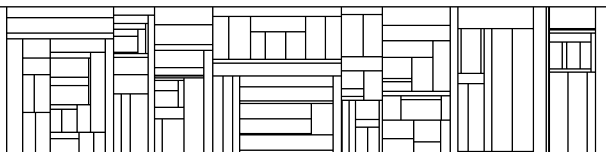
ISSN 0105-8517

Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools
Aarhus, Denmark, August 28-30, 2002

Kurt Jensen (Ed.)

DAIMI PB - 560
August 2002

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF AARHUS
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C, Denmark



Preface

This booklet contains the proceedings of the Fourth Workshop on Practical Use of Coloured Petri Nets and the CPN Tools, August 28-30, 2002. The workshop is organised by the CPN group at Department of Computer Science, University of Aarhus, Denmark. The papers are also available in electronic form via the web pages: http://www.daimi.au.dk/CPnets/workshop02/

Coloured Petri Nets and the CPN tools are now used by 1000 organisations in 60 countries all over the world (including 200 commercial enterprises). 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:

Jonathan Billington	Australia	(j.billington@unisa.edu.au)
Søren Christensen	Denmark	(schristensen@daimi.au.dk)
Jorge de Figueiredo	Brazil	(abrantes@dsc.ufpb.br)
Nisse Husberg	Finland	(Nisse.Husberg@hut.fi)
Kurt Jensen	Denmark (chair)	(kjensen@daimi.au.dk)
Daniel Moldt	Germany	(moldt@informatik.uni-hamburg.de)
Laure Petrucci	France	(petrucci@lsv.ens-cachan.fr)
Dan Simpson	UK	(Dan.Simpson@brighton.ac.uk)
Edwin Stear	USA	(estear@aol.com)
Robert Valette	France	(robert@laas.fr)
Rüdiger Valk	Germany	(valk@informatik.uni-hamburg.de)
Klaus Voss	Germany	(klaus.voss@gmd.de)
Lee Wagenhals	USA	(lwagenha@gmu.edu)
Jianli Xu	Finland	(jianli.xu@research.nokia.com)
Wlodek Zuberek	Canada	(wlodek@cs.mun.ca)

The programme committee has accepted 8 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 three CPN Workshops can be found via the web pages: http://www.daimi.au.dk/CPnets/. After an additional round of reviewing and revision, some of the papers have also been published as a special section in the International Journal on Software Tools for Technology Transfer (STTT). For more information see: http://sttt.cs.uni-dortmund.de/

Kurt Jensen

Table of Contents

Invited Talk:
W.M.P. van der Aalst and A.H.M. ter Hofstede
Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages 1

Mathew Elliot, Jonathan Billington and Lars Michael Kristensen
Using Design/CPN to Design Extensions to Design/CPN 21

Bo Lindstrøm and Lisa Wells
Annotating Coloured Petri Nets 39

Invited Talk:
Lin Zhang
Model-based Operational Planning Using Coloured Petri Nets 59

Jens Bæk Jørgensen
Coloured Petri Nets in UML-Based Software Development - Designing Middleware for Pervasive Healthcare 61

Wlodek Zuberek
Performance Study of Distributed Generation of State Spaces using Colored Petri Nets 81

Invited Talk:
Kim Guldstrand Larsen
Verification of Timed and Hybrid Systems 99

Guy Edward Gallasch, Lars Michael Kristensen and Thomas Mailund
Sweep-Line State Space Exploration for Coloured Petri Nets 101

Louise Lorentsen
Coloured Petri Nets and State Space Generation with the Symmetry Method 121

Thomas Jacob, Olaf Kummer, Daniel Moldt, and Ulrich Ultes-Nitsche
Implementation of Workflow Systems using Reference Nets - Security and Operability Aspects 139

Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages

W.M.P. van der Aalst[1,2] and A.H.M. ter Hofstede[2]

[1] Department of Technology Management, Eindhoven University of Technology
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands.
w.m.p.v.d.aalst@tm.tue.nl

[2] Queensland University of Technology, School of Information Systems
P.O. Box 2434, Brisbane Qld 4001, Australia.
a.terhofstede@qut.edu.au

Abstract. Contemporary workflow management systems are driven by explicit process models, i.e., a completely specified workflow design is required in order to enact a given workflow process. Creating a workflow design is a complicated time-consuming process which is often hampered by the limitations of the workflow language being used. To identify the differences between the various languages, we have collected a fairly complete set of workflow patterns. Based on these patterns we have evaluated 15 workflow products and detected considerable differences in expressive power. Languages based on Petri nets perform better when it comes to state-based workflow patterns. However, some patterns (e.g. involving multiple instances, complex synchronizations or non-local withdrawals) are not easy to map onto (high-level) Petri nets. These patterns pose interesting modeling problems and are used for developing the Petri-net-based language YAWL (Yet Another Workflow Language).

1 Introduction

Workflow technology continues to be subjected to on-going development in its traditional application areas of business process modeling and business process coordination, and now in emergent areas of component frameworks and inter-workflow, business-to-business interaction. Addressing this broad and rather ambitious reach, a large number of workflow products, mainly workflow management systems (WFMS), are commercially available, which see a large variety of languages and concepts based on different paradigms (see e.g. [1, 4, 12, 19, 23, 28, 31, 30, 40, 47]).

As current provisions are compared and as newer concepts and languages are embarked upon, it is striking how little, other than standards glossaries, is available for central reference. One of the reasons attributed to the lack of consensus of what constitutes a workflow specification is the variety of ways in which business processes are otherwise described. The absence of a universal organizational "theory", and standard business process modeling concepts, it is contended, explains and ultimately justifies the major differences in workflow languages - fostering up a "horses for courses" diversity in workflow languages. What is more, the comparison of different workflow products winds up being more of a dissemination of products and less of a critique of

workflow language capabilities - “bigger picture” differences of workflow specifications are highlighted, as are technology, typically platform dependent, issues.

Workflow specifications can be understood, in a broad sense, from a number of different perspectives (see [4, 23]). The *control-flow* perspective (or process) perspective describes activities and their execution ordering through different constructors, which permit flow of execution control, e.g., sequence, choice, parallelism and join synchronization. Activities in elementary form are atomic units of work, and in compound form modularize an execution order of a set of activities. The *data perspective* layers business and processing data on the control perspective. Business documents and other objects which flow between activities, and local variables of the workflow, qualify in effect pre- and post-conditions of activity execution. The *resource perspective* provides an organizational structure anchor to the workflow in the form of human and device roles responsible for executing activities. The *operational* perspective describes the elementary actions executed by activities, where the actions map into underlying applications. Typically, (references to) business and workflow data are passed into and out of applications through activity-to-application interfaces, allowing manipulation of the data within applications.

Clearly, the control flow perspective provides an essential insight into a workflow specification’s effectiveness. The data flow perspective rests on it, while the organizational and operational perspectives are ancillary. If workflow specifications are to be extended to meet newer processing requirements, control flow constructors require a fundamental insight and analysis. Currently, most workflow languages support the basic constructs of sequence, iteration, splits (AND and XOR) and joins (AND and XOR) - see [4, 30]. However, the interpretation of even these basic constructs is not uniform and it is often unclear how more complex requirements could be supported. Indeed, vendors are afforded the opportunity to recommend implementation level “hacks” such as database triggers and application event handling. The result is that neither the current capabilities of workflow languages nor insight into more complex requirements of business processes is advanced.

We indicate requirements for workflow languages through workflow *patterns* [5–8, 48]. As described in [36], a pattern “is the abstraction from a concrete form which keeps recurring in specific nonarbitrary contexts”. Gamma et al. [17] first catalogued systematically some 23 design patterns which describe the smallest recurring interactions in object-oriented systems. The design patterns, as such, provided independence from the implementation technology and at the same time independence from the essential requirements of the domain that they were attempting to address (see also e.g. [15]).

We have collected a set of about 30 workflow patterns and have used 20 of these patterns to compare the functionality of 15 workflow management systems (COSA, Visual Workflow, Forté Conductor, Lotus Domino Workflow, Meteor, Mobile, MQSeries/Workflow, Staffware, Verve Workflow, I-Flow, InConcert, Changengine, SAP R/3 Workflow, Eastman, and FLOWer). The result of this evaluation reveals that (1) the expressive power of contemporary systems leaves much to be desired and (2) the systems support different patterns. Note that we do not use the term “expressiveness” in the traditional or formal sense. If one abstracts from capacity constraints, any workflow language is Turing complete. Therefore, it makes to sense to compare these languages using for-

mal notions of expressiveness. Instead we use a more intuitive notion of expressiveness which takes the modeling effort into account. This more intuitive notion is often referred to as suitability. See [27] for a discussion on the distinction between formal expressiveness and suitability.

The observation that the expressive power of the available workflow management systems leaves much to be desired, triggered the question: *How about high-level Petri nets (i.e., Petri nets extended with color, time, and hierarchy) as a workflow language?*

Petri nets have been around since the sixties [35] and have been extended with color [24, 25] and time [32, 33] to improve expressiveness. High-level Petri nets tools such as Design/CPN (University of Aarhus, http://www.daimi.au.dk/designCPN/) and ExSpect (EUT/D&T Bakkenist, http://www.exspect.com/) incorporate these extensions and support the modeling and analysis of complex systems. There are at least three good reasons for using Petri nets as a workflow language [1]:

1. Formal semantics despite the graphical nature.
2. State-based instead of (just) event-based.
3. Abundance of analysis techniques.

Unfortunately, a straightforward application of high-level Petri nets does not yield the desired result. There seem to be three problems relevant for modeling workflow processes:

1. In a high-level Petri net it is possible to use colored tokens. Although it is possible to use this to identify multiple instances of a subprocess, there is no specific support for *patterns involving multiple instances* and the burden of keeping track, splitting, and joining is carried by the designer.
2. Sometimes two flows need to be joined while it is not clear whether synchronization is needed, i.e., if both flows are active an AND-join is needed otherwise an XOR-join. Such *advanced synchronization patterns* are difficult to model in terms of a high-level Petri net because the local transition rule is either an AND-join or an XOR-join.
3. The firing of a transition is always local only based on the tokens in the input places and only affecting the output places. However, some events in the workflow may have an effect which is not local, e.g., because of an error tokens need to be removed from various places without knowing where the tokens reside. Everyone who has modeled such a *cancellation pattern* (e.g., a global timeout mechanism) in terms of Petri nets knows that it is cumbersome to model a so-called “vacuum cleaner” removing tokens from selected parts of the net.

In this paper, we discuss the problems when supporting the workflow patterns with high-level Petri nets. We also briefly introduce a workflow language under development: *YAWL (Yet Another Workflow Language)*. YAWL is based on Petri nets but extended with additional features to facilitate the modeling of complex workflows.

2 Workflow patterns

Since 1999 we have been working on collecting a comprehensive set of workflow patterns [5–8]. The results have been made available through the “Workflow patterns

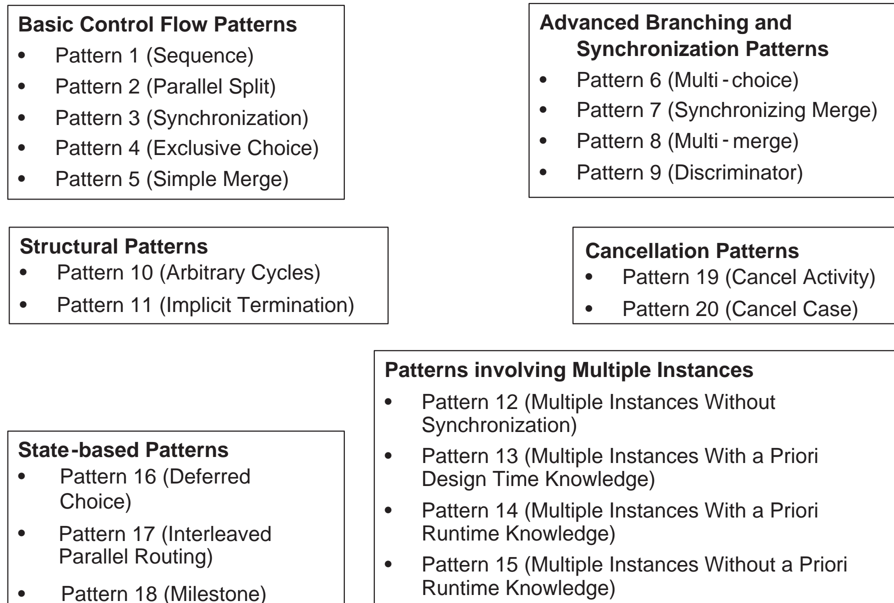


Fig. 1. Overview of the 20 most relevant patterns.

WWW site" [48]. The patterns range from very simple patterns such as sequential routing (Pattern 1) to complex patterns involving complex synchronizations such as the discriminator pattern (Pattern 9). In this paper, we restrict ourselves to the 20 most relevant patterns. These patterns can be classified into six categories:

1. *Basic control flow patterns.* These are the basic constructs present in most workflow languages to model sequential, parallel and conditional routing.
2. *Advanced branching and synchronization patterns.* These patterns transcend the basic patterns to allow for more advanced types of splitting and joining behavior. An example is the Synchronizing merge (Pattern 7) which behaves like an AND-join or XOR-join depending on the context.
3. *Structural patterns.* In programming languages a block structure which clearly identifies entry and exit points is quite natural. In graphical languages allowing for parallelism such a requirement is often considered to be too restrictive. Therefore, we have identified patterns that allow for a less rigid structure.
4. *Patterns involving multiple instances.* Within the context of a single case (i.e., workflow instance) sometimes parts of the process need to be instantiated multiple times, e.g., within the context of an insurance claim, multiple witness statements need to be processed.
5. *State-based patterns.* Typical workflow systems focus only on activities and events and not on states. This limits the expressiveness of the workflow language because it is not possible to have state dependent patterns such as the Milestone pattern (Pattern 18).

