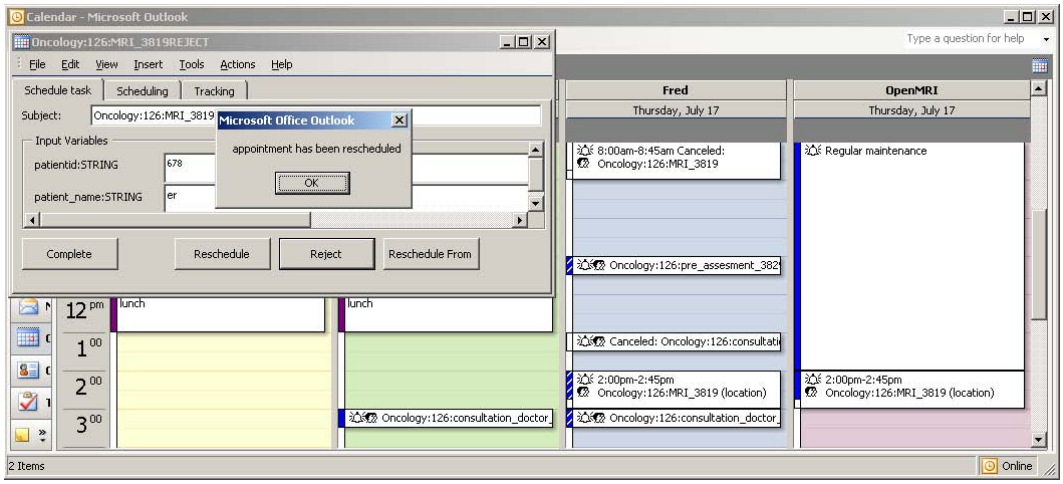


Fig. 11. Result after rescheduling the "MRI" task.

.

short in providing flexibility [10, 11] which seriously hampers the application of workflow technology in the healthcare domain. In addition to this, support is needed for the cross-departmental nature of healthcare processes [12]. Currently, administrative workflows are typically limited to single departments [19]. Successful implementation of workflow management exists but widespread adoption and dissemination is the exception rather than the rule [14]. It is expected that the use of workflow technology by healthcare institutions will grow dramatically in the future [14] and it is likely that it will become a core component in future healthcare systems [7].

Despite all these efforts, no work has been performed on the combination of appointment scheduling and workflow management systems, i.e., existing approaches are either focusing on planning with little consideration for workflow aspects or are focusing on workflow while ignoring that much work is done via appointments rather than worklists. We are not aware of any research looking at the mixture of flow and schedule tasks.

For various systems, CPNs have been used to formalize and validate functional requirements. For example, the formalization of the design of the so-called worklet service, which adds flexibility and exception handling capabilities to the YAWL workflow system [3], formalizing the implementation of a healthcare process in a workflow management system [13], and presenting a model-based approach to requirements engineering for reactive systems, in which CPNs are used for validating the functional requirements [8]. Related to this is [17], in which CPNs are used for specifying the operational semantics of *new*YAWL, a business process modeling language founded on the well-known workflow patterns[5].

7 Experiences and Conclusions

In this paper, we have discussed the design and implementation of a workflow management system offering planning and monitoring facilities. As approach, we started with a workflow language, followed by a conceptual model in CPNs and finally a concrete implementation

[5] For more information about workflow patterns see http://www.workflowpatterns.com

of the system. The conceptual model consists of 27 distinct nets, 377 transitions, 169 places and over 1000 lines of ML code. The construction of the whole model took around three months of work. These figures indicate that a workflow system augmented with planning facilities is a fairly complex system and the task of developing it is far from trivial.

One of the main benefits of building the conceptual model in CPNs is that it can be executed in the CPN Tools offering. In this way, it allows for experimentation during which comprehensive insights can be obtained about the design and behavior of the system to be realized which probably would not have been possible to obtain by pursuing other approaches to designing the system. Parts of the system can be tested early in the development process, thus enabling early detection of design errors. The costs of repairing these errors in this phase of the development process is far less than would be the case in a later phase. For example, when experimenting with the subnet of the planning service we identified errors with regard to the correct planning of appointments.

Another advantage of modeling the conceptual model in CPNs is that it completely specifies the behavior of the system to be implemented while abstracting from implementation details and language specific issues. So, for the conceptual model we only needed to worry about the behavior of the system, while for the implementation we focused on the realization. In this way, these kinds of issues are distinguished, allowing for a separation of concerns. The importance of this distinction can probably best be illustrated by the fact that it took more than 3 months to build the conceptual model, and just 3 months to implement the whole system. For the implementation of the system it was necessary to produce over 8000 lines of code by hand. Although the main functionality of the system was fully implemented during the implementation phase, a significant amount of time still needs to be spent on component selection, coding, and dealing with residual implementation issues.

The fact that we started completely from scratch ending up with a concrete implementation of the system with the proposed functionality shows both the applicability and feasibility of our approach. However, the developed system has only been tested in a limited set of scenarios. As future work, we plan to systematically test parts of the system by "replacing" one or more components in the conceptual model by a complete implementation for it, based on third party software, allowing for the testing of thousands of scenarios. In this way by simply executing the CPN model, we are able to identify errors in the components which probably would not have been found with using a scenario based approach of testing. In addition to this, we plan to use the conceptual model for evaluating alternative planning approaches using various performance indicators.

References

1. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2002.
2. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.
3. M.J. Adams. *Facilitating Dynamic Flexibility and Exception Handling for Workflows*. PhD thesis, Faculty of Information Technology, Queensland University of Technology, 2007.
4. L. Ardissono, A.D. Leva, G. Petrone, M. Segnan, and M. Sonnessa. Adaptive medical workflow management for a context-dependent home healthcare assistance service. *Electronic Notes in Theoretical Computer Science*, 146(1):59–68, 2006.
5. B. Cardoen, E. Demeulemeester, and J. Beliën. Operating room planning and scheduling: A literature review. FEB Research Report KBI 0807, Katholieke Universiteit Leuven, Leuven, 2008.

6. T. Cayirli and E. Veral. Outpatient scheduling in health care: a review of literature. *Product Operations Management*, 12(4):519–549, 2003.
7. A. Dwivedi, R. Bali, A. James, and R. Naguib. Workflow Management Systems: the Healthcare Technology of the Future? In *the 23rd Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, volume 4, pages 3887–3890, 2001.
8. J.M. Fernandes, S. Tjell, and J.B. Jorgensen. Requirements Engineering for Reactive Systems with Coloured Petri Nets: the Gas Pump Controller Example. In K. Jensen, editor, *Proceedings of the Eight Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 207–222, 2007.
9. K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *STTT*, 9(3-4):213–254, 2007.
10. M. Klein, C. Dellarocas, and A. Bernstein, editors. *Adaptive Workflow Systems*, Special Issue of Computer Supported Cooperative Work, 2000.
11. R. Lenz, T. Elstner, H. Siegele, and K. Kuhn. A Practical Approach to Process Support in Health Information Systems. *JAMIA*, 9(6):571–585, December 2002.
12. R. Lenz and M. Reichert. IT Support for Healthcare Processes - Premises, Challenges, Perspectives. *Data and Knowledge Engineering*, 61:49–58, 2007.
13. R.S. Mans, W.M.P. van der Aalst, P.J.M. Bakker, A.J. Moleman, K.B. Lassen, and J.B. Jorgensen. From Requirements via Colored Workflow Nets to an Implementation in Several Workflow Systems. In K. Jensen, editor, *Proceedings of the Eight Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 187–206, 2007.
14. M. Murray. Strategies for the Successful Implementation of Workflow Systems within Healthcare: A Cross Case Comparison. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, pages 166–175, 2003.
15. M. Poulymenopoulou and G. Vassilacopoulos. A Web-based Workflow System for Emergency Healthcare. In *Proceedings of the MIE 2002 conference*, 2002.
16. S. Quaglini, M. Stefanelli, G. Lanzola, V. Caporusso, and S. Panzarasa. Flexible Guideline-based Patient Careflow Systems. *Artificial Intelligence in Medicine*, 22(1):65–80, 2001.
17. N.C. Russell, A.H.M. ter Hofstede, and W.M.P. van der Aalst. newYAWL: specifying a workflow reference language using coloured petri nets. In K. Jensen, editor, *Proceedings of the Eight Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2006)*, volume 584 of *DAIMI*, pages 107–126, Aarhus, Denmark, October 2007. University of Aarhus.
18. G. Russello, C. Dong, and N. Dulay. Consent-Based Workflows for Healthcare Management. In *Proceedings of 2008 IEEE Workshop on Policies for Distributed Systems and Networks (Policy 08)*, pages 153–161, Palisades, NY, US, 2008.
19. X. Song, B. Hwong, G. Matos, and A. Rudorfer. Understanding and classifying requirements for computer-aided healthcare workflows. In *COMPSAC (1)*, pages 137–144. IEEE Computer Society, 2007.
20. M. Stefanelli. Knowledge and Process Management in Health Care Organizations. *Methods Inf Med*, 43:525–535, 2004.
21. S.W. Tu, M.A. Musen, R. Shankar, J. Campbell, K. Hrabak, J. McClay, S.M. Huff, R. McClure, C. Parker, and R. Rocha. Modeling Guidelines for Integration into Clinical Workflow. *Studies in Health Technology and Informatics*, 107:174–178, 2005.
22. I. Vermeulen, H. La Poutré, S.M. Bohte, S.G. Elkhuizen, and P.J. Bakker. Decentralized Online Scheduling of Combination-Appointments in Hospitals. In *Proceedings of ICAPS-2008, the International Conference on Automated Planning and Scheduling*, Sydney, Australia, 2008. AAAI Press.

Towards Formal Modelling and Analysis of SCTP Connection Management

Somsak Vanit-Anunchai

School of Telecommunication Engineering
Institute of Engineering
Suranaree University of Technology
Muang, Nakhon Ratchasima, Thailand
Email: somsav@sut.ac.th

Abstract. The Stream Control Transmission Protocol (SCTP - RFC 2960) is a reliable unicast transport protocol originally specified by the Internet Engineering Task Force (IETF) in 2000. Its design rationale aims to overcome the weaknesses of the Transmission Control Protocol (TCP). However, after years of implementing and testing, defects and errors in RFC 2960 were reported and later fixed in RFC 4460. Incorporating those suggested fixes, IETF revised the SCTP specification and in September 2007 published RFC 4960, which replaces RFC 2960. This paper presents a Coloured Petri Net (CPN) model of the revised version of SCTP's connection management from RFC 4960. In particular we model the revised Tie-Tag mechanisms that differ significantly from those specified in RFC 2960. By following a procedure-based approach, each CPN page relates to only a few sections in RFC 4960, which aids validation. Preliminary results from state space analysis reveal two potential problems. First, SCTP association establishment can terminate in a half open state, with one side in CLOSED but the other in ESTABLISHED. Second, simultaneous establishment can lead to states in which both sides are in ESTABLISHED but the verification tags in both Transmission Control Blocks do not match.

Keywords: SCTP, Formal Methods, Coloured Petri Nets, State space methods.

1 Introduction

Motivation The Stream Control Transmission Protocol (SCTP) [6] is a transport protocol that was approved by the Internet Engineering Task Force (IETF) as Request for Comment (RFC) 2960 in 2000. It was originally designed by the Signalling Transport working group for transporting telephony signalling messages over UDP. These signalling messages have stringent timing requirements which are difficult to meet when using TCP. Foreseeing its significance and great potential to become a major transport protocol, IETF decided to operate SCTP over IP instead. Despite its potential to replace TCP, several years of implementation and testing revealed fifty-two defects in the SCTP specification, RFC 2960. Solutions were gathered and discussed in RFC 4460 [5]. The IETF has published a revised version of the SCTP specification, RFC4960 [4], in September 2007, and RFC 2960 has become obsolete. This revised specification raises two questions. Firstly, are there any unknown defects left? Secondly, are any new defects introduced in the new specification?

In addtion to the motivation for formal validation of SCTP, we wish to experiment with a new modelling approach called the "procedure-based" approach. In [2] Billington and Vanit-Anunchai discussed the advantages and

disadvantages of state-based and event-processing CPN modelling styles. The state-based modelling approach has the advantage of readability, and unspecified actions can be easily discovered. Its disadvantage comes from redundant specification of actions that are common to several states. While the event processing modeling approach has the advantage of folding common actions across several states, which makes the CPN model easier to maintain, it has some drawbacks with respect to readability. Thus [2] proposed the procedure-based modelling approach, which structures the CPN model according to the protocol's functionality. This modelling style has two merits. Firstly, the CPN model is easy to maintain. Secondly, the procedure-based CPN model comprises typical - simple procedures and unexpected - complex procedures (error handling). Beginners can pay attention to the typical scenarios before getting into the complex procedures later. The procedure-based CPN model of DCCP connection management presented in [2] evolved from a previous version modelled using the state-based approach. In this paper, we wish to gain experience building a procedure-based CPN model directly from an informal specification.

Previous work Since published in 2000, SCTP has been an attractive research topic. Although there has been a lot of work on SCTP regarding its security, performance, and extensions of SCTP functionality, we have found only one article (in Portuguese) [3] modelling SCTP connection management using Coloured Petri Nets (CPN). The CPN model we propose in this paper differs from [3] in three aspects. Firstly, we build the CPN model according the revised specification, RFC 4960, while [3] uses RFC 2960 which is now obsolete. Secondly, while the CPN model in [3] did not include the procedures for when SCTP nodes receive duplicated or unexpected messages, our model includes these events (described in Sections 5.2 and 9.2 in RFC 4960). Thirdly, the CPN model in [3] follows the state-based approach, whereas our model uses the procedure-based approach of [2].

Contribution The difficulty of designing a protocol is again witnessed by the defect list in RFC 4460 [5]. Despite many years of implementing and testing, it is still important to have a proper formal model and to perform formal analysis of SCTP connection management, especially when SCTP is designed for reliable data transfer such as signalling in Public Switching Telephone Networks. The contribution of this paper is three-fold. Firstly, we propose a CPN model of SCTP connection management based on Internet Standard RFC 4960. The model provides good insight into how to manipulate and use the Tie-Tags. Secondly, even though no errors in the shutdown scenarios are found, we discover an error in section 5.2.4 of RFC 4960 and two potential defects in the association establishment scenarios. Thirdly, for readers who are not interested in SCTP, this paper demonstrates an example of modelling a transport protocol using the procedure-based approach.

Organisation This paper is organised as follows. Section 2 provides an overview of SCTP association set up and graceful shutdown. Modelling assumptions are

listed in Section 3. The description of the CPN model of SCTP connection management and its declarations is given in Section 4. Section 5 presents the analysis results and a discussion of terminal markings. Section 6 presents conclusions and future work.

2 Stream Control Transmission Protocol

The Stream Control Transmission Protocol (SCTP) [6] is a unicast connection oriented transport protocol. Like TCP, SCTP provides an error-free reliable flow of data, without loss or duplication, between a client and a server. To ensure that the behaviour of SCTP's traffic mimics that of TCP, it uses the same congestion control algorithm as TCP.

SCTP has three distinctive features. Firstly, SCTP introduces the concept of multiple streams to reduce the problem of head-of-line blocking. It delivers its user messages in sequence within a given stream. Multiple streams are bundled into a single SCTP packet. The number of streams in an association is set up during startup. If one stream is blocked, delivery on the other streams can still proceed. Secondly, SCTP has the ability to support multiple IP addresses, called multi-homing. Several paths may exist between two nodes but only the primary path is used for transferring data. Other paths are redundant and are used when the network fails. Thus an SCTP connection is referred as an "association" between two sets of IP addresses. Thirdly, it defends against state-exhaustion attacks using a *cookie* mechanism and a four way handshake during connection establishment.

2.1 SCTP Packet Format

An SCTP packet comprises a common header and one or more chunks as shown in Fig. 1. The SCTP header contains 16 bit source and destination port numbers, a 32 bit verification tag and a 16 bit checksum. The verification tag is used to protect an association from blind attacks. Each end point keeps two values of verification tag: "My Verification Tag" and "Peer's Verification Tag". In general, any received packets containing a verification tag differing from "My Verification Tag" will be discarded. On the other hand, sent packets will carry a verification tag equal to "Peer's Verification Tag". These tag values are randomly selected at initialization and exchanged between the end points during association set up.

A Chunk is an information unit. There are 12 different control chunks but only one data chunk. The control chunks are Init[1], InitAck, SACK, Heartbeat, HeartbeatAck, Abort, Shutdown, ShutdownAck, Error, CookieEcho, CookieAck and ShutdownComplete. Control chunks are used to setup and shutdown the association, selectively acknowledge, report error messages, monitor reachability of the peer, etc. Association setup uses a four-way handshake comprising four control chunks: Init; InitAck; CookieEcho and CookieAck. Graceful closing uses

[1] Chunk names in the RFC are shown in all uppercase letters. To increase readability and distinguish them from SCTP States, the chunk names in this paper are given with only the first letters capitalized instead.