6. *Cancellation patterns.* The occurrence of an event (e.g., a customer canceling an order) may lead to the cancellation of activities. In some scenarios such events can even cause the withdrawal of the whole case.

Figure 1 shows an overview of the 20 patterns grouped into the six categories. A detailed discussion of these patterns is outside the scope of this paper. The interested reader is referred to [5–8, 48].

We have used these patterns to evaluate 15 workflow systems: COSA (Ley GmbH, [43]) , Visual Workflow (Filenet, [13]), Forté Conductor (SUN, [14]), Lotus Domino Workflow (IBM/Lotus, [34]), Meteor (UGA/LSDIS, [41]), Mobile (UEN, [23]), MQ-Series/Workflow (IBM, [22]), Staffware (Staffware PLC, [44]), Verve Workflow (Versata, [46]), I-Flow (Fujitsu, [16]), InConcert (TIBCO, [45]), Changengine (HP, [21]), SAP R/3 Workflow (SAP, [39]), Eastman (Eastman, [42]), and FLOWer (Pallas Athena, [9]). Tables 1 and 2 summarize the results of the comparison of the workflow management systems in terms of the selected patterns. For each product-pattern combination, we checked whether it is possible to realize the workflow pattern with the tool. If a product directly supports the pattern through one of its constructs, it is rated +. If the pattern is not *directly* supported, it is rated +/-. Any solution which results in spaghetti diagrams or coding, is considered as giving no direct support and is rated -.

pattern	product							
	Staffware	COSA	InConcert	Eastman	FLOWer	Domino	Meteor	Mobile
1 (seq)	+	+	+	+	+	+	+	+
2 (par-spl)	+	+	+	+	+	+	+	+
3 (synch)	+	+	+	+	+	+	+	+
4 (ex-ch)	+	+	+/-	+	+	+	+	+
5 (simple-m)	+	+	+/-	+	+	+	+	+
6 (m-choice)	-	+	+/-	+/-	-	+	+	+
7 (sync-m)	-	+/-	+	+	-	+	-	-
8 (multi-m)	-	-	-	+	+/-	+/-	+	-
9 (disc)	-	-	-	+	+/-	-	+/-	+
10 (arb-c)	+	+	-	+	-	+	+	-
11 (impl-t)	+	-	+	+	-	+	-	-
12 (mi-no-s)	-	+/-	-	+	+	+/-	+	-
13 (mi-dt)	+	+	+	+	+	+	+	+
14 (mi-rt)	-	-	-	-	+	-	-	-
15 (mi-no)	-	-	-	-	+	-	-	-
16 (def-c)	-	+	-	-	+/-	-	-	-
17 (int-par)	-	+	-	-	+/-	-	-	+
18 (milest)	-	+	-	-	+/-	-	-	-
19 (can-a)	+	+	-	-	+/-	-	-	-
20 (can-c)	-	-	-	-	+/-	+	-	-

Table 1. The main results for Staffware, COSA, InConcert, Eastman, FLOWer, Lotus Domino Workflow, Meteor, and Mobile.

pattern	product						
	MQSeries	Forté	Verve	Vis. WF	Changeng.	I-Flow	SAP/R3
1 (seq)	+	+	+	+	+	+	+
2 (par-spl)	+	+	+	+	+	+	+
3 (synch)	+	+	+	+	+	+	+
4 (ex-ch)	+	+	+	+	+	+	+
5 (simple-m)	+	+	+	+	+	+	+
6 (m-choice)	+	+	+	+	+	+	+
7 (sync-m)	+	-	-	-	-	-	-
8 (multi-m)	-	+	+	-	-	-	-
9 (disc)	-	+	+	-	+	-	+
10 (arb-c)	-	+	+	+/-	+	+	-
11 (impl-t)	+	-	-	-	-	-	-
12 (mi-no-s)	-	+	+	+	-	+	-
13 (mi-dt)	+	+	+	+	+	+	+
14 (mi-rt)	-	-	-	-	-	-	+/-
15 (mi-no)	-	-	-	-	-	-	-
16 (def-c)	-	-	-	-	-	-	-
17 (int-par)	-	-	-	-	-	-	-
18 (milest)	-	-	-	-	-	-	-
19 (can-a)	-	-	-	-	-	-	+
20 (can-c)	-	+	+	-	+	-	+

Table 2. The main results for MQSeries, Forté Conductor, Verve, Visual WorkFlo, Changengine, I-Flow, and SAP/R3 Workflow.

Please apply the results summarized in tables 1 and 2 with care. First of all, the organization selecting a workflow management system should focus on the patterns most relevant for the workflow processes at hand. Since support for the more advanced patterns is limited, one should focus on the patterns most needed. Second, the fact that a pattern is not directly supported by a product does not imply that it is not possible to support the construct at all.

From the comparison it is clear that no tool supports all the of the 20 selected patterns. In fact, many of the tools only support a relatively small subset of the more advanced patterns (i.e., patterns 6 to 20). Specifically the limited support for the discriminator, and its generalization, the N-out-of-M-join, the state-based patterns (only COSA), the synchronization of multiple instances (only FLOWer) and cancellation activities/cases, is worth noting.

The goal of providing the two tables is not to advocate the use of specific tools. However, they illustrate that existing tools and languages are truly different and that most languages provide only partial support for the patterns appearing in real life workflow processes. These observations have been our main motivation to look into the expressiveness of high-level Petri nets (Section 3) and come up with a new language (Section 4).

3 Limitations of Petri nets

Given the fact that workflow management systems have problems dealing with workflow patterns it is interesting to see whether established process modeling techniques such as Petri nets can cope with these patterns. The table listed in the appendix shows an evaluation of high-level Petri nets with respect to the patterns. (Ignore the column under YAWL for the time being.) We use the term high-level Petri nets to refer to Petri nets extended with color (i.e., data), time, and hierarchy [4]. Examples of such languages are the colored Petri nets as described in [25], the combination of Petri nets and Z specification described in [20], and many more. These languages are used by tools such as Design/CPN (University of Aarhus, http://www.daimi.au.dk/designCPN/) and ExSpect (EUT/D&T Bakkenist, http://www.exspect.com/). Although these languages and tools have differences when it comes to for example the language for data transformations (e.g., arc inscriptions) there is a clear common denominator. When we refer to high-level Petri nets we refer to this common denominator. To avoid confusion we use the terminology as defined in [25] as much as possible. It is important to note that for the table shown in the appendix we have used the same criteria as used in tables 1 and 2 for the 15 workflow systems (i.e., a "+" is only given if there is direct support).

Compared to existing languages high-level Petri nets are quite expressive. Recall that we use the term "expressiveness" not in the formal sense. High-level Petri nets are Turing complete, and therefore, can do anything we can define in terms of an algorithm. However, this does not imply that the modeling effort is acceptable. By comparing the table in the appendix with tables 1 and 2, we can see that high-level nets, in contrast to many workflow languages, have no problems dealing with state-based patterns. This is a direct consequence of the fact that Petri nets use places to represent states explicitly. Although high-level Petri nets outperform most of the existing languages, the result is not completely satisfactory. As indicated in the introduction we see serious limitations when it comes to (1) patterns involving multiple instances, (2) advanced synchronization patterns, and (3) cancellation patterns. In the remainder of this section we discuss these limitations in more detail.

3.1 Patterns involving multiple instances

Suppose that in the context of a workflow for processing insurance claims there is a subprocess for processing witness statements. Each insurance claim may involve zero or more witness statements. Clearly the number of witness statements is not known at design time. In fact, while a witness statement is being processed other witnesses may pop up. This means that within one case a part of the process needs to be instantiated a variable number of times and the number of instances required is only known at run time. The required pattern to model this situation is Pattern 15 (Multiple instances without a priori runtime knowledge). Another example of this pattern is the process of handling journal submissions. For processing journal submissions multiple reviews are needed. The editor of the journal may decide to ask a variable number of reviewers depending on the nature of the paper, e.g., if it is controversial, more reviewers are selected. While the reviewing takes place, the editor may decide to involve more reviewers. For example, if reviewers are not responsive, have brief or conflicting reviews,

then the editor may add an additional reviewer. Other examples of multiple instances include orders involving multiple items (e.g., a customer orders three books from an electronic bookstore), a subcontracting process with multiple quotations, etc.

It is possible to model a variable number of instances executed in parallel using a high-level Petri net. However, the designer of such a model has to keep track of two things: (1) case identities and (2) the number of instances still running.

At the same time multiple cases are being processed. Suppose x and y are two active cases. Whenever, there is an AND-join only tokens referring to the same case can be synchronized. If inside x part of the process is instantiated n times, then there are n "child cases" $x.1 \ldots x.n$. If for y the same part is also instantiated multiple times, say m, then there are m "child cases" $y.1 \ldots y.m$. Inside the part which is instantiated multiple times there may again be parallelism and there may be multiple tokens referring to one child case. For a normal AND-join only tokens referring to the same child case can be synchronized. However, at the end of the part which is instantiated multiple times all child cases having the same parent should be synchronized, i.e., case x can only continue if for each child case $x.1 \ldots x.n$ the part has been processed. In this synchronization child cases $x.1 \ldots x.n$ and child cases $y.1 \ldots y.m$ should be clearly separated. To complicate matters the construct of multiple instances may be nested resulting in child-child cases such as x.5.3 which should be synchronized in the right way. Clearly, a good workflow language does not put the burden of keeping track of these instances and synchronizing them at the right level on the workflow designer.

Besides keeping track of identities and synchronizing them at the right level, it is important to know how many child cases need to be synchronized. This is of particular relevance if the number of instances can change while the instances are being processed (e.g., a witness which points out another witness causing an additional witness statement). In a high-level Petri net this can be handled by introducing a counter keeping track of the number of active instances. If there are no active instances left, the child cases can be synchronized. Clearly, it is also not acceptable to put the burden of modeling such a counter on the workflow designer.

3.2 Advanced synchronization patterns

Consider the workflow process of booking a business trip. A business trip may involve the booking of flights, the booking of hotels, the booking of a rental car, etc. Suppose that the booking of flights, hotels, and cars can occur in parallel and that each of these elements is optional. This means that one trip may involve only a flight, another trip may involve a flight and a rental car, and it is even possible to have a hotel and a rental car (i.e., no flight). The process of booking each of these elements has a separate description which may be rather complex. Somewhere in the process these optional flows need to be synchronized, e.g., activities related to payment are only executed after all booking elements (i.e., flight, hotel, and car) have been processed. The problem is that it is not clear which subflows need to be synchronized. For a trip not involving a flight, one should not wait for the completion of booking the flight. However, for a business trip involving all three elements, all flows should be synchronized. The situation where there is sometimes no synchronization (XOR-join), sometimes full synchronization (AND-

join), and sometimes only partial synchronization (OR-join) needed is referred to as Pattern 7 (Synchronizing merge).

It is interesting to note that the Synchronizing merge is directly supported by InConcert, Eastman, Domino Workflow, and MQSeries Workflow. In each of these systems, the designer does not have to specify the type of join; this is automatically handled by the system.

In a high-level Petri net each construct is either an AND-join (transition) or an XOR-join (place). Nevertheless, it is possible to model the Synchronizing merge in various ways. First of all, it is possible to pass information from the split node to the join node. For example, if the business trip involves a flight and a hotel, the join node is informed that it should only synchronize the flows corresponding to these two elements. This can be done by putting a token in the input place of the synchronization transition corresponding to the element car rental. Second, it is possible to activate each branch using a "Boolean" token. If the value of the token is true, everything along the branch is executed. If the value is false, the token is passed through the branch but all activities on it are skipped. Third, it is possible to build a completely new scheduler in terms of high-level Petri nets. This scheduler interprets workflow processes and uses the following synchronization rule: "Fire a transition t if at least one of the input places of t is marked and from the current marking it is not possible to put more tokens on any of the other input places of t." In this last solution, the problem is lifted to another level. Clearly, none of the three solutions is satisfactory. The workflow designer has to add additional logic to the workflow design (case 1), has to extend the model to accommodate true and false tokens (case 2), or has to model a scheduler and lift the model to another level (case 3).

It is interesting to see how the problem of the Synchronizing merge has been handled in existing systems and literature. In the context of MQSeries workflow the technique of "dead-path elimination" is used [31, 22]. This means that initially each input arc is in state "unevaluated". As long as one of the input arcs is in this state, the activity is not enabled. The state of an input arc is changed to true the moment the preceding activity is executed. However, to avoid deadlocks the input arc is set to false the moment it becomes clear that it will not fire. By propagating these false signals, no deadlock is possible and the resulting semantics matches Pattern 7. The solution used in MQSeries workflow is similar to having true and false tokens (case 2 described above). The idea of having true and false tokens to address complex synchronizations was already raised in [18]. However, the bipolar synchronization schemes presented in [18] are primarily aimed at avoiding constructs such as the Synchronizing merge, i.e., the nodes are pure AND/XOR-splits/joins and partial synchronization is not supported nor investigated. In the context of Event-driven Process Chains (EPC's, cf. [26]) the problem of dealing with the Synchronizing merge also pops up. The EPC model allows for so-called $\vee$-connectors (i.e., OR-joins which only synchronize the flows that are active). The semantics of these $\vee$-connectors have been often debated [3, 11, 29, 37, 38]. In [3] the explicit modeling is advocated (case 1). Dehnert and Rittgen [11] advocate the use of a weak correctness notion (relaxed soundness) and an intelligent scheduler (case 3). Langner et al. [29] propose an approach based on Boolean tokens (case 2). Rump [38] proposes an intelligent scheduler to decide whether an $\vee$-connector should synchronize

or not (case 3). In [37] three different join semantics are proposed for the $\vee$-connector: (1) *wait for all to come* (corresponds to the Synchronizing merge, Pattern 7), (2) *wait for first to come and ignore others* (corresponds to the Discriminator, Pattern 9), and (3) *never wait, execute every time* (corresponds to the Multi merge, Pattern 8). The extensive literature on the synchronization problems in EPC's and workflow systems illustrates that patterns like the Synchronizing merge are relevant and far from trivial.

3.3 Cancellation patterns

Most workflow modeling languages, including high-level nets, have local rules directly relating the input of an activity to output. For most situations such local rules suffice. However, for some events local rules can be quite problematic. Consider for example the processing of Customs declarations. While a Customs declaration is being processed, the person who filed the declaration can still supply additional information and notify Customs of changes (e.g., a container was wrecked, and therefore, there will be less cargo as indicated on the first declaration). These changes may lead to the withdrawal of a case from specific parts of the process or even the whole process. Such cancellations are not as simple as they seem when for example high-level Petri nets are used. The reason is that the change or additional declaration can come at any time (within a given time frame) and may affect running and/or scheduled activities. Given the local nature of Petri net transitions, such changes are difficult to handle. If it is not known where in the process the tokens reside when the change or additional declaration is received, it is not trivial to remove these tokens. Inhibitor arcs allow for testing whether a place contains a token. However, quite some bookkeeping is required to remove tokens from an arbitrary set of places. Consider for example 10 parallel branches with 10 places each. To remove 10 tokens (one in each parallel branch) one has to consider 10^{10} possible states. Modeling a "vacuum cleaner", i.e., a construct to remove the 10 tokens, is possible but results in a spaghetti-like diagram. Therefore it is difficult to deal with cancellation patterns such as Cancel activity (Pattern 19) and Cancel case (Pattern 20) and anything in-between.

In this section we have discussed serious limitations of high-level Petri nets when it comes to (1) patterns involving multiple instances, (2) advanced synchronization patterns, and (3) cancellation patterns. Again, we would like to stress that high-level Petri nets are able to express such routing patterns. However, the modeling effort is considerable, and although the patterns are needed frequently, the burden of keeping track of things is left to the workflow designer.

4 YAWL: Yet Another Workflow Language

In a joint effort between Eindhoven University of Technology and Queensland University of Technology we are currently working on a new workflow language based on Petri nets. The goal of this joint effort is to overcome the limitations mentioned in the previous section by adding additional constructs. A detailed description of the language is beyond the scope of this paper. Moreover, the language is still under development.

The goal of this section is to briefly sketch the features of this language named *YAWL (Yet Another Workflow Language)*.

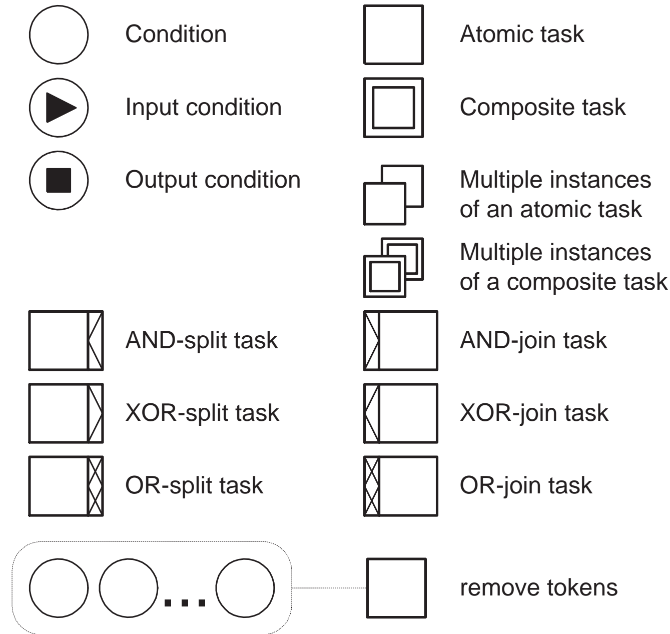


Fig. 2. Symbols used in YAWL.

Figure 2 shows the modeling elements of YAWL. YAWL extends the class of workflow nets described in [2, 4] with multiple instances, composite tasks, OR-joins, removal of tokens, and directly connected transitions. A *workflow specification* in YAWL is a set of *extended workflow nets* (EWF-nets) which form a hierarchy, i.e., there is a tree-like structure. *Tasks*[1] are either *(atomic) tasks* or *composite tasks*. Each composite task refers to a unique EWF-net at a lower level in the hierarchy. Atomic tasks form the leaves of the tree-like structure. There is one EWF-net without a composite task referring to it. This EWF-net is named the *top level workflow* and forms the root of the tree-like structure.

Each EWF-net consists of tasks (either composite or atomic) and *conditions* which can be interpreted as places. Each EWF-net has one unique *input condition* and one unique *output condition* (see Figure 2). In contrast to Petri nets, it is possible to connect "transition-like objects" like composite and atomic tasks directly to each other without using a "place-like object" (i.e., conditions) in-between. For the semantics this construct can be interpreted as a hidden condition, i.e., an implicit condition is added for every direct connection.

[1] Note that in YAWL we use the term *task* rather than *activity* to remain consistent with earlier work on workflow nets [2, 4].

Each task (either composite or atomic) can have multiple instances as indicated in Figure 2. It is possible to specify a lower bound and an upper bound for the number of instances created after initiating the task. Moreover, it is possible to indicate that the task terminates the moment a certain threshold of instances has completed. The moment this threshold is reached, all running instances are terminated and the task completes. If no threshold is specified, the task completes once all instances have completed. Finally, there is a fourth parameter indicating whether the number of instances is fixed after creating the instance. The value of the parameter is ”fixed” if after creation no instances can be added and ”var” if it is possible to add additional instances while there are still instances being processed. Note that by extending Petri-nets with this construct with four parameters (lower bound, upper bound, threshold, and fixed/var), we directly support all patterns involving multiple instances (cf. Section 3.1, and in addition, the Discriminator pattern (Pattern 9) under the assumption of multiple instances of the same task. In fact, we also support the more general n-out-of-m join [6].

We adopt the notation described in [2, 4] for AND/XOR-splits/joins as shown in Figure 2. Moreover, we introduce OR-splits and OR-joins corresponding to respectively Pattern 6 (Multi choice) and Pattern 7 (Synchronizing merge), cf. Section 3.2.

Finally, we introduce a notation to remove tokens from places independent of the fact if and how many tokens there are. As Figure 2 shows this is denoted by dashed circles/lines. The enabling of the task does not depend on the tokens within the dashed area. However, the moment the task executes all tokens in this area are removed. Clearly, this extension is useful for the cancellation patterns, cf. Section 3.3. Independently, this extensions was also proposed in [10] for the purpose of modeling dynamic workflows.

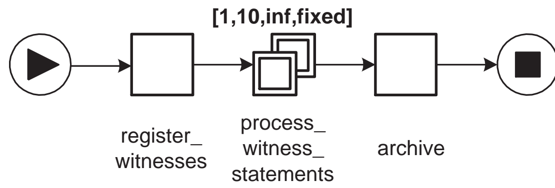
As indicated earlier, YAWL is still under development and the goal of this paper is not to introduce the language in any detail. Therefore, we restrict ourselves to simply applying YAWL to some of the examples used in the previous section.

4.1 Example: Patterns involving multiple instances

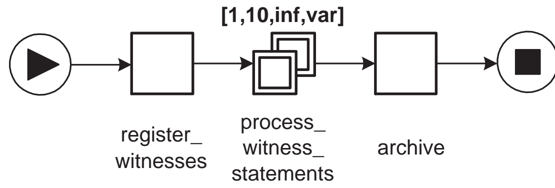
Figure 3 shows three workflow specifications dealing with multiple witness statements in parallel. The first workflow specification (a), starts between 1 and 10 instances of the composite task *process_witness_statement* after completing the initial task *register_witness*. When all instances have completed, task *archive* is executed. The second workflow specification shown in Figure 3(b), starts an arbitrary number of instances of the composite task and even allows for the creation of new instances. The third workflow specification (c) starts between 1 and 10 instances of the composite task *process_witness_statement* but the finishes if all have completed or at least three have completed. The three examples illustrate that YAWL allows for a direct specification of the patterns 14, 15, and 9.

4.2 Example: Advanced synchronization patterns

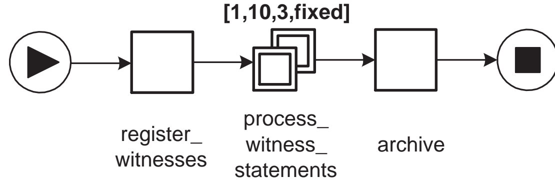
As explained in Section 3.2 an OR-join can be interpreted in many ways. Figure 4 shows three possible interpretations using the booking of a business trip as an example. The first workflow specification (a) starts with an OR-split *register* which enables tasks *flight*, *hotel* and/or *car*. Task *pay* is executed for each time one of the three tasks (i.e.,



(a) A workflow processing between 1 and 10 witness statements without the possibility to add witnesses after registration (Pattern 14).

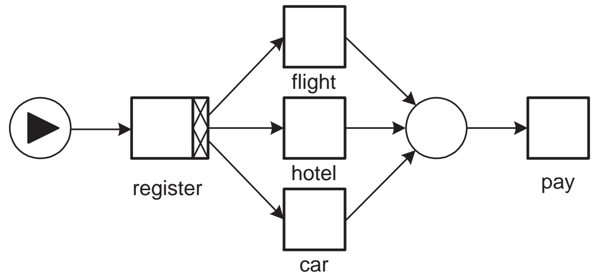


(b) A workflow processing and arbitrary number of witnesses with the possibility to add new batches of witnesses (Pattern 15).

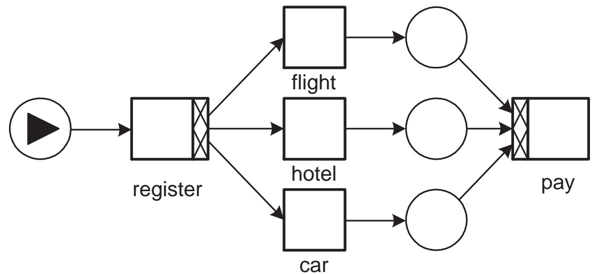


(c) A workflow processing between 1 and 10 witness statements with a threshold of 3 witnesses (extension of Pattern 9).

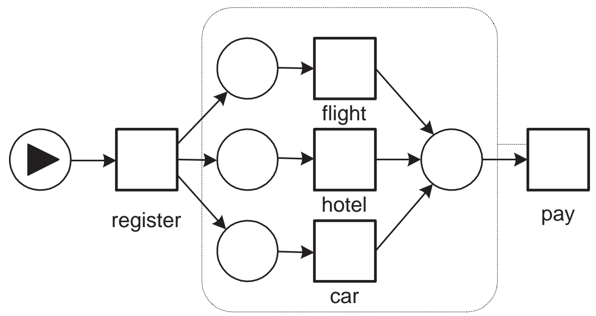
Fig. 3. Some examples illustrating the way YAWL deals with multiple instances.



(a) Task pay is executed each time one of the three preceding task completes (Pattern 8).



(b) Task pay is executed only once, i.e., when all started tasks have completed (Pattern 7).



(c) Task pay is executed only once, i.e., when the first task has completed (Pattern 9).

Fig. 4. Some examples illustrating the way YAWL deals with advanced synchronization patterns.

flight, *hotel*, and *car*) completes. This construct corresponds to the Multi merge (Pattern 8). The second workflow specification shown in Figure 4(b) is similar but combines the individual payments into one payment. Therefore, it waits until each of the tasks enabled by *register* completes. Note that if only a flight is booked, there is no synchronization. However, if the trip contains two or even three elements, task *pay* is delayed until all have completed. This construct corresponds to the Synchronizing merge (Pattern 7). The third workflow specification (c) enables all three tasks (i.e., *flight*, *hotel*, and *car*) but pays after the first task is completed. After the payment all running tasks are canceled. Although this construct makes no sense in this context it has been added to illustrate how the Discriminator can be supported (Pattern 9) assuming that all running threads are canceled the moment the first one completes.

4.3 Example: Cancellation patterns

Figure 5 illustrates the way YAWL supports the two cancellation patterns (patterns 19 and 20). The first workflow specification (a) shows the Cancel activity pattern which removes all tokens from the input places of task *activity*. In the second workflow speci-

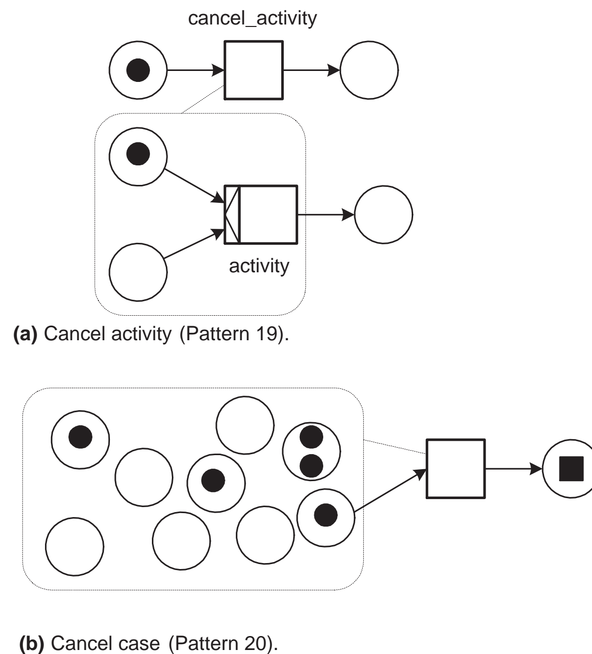


Fig. 5. Some examples illustrating the way YAWL deals with cancellation patterns.

fication (b) there is a task removing all tokens and putting a token in the output condition thus realizing the Cancel case pattern.

The examples given in this section illustrate that YAWL solves many of the problems indicated in Section 3. The table in the appendix shows that YAWL supports 19 of the 20 patterns used to evaluate contemporary workflow systems. Implicit termination (i.e., multiple output conditions) is not supported to force the designer to think about termination properties of the workflow. It would be fairly easy to extend YAWL with this pattern (simply connect all output conditions with an OR-join having a new and unique output condition). However, implicit termination also hides design errors because it is not possible to detect deadlocks. Therefore, there is no support for this pattern.