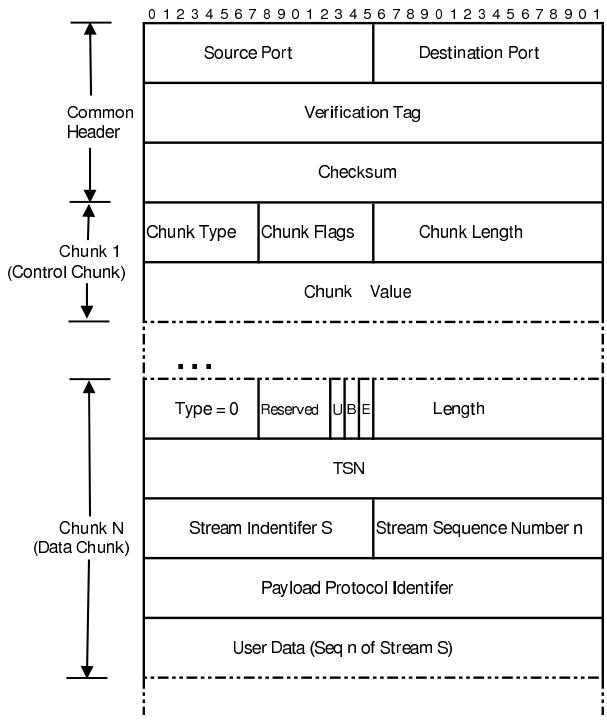


Fig. 1. SCTP Packet Format.

a three-way handshakes comprising three control chunks: the Shutdown; ShutdownAck and ShutdownComplete chunks. The Data transfer phase involves Data and SACK (Selective Acknowledgement) chunks. Further detail of the structure of chunks can be found in [4].

2.2 SCTP Connection Management Procedures

The state diagram shown in Fig. 2 illustrates the connection management procedures of SCTP. It comprises eight states: CLOSED; COOKIE-WAIT; COOKIE-ECHOED; ESTABLISHED; SHUTDOWN PENDING; SHUTDOWN-SENT; SHUTDOWN-RECEIVED and SHUTDOWN-ACK-SENT. The typical association establishment and close down procedures are shown in Fig. 3.

Normal Association Establishment Figure 2 (a) shows the association set up state diagram, and Fig. 3 (a) shows a typical set up procedure. An association between two nodes, A and Z, is initiated by a SCTP user on node "A" issuing an "ASSOCIATE" command. After receiving the "ASSOCIATE" primitive, node A sends an SCTP packet with a verification tag (VTAG) equal to zero. This SCTP packet contains only an Init chunk with an initial tag to specify the verification tag of incoming packets. Then node A enters the COOKIE-WAIT state. On receiving the Init chunk, node Z replies with an InitAck chunk indicating that it is willing to communicate with node A. The response includes node Z's initial tag number and encrypted cookie containing enough information to create node Z's Transmission Control Block (TCB). To prevent state

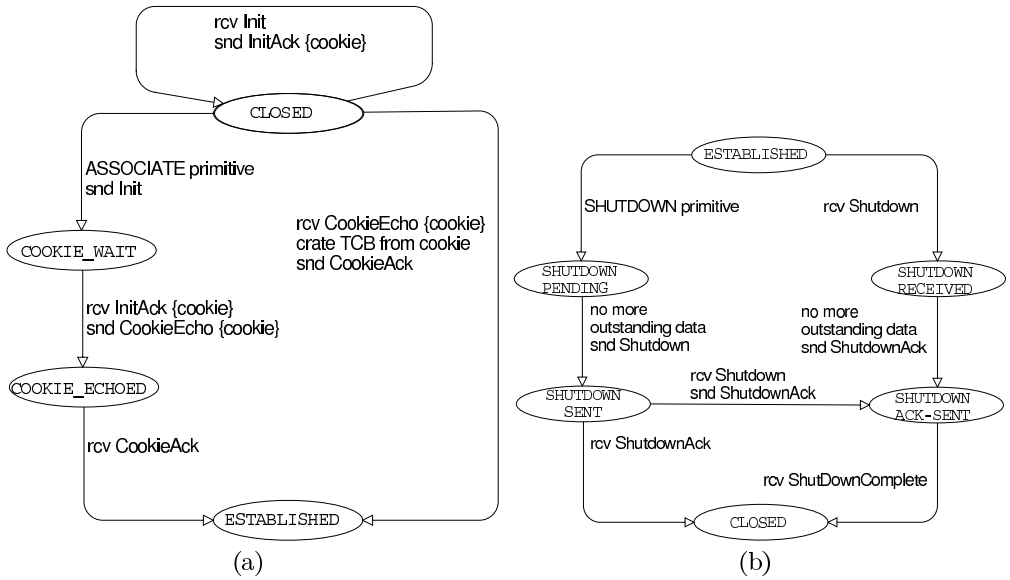


Fig. 2. SCTP State Diagram (a) association set up (b) closing down.

exhaustion attacks node Z is still in CLOSED after replying with an InitAck. To acknowledge the InitAck, node A returns the cookie in a CookieEcho chunk and enters COOKIE-ECHOED. When carrying an Init or InitAck chunk, the SCTP packet comprises only one chunk. When sending a CookieEcho chunk, the SCTP packet may enclose Data chunks after the CookieEcho chunk. On receiving a CookieEcho from node A, node Z creates its TCB from the received cookie, enters the ESTABLISHED state, replies with CookieAck and is ready for data transfer. After receiving CookieAck, node A enters ESTABLISHED indicating that the association is established. During data transfer, endpoint nodes A and Z may exchange Data and SACK chunks.

Graceful Association Shutdown Figure 2 (b) shows the association close down state diagram, and Fig. 3 (b) shows a typical graceful close down procedure. When the application at node A issues a "SHUTDOWN" command, node A enters the SHUTDOWN PENDING state and waits for all outstanding data chunks to be acknowledged. This end point stops accepting new data from the user. After all remaining data is acknowledged, node A sends a Shutdown chunk and enters the SHUTDOWN-SENT state. When node Z receives a Shutdown chunk, it must enter SHUTDOWN-RECEIVED, stop accepting new data from its user, and remain in this state until all outstanding data chunks are acknowledged. After all remaining data is acknowledged, node Z sends a ShutdownAck chunk and enters the SHUTDOWN-ACK-SENT state. After node A receives a ShutdownAck chunk, it must respond with a ShutdownComplete chunk and enter the CLOSED state. When node Z receives the ShutdownComplete chunk in SHUTDOWN-ACK-SENT, it enters the CLOSED state.

Besides the typical association set up and closing down procedures, the SCTP specification allows the simultaneous opening and simultaneous closing down of associations. For instance, when an endpoint in SHUTDOWN-SENT

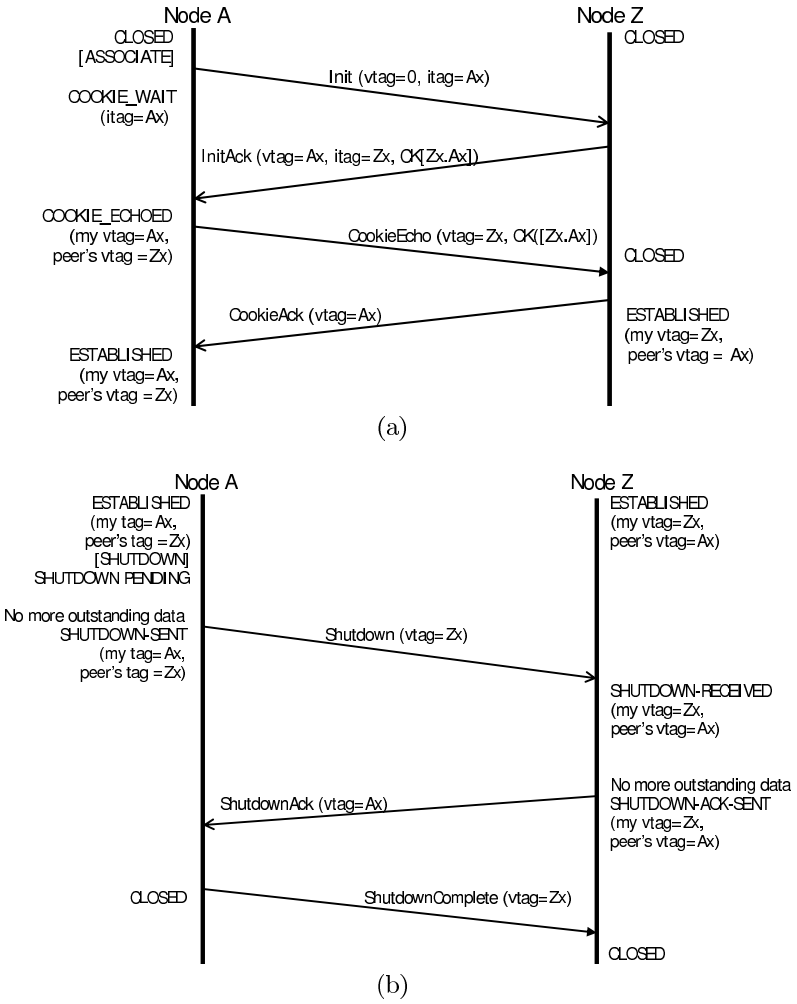


Fig. 3. Typical message sequence charts (a) association set up (b) closing down.

receives a Shutdown chunk, it sends a ShutdownAck in response and enters SHUTDOWN-ACK-SENT.

Handling Unexpected Init, InitAck, CookieEcho, and CookieAck Besides typical set up and closedown procedures, RFC 4960 specifies the rules to handle duplicate and unexpected Init, InitAck, CookieEcho, and CookieAck chunks in Section 5.2. These rules are intended to identify and solve problems that occur in the following scenarios.

1) An association is already established and both sides are in ESTABLISHED. An end point crashes and attempts to restore the association by sending a new Init chunk with a new Initial Tag.

2) Both end points attempt to open the association simultaneously.

3) The control chunk used to establish the association is stale.

4) An attacker generates a false SCTP packet.

5) The peer keeps retransmitting CookieEcho chunks and never receives a CookieAck.

Section 5.2 of RFC 4960 discusses the definition of *Tie-Tags*. *Tie-Tags* are copies of two verification tags (my verification tag and peer's verification tag). Actions specified in RFC 4960 that significantly differ from RFC 2960 are how to store and use the Tie-Tags. RFC 2960 specifies the Tie-Tags being stored in the cookie only but RFC 4960 requires to store the Tie-Tags in both cookie and TCB. The Tie-Tags in the TCB are called "Local Tag" and "Peer's Tag". The Tie-Tags in the cookie are called "Local Tie-Tag" and "Peer's Tie-Tag". The Tie-Tags are used to tie the received cookie of the new association with the old association. The Local Tie-Tag and the Peer's Tie-Tag (in the cookie) are compared with the Local Tag and the Peer's Tag (in the TCB) to ensure that the cookie belongs to the current association.

Section 5.2.4 of RFC 4960 discusses how SCTP responds when receiving an unexpected CookieEcho chunk. The Tie-Tags and verification tags in the cookie are compared with the verification tags in the existing TCB to identify which scenario occurs. Thus the received CookieEcho chunk can be correctly handled. An example of the scenarios is an association restart. When one side crashes and loses its existing TCB, Tie-Tags are used to link the restart association to the original association without shutting down and starting a new association.

3 Modelling Scope and Assumptions

Our model comprises all the state transitions of Fig. 2, and incorporates the narrative description from the RFC 4960 [4] Section 5.1, 5.2, 8.4, 8.5, 9.1 and 9.2. We also make the following assumptions regarding SCTP connection management when creating our CPN model.

1. We only consider a single association instance, while ignoring the procedures for data transfer, congestion control and other options. One SCTP packet contains only one chunk. A SCTP packet is modelled by chunk type, verification tag, initial tag and cookie. A cookie is modelled by "My Verification Tag" and "Peer's Verification Tag", "Local-Tie-Tag" and "Peer's-Tie-Tag".

2. Other fields in the SCTP packet are omitted because they do not affect the operation of the connection management procedure.

3. We do not consider misbehaviour or malicious attack.

4. Reordered or lossy channels may mask out possible deadlock, such as unspecified receptions. Thus we follow the incremental procedure outlined in [1] and analyse the CPN model with the following channel characteristics: FIFO without loss, reordered without loss, FIFO with loss, and reordered with loss, using the method proposed in [8]. However due to space limitations and for the sake of readability, we only discuss the case when the communication channels can delay and reorder packets without loss.

4 CPN Model of SCTP Connection Management

This section describes our CPN model of SCTP association establishment and shutdown procedures. Influenced by [2,7], the hierarchical structure of our CPN model is a procedure-based style. It comprises four hierarchical levels, 6 places,

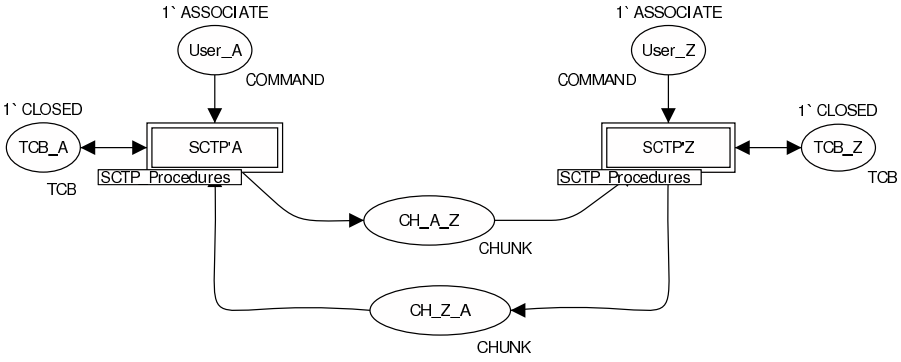


Fig. 4. The Top-level CPN page.

16 substitution transitions, 54 executable transitions and 2 ML functions. The top-level page of the SCTP-CPN model is illustrated in Fig. 4. Two substitution transitions (SCTP'A and SCTP'Z) represent the SCTP end point nodes, A and Z. Each side connects to four places. One place represents an application user typed by COMMAND. Another models a Transmission Control Block typed by TCB. Both end points are connected via two channel places, CH_A_Z and CH_Z_A. We assume that during association set up and closing down a packet contains only one chunk. Thus the channel places are typed by CHUNK, defined in Fig. 7. The layout of the top level CPN page also reflects the well-known model of the n-layer in a layered protocol architecture. The application layer is placed on the top while the underlying medium layer is below the protocol entity.

The substitution transitions, SCTP'A and SCTP'Z, are linked to the second level page named SCTP_Procedures, shown in Fig. 5. We divide SCTP_Procedures into five categories: normal events; unexpected events; retransmission; abort and checking Invalid Tags. Shown in Fig. 6 (a), the normal events comprise NormalEstablish and NormalShutDown of an association. The unexpected events are when the end points receive unexpected packets. We group the unexpected events into three CPN substitution transitions according to chunk types: UnexpectedIntIntAck; UnexpectedCookieEchoCookieAck and UnexpectedShutdown, shown in Fig. 6 (b). Space limitation prevents us from including all CPN model pages. Thus this paper will illustrate only five CPN pages: Normal'Establish, Normal'Shut_Down, Unexpected'Int_IntAck, Unexpected'CookieEcho_CookieAck and Unexpected'Shutdown.

With a state-based approach a CPN page represents several actions that may be scattered through the narrative specification. When modelling SCTP connection management with the procedure-based style, actions in each CPN page are confined to only a few sections in RFC 4960, as illustrated in Table 1. This makes our CPN model easier to understand when reading it alongside RFC 4960.

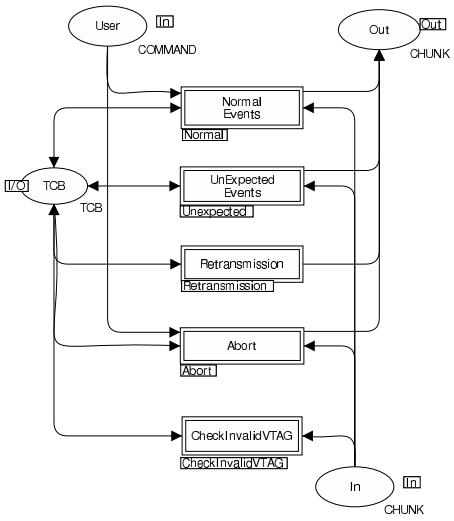


Fig. 5. The SCTP_Procedures page.

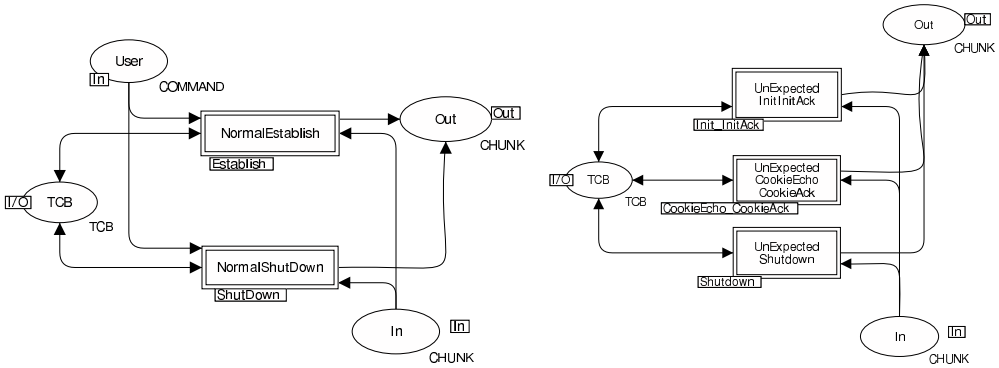


Fig. 6. (a) The Normal page (b) The Unexpected page.

4.1 Definition of CHUNK

When considering an abstract representation of SCTP packets, unlike the packets of TCP and DCCP, we found that Transmission Sequence Numbers (TSN) and Stream Sequence numbers are used only during the data transfer phase and are not relevant to SCTP connection management. On the other hand, Verification Tags (VTAG) play a major role during SCTP association establishment. Figure 7 defines the data structure of an SCTP packet called CHUNK (recall that we model each packet as a single chunk only). Many chunks have a different formats, thus we declare CHUNK (line 10) as the union of nine colour sets: VTAGxITAG (for Init chunk), VTAGxITAGxCOOKIE (for InitAck chunk), VTAGxCOOKIE (for CookieEcho chunk), four sets of VTAG (for CookieAck, Data, Shutdown and ShutdownAck chunks) and two sets of TFLGxVTAG (for Abort and ShutdownComplete chunks). Chunk types are distinguished by the ML selectors shown in line 10 to 14.

Procedures	CPN Page	Relevant Sections
Normal Event		
Normal establishment	Establish	5.1
Normal Shutdown	Shut_Down	9.2
Unexpected Events		
Receiving unexpected Init chunk	Init_InitAck	5.2.1, 5.2.2
Receiving unexpected InitAck chunk	Init_InitAck	5.2.3
Receiving unexpected CookieEcho chunk	CookieEcho_CookieAck	5.2.4
Receiving unexpected CookieAck chunk	CookieEcho_CookieAck	5.2.5
Receiving unexpected Shutdown chunk	Shutdown	9.2
Receiving unexpected ShutdownAck chunk	Shutdown	9.2
Receiving unexpected ShutdownComplete chunk	Shutdown	9.2
Abort	Abort	9.1
Retransmission	Retransmission	5.1, 9.2
Validation of VTAG	CheckInvalidVTAG	8.4, 8.5

Table 1. Relationship between SCTP procedures, CPN pages and sections in RFC 4960.

```
 1: colset VTAG = int;
 2: colset VTAGxITAG = product VTAG * VTAG; (* Init and InitAck Chunk *)
 3: colset MyVTAGxPeerVTAG = product VTAG * VTAG;
 4: colset COOKIE = record CK_TAG:MyVTAGxPeerVTAG
 5:                        * CK_TT:MyVTAGxPeerVTAG;
 6: colset VITAGxCOOKIE = record VI:VTAGxITAG * CK:COOKIE; (* InitAck chunk *)
 7: colset VTAGxCOOKIE = record VT:VTAG * CK:COOKIE; (* CookieEcho Chunk *)
 8: colset TFLAG = with T_ON | T_OFF;
 9: colset TFLAGxVTAG = product  TFLAG * VTAG;
10: colset CHUNK = union Init:VTAGxITAG + InitAck:VTAGxITAGxCOOKIE
11:                    + CookieEcho:VTAGxCOOKIE + CookieAck:VTAG
12:                    + Data:VTAG + Abort:TFLAGxVTAG
13:                    + Shutdown:VTAG + ShutdownAck:VTAG
14:                    + ShutdownComplete:TFLAGxVTAG;
```

Fig. 7. The definition of SCTP Chunk.

Line 1 defines the basic unit, VTAG, as the set of integers. Although an integer in CPN Tools is not a 32-bit unsigned integer like the VTAG field shown in Fig. 1, this does not affect the analysis of the model. Notice that according to our model abstraction, besides T-Flag, CHUNK comprises only VTAG and the compositions of VTAGs.

Each endpoint shall keep *my* and *peer's verification tags* in its TCB. When an endpoint sends out a packet, the value of *peer's verification tag* is copied into VTAG field of the outgoing chunks. If an endpoint receives an SCTP packet of which VTAG field does not match *my verification tags*, the packet shall be silently discarded. Peer's verification tags of both sides are initialized by exchanging the value in the Initial Tag (ITAG) field of the Init and InitAck chunks (see Fig. 3 (a)). Thus the data structure of the Init chunk is defined as the product of two verification tags (VTAGxITAG - line 2). The InitAck chunk (line 6) is modelled by a record of VTAGxITAG and COOKIE. Despite a cookie containing a lot of parameters, we model COOKIE (line 4) as a record of two pairs of VTAG: (*My Verification Tag*, *Peer's Verification Tag*) and (*Local-Tie-Tag*, *Peer-Tie-Tag*).

Line 7 defines CookieEcho chunk as the record of VTAG and COOKIE, whereas CookieAck, Shutdown and ShutdownAck are modelled by only VTAG.

Abort and ShutdownComplete chunks use T-Flag to indicate whether, when sending out these chunks, the TCB exists or not. If the TCB does exist, the Abort and ShutdownComplete chunks have T-Flag set to off and VTAG equal to the peer's verification tag. When TCB does not exist, T-flag is on and the VTAG field equals the verification tag of the received packet the SCTP entity is responding to. Thus Abort and ShutdownComplete chunks (line 9) are defined by the product of TFLAG and VTAG (TFLAGxVTAG).

4.2 Definition of TCB

During each stage of association establishment the TCB of a SCTP endpoint may record different data. CLOSED means there is no connection, and no state parameters exist. A SCTP node in the COOKIE-WAIT state has no knowledge of the *peer's verification tag* but knows only *my verification tag*. Both verification tags are known in the COOKIE-ECHOED state but a SCTP node may retransmit the CookieEcho chunk together with the echoed cookie. Thus the SCTP node needs to store the cookie for the purpose of retransmission because the content in the cookie is encrypted and cannot be recreated.

According to the differences in the TCB data structure we describe above, we divide SCTP states into four groups: CLOSED, COOKIE-WAIT, COOKIE-ECHOED and the states after association established. Figure 8 declares TCB in line 12 as a union set of empty set, COOKIEWAIT_CB,COOKIE_ECHOED_CB and SCTP_CB. SCTP's state are distinguished by ML selectors as defined in line 12. Similar to CHUNK, besides the retransmission counter, a TCB comprises only VTAG and the compositions of VTAGs.

```
 1: colset RCNT = int;
 2: colset COOKIEWAIT_CB = record Rcnt:RCNT * myvtag:VTAG
 3:                              * SV_TT:MyVTAGxPeerVTAG;
 4: colset SV_in_TCB = record Rcnt:RCNT
 5:                         * SV_VT:MyVTAGxPeerVTAG
 6:                         * SV_TT:MyVTAGxPeerVTAG;
 7: colset COOKIE_ECHOED_CB = product SV_in_TCB * COOKIE;
 8: colset TCB_EXIST_STATE = with ESTABLISH | SHUTDOWN_PENDING
 9:                             | SHUTDOWN_RECEIVED | SHUTDOWN_SENT
10:                             | SHUTDOWN_ACK_SENT;
11: colset SCTP_CB = product TCB_EXIST_STATE * SV_in_TCB;
12: colset TCB = union CLOSED + COOKIE_WAIT:COOKIEWAIT_CB
13:                + COOKIE_ECHOED:COOKIE_ECHOED_CB
14:                + TCBExist:SCTP_CB;
15: colset COMMAND = with ASSOCIATE | SHUTDOWN | ABORT;
```

Fig. 8. The definition of SCTP's Transmission Control Block (TCB).

Differing from RFC 2960, in addition to only storing Tie-Tags in the cookie, Section 5.2.2 of RFC 4960 defines the Local Tag and Peer's Tag be stored in association's TCB. Thus line 4 declares state variables in TCB, SV_in_TCB, as a record of retransmission counter (RCNT - line 1) and two products of verification tags: (my verification tag, peer's verification tag) and (Local Tag, Peer's Tag).

After association establishment, a SCTP node in our model has TCB defined as a product of state after the association established (TCB_EXIST_STATE- line 8) and state variables SV_in_TCB.

COOKIE_ECHOED_CB (Line 7) is defined as a product of SV_in_TCB and COOKIE (line 4 of Fig. 7). Line 2 declares COOKIE_WAIT_CB as a record of retransmission counter (RCNT), my verification tag (VTAG) and a pair of VTAGs (Local Tag, Peer's Tag). Line 15 defines user commands which are ASSOCIATE, SHUTDOWN and ABORT.

4.3 CPN Model of SCTP Normal Procedures

This subsection illustrates two CPN pages of typical scenarios of SCTP connection management: association establishment and graceful shutdown. Beginners who have just studied this protocol can easily understand these two CPN pages because they are similar to state diagrams (Fig. 2) and the typical message sequence chart in Fig. 3.

Normal Establishment Page The normal establishment procedures described in section 5.1 of RFC 4960 are modelled in the Normal'Establish page shown in Fig. 9. Five transitions represent a sequence of actions directly mapping from the message sequence chart in Fig. 3 (a). In addition to sending outgoing chunks and changing states, SCTP nodes validate VTAG fields and populates the values of Tie-Tags into the cookie and TCB. The guard on transition Rcv_Init requires the VTAG of the Init chunk to be equal to zero. According to last paragraph of section 5.2.2 page 67 of RFC 4960 [4], the value of Tie-Tags in CLOSED, COOKIE-WAIT and SHUTDOWN-ACK-SENT shall be set to zeros. Thus the Tie-Tags in the InitAck chunk and in TCB of COOK-WAIT state are set to zeros.

Normal Shutdown Page Figure 10 shows a CPN page of the normal graceful shutdown procedures described in section 9.2 of RFC 4960. Despite the normal shutdown procedure shown in Fig. 3 (b) comprising only SHUTDOWN primitives and three-way handshakes, the Normal'Shut_Down page contains 9 transitions. Unlike association establishment, section 9.2 of RFC 4960 does not separate the normal closing down and unexpected closing down procedures. Hence, in addition to the typical actions of SHUTDOWN primitives and three-way handshakes, this page includes

1) Clearing outstanding data when receiving Data chunks in SHUTDOWN-PENDING and SHUTDOWN-RECEIVED;

2) Retransmitting Shutdown chunks when receiving Data chunks in the SHUTDOWN-SENT state;

3) Receiving Shutdown chunks in the SHUTDOWN-RECEIVED state.

4.4 CPN Models of Unexpected Procedures

Handling unexpected receptions of SCTP control chunks is modelled by three CPN pages: Unexpected'Int_IntAck, Unexpected'CookieEcho_CookieAck, and Unexpected'Shutdown. These are illustrated in this section.

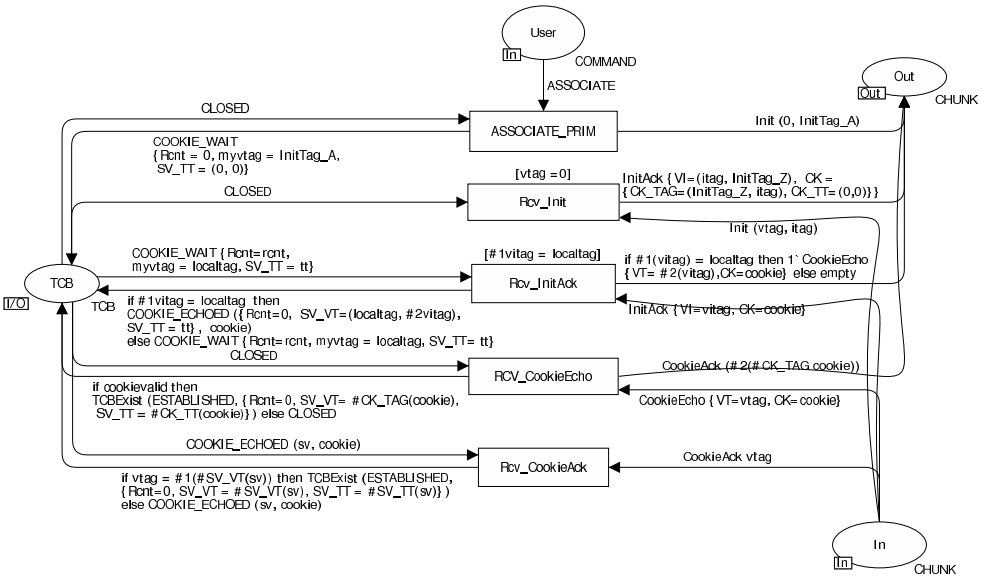


Fig. 9. The Normal'Establish page.

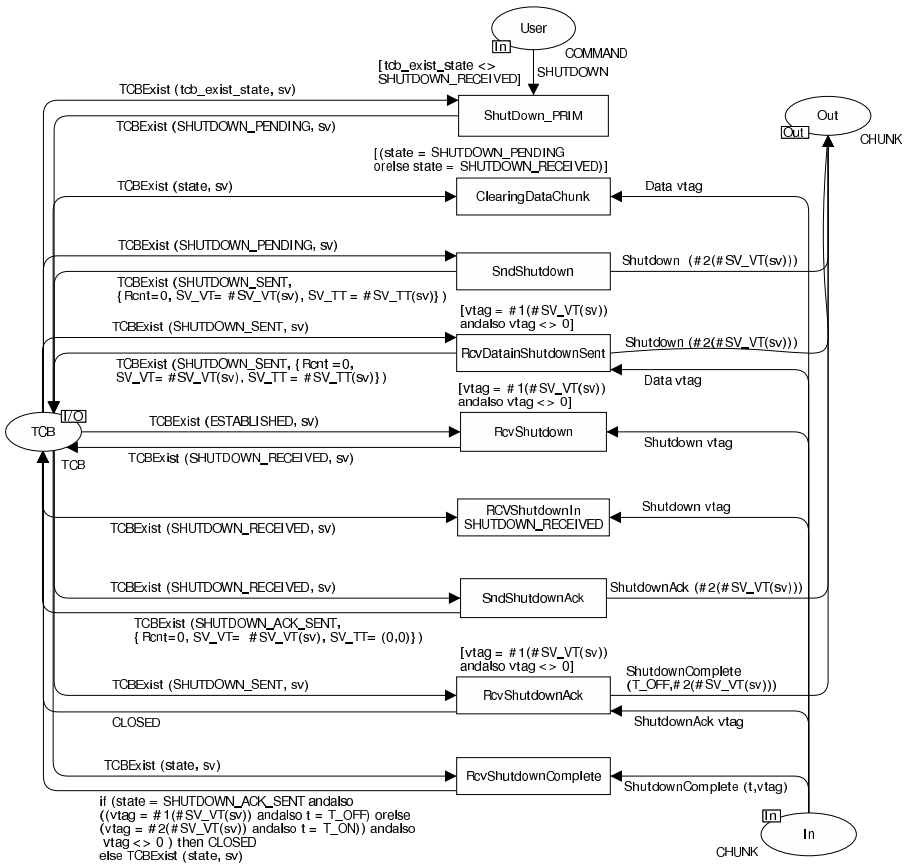


Fig. 10. The Normal'Shut_Down page.

Unexpected Init and InitAck Page Figure 11 shows the CPN page dealing with the unexpected events of receiving Init and InitAck chunks in states other than CLOSED. Transitions RcvInit_CK_WAIT and RcvInit_CK_ECHOED model the actions according to section 5.2.1 of RFC 4960 [4] when an endpoint receives an Init chunk in the COOKIE-WAIT or COOKIE-ECHOED state. The difference between these actions is that the Tie-Tags from the COOKIE-WAIT state are set to zeros but from COOKIE-ECHOED, they are set to the current verification tags. Transitions Rcv_InitOtherThan models the action according to section 5.2.2 of RFC 4960 when the endpoints receive unexpected Init chunks in states other than CLOSED, COOKIE-WAIT and COOKIE-ECHOED. The action is similar to that of transition RcvInit_CK_ECHOED but the "my verification tag" in the cookie and Initial Tag in the InitAck chunk are set to a new value instead of the old value of the Initial tag (InitTag_Z). Transition RcvInit_in_SHUTDOWN_ACK_SENT models the action according to the sixth paragraph of section 9.2 of RFC 4960. After receiving an Init chunk in SHUTDOWN-ACK-SENT, the SCTP node discards the Init chunk but retransmits a ShutdownAck chunk. Transition Rcv_InitAck models the action according to section 5.2.3 of RFC 4960. The SCTP node silently discards any unexpected InitAck chunks if receiving them in states other than COOKIE-WAIT

Unexpected CookieEcho and CookieAck Page Figure 12 shows the CPN page dealing with the unexpected events of receiving CookieEcho chunks in states other than CLOSED; and receiving CookieAck in states other than COOKIE-ECHOED. When receiving CookieAck in states other than COOKIE-ECHOED, (transition Rcv_CookieAck), the SCTP node silently discards the

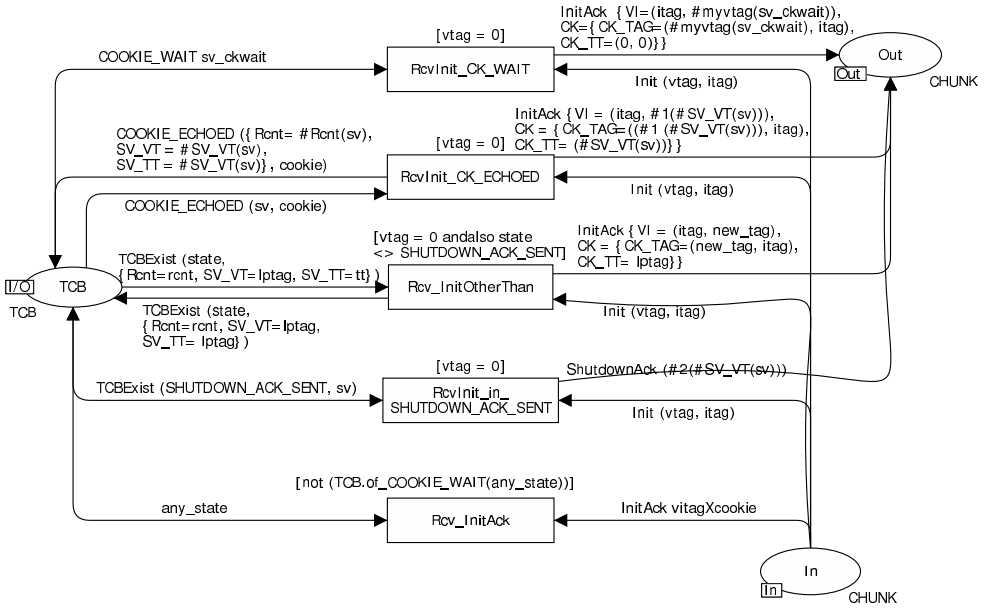


Fig. 11. The Unexpected'Init_InitAck page.