5 Conclusion

The workflow patterns described in previous publications [5–8, 48] provide functional requirements for workflow languages. Unfortunately, existing workflow languages only offer partial support for these patterns. Compared with the workflow languages used by commercial tools, high-level Petri nets are acceptable. Nevertheless, when it comes to patterns involving multiple instances, advanced synchronization, and cancellation, high-level Petri nets offer little support. Therefore, we are working towards a more expressive Petri-net-based language supporting most patterns. Moreover, we hope that the modeling problems collected in this paper will stimulate other researchers working on high-level Petri nets to develop mechanisms, tools, and methods providing more support.

References

1. W.M.P. van der Aalst. Chapter 10: Three Good reasons for Using a Petri-net-based Workflow Management System. In T. Wakayama, S. Kannapan, C.M. Khoong, S. Navathe, and J. Yates, editors, *Information and Process Integration in Enterprises: Rethinking Documents*, volume 428 of *The Kluwer International Series in Engineering and Computer Science*, pages 161–182. Kluwer Academic Publishers, Boston, Massachusetts, 1998.
2. 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.
3. W.M.P. van der Aalst. Formalization and Verification of Event-driven Process Chains. *Information and Software Technology*, 41(10):639–650, 1999.
4. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2002.
5. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Advanced Workflow Patterns. In O. Etzion and P. Scheuermann, editors, *7th International Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 18–29. Springer-Verlag, Berlin, 2000.
6. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. BETA Working Paper Series, WP 47, Eindhoven University of Technology, Eindhoven, 2000.

7. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. Technical report, Eindhoven University of Technology, Eindhoven, 2002. http://www.tm.tue.nl/it/research/patterns.
8. W.M.P. van der Aalst and H. Reijers. Adviseurs slaan bij workflow-systemen de plank regelmatig mis. *Automatisering Gids*, 36(15):15–15, 2002.
9. Pallas Athena. *Flower User Manual*. Pallas Athena BV, Apeldoorn, The Netherlands, 2001.
10. P. Chrzastowski-Wachtel. Top-down Petri Net Based Approach to Dynamic Workflow Modeling (Work in Progress). University of New South Wales, Sydney, 2002.
11. J. Dehnert and P. Rittgen. Relaxed Soundness of Business Processes. In K.R. Dittrich, A. Geppert, and M.C. Norrie, editors, *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01)*, volume 2068 of *Lecture Notes in Computer Science*, pages 157–170. Springer-Verlag, Berlin, 2001.
12. C.A. Ellis and G.J. Nutt. Modelling and Enactment of Workflow Systems. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, Berlin, 1993.
13. FileNet. *Visual WorkFlo Design Guide*. FileNet Corporation, Costa Mesa, CA, USA, 1997.
14. Forté. *Forté Conductor Process Development Guide*. Forté Software, Inc, Oakland, CA, USA, 1998.
15. M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Reading, Massachusetts, 1997.
16. Fujitsu. *i-Flow Developers Guide*. Fujitsu Software Corporation, San Jose, CA, USA, 1999.
17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison Wesley, Reading, MA, USA, 1995.
18. H. J. Genrich and P. S. Thiagarajan. A Theory of Bipolar Synchronization Schemes. *Theoretical Computer Science*, 30(3):241–318, 1984.
19. D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.
20. K.M. van Hee. *Information System Engineering: a Formal Approach*. Cambridge University Press, 1994.
21. HP. *HP Changengine Process Design Guide*. Hewlett-Packard Company, Palo Alto, CA, USA, 2000.
22. IBM. *IBM MQSeries Workflow - Getting Started With Buildtime*. IBM Deutschland Entwicklung GmbH, Boeblingen, Germany, 1999.
23. S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996.
24. K. Jensen. Coloured Petri Nets: A High Level Language for System Design and Analysis. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 342–416. Springer-Verlag, Berlin, 1990.
25. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1992.
26. G. Keller, M. Nüttgens, and A.W. Scheer. Semantische Processmodellierung auf der Grundlage Ereignisgesteuerter Processketten (EPK). Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 89 (in German), University of Saarland, Saarbrücken, 1992.
27. B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2002.
28. T.M. Koulopoulos. *The Workflow Imperative*. Van Nostrand Reinhold, New York, 1995.
29. P. Langner, C. Schneider, and J. Wehler. Petri Net Based Certification of Event driven Process Chains. In J. Desel and M. Silva, editors, *Application and Theory of Petri Nets 1998*, volume 1420 of *Lecture Notes in Computer Science*, pages 286–305. Springer-Verlag, Berlin, 1998.

30. P. Lawrence, editor. *Workflow Handbook 1997, Workflow Management Coalition*. John Wiley and Sons, New York, 1997.
31. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999.
32. M. Ajmone Marsan, G. Balbo, and G. Conte. A Class of Generalised Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(2):93–122, May 1984.
33. M. Ajmone Marsan, G. Balbo, and G. Conte et al. *Modelling with Generalized Stochastic Petri Nets*. Wiley series in parallel computing. Wiley, New York, 1995.
34. S.P. Nielsen, C. Easthope, P. Gosselink, K. Gutsze, and J. Roele. *Using Lotus Domino Workflow 2.0, Redbook SG24-5963-00*. IBM, Poughkeepsie, USA, 2000.
35. C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.
36. D. Riehle and H. Züllighoven. Understanding and Using Patterns in Software Development. *Theory and Practice of Object Systems*, 2(1):3–13, 1996.
37. P. Rittgen. Modified EPCs and their Formal Semantics. Technical report 99/19, University of Koblenz-Landau, Koblenz, Germany, 1999.
38. F. Rump. Erreichbarkeitsgraphbasierte Analyse ereignisgesteuerter Prozessketten. Technischer Bericht, Institut OFFIS, 04/97 (in German), University of Oldenburg, Oldenburg, 1997.
39. SAP. *WF SAP Business Workflow*. SAP AG, Walldorf, Germany, 1997.
40. T. Schäl. *Workflow Management for Process Organisations*, volume 1096 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1996.
41. A. Sheth, K. Kochut, and J. Miller. Large Scale Distributed Information Systems (LSDIS) laboratory, METEOR project page. http://lsdis.cs.uga.edu/proj/meteor/meteor.html.
42. Eastman Software. *RouteBuilder Tool User's Guide*. Eastman Software, Inc, Billerica, MA, USA, 1998.
43. Software-Ley. *COSA 3.0 User Manual*. Software-Ley GmbH, Pullheim, Germany, 1999.
44. Staffware. *Staffware 2000 / GWD User Manual*. Staffware plc, Berkshire, United Kingdom, 2000.
45. Tibco. *TIB/InConcert Process Designer User's Guide*. Tibco Software Inc., Palo Alto, CA, USA, 2000.
46. Verve. *Verve Component Workflow Engine Concepts*. Verve, Inc., San Francisco, CA, USA, 2000.
47. WFMC. Workflow Management Coalition Terminology and Glossary (WFMC-TC-1011). Technical report, Workflow Management Coalition, Brussels, 1996.
48. Workflow Patterns Home Page. http://www.tm.tue.nl/it/research/patterns.

A A comparison of high-level Petri nets and YAWL using the patterns

The table shown in this appendix indicates for each pattern whether high-level Petri nets/YAWL offers direct support (indicated by a "+"), partial direct support (indicated by a "+/-"), or no direct support (indicated by a "-").

pattern	high-level Petri nets	YAWL
1 (seq)	$+$	$+$
2 (par-spl)	$+$	$+$
3 (synch)	$+$	$+$
4 (ex-ch)	$+$	$+$
5 (simple-m)	$+$	$+$
6 (m-choice)	$+$	$+$
7 (sync-m)	$-^{(i)}$	$+$
8 (multi-m)	$+$	$+$
9 (disc)	$-^{(ii)}$	$+$
10 (arb-c)	$+$	$+$
11 (impl-t)	$-^{(iii)}$	$-^{(iv)}$
12 (mi-no-s)	$+$	$+$
13 (mi-dt)	$+$	$+$
14 (mi-rt)	$-^{(v)}$	$+$
15 (mi-no)	$-^{(vi)}$	$+$
16 (def-c)	$+$	$+$
17 (int-par)	$+$	$+$
18 (milest)	$+$	$+$
19 (can-a)	$+/-^{(vii)}$	$+$
20 (can-c)	$-^{(viii)}$	$+$

(i) The synchronizing merge is not supported because the designer has to keep track of the number of parallel threads and decide to merge or synchronize flows (cf. Section 3.2).

(ii) The discriminator is not supported because the designer needs to keep track of the number of threads running and the number of threads completed and has to reset the construct explicitly by removing all tokens corresponding to the iteration (cf. Section 3.2).

(iii) Implicit termination is not supported because the designer has to keep track of running threads to decide whether the case is completed.

(iv) Implicit termination is not supported because the designer is forced to identify one unique final node. Any model with multiple end nodes can be transformed into a net with a unique end node (simply use a synchronizing merge). This has not been added to YAWL to force the designer to think about successful completion of the case. This requirement allows for the detection of unsuccessful completion (e.g., deadlocks).

(v) Multiple instances with synchronization are not supported by high-level Petri nets (cf. Section 3.1).

(vi) Also not supported, cf. Section 3.1.

(vii) Cancel activity is only partially supported since one can remove tokens from the input place of a transition but additional bookkeeping is required if there are multiple input places and these places may be empty (cf. Section 3.3).

(viii) Cancel activity is not supported because one needs to model a vacuum clearer to remove tokens which may of may not reside in specific places (cf. Section 3.3).

Using Design/CPN to Design a Visualisation Extension for Design/CPN

Mathew Elliot, Jonathan Billington, and Lars M. Kristensen

Computer Systems Engineering Centre
School of Electrical and Information Engineering
University of South Australia
Mawson Lakes Campus, SA 5095, AUSTRALIA
Email: {Mathew.Elliot@dsto.defence.gov.au, Jonathan.Billington@unisa.edu.au, lars.kristensen@unisa.edu.au}

Abstract. This paper reports on the use of Coloured Petri Nets (CPNs) and DESIGN/CPN in the development of a protocol to support the visualisation and animation of a range of systems that can be modelled using CPNs. The developed protocol facilitates the co-ordination between a DESIGN/CPN simulation, and an *External Visualisation Package*. The protocol is modelled and analysed using CPNs created using DESIGN/CPN. The analysis shows that the protocol works as required and contains no deadlocks. The paper also analyses a more robust version of the protocol.

Keywords: Design/CPN Extensions, Visualisation, Animation of CPN Models

1 Introduction

When using the Coloured Petri Net (CPN) [7,8] tool, DESIGN/CPN [11], users are limited in the visualisations that can be performed. Whilst DESIGN/CPN provides very detailed visualisations in the form of the "token game" these are not easily understood by people unfamiliar with CPNs. There are other software packages that provide higher level visualisations for DESIGN/CPN, such as MIMIC/CPN [1] and the Message Sequence Chart library [10], however these packages are not always satisfactory for various reasons, including the fact that they rely on the DESIGN/CPN graphical user interface (GUI).

Prompted by the limitations imposed by the visualisation tools currently available for DESIGN/CPN we decided to create an *External Visualisation Package*. The key ideas behind this *External Visualisation Package* are: that visualisations can be developed with greater graphical capabilities (for example using Java libraries) than those of DESIGN/CPN; and the visualisation could be performed at a remote location, independently of the DESIGN/CPN GUI.

Such a visualisation tool has been created as part of the *Animation Facilities for Defence Systems Modelling* project [4] conducted by the University of South Australia for the Australian Defence Science and Technology Organisation (DSTO). The work was undertaken by a group of two students for their final year honours project, a mandatory component of a 4 year Bachelor of Engineering in Computer Systems Engineering at the University of South Australia. The project had two supervisors from the University of South Australia, and an external supervisor from DSTO, who provided requirements for the visualisation package. A requirement of final year computer systems engineering projects is that they must follow a full systems engineering approach. This involves fortnightly meetings with the supervisors, chaired by the students, to discuss project progress. Formal minutes of the meetings are taken

to record decisions and action items, and progress reports are submitted for each meeting. The students must submit for assessment a full set of documentation for the project that includes: a project plan, requirements specification, design description, test plan, test specification, test report, user manual, and system manual. The students also give a seminar on the project, and provide demonstrations of a working product, which is assessed by independent academic staff. In addition, honours students need to write a research proposal and paper.

The project officially comprises a quarter of a year's work for the two students, so that the total effort for this project was of the order of 5 person months. During this time, the students produced about 900 pages of A4 documentation for the project. This paper is a significant revision of the research paper written by the first author to fulfill the research paper requirement for honours students.

The goal of the project was to add visualisation facilities to a CPN model of an avionics mission system (AMS). An AMS is essentially a local area computer network consisting of a Serial Data Bus (SDB) that connects a set of components including a Mission Control Computer (MCC), various sensors, displays, and navigation systems.

The visualisation tool utilises the COMMS/CPN library [5, 6], to send information to the *External Visualisation Package (EVP)* via TCP/IP [2]. Further, an application protocol is required to support the exchange of visualisation commands between DESIGN/CPN and the EVP. The students developed and implemented a protocol for this purpose which is specified in the Software Design Description [3]. The students followed a rapid prototyping paradigm, to provide a rough first cut implementation to test out visualisation ideas, so that feedback could be obtained from the customer (DSTO) at the earliest opportunity. To add rigour to the approach for the honours component, it was decided to create a formal model of this application protocol using CPNs.

The purpose of this paper is to present our work on modelling this *visualisation* protocol and to verify that it is deadlock and livelock free and conforms to a sequencing constraint. The visualisation protocol is then extended and re-analysed.