CookieAck chunk. Four substitution transitions, Restart, Simultaneous_Open, Delayed_Cookie and Tags_match, model the actions described in section 5.2.4 of RFC 4960. While we modeled this CPN page, we found an error[2] in Table 2 of section 5.2.4 of RFC 4960. The column headers of the Table, "Local Tag" should be "Local verification tag in the received cookie" and "Peer's tag" should be "Peer's verification tag in the received cookie".

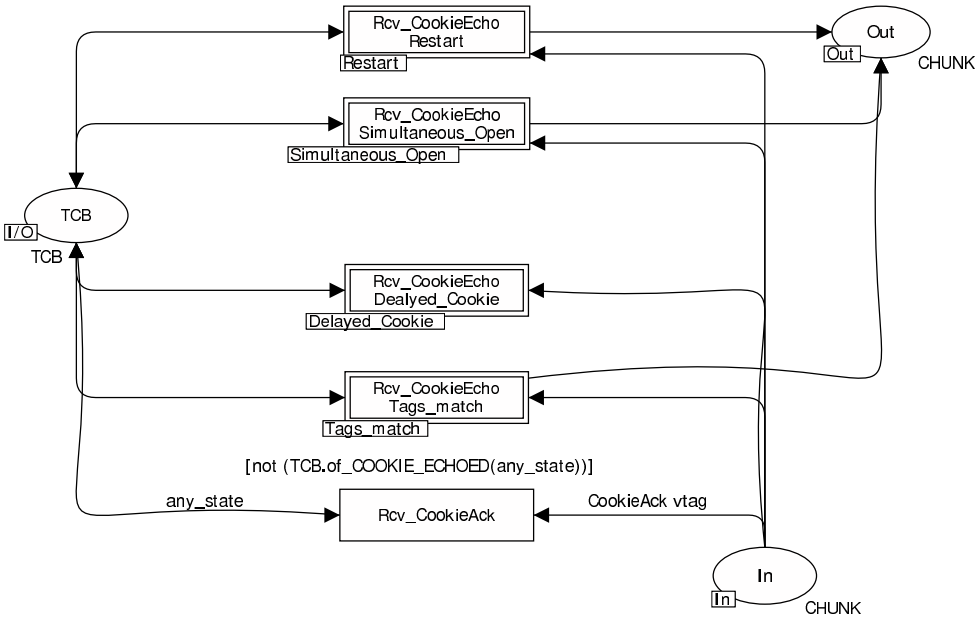


Fig. 12. The Unexpected'CookieEcho_CookieAck page.

Unexpected Shutdown Page This CPN page represents the events when a SCTP endpoint unexpectedly receives shutdown control chunks (Shutdown, ShutdownAck and ShutdownComplete). The states in the establishment phase, COOKIE-WAIT and COOKIE-ECHOED, should not receive the shutdown control chunks. SHUTDOWN-SENT should not receive the Shutdown chunk. SHUTDOWN-ACK-SENT should not receive the ShutdownAck chunk. The reception of shutdown control chunks in CLOSED is modelled by the first transition. The second, third and fourth transitions model the receptions of Shutdown, ShutdownAck and ShutdownComplete respectively in the COOKIE-WAIT and COOKIE-ECHOED states. When a node receives ShutdownAck in either COOKIE-WAIT or COOKIE-ECHOED, the node replies with ShutdownComplete. The T-Flag is set and the VTAG of the outgoing packet is set equal to the VTAG of the incoming packet.

The fifth transition models the reception of a Shutdown chunk in the SHUTDOWN-SENT state. The node replies with ShutdownAck and enters the SHUTDOWN-ACK-SENT state. The sixth transition represents the reception

[2] See Transport Area Discussion Archive
http://www.ietf.org/mail-archive/web/tsvwg/current/msg08603.html.

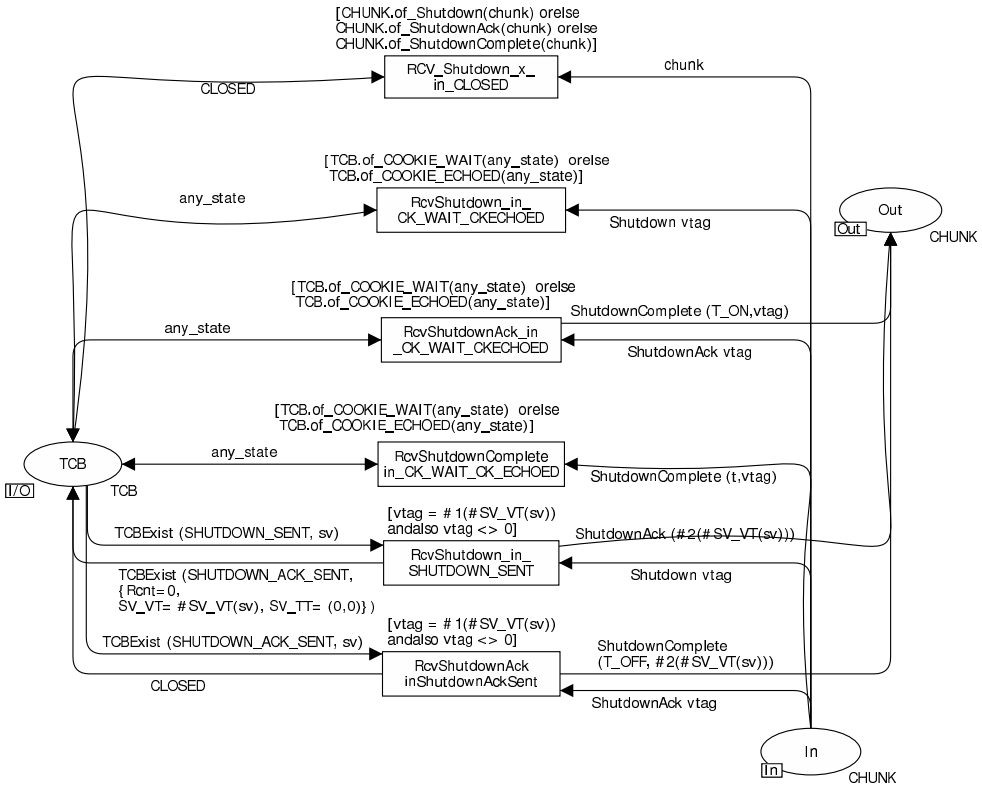


Fig. 13. The Unexpected'Shutdown page.

of a ShutdownAck chunk in the SHUTDOWN-ACK-SENT state. The node replies with ShutdownComplete and goes to CLOSED.

5 Analysis of DCCP-CPN Connection Management Model

5.1 Initial configuration

We analyse our SCTP connection management model using CPN Tools version 2.2.0 on an AMD Athlon 1.79 GHz computer with 2GB RAM. The SCTP-CPN model is initialised by distributing initial tokens to places User_A and User_Z. TCB_A and TCB_Z of the model to create the initial marking. Table 2 shows the initial values of the user commands and TCB. All channel places are empty. All initial tags used during association establishment are randomly generated.

We analyse five cases. Case A is association establishment. Both node A and Z are initially in CLOSED. The user of node A issues an "ASSOCIATE" command to start an association. Case B is simultaneous establishment, where both sides attempt to initiate the association at about the same time. Case C is graceful shutdown. Both sides are in ESTABLISHED, and the user at node A issues a SHUTDOWN command. Case D is simultaneous graceful shutdown when both sides try to close the association at about the same time. Case E is ungraceful abort. Both sides are in ESTABLISHED, and the user at node A

Case	Initial Markings in place			
	User_A	User_Z	TCB_A	TCB_Z
A	1‘ASSOCIATE		CLOSED	CLOSED
B	1‘ASSOCIATE	1‘ASSOCIATE	CLOSED	CLOSED
C	1‘SHUTDOWN		TCBExist (ESTABLISHED,	TCBExist (ESTABLISHED,
D	1‘SHUTDOWN	1‘SHUTDOWN	{Rcnt=0,SV_VT=(2,3)	{Rcnt=0, SV_VT=(3,2)
E	1‘ABORT		SV_TT=(2,3)})	SV_TT=(3,2)})

Table 2. Initial Configurations.

issues an ABORT command. The initial markings for these cases are shown in Table 2.

5.2 Analysis Results

The analysis results of our SCTP Connection Management CPN model using the initial configurations in Table 2 are shown in Table 3. The 4-tuple in the first column is the maximum retransmissions allowed for Init, CookieEcho, Shutdown and ShutdownAck respectively. An “x” indicates that the retransmission of those chunk types does not occur in that configuration. The state space tool (in CPN Tools) provides the number of nodes, arcs and terminal markings. In all cases (A to E) in Table 2 the number of nodes and arcs in the Strongly Connected Component (SCC) Graph are the same as the number of nodes and arcs in the state space. Thus no livelocks are found.

We classify the terminal markings into four categories based on the SCTP endpoint states:

TYPE-I CL-CL: both sides terminate in CLOSED.

TYPE-II EST-EST: both sides terminate in ESTABLISHED.

TYPE-III CL-EST: node A terminates in CLOSED but node Z in ESTABLISHED.

TYPE-IV EST-CL: node A terminates in ESTABLISHED but node Z in CLOSED.

Cases C to E in Table 2 are when the association is terminated. All terminal markings of cases C to E are TYPE-I (CLOSED-CLOSED) which is desirable, and no other deadlocks are found. Case D has three terminal markings because there is the possibility of one user command token (SHUTDOWN) remaining in either place User_A or User_Z.

Case A and B are when SCTP’s nodes attempt to establish an association. TYPE-I terminal marking (CLOSED-CLOSED) is a desirable terminal marking when the association can not be established thus both sides go to CLOSED state (No connection). TYPE-III and TYPE-IV terminal markings occur when one side is in ESTABLISHED while the other is in CLOSED. This can happen when the maximum number of retransmissions of the CookieEcho chunk is reached and the node enters CLOSED before CookieAck arrives. An example of this scenario is shown in Fig. 14. Although TYPE-III and TYPE-IV are unwanted, they are not harmful. This is because when the peer is considered unreachable, SCTP will report the failure to its user so that the user may decide to re-initiate

Case	Nodes	Arcs	Time	Terminal Markings			
			(sec)	(I)CL-CL	(II)EST-EST	(III)CL-EST	(IV)EST-CL
A-(0,0,x,x)	12	13	0	1	1	1	0
A-(1,0,x,x)	38	58	0	1	2	2	0
A-(0,1,x,x)	20	26	0	1	1	1	0
A-(1,1,x,x)	78	147	0	1	2	2	0
A-(2,2,x,x)	345	905	1	1	2	2	0
A-(3,3,x,x)	1,124	3,535	1	1	2	2	0
B-(0,0,x,x)	380	756	0	1	16(11)	4	4
B-(1,0,x,x)	8,206	25,778	41	1	36(20)	7	7
B-(0,1,x,x)	1,087	2,498	1	1	16(11)	4	4
B-(1,1,x,x)	31,295	113,302	579	1	36(20)	7	7
C-(x,x,0,0)	14	20	0	1	0	0	0
C-(x,x,1,0)	28	50	0	1	0	0	0
C-(x,x,0,1)	22	36	0	1	0	0	0
C-(x,x,1,1)	45	87	0	1	0	0	0
D-(x,x,0,0)	106	234	0	3	0	0	0
D-(x,x,1,0)	415	1,178	0	3	0	0	0
D-(x,x,0,1)	253	598	0	3	0	0	0
D-(x,x,1,1)	1,125	3,604	1	3	0	0	0
D-(x,x,2,2)	6,024	22,294	16	3	0	0	0
E-(x,x,x,x)	3	2	0	1	0	0	0

Table 3. State space analysis results.

the ASSOCIATE command. Thus the association can be restored as described in Fig. 5 of RFC 4960.

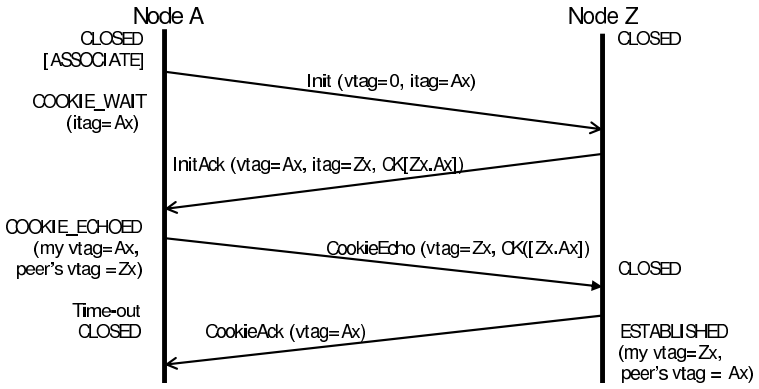


Fig. 14. A scenario leads to a terminal marking TYPE III (half open state).

TYPE-II terminal markings should be desirable when both sides successfully establish the association. However when we check the verification tags stored in the TCBs, some terminal markings of TYPE-II are undesirable. #1SV_VT in TCB_A must equal #2SV_VT in TCB_Z and vice versa, otherwise all received data packets will be discarded. In column TYPE-II, the number in parenthesis is the number of TYPE-II terminal markings in which verification tags between both TCBs match each other. For example, in case B(0,0,x,x), eleven terminal markings of TYPE-II have verification tags matched to each other.

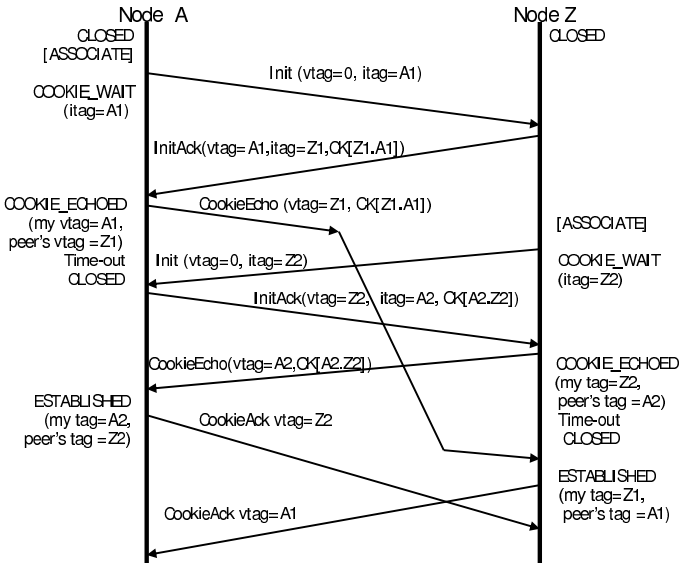


Fig. 15. A scenario when both sides reach ESTABLISHED with mismatched verification tags.

According our investigation, one cause of this problem is shown in Fig. 15. Node A starts initiating the association. The setup sequence proceeds according to the typical scenario but the CookieEcho chunk from node A is delayed. While waiting for CookieAck, node A reaches the maximum number of retransmission of the CookieEcho chunk, thus node A goes to CLOSED. Meanwhile after replying InitAck, node Z initiates the association establishment (simultaneous establishment) using a different set of verification tags. Similar to node A, while waiting for CookieAck, node Z retransmits the maximum number of retransmission of CookieEcho chunks, and goes to CLOSED. After both sides are in CLOSED, CookieEcho's arrive at both sides. Both sides process the cookies and authenticate the State Cookie if it is the one that it has just generated. Both sides create their TCBs from the received cookies and go to ESTABLISHED with mismatched verification tags. Notice that this problem does not relate to Tie-Tags mechanism or section 5.2 of RFC 4960.

6 Conclusions and future work

This paper has presented a Coloured Petri Nets model and analysis of SCTP connection management. Our CPN model is based on the recent RFC 4960 rather than the obsolete RFC 2960. Besides the typical association establishment, graceful shutdown and abort, our CPN model includes the procedures for handling the reception of unexpected control chunks. In particular we attempt to model the use of Tie-Tags from RFC 4960 that differs significantly from the description in RFC 2960.

We build the procedure-based CPN model of SCTP connection management directly from RFC 4960. It took about two months (part time) to study the SCTP procedures, create the model and debug the model. The most critical problem of this project is to understand how to use Tie-Tags. When SCTP receives a CookieEcho chunk in the states other than CLOSED (section 5.2.4

of RFC 4960), Tie-Tags are used to identify complex scenarios such as an association restart and simultaneous establishment. Although we found an error in section 5.2.4 of RFC 4960 while we developed the SCTP-CPN model, to gain an insight into these complex scenarios requires more exhaustive analysis and more time.