The paper is organised as follows. Section 2 provides some insight into the nature of an AMS, and the visualisations that were implemented. We describe the visualisation protocol and its sequencing constraints in section 3 and a CPN model of the visualisation protocol in section 4. The model is analysed in section 5. Section 6 describes an enhanced protocol, which allows DESIGN/CPN to be informed of the success or failure of each visualisation command. The enhanced CPN model and its analysis are presented in sections 7 and 8, with our conclusions provided in section 9. The reader is assumed to be familiar with CPNs [7] and DESIGN/CPN [11].

2 Overview of the External Visualisation Package

The purpose of the external visualisation package is to provide domain specific visualisation facilities for the avionics mission system model at a higher, more abstract level than the DESIGN/CPN tool can provide. These facilities are to produce visualisations that are more strongly related to the physical system being modelled than is capable with DESIGN/CPN. This is important from a system development perspective as well as a marketing and management perspective because it is likely than these people would be unfamiliar CPN specific concepts. The four visualisations required to be supplied by the *EVP* were:

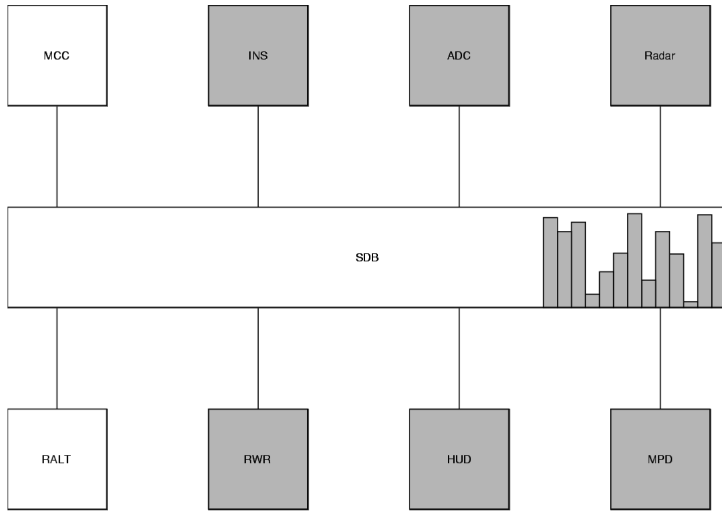


Fig. 1. Block diagram showing components of the AMS.

1. A block diagram of the AMS and its components. An example of this can be seen in Figure 1, which shows the Mission Control Computer (MCC), Inertial Navigation System (INS), Air Data Computer (ADC), Radar, Radar Altimeter (RALT), Radar Warning Receiver (RWR), Heads Up Display (HUD), Multi-Purpose Display (MPD), and Serial Data Bus (SDB) components of the AMS.
2. A diagram showing the progress throughout a mission. This is simply a bar chart with a line showing the progress through the mission as a percentage between 0 and 100.
3. More detailed information on the MCC and SDB components within the AMS as illustrated in Figure 2.
4. Message sequence charts to show instructions sent between components of the AMS.

A further decision was made to design the *EVP* architecture to be flexible enough to allow the use of different visualisation libraries. The rationale behind this was that the *EVP* could be reused to visualise many different systems rather than just avionics mission systems.

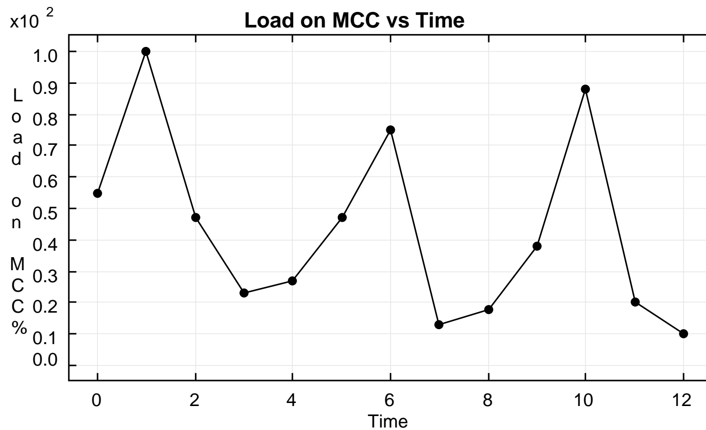


Fig. 2. Detailed view of the MCC component.

3 Visualisation Protocol

Figure 3 shows the overall architecture of the system that includes the *EVP*. The visualisation protocol is between the *AMS Visualisation Primitives* block on the DESIGN/CPN side and the *Visualisation Display* on the EVP side. The protocol itself comprises:

1. Client side: this is DESIGN/CPN in this implementation but could be any application.
2. Server side: the *EVP* which receives the visualisation instructions.

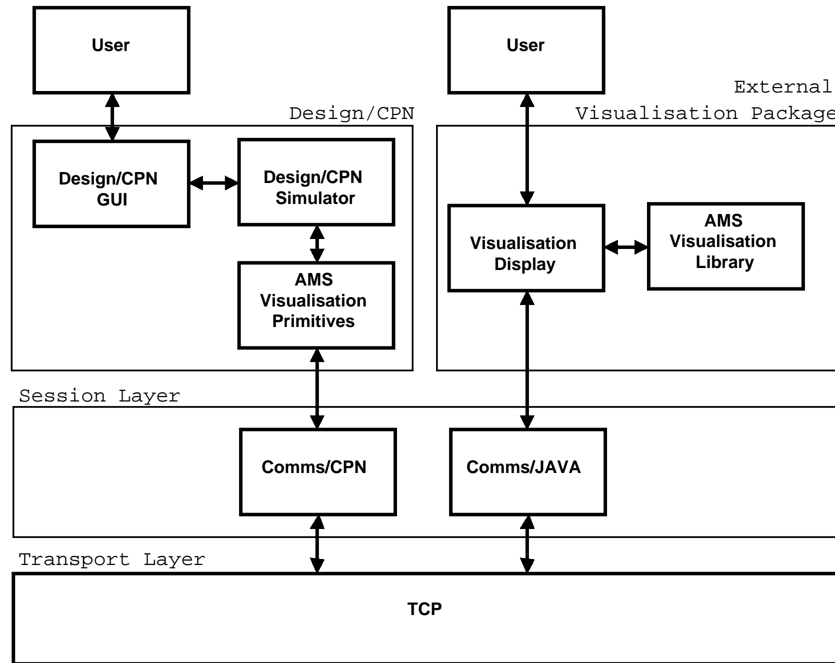


Fig. 3. Overall Decomposition of the External Visualisation Package.

3.1 Visualisation Instructions

A detailed specification for this protocol as used in the External Visualisation Package can be found in the Software Design Description [3]. The protocol facilitates the sending of instructions between the client, DESIGN/CPN, and the server, the External Visualisation Package. These instructions inform the visualisation package of the components required for visualisation and the visualisations that are to be performed. On the DESIGN/CPN side, the instructions are sent and received using the COMMS/CPN library [5,6] and hence communication between the client and server is via TCP/IP [2]. As TCP/IP is considered to provide a reliable transport service, it is assumed that all instructions sent arrive successfully and in order.

Two sets of instructions are sent between DESIGN/CPN and the External Visualisation Package. *Initialisation Instructions* as listed in Table 1 are used to initialise the External Visualisation Package before any visualisation can occur. The second set of instructions enables the *EVP* to perform animations involving the components specified using the initialisation commands. As the visualisation protocol is designed for visualisation of Avionics Mission Systems, many of these instructions are specific to these systems. The *Visualisation Instructions* are given in Table 2.

Instruction	Purpose
useLibrary	Specifies the visualisation library to be used by the *EVP*.
createComponent	Creates an instance of a component to be used by the *EVP*.
initComplete	Specifies that the initialisation is complete, (i.e. that all components to be used by the *EVP* have been created.)

Table 1. The *Initialisation Instructions* used within the visualisation protocol.

Instruction	Purpose
isCommunicating	displays that a specified component is communicating.
stoppedCommunicating	displays that a specified component is not communicating.
createLink	creates a visible link between two components used within a visualisation.
MCCLoadIndication	displays the load on the MCC component.
SDBLoadIndication	displays the load on an SDB component.
getInstruction	This is the only visualisation command that expects a response from the visualisation server. This instruction expects the server to reply with either a "STOP" or a "CONT" message. If a "STOP" signal is received the transfer of further visualisation commands is stopped. The "CONT" signal allows more visualisation commands to be sent.

Table 2. The *Visualisation Instructions* used within the visualisation protocol.

The order in which these instructions must be sent for successful visualisation is defined by the regular expression [9]:

$$useLibrary\ createComponent^{*}\ initComplete\ [VisualisationInstructions]^{*} \tag{1}$$

where [Visualisation Instructions] are any of the *Visualisation Instructions* defined in Table 2.

3.2 Client

The client Design/CPN sends out *Initialisation Instructions*, as defined in Table 1, to initialise the visualisation tool. It then sends *Visualisation Instructions* (listed in Table 2), until it is informed to stop by a reply of STOP (to the *getInstruction*) or there are no more visualisation commands to send. This progression of instructions is specified by the regular expression (Equation 1) given in section 3.1.

3.3 Server

The *Visualisation Server* receives instructions from Design/CPN via the session service. The server is responsible for determining if the instruction is valid (ie., is to be acted upon). If an instruction is determined to be invalid, (for example a useLibrary instruction is received after the library to be used has already been specified), the instruction is simply discarded.

4 Protocol CPN Model

4.1 Modelling Approach

Although the CPN model of the protocol is constructed on one level, (there are no substitution transitions), it is clearly separated into three sections: the Design/CPN Simulator; the

Session Service; and the External Visualisation Package, as can be seen in Figure 4. Figure 4 will be explained in more detail in the following subsections. We model the protocol at a systems design level and make the following assumptions:

1. The content of the instructions sent to and from the server are not important for the operation of the protocol.
2. Only one instruction can be processed at a time at the server.
3. The session service is free of errors, and preserves the sequence of messages.
4. A session is already established between both ends.
5. All the *Visualisation Instructions* defined in Table 2, except getInstruction as this requires a reply from the server, are abstracted into one message called UPDATEGUI. This is because these Visualisation Instructions just update the visualisation without waiting for a reply from the server. Therefore the behaviour for all of these instructions is the same.

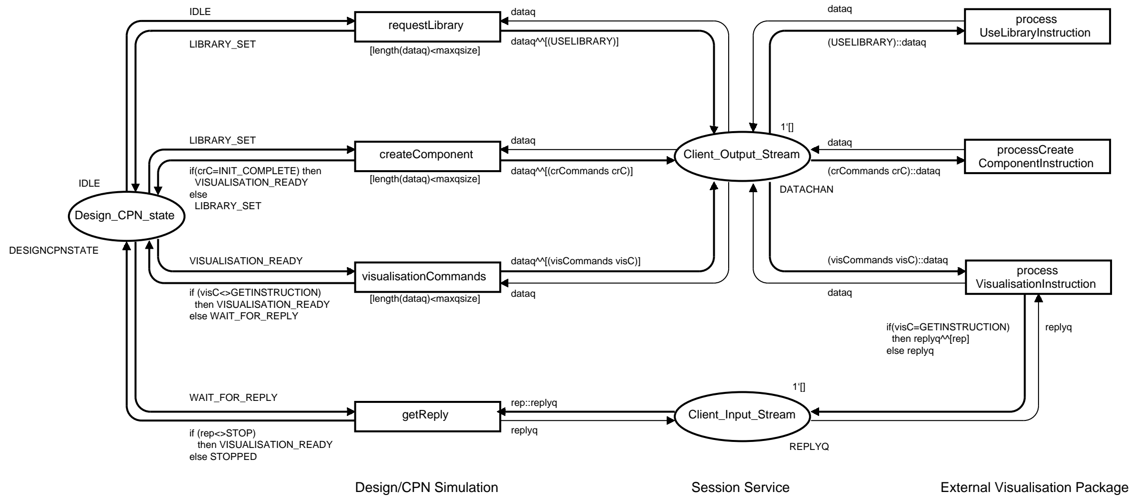


Fig. 4. Model of the basic protocol.

4.2 Data Structures and Declarations

Figure 5 shows the colour sets used in the CPN model. The CR_COMPONENT_COMMAND colour set is used to define the two initialisation instructions *createComponent and initComplete* as previously defined. The VISCOMMANDS colour set defines the visualisation instructions that can be sent to the *EVP*. The colour set DATA is the union of the initialisation instructions, the visualisation instructions and the *useLibrary* initialisation instruction, so that all types of instructions may be put in the client output queue. The DATACHAN colour set is simply a list of the DATA colour set, which is used to model the session service. The REPLY colour set enumerates the valid replies from the *EVP* (STOP or CONTINUE). The REPLYQ colour set is used to create a list of the REPLY colour set for the output queue of the *EVP*. The final colour set, DESIGNCPNSTATE, is used to enumerate the states of the DESIGN/CPN side. The client is either IDLE, waiting to send the USELIBRARY initialisation instruction, LIBRARY_SET when the USELIBRARY instruction has been sent and the client is now either creating components or sending the *initComplete* instruction, VISUALISATION_READY, when sending visualisation instructions, WAIT_FOR_REPLY when

```
color CR_COMPONENT_COMMAND = with CREATECOMPONENT | INIT_COMPLETE;
color VISCOMMANDS = with GETINSTRUCTION | UD
color DATA = union crCommands : CR_COMPONENT_COMMAND + visCommands : VISCOMMANDS + USELIBRARY;
color DATACHAN = list DATA;
color REPLY = with CONTINUE | STOP;
color REPLYQ=list REPLY;
color DESIGNCPNSTATE = with IDLE | LIBRARY_SET | VISUALISATION_READY
                         | WAIT_FOR_REPLY | STOPPED;

val maxqsize=1;
var replyq : REPLYQ;
var rep : REPLY;
var dataq : DATACHAN;
var visC : VISCOMMANDS;
var crC : CR_COMPONENT_COMMAND;
```

Fig. 5. Colour set declarations.

waiting for a reply, or STOPPED when no more instructions are to be sent. A set of variables needed in arc expressions and guards, are declared, and the value of the constant, maxqsize, is defined, to limit the size of the session service queues.