Our initial state space analysis shows that the shutdown procedures have no deadlocks but the establishment procedures have undesired deadlocks. The undesired deadlocks are half open states, where one SCTP node is in CLOSED while the other is in ESTABLISHED. This deadlock could be easily solved by restarting the association. When the maximum number of retransmissions has been reached, SCTP must report to its user. Then the user can restart the association.

The second problem seems more severe because both sides are in ESTABLISHED with mismatched verification tags stored in their TCB. As far as we aware there is no existing discussion of this problem. The solution to the second problem seems to involve cookie authentication, which needs further investigation. In future, we are interested in modelling security attacks against SCTP as well as multi-homing.

Acknowledgments The author are thankful to the anonymous reviewers and also to Professor Jonathan Billington and Dr. Guy Gallasch. Their constructive feedback has helped to improve the quality of this paper.

References

1. J. Billington, G. E. Gallasch, and B. Han. A Coloured Petri Net Approach to Protocol Verification. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 210–290. Springer, Heidelberg, 2004.
2. J. Billington and S. Vanit-Anunchai. Coloured Petri Nets Modelling of an Evolving Internet Standard: the Datagram Congestion Control Protocol . *Fundamenta Informaticae*, In Press, 2008.
3. M. Martins M. G. Modelagem e Análise Formal de algumas Funcionalidades de um Protocolo de Transporte Atrvés das Redes de Petri. Master's thesis, Instituto Nacional de Telecomunicações (INATEL) , Santa Rita do Sapucaí, Brazil, December 2003.
4. R. Stewart Ed. Stream Control Transmission Protocol (SCTP), RFC4960. Available via http://www.rfc-editor.org/rfc/rfc4960.txt, September 2007.
5. R. Stewart, I. Arias-Rodriguez, K. Poon, A. Caro, M. Tuexen. Stream Control Transmission Protocol (SCTP) Specification Errata and Issues, RFC4460. Available via http://www.rfc-editor.org/rfc/rfc4460.txt, September 2007.
6. R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang and V. Paxson. Stream Control Transmission Protocol (SCTP), RFC2960. Available via http://www.rfc-editor.org/rfc/rfc2960.txt, October 2000.
7. S. Vanit-Anunchai and J. Billington. Modelling the Datagram Congestion Control Protocol's Connection Management and Synchronisation Procedures. In J. Kleijn and A. Yakovlev, editors, *Proceedings of the 28th International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN'07)*, volume 4546 of *Lecture Notes in Computer Science*, pages 423–444, Siedlce, Poland, 25-29 June 2007. Springer, Heidelberg.
8. S. Vanit-Anunchai, J. Billington, and G.E. Gallasch. A Combined Protocol Channel Model and its Application to the Datagram Congestion Control Protocol. In D. Moldt N. Sidorova and H. Rolke, editors, *Proceeding of the International Workshop on Petri Nets and Distributed Systems (PNDS08)*, pages 32–46, Xian, China,, 23-24 June 2008.

A discretization method from coloured to symmetric nets: application to an industrial example

Fabien **Bonnefoi**
DSO/DSETI,
Cofiroute,
6 - 10 rue Troyon,
92310 Sèvres, France
Fabien.Bonnefoi@cofiroute.com

Christine **Choppy**
LIPN, CNRS UMR 7030,
Université Paris XIII,
99 av. J-B Clément,
93430 Villetaneuse, France
Christine.Choppy@lipn.univ-paris13.fr

Fabrice **Kordon**
LIP6 - CNRS UMR 7606,
Université P. & M. Curie,
4 Place Jussieu,
75252 Paris Cedex 05, France
Fabrice.Kordon@lip6.fr

1 Introduction

Future supervision systems tend to be distributed and at least partially embedded. Distribution brings a huge complexity and then, a strong need to deduce possible (good and bad) behaviours on the global system, from the known behaviour of its actors. This is crucial since mission critical or life critical missions are more and more supervised by such systems. Intelligent Transport Systems (ITS) are a typical example: more and more functions tend to be integrated in vehicles and road infrastructure.

Moreover, in many cases (like ITS), physical constraints are part of the system description. Analysis techniques based on discrete models must integrate such constraints : we then speak of *hybrid* systems.

So, a major trend in formal analysis is to cope with such systems. This raises many issues in terms of analysis complexity. Some techniques are dedicated to continuous analysis such as algebraic approaches like B [1]. However, such approaches are difficult to set up and most industries prefer push-button tools.

Model checking easily offers such push-button tools but does not cope well with continuous systems. Most model checking techniques deal with discrete (finite) systems. Thus, management of hybrid systems is not easy or leads to potentially infinite systems that are difficult to verify (for example, management of continuous time requires much care, even to only have decidable models). Hybrid Petri Nets [15] might be a solution to model and analyze hybrid systems but no tool is available to test neither safety nor temporal logic properties [11].

In this paper, we propose a methodology to handle hybrid systems with model checking on Petri Nets and algebraic methods. Our methodology is based on transformations from Coloured Petri Nets (CPN) [25, 26] to Symmetric Petri Nets (SN) [9, 7].

CPN allow an easy modelling of the system to be analyzed. SN are of interest for their analysis because of the symbolic reachability graph that is efficient to represent the state space of large systems. Moreover, since SN only offer a limited set of operations on colours, transformation from CPN requires much care from the designer as regards the types to be discretized.

Our methodology also addresses an important question: what is the impact of discretization on the precision of verification? As in scientific computing, the discretization process may generate “precision errors” that could turn a given verified property into a wrong one. In that case, the property to be verified might have to be transformed to take into consideration such precision errors.

Section 2 briefly recalls the notions of CPN, SN and abstraction/refinement, type issues. Our methodology which involves modelling, discretization and verification is presented in Section 3, and we show in Section 4 how we model our Emergency Braking application. The various issues regarding discretization on our case study are detailed in Section 5, and issues on net analysis are presented in Section 6. Some open issues are discussed in Section 7 before a conclusion (Section 8).

2 Building Blocks

This section presents the building blocks from the state of the art used to set up our transformation methodology.

2.1 Coloured Petri Nets

Coloured Petri nets [25, 26] are high level Petri nets where tokens in a place carry data (or colours) of a given type. Since several tokens may carry the same value, the concept of multiset (or bag) is used to describe the marking of places.

In this paper, we assume the reader is familiar with the concept of multisets. We thus recall briefly the formal definition of coloured Petri nets as in [26]. It should be noted however that the types considered for the place tokens may be basic types (e.g. boolean, integers, reals, strings, enumerated types) or structured types – also called compound colour sets – (e.g. lists, product, union, etc.). In both cases, the type definition includes the appropriate (or usual) functions.

Different languages were proposed to support the type definition for coloured Petri nets (e.g. algebraic specification languages as first introduced in [33], object oriented languages [5]), and an extension of the Standard ML language was chosen for CPN Tools [13]. As always, there may be a tradeoff between the expressivity of a specification language, and efficiency when tools are used to compute executions, state graphs, etc. If expressivity is favored, it could be desirable to allow any appropriate type and function, while when tools should be used to check the behaviour and the properties of the system studied, the allowed types and functions are restricted (as the language allowed for CPN Tools or as in Symmetric Nets presented in Section 2.2). Here, we want to allow a specification language that fits as much as possible what is needed to describe the problem under study, and then show how the specification is transformed so as to allow computations and checks by tools.

In the following, we refer to $EXPR$ as the set of expressions provided by the net inscription language (net inscriptions are arcs expressions, guards, colour sets and initial markings), and to $EXPR_V$ as the set of expressions $e \in EXPR$ such that $Var[e] \subseteq V$.

Definition 2.1. *A non-hierarchical coloured Petri net CPN [26] is a tuple* $CPN = (P, T, A, \Sigma, V, N, C, G, E, I)$ *such that:*

1. P *is a finite set of places.*
2. T *is a finite set of transitions such that* $P \cap T = \emptyset$.
3. $A \subseteq P \times T \cup T \times P$ *is a set of directed arcs*
4. Σ *is a finite set of non empty colour sets (types).*
5. V *is a finite set of typed variables such that* $Type[v] \in \Sigma$ *for all variables* $v \in V$.
6. $C : P \rightarrow \Sigma$ *is a colour set function assigning a colour set (or a type) to each place.*
7. $G : T \rightarrow EXPR_V$ *is a guard function assigning a guard to each transition such that* $Type(G(t)) = Bool$, *and* $Var[G(t)] \subseteq V$, *where* $Var[G(t)]$ *is the set of free variables of* $G(t)$.
8. $E : A \rightarrow EXPR_V$ *is an arc expression function assigning an arc expression to each arc such that* $Type(E(a)) = C(p)_{MS}$, *where* p *is the place connected to the arc* a.
9. $I : A \rightarrow EXPR_V$ *is an initialisation function assigning an initial marking to each place such that* $Type(I(p)) = C(p)_{MS}$.

As explained in Section 3, the first step of our methodology is to produce a CPN model for the application under study. The next step is a transformation motivated by the discretization of continuous functions to obtain a symmetric net.

2.2 Symmetric Nets

Symmetric nets[1] were introduced in [9] and [7], with the goal of exploiting symmetries in distributed systems to provide a more compact representation of the state space.

[1] *Symmetric nets* were formerly known as *Well-Formed nets*, a subclass of *High-level Petri nets*. The new name was chosen in the context of the ISO standardisation of Petri nets [21].

The concept of symmetric nets is similar to the coloured Petri net one. However, the allowed types for the places as well as allowed colour functions are more restricted. These restrictions allow us to compute symmetries and obtain very compact representations of the state space, enabling the analysis of complex systems as in [22].

Basically, types must be finite enumerations and can only be combined by means of cartesian products. Allowed functions in arc valuation are: Id, successor, predecessor and broadcast (that generates one copy of any value in the type). These constraints affect points 4, 6, 7, 8, 9 in Definition 2.1.

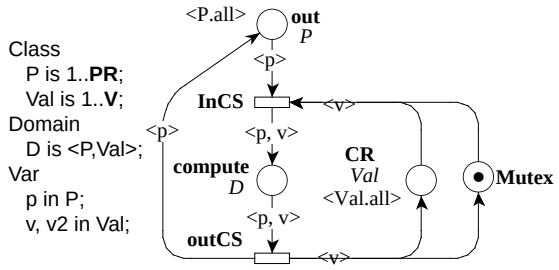


Figure 1: Example of Symmetric Net

The Symmetric net in Figure 1 represents a class of threads (identified by an identity in type *P*) accessing a critical resource *CR*. Threads can get a value within the type *Val* from **CR**. Constants **PR** and **V** are integer parameters for the system. The class of threads is represented by places **out** and **compute**. Place **compute** corresponds to some computation on the basis of the value provided by **CR**. At this stage, each thread holds a value that is replaced when the computation is finished. Place **Mutex** handles mutual exclusion between threads and contains token with no data (”black tokens” in the sense of the Petri Net standard [23]). Place **out** initially holds one token for each value in *P* (the marking is then denoted $< P.all >$) and place **CR** holds one value for each value in *Val*.

Verification of properties can be achieved either by a structural analysis, on the symbolic reachability graph (model checking), or on the unfolded associated Place/Transition (PT) net (model checking as well as structural properties).

2.3 Transformation, abstraction and refinement

Abstraction and refinement are part of the use of formal specifications. While abstraction is crucial to concentrate on essential aspects of the problem to be solved (or the system to be built), and to reason about them, more elaborate details need to be further introduced in the refinement steps. A similar evolution is taking place when a general pattern or template is established to describe the common structure of a family of problems, and when this template is instantiated to describe a single given problem.

Three kinds of refinement for coloured Petri nets are introduced in [28, 29], the type refinement, the node refinement and the subnet refinement. The idea for these refinements to be correct is that behaviours should be preserved, and to any behaviour of a refined net it should be possible to match a behaviour of the abstract net.

We have here another motivation that is raised by the use of tools to check the behaviour and properties of the model, and that may involve the discretization of some domains so as to reduce the number of possible values to consider in the state space. It thus involves a simplification of some domains that may be considered as an abstraction.

3 Methodology for Discretization

This section presents our methodology to model and analyse a complex system. We first give an overview of the approach and then detail its main steps and the involved techniques.

3.1 Overview of the Methodology

Figure 2 sketches our methodology. It takes as input a set of requirements structured following the FRAME method [20]. It is thus divided in two parts:

- the *specification* describes the system (we only consider in this work the behavioural aspects),
- the *required properties* establish a set of assertions to be verified by the system.

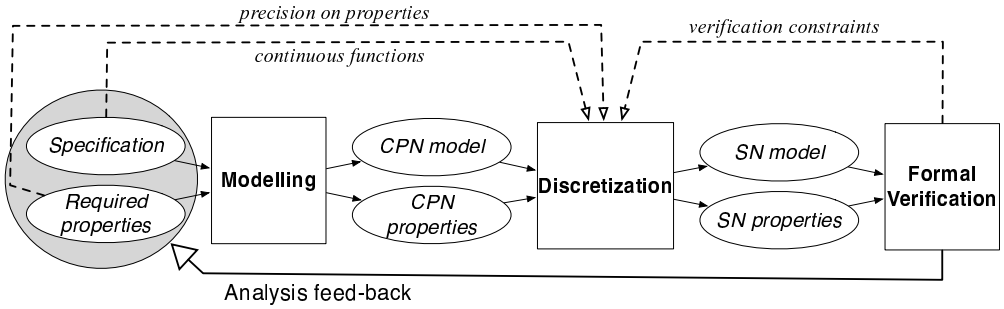


Figure 2: Overview of our methodology

Once the specification written using "classical" techniques, the system is modelled using high-level Petri Nets (CPN) that allow one to insert complex colour functions such as one involving real numbers. These functions come from the specifications of the system (in Intelligent Transport Systems, numerous behaviours are described by means of equations describing physical models). These functions are inserted in arc labels into the CPN-model produced by the **Modelling** step. Required properties are also set in terms of CPN. However, the CPN system cannot be analyzed in practice since the system is too complex (due to the data and functions involved). So, the **Discretization** step is dedicated to the generation of an associated system expressed using Symmetric Nets. Symmetric Nets are well suited to specify such systems that are intrinsically symmetric [3]. Operations such as structural analysis or model checking can be achieved for much larger systems. Formal analysis of the system is performed at the **Formal Verification** step.

The following sections present the three main steps of our methodology and especially focus on the **Discretization** step that is the most delicate one as well as the main contribution of this paper.

3.2 Modelling

There are heterogeneous elements to consider in Intelligent Transport Systems (ITS): computerized actors (such as cars or controllers in a motorway infrastructure) have to deal with physical variables such as braking distances, speed and weigth. In [3] we presented a methodology to model large and complex ITS starting from a specification mainly based on a subset of UML diagrams.

This methodology [3] is also based on the definition and use of an ITS template. To have a hierarchical and structured specification using a relevant subset of UML diagrams, we proposed an ITS template that allows variations of architectures and component variables. The architectures are defined, involving components and their interconnections through interfaces. This enables us to change and update components of the architecture and to generate the Petri Net model easily. This template was elaborated from the investigation of case studies of the SAFESPOT and TrafficView projects [4, 14].

The system high level architecture is specified using UML component diagrams. Interfaces between components are specified with class diagrams. This first step of the methodology is used to identify the different components of the system and their counterparts in Petri nets. It is also used to define how they should be assembled to compose a complete model. Then, the behaviour of each component can be specified either with UML activity diagrams, UML state machines or Petri nets. This methodology relies on the use of Petri scripts to assemble the complete model but also for modelling complex components.

This methodology is well suited to have a fast, efficient, modular and incremental approach in modelling large systems. But only a subpart of the "required properties" of the system could be checked. Especially, it

was not possible to verify properties related to quantitative variables as they are usually abstracted in the Petri nets.

The work presented in this paper aims at providing a more precise representation of the system in the Petri net models by representing those quantitative variables. To design the CPN model we used a template adapted to the case study presented in Section 4. The "interfaces" of the Petri net model, presented in Section 5, were already identified. The main task was to identify control and data flows that are involved in this subpart of the system, and that must be modeled to allow formal verification. Also, operations made on those flows were identified.

Then, the different selected variables of the system were represented using equivalent types in CPN. For example, continuous variables of the system were modelled with the real type of CPN formalism. The functions of the system that manipulate the continuous variables were represented using arc expressions.

3.3 Discretization

The discretization step takes CPN with their properties as inputs, and produces SN with their properties as outputs. To achieve this goal, a discretization of the real data and functions involved is performed. As a result, the types involved in the CPN are abstracted, and the real functions are represented by a place providing tuples of appropriate result values.

We propose different steps to manage the discretization of continuous functions in Symmetric Nets

- Continuous feature discretization.
- Error propagation computing
- Type transformation and modelling of complex functions in Symmetric Nets.

Continuous feature discretization Discretization is the process of transforming continuous models and equations into discrete counterparts. Depending on the domain to which this process is applied we use also the words "digitizing", "digitization", "sampling", "quantization" or "encoding". Techniques for discretization differ according to application domains and objectives.

Let us introduce the following definitions that are used in this paper to avoid ambiguity:

Definition 3.1. *A **region** is a n-dimentional polygon (i.e. a polytope) made by adjacent points of an n-dimentional discretized function.*

Definition 3.2. *A **mesh** is a set of regions used to represent a n-dimentional discretized function for modeling or analysis.*

There exist many discretization methods that can be classified between global or local, supervised or unsupervised, and static or dynamic methods [17].

- **Local methods** produce partitions that are applied to localized regions of the instance space. Those methods usually use decision trees to produce the partitions (i.e. the classification).
- **Global methods** (like binning) [17] produce a mesh over the entire n-dimentional continuous instance space, where each feature is partitioned into regions. The mesh contain $\prod_{i=1}^{n} k_i$ regions, where k_i is the number of partitions of the ith feature.

In our study we consider the **equal width interval binning method** as a first approach to discretize the continuous features. Equal width interval binning is a global unsupervised method that involves dividing the range of observed values for the variable into k equally sized intervals, where k is a parameter provided by the user. If a variable x is bounded by x_{min} and x_{max}, the interval width is:

$$\Delta = \frac{x_{max} - x_{min}}{k} \tag{3.1}$$

Error propagation computing To model a continuous function in Symmetric Nets it is necessary to convert it into an equivalent discrete function. This operation introduces inaccuracy (or error) which must be taken into account during the formal verification of the model. This inaccuracy can be taken into account in the Symmetric Net properties in order to keep them in accordance with the original system required properties. The other solution is to change the original required properties taking into account the introduced inaccuracy.

The issues are well expressed below [6]:

In science, the terms uncertainties or errors do not refer to mistakes or blunders. Rather, they refer to those uncertainties that are inherent in all measurements and can never be completely eliminated.(...) A large part of a scientist's effort is devoted to understanding these uncertainties (error analysis) so that appropriate conclusions can be drawn from variable observations. A common complaint of students is that the error analysis is more tedious than the calculation of the numbers they are trying to measure. This is generally true. However, measurements can be quite meaningless without knowledge of their associated errors.

There are different methods to compute the error propagation in a function [30, 6]. The most current one is to determine the separate contribution due to errors on input variables and to combine the individual contributions in quadrature.

$$\Delta_{f(x,y,..)} = \sqrt{\Delta_{fx}^2 + \Delta_{fy}^2 + ...} \tag{3.2}$$

Then, different methods to compute the contribution of input variables to the error in the function are possible, like the "derivative method" or the "computational method".

- The derivative method evaluates the contribution of a variable x to the error on a function f as the product of error on x (i.e. Δ_x) with the partial derivative of $f(x,y,..)$:

$$\Delta_{fx} = \frac{\partial f(x,y,..)}{\partial x}\Delta_x \tag{3.3}$$

- The computational method computes the variation directly by a finite difference:

$$\Delta_{fx} = \mid f(x+\Delta_x,y,..) - f(x,y,..) \mid \tag{3.4}$$

The use of individual contribution in a quadrature relies on the assumption that the variables are independent and that they have a Gaussian distribution for their mean values. This method is interesting as it gives a good evaluation of the error. But we do not have a probabilistic approach, and we do not have a Gaussian distribution of the "measured" values.

In this paper, we prefer to compute the maximum error bounds on f due to the errors on variables as it gives an exact evaluation of the error propagation. Let $f(x)$ be a continuous function, x be the continuous variable, and x_{disc} the discrete value of x. If we choose a discretization step of $2 * \Delta_x$ we can say that for each x_{disc} image of x by the discretization process, $x \in [x_{disc} - \Delta_x, x_{disc} + \Delta_x]$ (which is usually simplified by the expression $x = x_{disc} \pm \Delta_x$). We can compute the error $\Delta_{f(x)}$ introduced by the discretization:

$$f(x) = f(x_{disc}) \pm \Delta_{f(x)} \tag{3.5}$$

$$\Delta_{f(x)} = f(x \pm \Delta_x) - f(x) \tag{3.6}$$

We can also say that the error on $f(x)$ is inside the interval :

$$\Delta_{f(x)} \in [Min(f(x \pm \Delta_x) - f(x)), Max(f(x \pm \Delta_x) - f(x))] \tag{3.7}$$

This method can also be applied with functions of multiple variables. In this case, for a function f of n variables $f(x \pm \Delta_x, y \pm \Delta_y, ..)$ has 2^n solutions. The maximum error bounds on f are:

$$\Delta_f \in [Min(f(x \pm \Delta_x, y \pm \Delta_y, ..) - f(x,y,..)), Max(f(x \pm \Delta_x, y \pm \Delta_y, ..) - f(x,y,..))] \tag{3.8}$$

An example of this method applied to an emergency braking function is presented in Section 5.2.

Type transformation Once the best discretization actions are decided upon as regards our goals, the CPN specification may be transformed. The resulting net is a symmetric net.

Let us first note that some types do not need to be transformed because they are simple enough (e.g. enumerated types) and do not affect the state graph complexity.

When the types are more complex, two kinds of transformation are involved in this process, that concern the value set (also called carrier set), and the complex functions. The value set transformation results from the discretization of all infinite domains into an enumerated domain.

A node refinement is applied to transitions that involve a complex function on an output arc expression. As explained below and in Figure 3, there are two possibilities to handle this. In our method, such functions are represented by tuples of discrete values (values of the function arguments and of the result) that are stored in a **values** place. The **values** place is both input and output of the refined transition, thus for any input data provided by the original input arc(s), the **values** place yields the appropriate tuple with the function result.

Modelling of complex functions in Symmetric Nets To cope with the modelling of complex functions in Symmetric Nets (for example, the computation of braking distance according to the current speed of a vehicle), we must discretize and represent them either in a specific place or as a guard of a transition. When a place is used, it can be held in an SN-module ; it then represents the function and can be stored in a dedicated library.

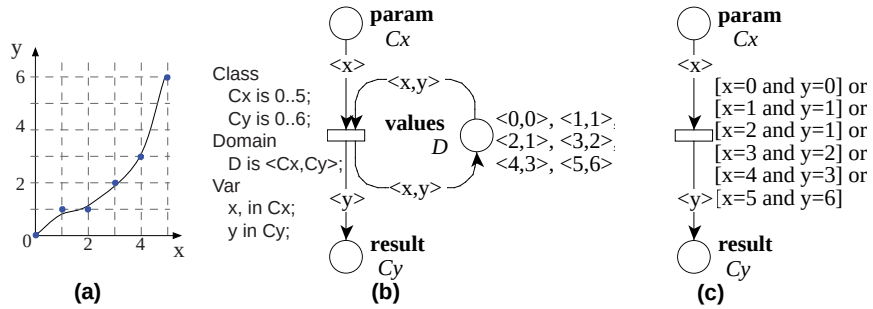


Figure 3: Example of complex function discretization by means of a place or a transition guard

Figure 3 represents an example of function discretization. The left side (a) of Figure 3 shows a function that is discretized, and the right side shows the corresponding Petri net models : in model (b), the function is discretized by means of a place, in model (c), it is discretized by mean of a transition guard. In both cases, correct associations between x and y are the only ones to be selected when the transition fires. Note that in model (b) **values** markings remain constants.

This technique can be generalized to any function $x = f(x_1, x_2, ..., x_n)$, regardless of its complexity. Non deterministic functions can also be specified in the same way (for example, to model potential errors in the system). Let us note that:

- the discretization of any function becomes a modelling hypothesis and must be validated separately (to evaluate the impact of imprecision due to discretization),
- given a function, it is easy to automatically generate the list of values to be stored in the initial marking of the place representing the function, or to be put in the guard of the corresponding transition.

The only drawback of this technique is a loss in precision compared to continuous systems that require appropriate hybrid techniques [10]. Thus, the choice of a discretization schema must be evaluated, for example to ensure that uncertainty remains in a safe range.

3.4 Verification

We use CPN-AMI [31] to perform verification. So far, our models can be analyzed using:

- *Structural techniques* (invariant computation, structural bounds, etc) on P/T nets. Since our nets are coloured, an unfolding tool able to cope with large systems [27] is used to derive the corresponding P/T net to compute structural properties.
- *Model checking*, we designed efficient model checking techniques that are dedicated to this kind of systems and make intensive use of symmetries as well as of decision diagrams. Such techniques revealed to be very efficient for this kind of systems by exploiting their regularity [22, 3].

However, due to the complexity of such systems, discretization is a very important point. If Symmetric net coloured classes are too large (i.e. the discretization interval is too small), we face a combinatorial explosion (for both model checking or structural analysis by unfolding). On the other hand, if the error introduced by the discretization is too high, the property loses its ”precision” and the verification of properties may lose its significance.

This is why in Figure 2, the discretization step needs *verification constraints* as inputs from the verification step. A compromise between combinatorial explosion and precision in the model must be found.

4 Modelling the Emergency Braking Problem

The case study presented in this paper is a subpart of an application from the “Intelligent Road Transport System” domain. It is inspired from the European project SAFESPOT [4]. This application is called “Hazard and Incident Warning” (H&IW), and its objective is to warn the driver when an obstacle is located on the road. Different levels of warning are considered, depending on the criticality of the situation. This section presents the “Emergency Braking module” of the application and how it can be specified using the CPN formalism.

4.1 Presentation of the Case Study

SAFESPOT is an Integrated Project funded by the European Commission, under the strategic objective “Safety Cooperative Systems for Road Transport”. The Goal of SAFESPOT is to understand how “intelligent” vehicles and “intelligent” roads can cooperate to produce a breakthrough in road safety. By combining data from vehicle-side and road-side sensors, the SAFESPOT project will allow to extend the time in which an accident is foreseen. The transmission of warnings and advices to approaching vehicles (by means of vehicle-to-vehicle and vehicle-to-infrastructure communications [34, 19, 24]), will extend in space and time the driver’s awareness of the surrounding environment.

The SAFESPOT applications [2] rely on a complex functional architecture. If the sensors and warning devices differ between SAFESPOT vehicles and SAFESPOT infrastructure, the functional architecture is designed to be almost the same for these two main entities of the system providing a peer-to-peer network architecture. It enables real-time exchange of vehicles’ status and of all detected events or environmental conditions from the road. This is necessary to take advantage of the cooperative approach and thus enable the design of effective safety applications.

As presented in Figure 4, information measured by sensors is provided to the “Data Processing / Fusion” module or transmitted through the network to the “Data Fusion Processing / Fusion” module of other entities. This module analyses and processes arriving data to put them on the “Local Dynamic Map” (LDM) of the system. The “Local Dynamic Map” enables the cooperative applications of the system to retrieve relevant variables and parameters depending on their purpose. The applications are then able to trigger relevant warnings to be transmitted to appropriate entities and displayed via an onboard Human Machine Interface (HMI) or road side Variable Message Signs (VMS). In SAFESPOT, five main infrastructure-based applications were defined: “Speed Alert”, “Hazard and Incident Warning”, “Road Departure Prevention”, “Co-operative Intersection Collision Prevention” and “Safety Margin for Assistance and Emergency Vehicles”. These applications are designed to provide the most efficient recommendations to the driver.

The aim of the “Hazard and Incident Warning” application is to warn the drivers in case of dangerous events on the road. Selected events are: accident, presence of unexpected obstacles on the road, traffic jam ahead, presence of pedestrians, presence of animals and presence of a vehicle driving in the wrong direction or dangerously overtaking. This application also analyses all environmental conditions that may influence the road friction or decrease the drivers’ visibility. Based on the cooperation of vehicles and road side sensors, the

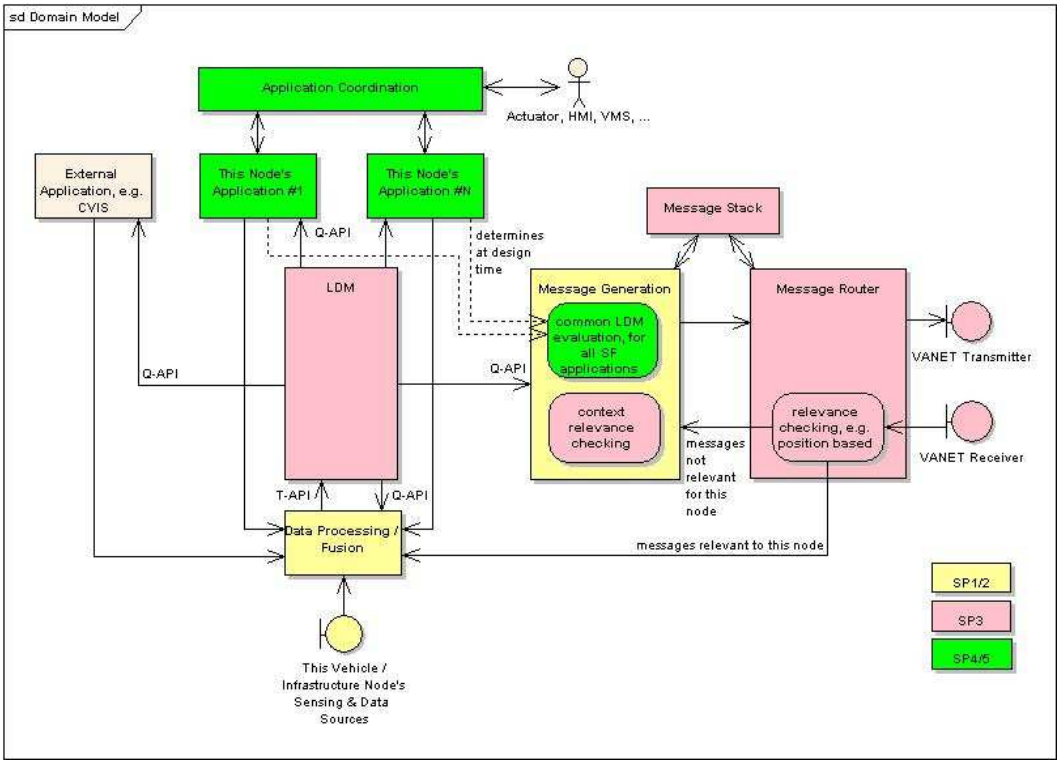


Figure 4: SAFESPOT High Level Architecture

"Hazard and Incident Warning" application provides warnings to the drivers and feeds the SAFESPOT road side systems and vehicles with information on new driving situations. This application is essential to provide other applications with the latest relevant road description.

The emergency braking module The emergency braking module is one subsystem in the "Hazard and Incident Warning" distributed application. It communicates with other subsystems. The behavior of this subsystem is significant in the SAFESPOT system and must be analyzed.

Petri nets are well suited to describe and analyse this type of application. However, a part of the "Hazard and Incident Warning" application algorithm is based on the analysis of continuous variables like vehicle speed or position of an obstacle. Those data are part of the data flow of the system ; they are also determinant for the control flow of the system. Many properties of the application can be verified with Petri nets by making an abstraction of the data flow where "continuous" variables are involved. This is where we face a huge combinatorial explosion and have to enhance the Petri net formalism and modelling methodology to enable the modelisation and verification of this kind of systems.

In the case of an obstacle on the road, the emergency braking module receives/retrieves the speed, deceleration capability and the relative distance to a static obstacle for the monitored vehicle. With these data, it will compute a safety command to be transmitted to the driver and to other applications of the system. Those commands represent the computed safety status of a vehicle. The three commands (or warnings) issued by this module are "Comfort" if no action is required from the driver, "Safety" if the driver is supposed to start decelerating, and "Emergency" if the driver must quickly start an emergency braking. This is illustrated in Figure 5. Note that if a driver in an "Emergency" status does not brake within one second, an automated braking should be triggered by the "Prevent" system (which is another European project).

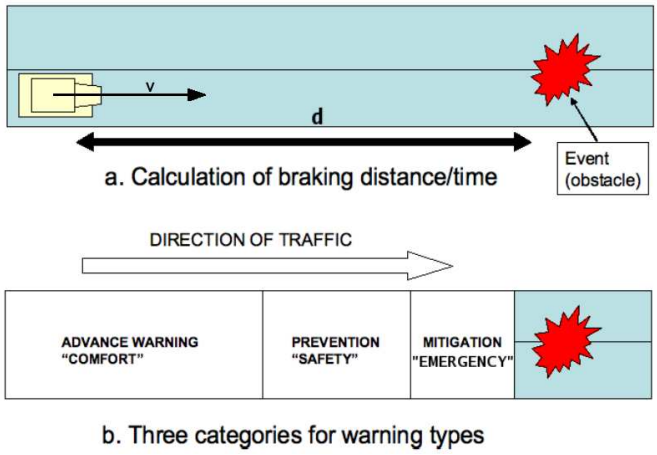


Figure 5: Emergency braking safety strategy

4.2 Mathematical model of the emergency braking module

The "emergency braking module" implements a strategy function to determine the safety status of a given vehicle. This function computes the "braking distance" of a vehicle from its speed and deceleration capabilities.

Let $v \in V$ be the velocity (speed) of a vehicle with $V \subset \mathbb{R}^{+*}$. Let also $b \in B$ be the braking capability of the vehicle with $B \subset \mathbb{R}^{+*}$. The braking distance function is then:

$$f(v,b) = \frac{v^2}{2b} \tag{4.1}$$

Let then $d \in D$ be the relative distance of the obstacle to the vehicle with $D \subset \mathbb{R}^{+}$. The main algorithm of the "Emergency braking module" defines two thresholds to determine when a vehicle goes from a "Comfort sate" to a "Safety state", and from a "Safety state" to an "Emergency state". Those thresholds are based on the time left to the driver to react. According to the application specification, if the driver has more than three seconds to react he is in a "Comfort state", then if he has less than three seconds but more than one second he is in the "Safety state", if he has less than one second to react, he is in the "Emergency State". The values of those thresholds are expressed as follow:

$$EB\_Safety = \frac{v^2}{2b} + v * 3 - d \tag{4.2}$$

$$EB\_Emergency = \frac{v^2}{2b} + v * 1 - d \tag{4.3}$$

The resulting algorithm of the strategy function can be represented with this pseudocode:

```
Eb_Strategy(d,v,b){
  Eb_Safety = (v^2)/(2b) + v * 3 - d;
  Eb_Emergency = (v^2)(2b) + v * 1 - d;
  if (Eb_Safety < 0) then
     Command = 'Comfort';
  else
     if (Eb_Emergency < 0) then
         Command = 'Safety';
     else
         Command = 'Emergency';
  endif
  return Command;
}
```

In SAFESPOT, v values are considered to be in $[0,46]m/s$, b in $[3,9]m/s^{-2}$ and d in $[0,500]m$. If variables are outside those sets, other applications are triggered (this becomes out of the scope of the emergency braking module). For exemple, speeds above $46m/s$ are managed by the "Speed Alert" application.