4.3 Places

The DESIGN/CPN side of the protocol consists of a single place, *Design_CPN_state* that describes the state of the client. Its initial marking is IDLE. Two places (*Client_Input_Stream* and *Client_Output_Stream*) are used to represent the transmission of instructions between DESIGN/CPN and the *External Visualisation Package*. These two places have been implemented as FIFO queues (Assumption 3 in Section 4.1). The initial marking for both of these places is an empty list. The External Visualisation Package side contains no places and thus, no state information. This is due to the design of the server, which handles instructions received in any order.

4.4 Transitions

The DESIGN/CPN simulator side consists of four transitions. Three of these transitions are used to send instructions to the *External Visualisation Package* ordered to satisfy the regular expression shown in equation (1) in section 3.1. The *requestLibrary* transition is used to place a USELIBRARY instruction into the queue. The transition, *createComponent*, is used to put either a CREATECOMPONENT or INITCOMPLETE instruction into the queue. The *visualisationCommands* transition places either a GETINSTRUCTION or UPDATEGUI instruction into the queue. All three transitions have a guard to limit the number of instructions placed into the queue to prevent an infinite state space when performing analysis. The final transition, *getReply*, is used to receive a response from the EVP output stream.

The EVP side of the CPN contains three transitions *processUseLibraryInstruction*, *processCreateComponentInstruction*, and *processVisualisationInstruction*, which are used to handle the incoming commands. The *processUseLibraryInstruction*, and *processCreateComponentInstruction*, transitions simply remove the incoming instruction from the queue, as no information is sent back. The *processVisualisationInstruction* transition performs the same function;

however if a GETINSTRUCTION is received, either a STOP or CONT signal is sent back. This represents the user specifying whether to continue with the visualisation or stop, and is handled non determinisitically at this this level of abstraction.

5 Analysis of the Visualisation Protocol

Figure 6 shows the reachability graph of the CPN model described in section 3.1 using a queue size of one. As can be seen there is only one terminal marking for the CPN, which is located at node 11. The marking at node 11 shows that all the queues are empty and that only a STOPPED token exists in the *Design_CPN_state place*. This is the expected and desired result. The visualisation has ended with no instructions in the queues. This indicates that both ends are correctly synchronised. Node 11 is a home state, which is also desirable because this means that the protocol always has the possibility to terminate successfully from any state that the protocol may be in.

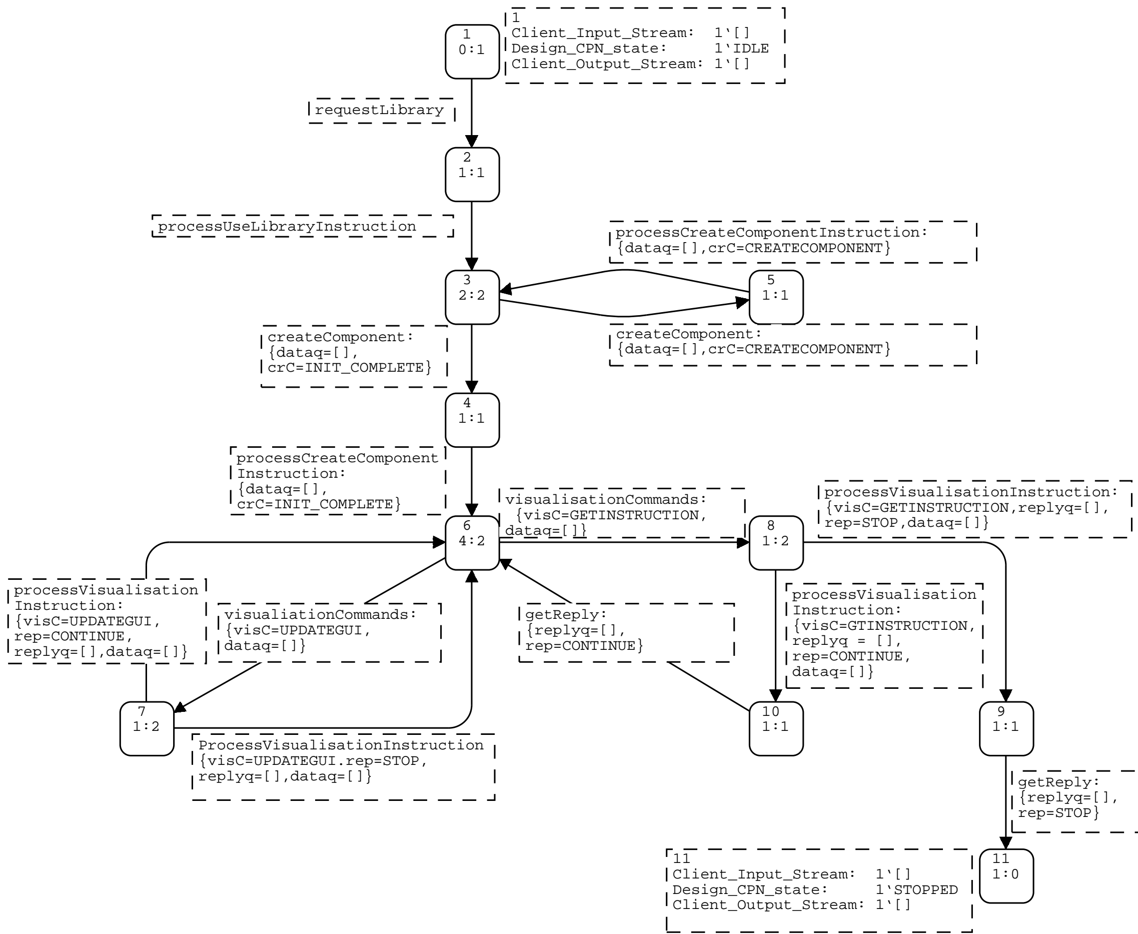


Fig. 6. Reachability graph of the Protocol CPN.

The *Strongly Connected Component (SCC)* graph was also calculated and is shown in Figure 7. The fact that the SCC graph contains less nodes than the reachability graph indicates that there are cycles within the protocol, although due to the fact that the terminal state

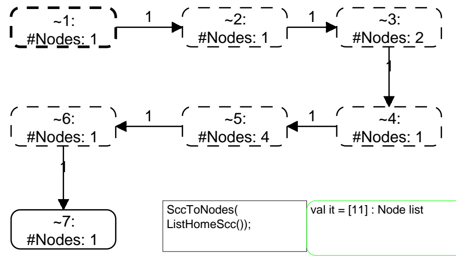


Fig. 7. SCC graph for the Basic Visualisation Protocol.

is a home state there always exists the possibility for the protocol to terminate successfully. An example of one of these cycles can be seen from nodes 6, 8 and 10 of the reachability graph. If the DESIGN/CPN simulator sends a GETINSTRUCTION and moves to the WAITING_FOR_REPLY state, then the *EVP* replies with CONTINUE, and the DESIGN/CPN simulator can again send the GETINSTRUCTION command. This cycle could continue forever. The protocol is proven to be livelock free because the SCC graph contains only one terminal node which contains the desired node 11 of the reachability graph.

Increasing the queue length beyond one causes the state space to grow. However in all examined cases up to a queue length of 40 (a node count of 3287) there exists only one terminal node with identical charateristics to node 11. We therefore conjecture that the queue size does not effect the operation of the protocol, but only affects the number of instructions that can be queued in the network at any time.

This model shows that the visualisation protocol satisfies its purpose of providing visualisation information to the *EVP* without being prone to deadlocks or livelocks, even though cycles are possible. However there is a major limitation of the protocol. This limitation is that the instructions sent are presumed to be correct. This means that users of the protocol are "left blind" not knowing if the instructions they just sent were valid. To overcome this, an attempt has been made to specify a more robust visualisation protocol, in Section 6, which sends replies back for each visualisation instruction received.

6 A More Robust Protocol for External Visualisation

The improved visualisation protocol includes a response for each instruction sent. This response indicates whether the instruction was successful or not, and if an instruction failed, the reason why is given to inform users of the protocol. The success or failure of these instructions is determined by the allowable sequences as specified by the regular expression in equation (1) of Section 3.1.

The *External Visualisation Package* side of the protocol has been modified significantly to determine if a visualisation instruction is acceptable. The server will either respond with S(uccess) or F(ail) depending on the current state of the *EVP* and the instruction received. The acceptability of an instruction is determined using the order that instructions can be received in for a successful visualisation to occur. This order is the same as the order specified in equation 1 using a FIFO queue ensures that instructions are received in the same order that they are sent.

The client has been modified to wait for a reply from the EVP each time an instruction is sent. The reply is then analysed to see if the instruction succeeded or failed, which allows users of the protocol to handle these events appropriately. This has the advantage of providing a flow

control, (which also eliminates the state space explosion), but also introduces round trip delay which could be significant on networks with large propagation delays. As the visualisation tool is intended to be used on a local area network this is not seen as a problem.

7 CPN Model of the Improved Protocol

Due to its greater complexity, the improved protocol CPN model has been constructed hierarchically. Figure 8 shows the top level of the hierarchy. The model of the improved visualisation protocol consists of four port places and four substitution transitions. The same assumptions have been made as were specified in Section 3.1.

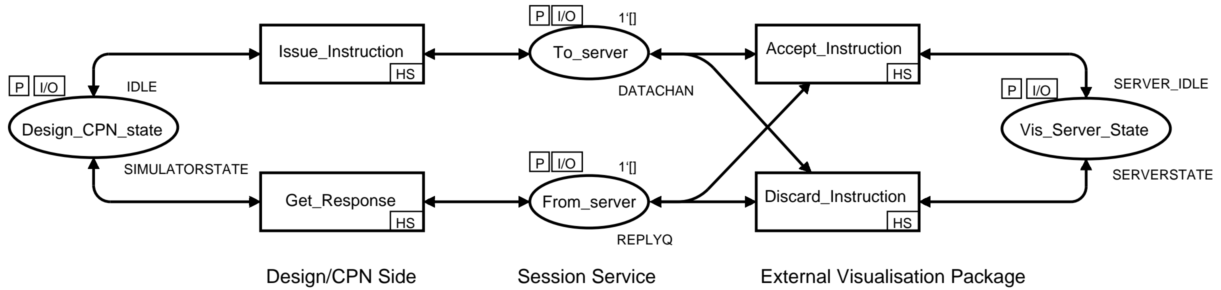


Fig. 8. Top Level of Improved Visualisation Protocol.

7.1 Data Structures and Declarations

Figure 9 shows the revised declarations. The changes with respect to Figure 5 are as follows.

The INSTRUCTION_STATUS colour set is used as a boolean to declare if an instruction was successful or not: **S** indicates success and **F** indicates a Fail. The colour set ARGS specifies the reason why an instruction failed. For example when a visualisation command is received before the initialisation is complete, the reason for failure would be INIT_NOT_COMPLETE. When an instruction is successful the ARG is NA (Not Applicable), unless it is a response to the AWAITING instruction in which case the ARG will be either CONT or STOP. The colour set REPLY is the product of the colour sets INSTRUCTION_STATUS and ARGS and defines a reply by the server. The SERVERSTATE colour set enumerates the three states of the Visualisation Package. SERVER_IDLE indicates when the server has not received a USELIBRARY instruction, LIBRARYSET indicates a USELIBRARY instruction has been received but the INITCOMPLETE instruction has not, and INITCOMPLETE indicates that the initialisation is complete.

The colour set SIMULATORSTATE is used to represent the various states of the DESIGN/CPN side. This state is IDLE indicating that a USELIBRARY command has not yet been successfully sent, WAIT_FOR_LIB_REPLY when the client is waiting for a reply to a USELIBRARY instruction, LIBRARY_SET indicating that the USELIBRARY command has been successfully sent and the client will send either CREATECOMPONENT or INITCOMPLETE instructions, WAIT_FOR_CR_COMPONENT_REPLY indicating that the client is waiting for a reply to the CREATECOMPONENT, WAIT_FOR_INIT_COMPLETE_REPLY indicating that the client is waiting for a reply to an INIT_COMPLETE instruction, VISUALISATION_READY when visualisation instructions are being sent, WAIT_FOR_REPLY when the client is waiting for a response to a visualisation instruction and finally GET_RESPONSE when the client is waiting for a response to either continue or stop the visualisation.

```
color CR_COMPONENT_COMMAND = with CREATECOMPONENT | INIT_COMPLETE;
color VISCOMMANDS = with AWAITING | UPDATEGUI;

color DATA = union crCommands  : CR_COMPONENT_COMMAND +
                   visCommands : VISCOMMANDS + USELIBRARY;
color DATACHAN = list DATA;
color INSTRUCTION_STATUS=with S|F;

color ARGS= with LIBRARY_NOT_SET | LIBRARY_ALREADY_SET |
                 INIT_NOT_COMPLETE | NA | CONT | STOP;

color REPLY = product INSTRUCTION_STATUS*ARGS;
color REPLYQ=list REPLY;

color SERVERSTATE = with SERVER_IDLE | LIBRARYSET | INITCOMPLETE;
color SIMULATORSTATE = with  IDLE | LIBRARY_SET |
          VISUALISATION_READY | WAIT_FOR_REPLY | STOPPED | GET_RESPONSE |
          WAIT_FOR_CR_COMPONENT_REPLY | WAIT_FOR_INIT_COMPLETE_REPLY |
          WAIT_FOR_LIB_REPLY;

var crC : CR_COMPONENT_COMMAND;
var dataq : DATACHAN;
var visC : VISCOMMANDS;
var replyq : REPLYQ;
var rep : REPLY;
var sf : INSTRUCTION_STATUS;
var args : ARGS;
var serState: SERVERSTATE;

val maxqsize=1;
```

Fig. 9. Declarations for the CPN of Improved Visualisation Protocol.

7.2 Places

The DESIGN/CPN side has a single place, *Design_CPN_state*, used to record the current state of the DESIGN/CPN side of the protocol, according to its type, SIMULATORSTATE. The *To_server* and *From_Server* places shown in Figure 8 represent the *Client_Output_Stream* and *Client_Input_Stream* places as respectively discussed in Section 4. The External Visualisation Package has a single place to store the state of the *EVP* server.