4.3 Required Properties

The SAFESPOT and H&IW application specifications are completed with required properties to be satisfied by the system. An analysis of the H&IW required properties shows that over the 47 main requirements, 18 involve continuous space and/or time constraints (i.e. 38%). The method presented in this paper focuses on those properties. Here are examples of this kind of properties for the emergency braking module:

- **Property 1**: When the braking distance of a vehicle is below its distance from a static obstacle plus one second of driver's reaction time, the H&IW application must trigger an "Emergency" warning.
- **Property 2**: When the braking distance of a vehicle is below its distance from a static obstacle plus three seconds of driver's reaction time, the H&IW application must trigger a "Safety" warning.

4.4 The coloured Petri net specification

Several modules in the H&IW application share the same architecture, namely for a given process, data is retrieved from the interface. Then, a command is computed, and sent to appropriate modules in the system. The coloured Petri net of Figure 6 exhibits this generic behaviour (i.e. the template mentioned in Section 3.2). Transition `Get_Data` has two input arcs from places `Interface_Call` and `Interface_Data`. Place `Interface_Call` is typed with `PROCESSID` which may be an integer subset (here the marking is a token with value 1). Once a process is called and data is retrieved, place `Step1` carries tokens that are couples (pid,data). Transition `Process_Strategy` provides a command resulting from computations from data.

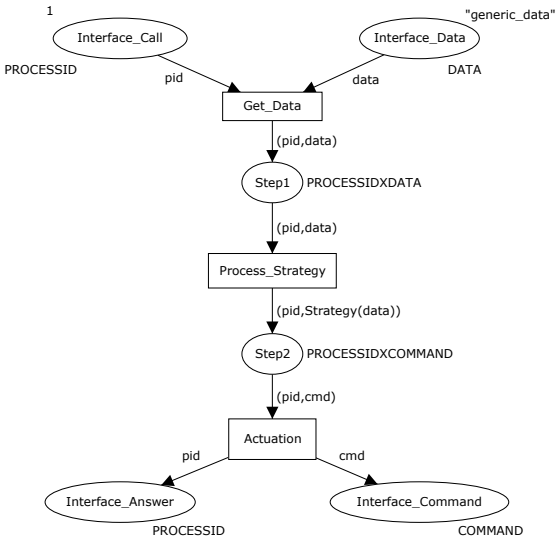


Figure 6: Template Coloured Petri net for the H&IW applications

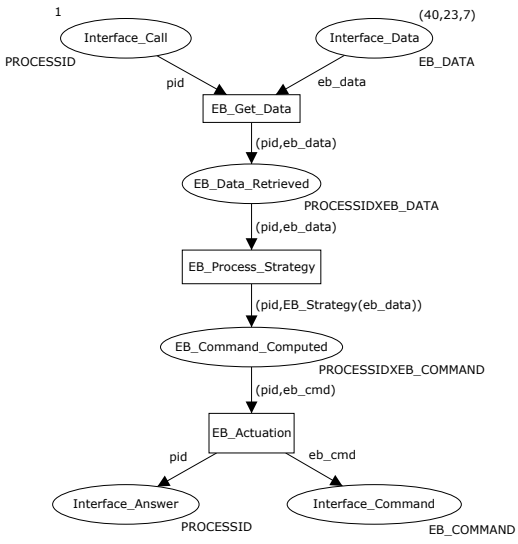


Figure 7: Coloured Petri net instantiated for the Emergency Braking application

In Figure 7 this generic schema is instanciated for the Emergency Braking Application (so, `generic_data` and `generic_command` become `EB_DATA` and `EB_COMMAND`). Data for this application are `Distance`, `Velocity`, and `Braking_Factor`, thus:

`EB_DATA = product Distance * Velocity * Braking_Factor`.

Data modelling physical entities are measured with a possible measurement error and are usually represented and computed in $\mathbb{R}^*$ in physics computations. For the CPN specification, we can keep this typing for

expressivity sake, while it is clear that it is not usable in practice (we would use integers for Petri nets tools and float in programming languages).

The `EB_COMMAND` type has three possible values related with the three levels of command or warning, therefore `EB_COMMAND = Comfort | Safety | Emergency`. The appropriate command results from the `EB_Strategy` function computation.

5 Discretization of the Problem

Discretization raises several issues. We propose a way to cope with these issues and apply our solutions to the emergency braking example.

5.1 Implementing complex functions in Symmetric Nets

Starting from the CPN model we use the methodology presented in Section 3.

First, CPN types must be transformed into discrete types. Using the equal width interval binning discretization method (presented in Section 3.3) with a number of k_v, k_b and k_d intervals for each variable we obtain a mesh of $k_v \times k_b \times k_d$ regions (as defined in definitions 3.2 and 3.1) in the resulting discretized function. The resulting sets for variables d, v and b are then composed of k ordered elements. For example, with $k = k_v = k_b = k_d = 10$ the resulting discretized type of v is $[0, 4.6, 9.2, ..., 46]$ and the discretized braking function contains 10^3 regions.

With $k = 100$, the domain of v is $[0, 0.46, 0.92, ..., 46]$ and the mesh is composed of 10^6 regions.

Section 3.3 presents two solutions to model complex functions in Symmetric Nets. We select solution b in Figure 3 because it is more efficiently represented in the Symbolic Reachability Graph. Therefore we add place "EB_Strategy_Table" in the Symmetric Petri net (Figure 8). Thus, cardinalities of the domains for places "Interface_Data" and "EB_Strategy_Table" (respectively named "EB_Data" and "EBStgyTable" in Figure 8) are computed using the formula: $Card(EBData) = Card(EBStgyTable) = Card(D \times V \times B)$. This means that these cardinalities are equal to the number of regions of the discretized function.

This method could provide very large markings (that is with a large number of tuples) in the resulting Symmetric Net. However, the use of a appropriate state space representation (by means of decision diagrams like in [12]) does not impact the size of the generated state space since the large marking is just represented once (the marking of places encoding complex functions is stable).

We chose a simple and generic discretization method that does not take into account the specificity of functions to be discretized. Other discretization methods like those using variable intervals can reduce the number of markings with the same level of accuracy in the resulting discretized function. Finally, depending on the kind of expected analysis, it is also possible to compute and use the equivalence classes. Those aspects are discussed later on this paper in sections 7.1 and 6.2.2.

5.2 Computation of the error propagation in Symmetric Nets

As presented in Section 3, we compute the precision error introduced by the discretization operation. The resulting error in the computation of the "Safety Threshold" is:

$$\Delta_{Eb\_Safety} = Eb\_Safety(v \pm \Delta_v, b \pm \Delta_b, d \pm \Delta_d) - Eb\_Safety(v, b, d) \tag{5.1}$$

$$\Delta_{Eb\_Safety} = (\frac{(v \pm \Delta_v)^2}{2(b \pm \Delta_b)} + 3(v \pm \Delta_v) - (d \pm \Delta_d)) - (\frac{v^2}{2b} + 3v - d) \tag{5.2}$$

$$\Delta_{Eb\_Safety} = \frac{(v \pm \Delta_v)^2}{2(b \pm \Delta_b)} - \frac{v^2}{2b} \pm 3\Delta_v \pm \Delta_d \tag{5.3}$$

For example, let us consider a classic private vehicle driving at $v = 14m/s$ (i.e. $50km/h$), on a dry road (i.e. $b = 8m/s^2$), at $d = 500m$ from an obstacle. If we consider $k = 100$ intervals and an error of respectively $\pm 0.45m/s$ for v, $\pm 0.06m/s^2$ for d and $\pm 5m$ for p. Then we obtain:

$$\Delta_{Eb\_Safety} \in [-7.25m, +7.29m]$$

$$\Delta_{Eb\_Emergency} \in [-6.33m, +6.37m]$$

For the same vehicle at 100 meters from the obstacle, driving at $v = 36m/s$ (i.e. $130km/h$), on a wet road (i.e. $b = 4m/s^2$), we obtain:

$$\Delta_{Eb\_Safety} \in [-12.85m, +13.10m]$$

$$\Delta_{Eb\_Emergency} \in [-11.93m, +12.19m]$$

Those results provide an information on the precision of the Symmetric Net properties. Table 1 gives some error bounds computed from four values for parameter k. As expected, precision of computed thresholds depends on k. However, precision also depends on the values of variables. For example, values of v and b are determinant on the computation of error bounds. Exploiting those precisions, to validate the Symmetric Net model and its properties, requires to consider carefully those values.

Discretization parameter	$v = 13m/s,\ b = 8m/s^{-2},$ $d = 500m$	$v = 36m/s,\ b = 4m/s^{-2},$ $d = 100m$
$k = 10$ $card(EBData) = 10^3$	$\Delta_{Eb\_Safety} \in [-70.83m, 74.84m]$ $\Delta_{Eb\_Emergency} \in [-61.64m, 65.64m]$	$\Delta_{Eb\_Safety} \in [-118.9m, 144.5m]$ $\Delta_{Eb\_Emergency} \in [-109.7m, 135.3m]$
$k = 20$ $card(EBData) = 8 * 10^3$	$\Delta_{Eb\_Safety} \in [-35.87m, 36.81m]$ $\Delta_{Eb\_Emergency} \in [-31.27m, 32.26m]$	$\Delta_{Eb\_Safety} \in [-61.97m, 68.28m]$ $\Delta_{Eb\_Emergency} \in [-57.37m, 63.68m]$
$k = 50$ $card(EBData) = 12.5 * 10^3$	$\Delta_{Eb\_Safety} \in [-14.45m, 14.61m]$ $\Delta_{Eb\_Emergency} \in [-12.62m, 12.77m]$	$\Delta_{Eb\_Safety} \in [-25.47m, 26.47m]$ $\Delta_{Eb\_Emergency} \in [-23.63m, 24.63m]$
$k = 100$ $card(EBData) = 10^6$	$\Delta_{Eb\_Safety} \in [-7.25m, 7.29m]$ $\Delta_{Eb\_Emergency} \in [-6.33m, 6.37m]$	$\Delta_{Eb\_Safety} \in [-12.85m, 13.10m]$ $\Delta_{Eb\_Emergency} \in [-11.93m, 12.19m]$

Table 1: Error bounds for different discretization parameters

5.3 Validating the discretization in Symmetric Nets

Discretization of variables and function in the Symmetric Net model in Figure 8 introduces imprecision. Depending on properties that need to be verified, this imprecision must be considered. For example, properties presented section 4.3 can be verified using CTL (Computation Tree Logic) [18] formulae. With a discretization factor of $k = 100$ values on input variables, property 1 can be verified with an accuracy smaller than $\pm 7,3m$ on a relative distance, for a velocity of $14m/s$ on a dry road.

If the introduced imprecision is acceptable with regards to the properties to be verified, then the system designer can state that the discretization is valid for those properties. Otherwise, a better accuracy may be required and a new discretization must be done.

It is also possible to integrate the imprecision in the CTL formulae. To do so, more constraining value of input variables must be chosen (i.e. an higher speed, a lower braking factor or a closer obstacle) in the CTL formula. In our case, the simplest way is to choose a lower value of obstacle position that cover the discretization error.

In some cases, it is possible to compute the discretization of input variables depending on the required precision on the function. This solution is discussed in section 7.2.

5.4 Transformation to obtain the Symmetric net

The SN in Figure 8 is derived from the CPN in Figure 7. Our purpose is to obtain a manageable state space for model checking, and, as presented in Section 3.3 and in Figure 3, this leads us to discretize some types and also to adopt some modelling for complex functions.

Thus, the different fields of `EB_DATA` in Figure 7 are now discretized. For example, type Distance is discretized into an enumeration: `_0`, `_50`, `_100`, etc.

`EB_DATA` is associated to *EBData* in the SN of Figure 8 that is a list of 3-uples (`Distance`, `Velocity`, `Braking_Factor`).

Now, as explained in Section 3.3 and shown in Figure 3 (b), the approach for modelling the function `EB_Strategy` is to add a place with a marking that is a conversion table for the discretized function. Thus, the `EB_Strategy` function in Figure 7 is associated in Figure 8 to a table *EBStgyTable* that represents the discretized function (`Emergency`, `Safety` or `Comfort`). This result is retrieved by means of place `EB_Strategy_Table` connected to transition `EB_Process_Strategy`.

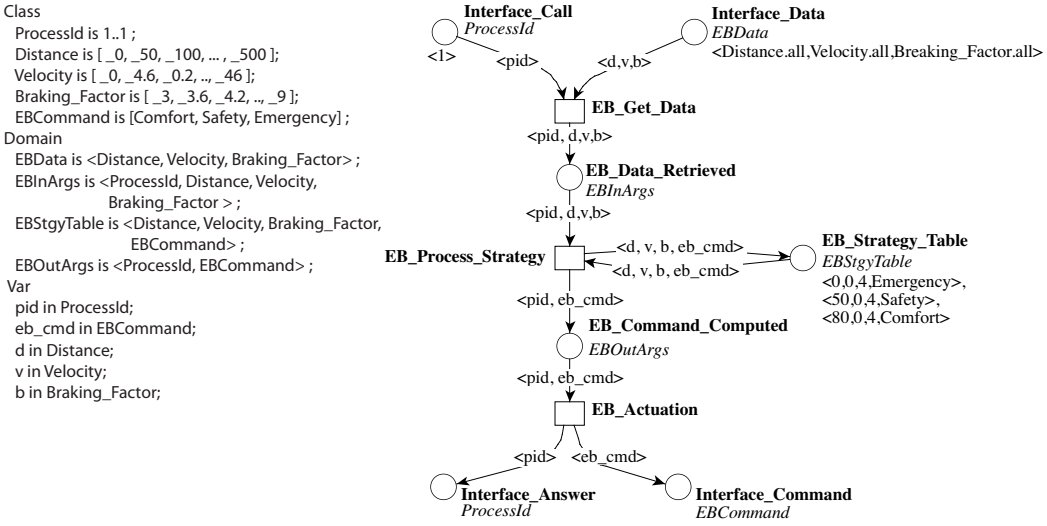


Figure 8: Symmetric Petri net for Emergency Braking module

Of course, the models presented in this paper are only sub-parts of a system. They can be independently verified but the purpose is to integrate them in a more complete representation of the system. This integration may introduce new discretization constraints and verification formulae must be rewritten. Those aspects are discussed in section 7 of this paper.

6 Net analysis

The use of a discretization method with symmetric nets generates complex models with large markings. It is important to know what are the consequences on the net analysis and model checking tools. In this section we present an overview of the analysis results obtained on the model.

Objectives The objectives of this analysis was first to analyse the properties of the net and sources of combinatorial explosion. Another interesting aspect of this analysis was to find the limitation of the tools used, which are not a priori suited to this type of net, and find some optimisation methods.

Experimental method As the complexity of the models presented in this paper is mainly dependent on the discretisation made on the three input variables, we focussed on the impact of this discretization. The symetric net model of Figure 8 was adapted to the experiment by connecting place "Interface_Answer" to place "Interface_Call" with two arcs and a transition with arcs expressions assigning variable $< pid >$ to the arcs. This allows the net to loop until the marking in place "Interface_Data" is empty. The marking of place "Interface_Data" was initialized with all values of domain "EBData" as presented Figure 8. Then scripts were used to initialize the class declaration and the marking of place "EB_Strategy_Table" depending on the chosen discretization level. Figure 9 gives an overview of the subset of properties that were tested.

Technical aspects The analysis of the Petri Net is a complex operation that requires different transformations of the model like unfolding or reduction. Different tools were used to make various analysis. First the CPN models were designed with CPN-Tools [13], then the symmetric models were designed using Coloane and

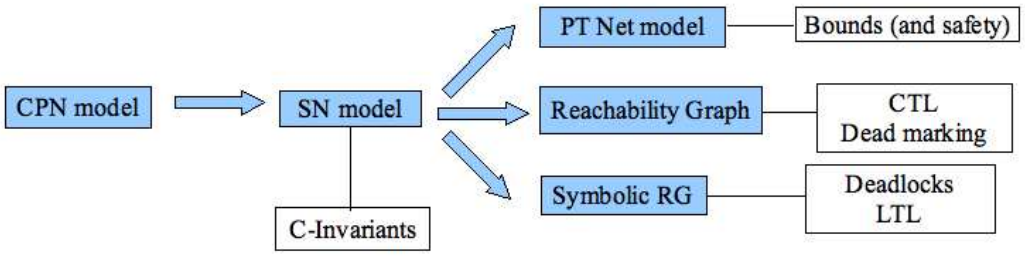


Figure 9: Overview of the analyses

Petriscript [31].To make the analysis of the model we chose the CPN-AMI [31] environment that provides a unified access to different tools like: a Petri net unfolding tool, PROD [32] or GreatSPN [8].