7.3 Substitution Transitions

The Issue_Instruction subpage is shown in Figure 10. Its purpose is to place commands into the queue in the order specified in Section 3.1.

Figure 11 defines the Get_Response subpage, which receives responses from the server side. The *getLibraryReply* transition retrieves a response from the USELIBRARY instruction - if the instruction was successful then the client will be added to the LIBRARY_SET state, otherwise the server is returned to IDLE. The *getCreateComponentReply* transition is responsible for determining whether or not a CREATECOMPONENT instruction was successful. The *getInitCompleteReply* places the client in the VISUALISATION_READY state if the server

replies with a success to the INIT_COMPLETE instruction, otherwise the server is returned to the LIBRARY_SET state. Responses to an instruction sent to the *EVP* are retrieved by the *getResponse* transition. The final transition, *getReply*, obtains a response from the EVP in response to an AWAITING instruction. This transition is distinct from the getResponse transition because the arguments of the S(uccess) response are inspected to see if the command is CONT or STOP.

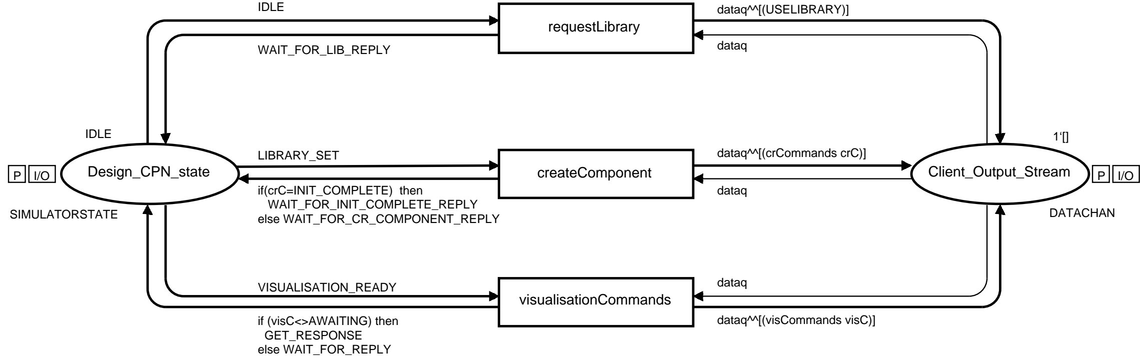


Fig. 10. Subpage for the Issue_Instruction transition.

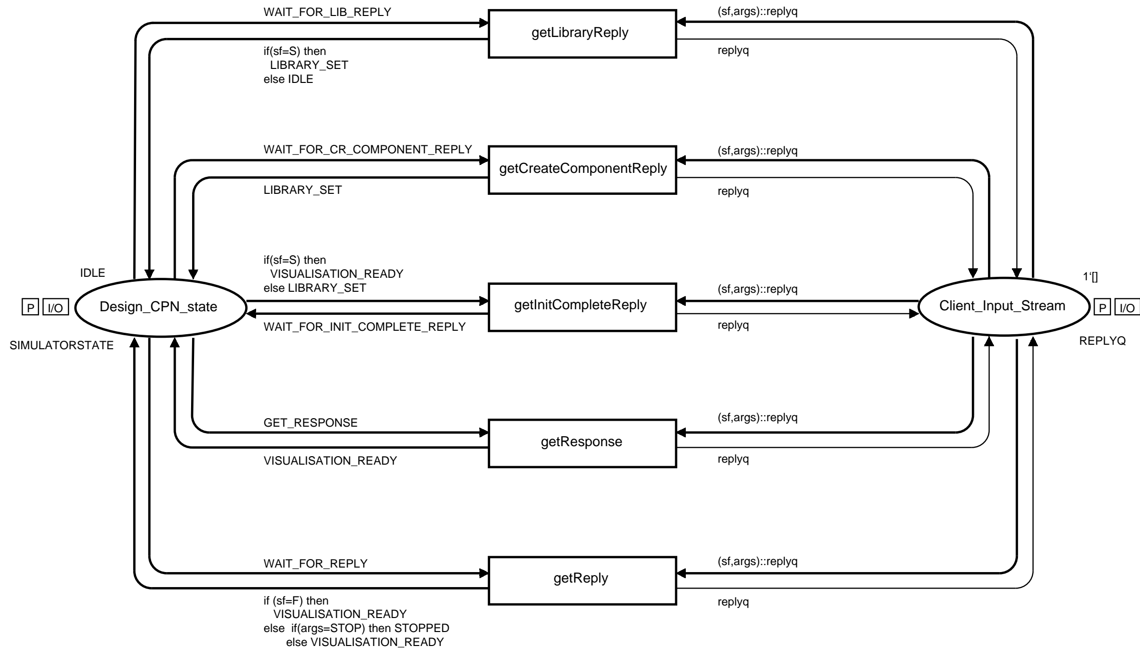


Fig. 11. Subpage for the Get_Response transition.

There are six transitions contained within the subpage of the Accept_Instruction substitution transition as shown in Figure 12. These transitions are activated when instructions arrive in the correct order. The *processUseLibraryInstruction* transition is enabled only when *EVP* is IDLE and a USELIBRARY instruction is received. The USELIBRARY instruction is removed from the queue and a *S* reply is issued to signify that the instruction was suc-

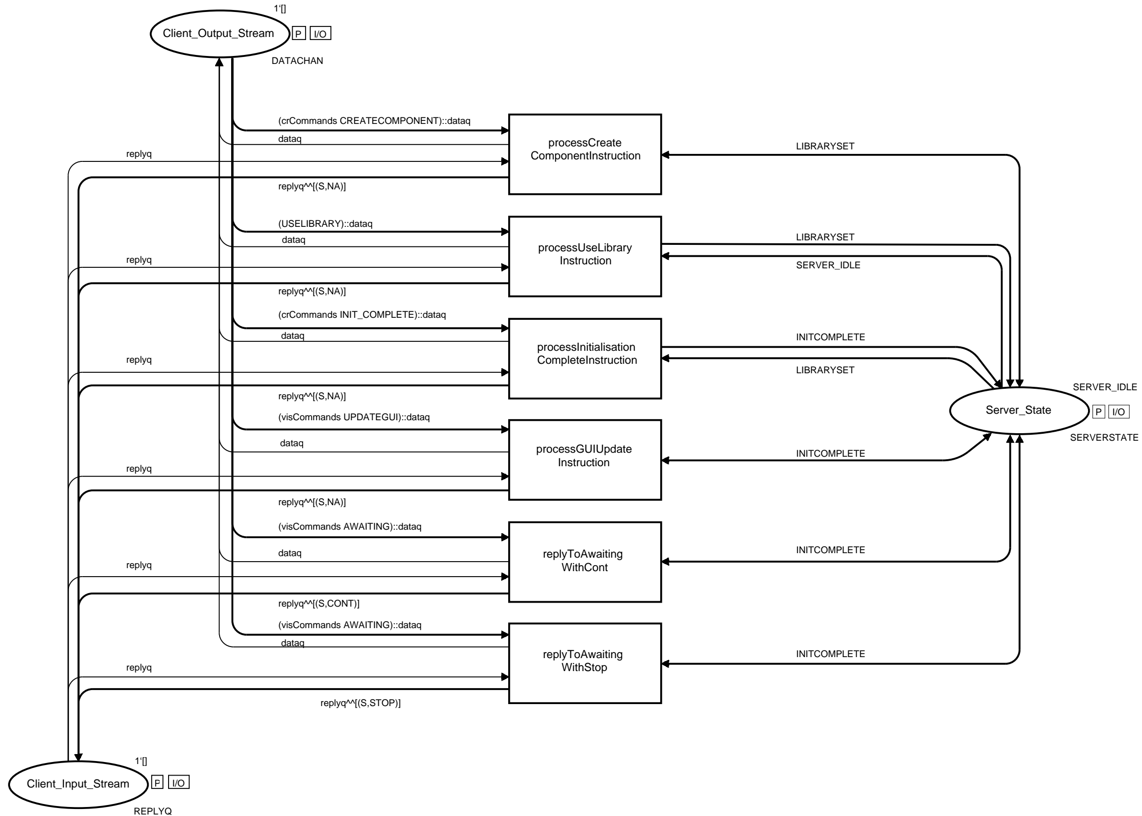


Fig. 12. Subpage for the Accept_Instruction transition.

cessful. The *processCreateComponentInstruction* transition is enabled only when the *EVP* is in the LIBRARYSET mode and a CREATECOMPONENT instruction is received. The CREATECOMPONENT instruction is removed from the queue and a *S* reply is issued. The *processInitialisationCompleteInstruction* transition is enabled only when the *EVP* is in the LIBRARYSET mode and an INIT_COMPLETE instruction is received. The instruction is removed from the queue and a *S* reply is issued, the *EVP* state is then changed to INITCOMPLETE. The *ProcessGUIUpdateInstruction* transition is enabled only when an UPDATEGUI instruction is received and the *EVP* is in the INITCOMPLETE mode. The UPDATEGUI instruction is removed from the queue and a *S* reply is issued. The *replyToAwaitingWithCont* transition is enabled when an AWAITING command is received and the server is in the INITCOMPLETE state. The AWAITING instruction is removed from the queue and a *(S,CONT)* reply is sent. The final transition, *replyToAwaitingWithStop*, is enabled when an AWAITING command is received and the server is in the INITCOMPLETE state. The AWAITING instruction is removed from the queue and a *(S,STOP)* reply is sent.

The four transitions contained within the Discard_Instruction substitution transition can be seen in Figure 13. These transitions are activated when an instruction arrives out of order and is hence discarded.

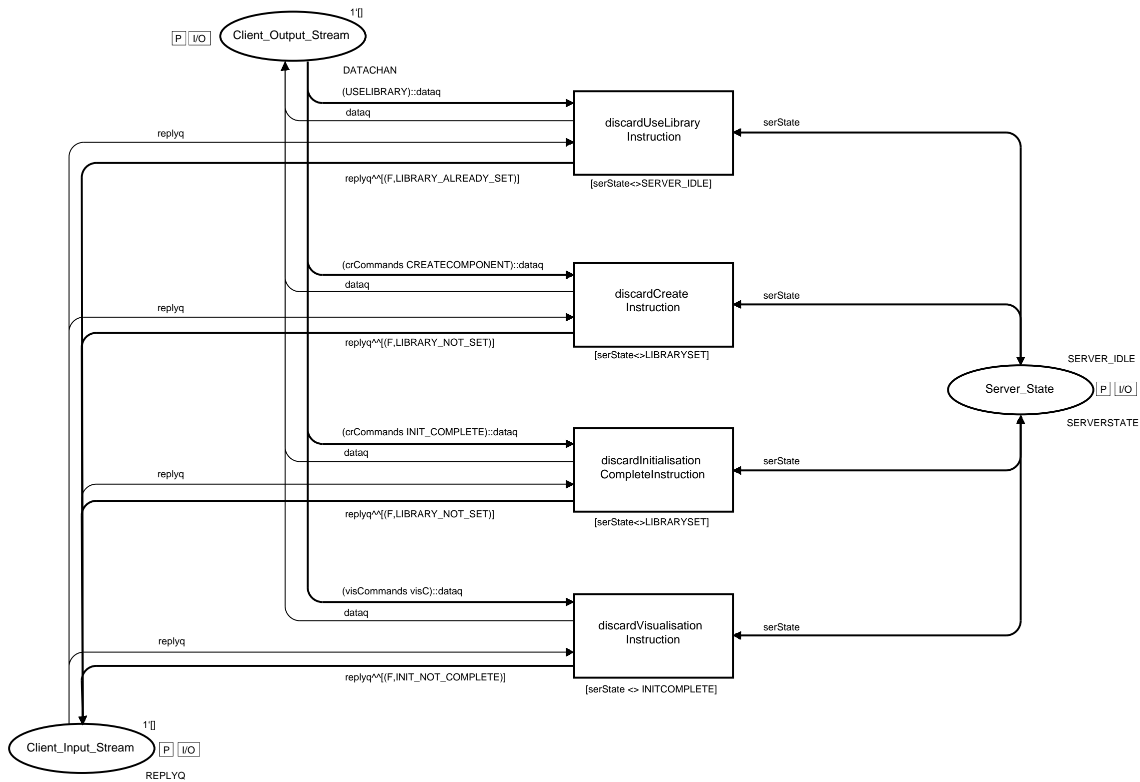


Fig. 13. Subpage for the Discard_Vis_Instruction transition.

8 Analysis of the Improved Visualisation Protocol

Figure 14 shows the reachability graph for the improved protocol with a queue length of one. As with the original protocol, there is only one terminal marking, which is at node 15. This node has similar characteristics to node 11 in Figure 6.

By examining the reachability graph we can observe the following:

- The occurence of the binding elements 1:1->2, 2:2->3, 3:3->4 shows a USELIBRARY instruction is sent and replied to sucessfully, this corresponds to the useLibaray part of the regular expression presented in Section 3.1.
- At node 4 two paths may be followed. By the occurrence of binding elements 5:4->6, 7:6->8 and 9:8->4, we can see that it is possible to send as many CREATE_COMPONENT instructions as we want with each being followed by a successful response. This path can be taken zero or more times. The only other path that can be taken from node 4 is by the occurrence of binding elements 4:4->5 6:5->7 and 8:7->9, which causes the sending of the INIT_COMPLETE command followed by a successful response from the server. These two possibilites correspond to the second and third terms of the regular expression - being createComponent* initComplete.
- The remaining biding elements show the sending and receiving responses to visualisation instructions, these correspond to the fourth term in the regular expression - [Visualisation Instructions]*.