6.1 Structural analysis

The first analyses made on the net are structural analyses. They do not require the construction of the reachability graph and then, do not require to apply firing rules. Therefore they are less complex than behavioural analyses.

6.1.1 Symmetric net analysis

We made the computation of Coloured-Invariant on the Symmetric net with different discretisations. The only invariant detected is the marking of place "EB_Strategy_Table", as expected. The results show that the discretization does not have a significant impact on the memory used for the computation. But due to the size of the marking of place "EB_Strategy_Table", which is composed of all associations between variables and commands, the tool is not able to show the invariant for large markings even if it claims to have made the computation.

6.1.2 Unfolding the net

The computation of the unfolded net requires an increasing amount of memory depending on the discretization of input variables. It also gives an increasing unfolded net. In fact, the size of the domain "EBData" has a cubic growth. An analysis of the symmetric net shows that the size of the unfolded net in terms of places (np) follows the law: $np = 5 * (k)^3 + 8$,where k is the discretization level. The use of the CPN-AMI unfolder confirms that the unfolding of the symmetric net did follow this law. It also appears that the memory used to compute the unfolded net grows even more quickly than the size of the unfolded net. This explains why we faced a combinatorial explosion in the computation of the unfolded net which has bounded the coverage of our experiment.

6.1.3 Bound computation

We were able to test the bounds and safety of the net. The net is effectively bounded but the complexity of the computation, in terms of memory and time used, is the same as the one of the unfolding operation.

6.2 Behavioural analysis

The behavioural analysis is based on the use of Great-SPN and Prod to produce the reachability graphs.

6.2.1 Computation of the reachability graph

The generation of rechability graphs seems, according to the few tests that we made, to have about the same complexity in terms of memory and time as that of the unfolding of the net. The size seems also to follow

a cubic growth. Using the symbolic rechability graph of GreatSPN is a little bit more efficient. We did not make enough LTL and CTL queries to provide conclusions but the detection of deadlocks is not generating an additionnal combinatorial explosion.

6.2.2 Semantic equivalence Classes in the Model

If we consider the computation of *safety properties* (also called reachability properties) in the reachability graph, we can deduce that numerous states correspond to similar execution path in the original program or specification.

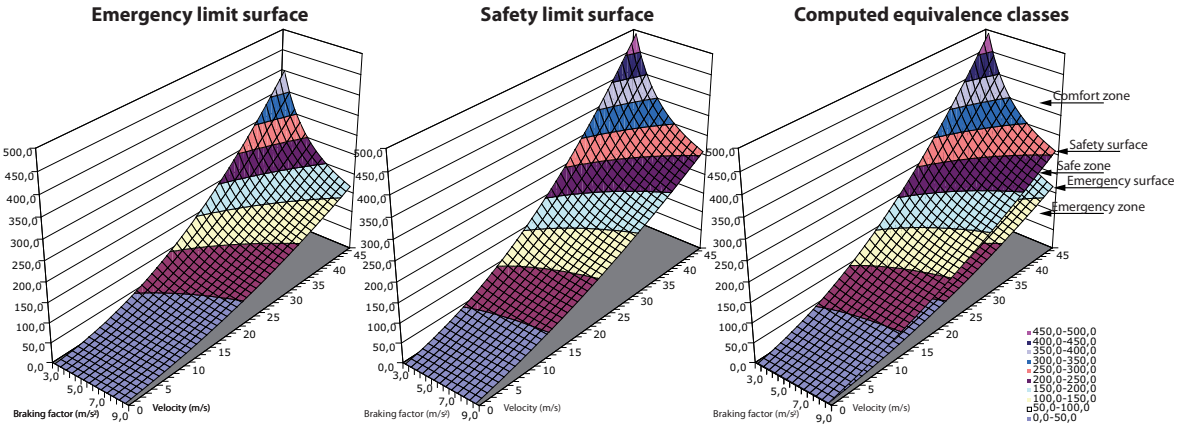


Figure 10: Building behavioural equivalence classes from the surfaces generated by the resolution of safety and emergency equations.

Thus, some accessibility properties could be preserved through equivalence classes. It is then of interest to exploit these, for example, when the computed subnet is integrated into a larger specification.

In our example, the equivalence classes are based on the net behaviour and must be computed from the physics equations that define the limits between the different situations of the system: “Comfort”, “Safety”, “Emergency”. To determine the surface that delimits equivalence classes in the state space, we compute solutions equations 4.2 and 4.3. We then get the two surfaces displayed on the left part of Figure 10.

From these surfaces, we can deduce five equivalence zones in the reachability graph as shown on the right part of Figure 10:

- the one above safety limit surface,
- the safety limit surface itself,
- the one between safety and emergency surfaces,
- the emergency surface itself,
- the one below the emergency surface.

For reachability properties, it is possible to provide a coherent discretization that reaches at least all these equivalence classes by randomly selecting any point in each zone and adding the corresponding value of b (from B axis), d (from D axis) and v (from V axis). Then, for each discretized colour B, D and V, we can take the coordinates of five points randomly chosen in these five zones. This approach is similar to the one proposed in [16] that was dealing with one colour domain only and used guards from the Petri nets to compute equivalence classes.

6.3 Conclusion on the analysis

The conclusions on the analysis of the nets are balanced. We were able to check properties but not for large models. The limitation comes from either the limitation of the tools, that are often not able to manage large markings, or from the complexity of the property to be analysed. We think anyway, that exploiting behavioural symmetries will solve some of those limitations.

7 Discussion and Open Issues

We have described and applied a discretization method to cope with hybrid systems and handle continuous variables in a safe and discrete manner. In this section, we open a discussion on several aspects.

7.1 Other discretization parameters

The methodology presented in this paper is based on the use of a discretization algorithm to discretize continuous variables. In section 5.2, we used "equal width interval binning" algorithm because it is simple to implement. This algorithm, like many others, relies on discretization parameters that can be optimized for a given set of continuous variables and functions.

However, in the emergency braking module example, we may study the partial derivates of the error on the two thresholds (Δ_{EB_Safety} and $\Delta_{EB\_Emergency}$). We then find that variables v and d are more influent than b. For example, the partial derivate of the error on the EB_Safety threshold (equation 5.3) with respect to the variable v is:

$$\frac{\partial \Delta_{Eb\_Safety}}{\partial v} = \frac{v \pm \Delta_v}{b \pm \Delta_b} - \frac{v}{b} \tag{7.1}$$

This allows to find optimized discretization parameters considering the respective influence of each involved variable. This is done by considering different parameters for each discretized variable depending on its influence on the error propagation.

Discretization parameters	$v = 14m/s, b = 8m/s^{-2}, d = 500m$	$v = 46m/s, b = 4m/s^{-2}, d = 100m$
$k_v = 114, k_b = 73, k_d = 120$	$\Delta_{Eb\_Safety} \in [-6.167m, 6.203m]$	$\Delta_{Eb\_Safety} \in [-12.09m, 12.40m]$
$card(EBData) < 10^6$	$\Delta_{Eb\_Emergency} \in [-5.367m, 5.403m]$	$\Delta_{Eb\_Emergency} \in [-11.29m, 11.60m]$

Table 2: Discretization with optimized criteria

Table 2 presents the resulting error when discretization parameters are optimized using partial derivates. It shows that we can reduce the resulting error of about 10% with discretization parameters based on partial derivates.

The study of the best discretization method and parameters for a given set of continuous variable and function is a complex problem which can give very interesting results. It is a promising field for future work on optimization of the methodology presented in this paper.

7.2 Tunning the discretisation

It is of interest to compute the discretization intervals of discretized types (here k_b, k_v and k_d) according to the maximum error tolerated on one type involved in a property where error must be bounded *a priori*.

Let us consider as an example the braking distance fonction (4.1) presented section 4.2. It is possible to compute the discretization intervals of variables v and b, based on the accuracy required for the function Δ_f. Let $\pm\Delta_f$ be the tolerated error on f, and $\pm\Delta_v$, $\pm\Delta_b$ be the resulting errors on v and b. Using the error bounds propagation as presented section 3.3 we get:

$$\pm\Delta_f = \frac{(v \pm \Delta_v)^2}{2 * (b \pm \Delta_b)} - \frac{v^2}{2 * b} \tag{7.2}$$

We then obtain[2]:

$$\Delta_b = -\frac{2*b^2*\Delta_f - 2*b*\Delta_v*v - b*\Delta_v^2}{2*b*\Delta_f + v^2} \tag{7.3}$$

and two solutions for Δ_v that are a little bit more complex.
Let v_{min},v_{max},b_{min} and b_{max} be the bounds of v and b. The cardinality of V and B sets are:

$$Card(V) = \frac{v_{max} - v_{min}}{2*\Delta_v} \tag{7.4}$$

$$Card(B) = \frac{b_{max} - b_{min}}{2*\Delta_b} \tag{7.5}$$

Now, consider that we want the same cardinalities for V and B colour sets ($k_b = k_v$). We obtain[3]:

$$\Delta_v = \frac{(v_{max} - v_{min})\Delta_b}{b_{max} - b_{min}} \tag{7.6}$$

Using the value of Δ_v of equation (7.6) in equation (7.3), it is now possible to compute Δ_b from the desired Δ_f.

For only two variables, this method is complex as it gives multiple solutions that need to be analyzed to choose the appropriate solutions. However, it provides a way to compute the discretization intervals of input variables depending on the desired output error.

8 Conclusion

In this paper, we proposed a way to integrate continuous aspects of complex specifications into a discretized Petri Net model. Our approach was studied in the context of Intelligent Transport Systems and, more precisely, management of emergency braking when an obstacle is identified on the road. An application to this case study is provided.

This discretization method relies on the use of equations modelling the problem. Such equations come from the physical models that interact with the system. We attach these equations to a CPN template and then proceed to its transformation in order to be able to have an analyzable model (i.e. that remains finite).

The equations modeling the problem are used to:

- Provide a discretized abstraction
- To evaluate the quality of this abstraction with regards to the proof of properties on the resulting model.

This is a key point in modeling and evaluating a system by means of formal specification. It is crucial for engineers to evaluate the quality of the proven properties and, if necessary when assumptions are done (here, they come from the discretization), to evaluate their impact on the system's properties. Typically, imprecision raised by discretization may have to be corrected by either applying a more precise discretization or adding constants in formulas expressing properties to be checked.

In our paper, discretization is applied on symmetric Nets deduced from CPN since our tools rely on symmetric nets. Of course, it is also valid on the CPN models.

In our methodology, different discretization algorithms can be applied. We used in this paper a simple algorithm as a first approach but other ones based on non-uniform discretization intervals are promising alternatives. This will introduce new constraints in formal verification and in error propagation computation but it is a interesting field for future works.

Also, managing more than one module is of interest. In the context of a SAFESPOT application, several modules run in parallels and may introduce more continuous types and variables. Future work will then have to evaluate how a larger number of variable (and constraints) could be managed. In particular, experimenting,

[2]We intentionally removed the $\pm$ operator to increase readability.
[3]Note that it is possible to choose another factor between $Card(V)$ and $Card(B)$ as explained section 7.1

propagation of discretization contraints between different modules need a particular attention.

Acknowledgements: We would like to thank the anonymous referees fro their careful reading and helpful comments.

References

[1] J-R. Abrial. *The B book - Assigning Programs to meanings*. Cambridge Univ. Press, 1996.

[2] F. Bonnefoi, F. Bellotti, T. Scendzielorz, and F. Visintainer. SAFESPOT Applications for Infrasructure-based Co-operative Road Safety . In *14th World Congress and Exhibition on Intelligent Transport Systems and Services*, Beijing, China, October 2007.

[3] F. Bonnefoi, L. Hillah, F. Kordon, and X. Renault. Design, modeling and analysis of ITS using UML and Petri Nets. In *10th International IEEE Conference on Intelligent Transportation Systems (ITSC'07)*, pages 314–319, Seattle, USA, September 2007. IEEE Press.

[4] R. Brignolo. Co-operative road safety - the SAFESPOT integrated project. In *APSN - APROSYS Conference*. Advanced Passive Safety Network, May 2006.

[5] Didier Buchs and Nicolas Guelfi. A formal specification framework for object-oriented distributed systems. *IEEE Trans. Software Eng.*, 26(7):635–652, 2000.

[6] R. Brown C. Covault and D. Driscoll. *Uncertainties and Error Propagation - Appendix V of Physics Lab Manual*. Case Western Reserve University, 2005.

[7] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. A symbolic reachability graph for coloured Petri nets. *Theoretical Computer Science*, 176(1–2):39–65, 1997.

[8] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets Performance Evaluation. *special issue on Performance Modeling Tools*, 24((1&2)):47–68, November 1995.

[9] Giovanni Chiola, Claude Dutheillet, Giuliana Franceschinis, and Serge Haddad. Stochastic well-formed colored nets and symmetric modeling applications. *IEEE Trans. Computers*, 42(11):1343–1360, 1993.

[10] P. Christofides and N. El-Farra. *Control Nonlinear And Hybrid Process Systems: Designs for Uncertainty, Constraints And Time-delays*. SPringer Verlag, 2005.

[11] Petri Nets Steering Committee. Petri nets tool database: quick and up-to-date overview of existing tools for petri nets `http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html`.

[12] J-M. Couvreur and Y. Thierry-Mieg. Hierarchical Decision Diagrams to Exploit Model Structure. *Formal Techniques for Networked and Distributed Systems - FORTE 2005*, pages 443–457, 2005.

[13] The CPN Tools Homepage, 2007. http://www.daimi.au.dk/CPNtools.

[14] S. Dashtinezhad, T. Nadeem, B. Dorohonceanu, C. Borcea, P. Kang, and L. Iftode. TrafficView: A Driver Assistant Device for Traffic Monitoring based on Car-to-Car Communication. In IEEE Computer Press, editor, *IEEE Semiannual Vehicular Technology Conference*, 2004.

[15] René David and Hassane Alla. On Hybrid Petri Nets. *Discrete Event Dynamic Systems: Theory and Applications*, 11(1-2):9–40, 2001.

[16] M. Doche, I. Vernier-Mounier, and F. Kordon. A modular approach to the specification and validation of an electrical flight control system. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 590–610. Springer-Verlag, 2001.

[17] James Dougherty, Ron Kohavi, and Mehran Sahami. Supervised and unsupervised discretization of continuous features. In *International Conference on Machine Learning*, pages 194–202, 1995.

[18] E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *J. Comput. Syst. Sci.*, 30(1):1–24, 1985.

[19] IEEE 802.11 Working Group for WLAN Standards. *IEEE 802.11 tm Wireless Local Arae Networks*. IEEE, 2008.

[20] Frame Forum. The FRAME forum home page, `http://www.frame-online.net`.

[21] L. Hillah, F. Kordon, L. Petrucci, and N. Trèves. PN standardisation : a survey. In *International Conference on Formal Methods for Networked and Distributed Systems (FORTE'06)*, pages 307–322, Paris, France, September 2006. IFIP.

[22] J. Hugues, Y. Thierry-Mieg, F. Kordon, L. Pautet, S. Baarir, and T. Vergnaud. On the Formal Verification of Middleware Behavioral Properties. In *9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04)*, pages 139–157. Elsevier, September 2004.

[23] ISO/IEC-JTC1/SC7/WG19. International Standard ISO/IEC 15909: Software and Systems Engineering - High-level Petri Nets, Part 1: Concepts, Definitions and Graphical Notation, December 2004.

[24] D.Luca J.Daniel. IEEE 802.11p: Towards an International Standard for Wireless Access in Vehicular Environments. In *Proceedings of Vehicular Technology Conference,VTC Spring, IEEE*, pages 2036–2040, May 2008.

[25] Kurt Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use, vol. 1, vol. 2 et vol. 3*. Springer-Verlag, London, UK, 1995.

[26] Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets, Modelling and Validation of Concurrent Systems*. Monograph to be published by Springer Verlag, 2008.

[27] F. Kordon, A. Linard, and E. Paviot-Adet. Optimized Colored Nets Unfolding. In *International Conference on Formal Methods for Networked and Distributed Systems (FORTE'06)*, volume 4229 of *LNCS*, pages 339–355, Paris, France, September 2006. Springer Verlag.

[28] Charles Lakos and Glenn Lewis. Incremental state space construction of coloured Petri nets. In *Proc. 22nd Int. Conf. Application and Theory of Petri Nets (ICATPN'01)*, volume 2075 of *Lecture Notes in Computer Science*, pages 263–282. Springer, 2001.

[29] Glenn Lewis. *Incremental specification and analysis in the context of coloured Petri nets*. PhD thesis, University of Hobart,Tasmania, 2002.

[30] Vern Lindberg. *Uncertainties and Error Propagation - Part I of a manual on Uncertainties, Graphing, and the Vernier Caliper*. Rochester Institute of Technology, 2000.

[31] LIP6/MoVe. The CPN-AMI home page, `http://www.lip6.fr/cpn-ami/`.

[32] K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. Prod reference manual. Technical report, Helsinki University of Technology, 1995.

[33] Jacques Vautherin. Parallel systems specifications with coloured Petri nets and algebraic specifications. In *Advances in Petri Nets 1987, covers the 7th European Workshop on Applications and Theory of Petri Nets, June 1986*, pages 293–308, London, UK, 1987. Springer-Verlag.

[34] ISO TC204 WG-16. *CALM architecture*. ISO, 2007.