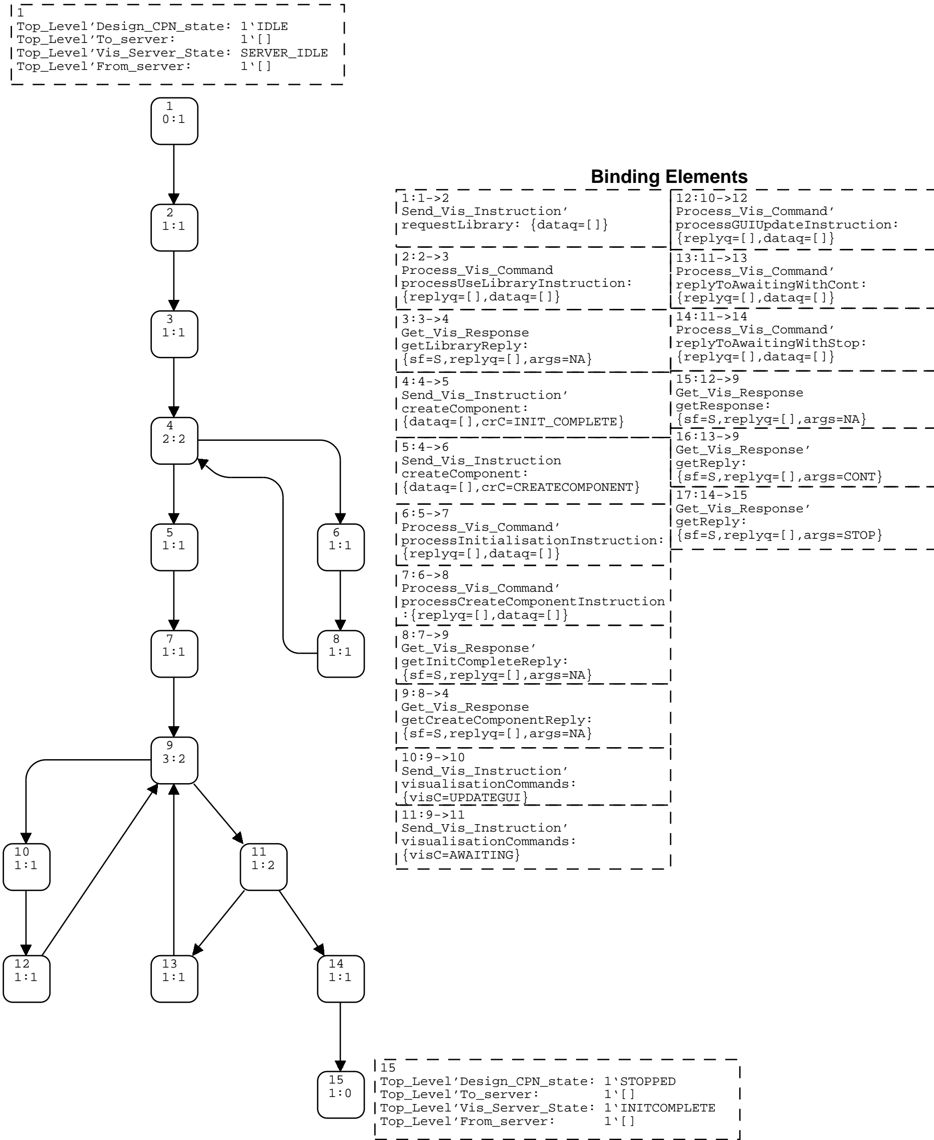


Fig. 14. Reachability Analysis for Model of Improved Protocol.

In summary, we can see that the client sends the instructions in the order specified by the regular expression (equation 1), and that they are also received by the server in the correct order, therefore we can say that the improved protocol behaves as expected.

Figure 15 shows the *Strongly Connected Component (SCC)* graph calculated from the CPN. As with the original protocol, the SCC graph contains less nodes than the reachability graph, indicating that there are cycles present. Also shown however is the fact that the only terminal node of the SCC contains the node that is the dead marking, node 15. This proves that there are no livelocks within the protocol.

Increasing the queue length beyond 1 has no effect as new instructions are not placed in the queue until the previous one is acknowledged. The only side effect of the new protocol is that it is reduced to a stop and wait protocol as after each instruction is sent to the *EVP* it waits for a response. This will lead to reduced performance on a network with high latency due to the fact that information has to travel in both directions whereas in the original protocol information was sent in only one direction a majority of the time.

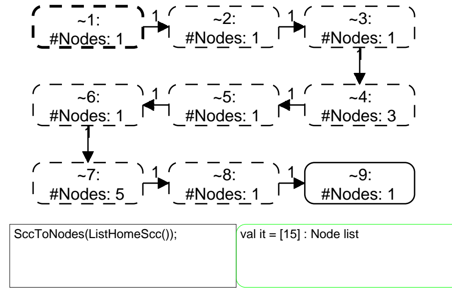


Fig. 15. SCC graph for the Improved Visualisation Protocol.

9 Conclusions

This paper shows that the visualisation protocol presented here is acceptable, if not perfect (due to the fact the sender of instructions is not informed of errors). The protocol allows a client application to request a remote a visualisation of system behaviour using the *External Visualisation Package*. It has been proved that the protocol is deadlock and livelock free and complies with the specification of the order in which instructions can be sent. It is therefore safe to use.

An important lesson learnt from this work was that the modelling and analysis exercise did reveal that the first cut implementation of the protocol (not modelled in this paper) did suffer from two problems. Firstly, that the order of instructions was not adhered to (any order was allowed) and that the implementation could deadlock. This has vindicated the use of formal methods in this project. The use of CPNs has proved to provide useful insight into the protocol's behaviour. This was achieved through several iterations of the model before the desired result was obtained.

Using CPNs, this paper has also specified an improved protocol, that returns information about the success of an instruction to the client. This will allow users of the protocol to provide for error handling. The improved protocol preserved the same desired properties as the original protocol. The drawback to feedback about the instruction sent is that the amount of data sent on the network is almost doubled as information is sent in both directions for each instruction, whereas in the original protocol, information flow was primarily in one direction. This is not seen as a problem for the envisaged application environment of a local area network.

It is also clear that the analysis of this practical application was well within the limits of DESIGN/CPN's capabilities, as the state spaces remained very small. This has allowed much greater confidence to be gained in the design of this protocol and should encourage the use of these techniques in similar distributed systems projects.

Acknowledgments We would like to thank Guy Gallasch, for assistance in the original design of the visualisation protocol and for assistance with the layout of this paper. We would also like to thank members of the Air Operations Division of DSTO for their feedback on the Visualisation tool.

References

1. Animation by Mimic/CPN. `http://www.daimi.au.dk/designCPN/libs/mimic/`.
2. D. E. Comer. *Computer Networks and Internets.* Prentice-Hall International, Inc., 1997.

3. Mathew Elliot and Guy Gallasch. Final Year Project Report: Software Design Description. Technical report, University of South Australia, 2001.
4. Mathew Elliot and Guy Gallasch. Final Year Project Report: Software Project Management Plan. Technical report, University of South Australia, 2001.
5. G. Gallasch and L. M. Kristensen. The comms/CPN library. `http://www.daimi.au.dk/designCPN/libs/commscpn/`.
6. G. Gallash and L. M. Kristensen. Comms/CPN: A Communication Infrastructure for External Communication with Design/CPN. In *Proc. of 3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'01)*, pages 79–93. Department of Computer Science, University of Aarhus, 2001. DAIMI PB-554.
7. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volumes 1-3.* Monographs in Theoretical Computer Science. Springer-Verlag, 1997.
8. L. M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.
9. Kenneth C. Louden. *Compiler Construction: Principles and Practice.* PWS Publishing Company, 1997.
10. Design/CPN Message Sequence Charts library. `http://www.daimi.au.dk/designCPN/libs/mscharts/`.
11. Design/CPN Online. `http://www.daimi.au.dk/designCPN/`.

Annotating Coloured Petri Nets

Bo Lindstrøm and Lisa Wells

Department of Computer Science, University of Aarhus,
IT-Parken, Aabogade 34, DK-8200 Aarhus N, Denmark
{blind,wells}@daimi.au.dk

Abstract. Coloured Petri nets (CP-nets) can be used for several fundamentally different purposes like functional analysis, performance analysis, and visualisation. To be able to use the corresponding tool extensions and libraries it is sometimes necessary to include extra auxiliary information in the CP-net. An example of such auxiliary information is a counter which is associated with a token to be able to do performance analysis. Modifying colour sets and arc inscriptions in a CP-net to support a specific use may lead to creation of several slightly different CP-nets – only to support the different uses of the same basic CP-net. One solution to this problem is that the auxiliary information is not integrated into colour sets and arc inscriptions of a CP-net, but is kept separately. This makes it easy to disable this auxiliary information if a CP-net is to be used for another purpose. This paper proposes a method which makes it possible to associate auxiliary information, called annotations, with tokens without modifying the colour sets of the CP-net. Annotations are pieces of information that are not essential for determining the behaviour of the system being modelled, but are rather added to support a certain use of the CP-net. We define the semantics of annotations by describing a translation from a CP-net and the corresponding annotation layers to another CP-net where the annotations are an integrated part of the CP-net.

1 Introduction

Coloured Petri nets (CP-nets or CPNs) were formulated by Kurt Jensen [8, 9] with the primary purpose of specifying, designing, and analysing concurrent systems. The tools Design/CPN [3, 5] and CPN Tools [4] have been developed to give tool-support for creating and analysing CP-nets. Ongoing practical use of CP-nets and Design/CPN in industrial projects [10] have identified the need for additional facilities in the tools.

One industrial project described in [2] illustrated that CP-nets can be used for performance analysis by predicting the performance of a web server using a CPN model. As part of this project, the Design/CPN Performance Tool [12] was developed as an integrated tool extension supporting data collection during simulations. Later, work was done to extend and generalise these data collection facilities to serve as a basis for a common so-called monitoring framework [13]. Other projects have shown that visualisation of behaviour using so-called message sequence charts (MSCs) [7] is very useful in combination with CP-nets. As a consequence, a library [14] has been developed for creating MSCs during simulations. Other similar libraries are Mimic [15], which is used for visualisation, and Comms/CPN [6], which is used for communicating with external processes.

The fact that a CPN model can be used for several fundamentally different purposes like functional analysis, performance analysis, and visualisation means that it is desirable that the tool extensions and libraries can be used without having to modify the CPN model itself. It should be possible to use a CPN model, for e.g. performance analysis, without having to add extra places, transitions, and colour sets purely for the purpose of collecting data. Optimally, the auxiliary information should not be integrated into colour set and arc inscriptions of a CPN model, but should be kept separately, so that it is easy to disable this information if the CPN model is to be used for something else.

Up to this point it has only been partially possible to use a CPN model for different purposes without having to change the CPN model itself. With the current tools, it is indeed possible to do, e.g. performance analysis without adding transitions and places for the sole purpose of doing the performance analysis. Unfortunately however, it is often necessary to add extra information to colour sets and arc inscriptions to hold, e.g. performance-related information such as the time at which a certain event happened.

This paper presents work on separating auxiliary information from a CPN model by proposing a method which makes it possible to associate auxiliary information, called *annotations*, with tokens without modifying the colour sets of the CPN model. Annotations are pieces of information that are not essential for determining the behaviour of the system being modelled, but rather are added to support a certain use of the CPN model. A CP-net that is equipped with annotations is referred to as an *annotated CP-net*. In an annotated CP-net, every token carries a token colour, and some tokens carry both a token colour and an annotation. A token that carries both a colour and an annotation is called an *annotated token*. Just like a token value, an annotation may contain any type of information, and it may be arbitrarily complex.

Annotations are defined in *annotation layers*. Defining annotations in layers makes it possible to make modular definitions of both a CP-net and one or more layers of auxiliary information that can be used for varying purposes. By defining several different layers of annotations, it is possible to maintain several versions of a CP-net and thereby to use the same basic CP-net for various purposes by adding, removing, or combining annotation layers. An advantage of the annotation layers is that they are defined so that they affect the behaviour of the original CP-net in a very limited and predictable way. Every marking of an annotated CP-net is the same as a marking in the original CP-net, if annotations are removed.

In the following, we will assume that the reader is familiar with CP-nets as defined in [8]. The first half of this paper provides an informal introduction to annotations and an example of how annotations can be used in practice. The second half of the paper provides a formal definition of annotations and proof of the fact that annotations affect the behaviour of a CP-net in a very limited way. In this paper, we will only discuss how to annotate non-hierarchical, untimed CP-nets. However, timed and hierarchical CP-nets can also be annotated using similar techniques.

The paper is structured as follows. Section 2 presents the well-known resource allocation system CP-net, which will be used as a running example throughout the paper, and discusses existing ways of including auxiliary information in CP-nets. In Sect. 3 we informally introduce our proposal for how to annotate CP-nets. Section 4 discusses how multiple annotation layers can be used for visualisation using MSCs. In Sect. 5 we give the formal definitions for annotating CP-nets. Finally, in Sect. 6 we conclude and give directions for future work.

2 Motivation

It is seldom the case that the exact same CP-net can be used for a variety of different purposes, as it is frequently necessary to make small modifications to a CP-net in order to obtain a CP-net that is appropriate for a given purpose. Consider for example the resource allocation system that is found in Jensen's volumes on CP-nets [8, 9]. At least three variations of the resource allocation CP-net can be found in these volumes: a *basic* version (shown in Fig. 1) suitable for full state space analysis; an *extended* version (shown in Fig. 2) which is extended with cycle counters for the p and q processes; and a *timed* version with cycle counters and timing information which could be used for performance analysis.

The basic version in Fig. 1 purely models the basic aspects of the resource allocation system, and thereby only models the parts of the system that are common for any use of the CP-net. However, even for such a simple system as the resource allocation system, it is indeed necessary to have slightly different versions of the same CP-net in order to support different kinds of use. In other words, modifications of the basic CP-net are made only to support a certain use, and the modifications may limit other uses of the modified CP-net because the modifications may change the behaviour.

An example of a situation where the basic version does not contain sufficient information is when we need to be able to count how many cycles each of the p and q processes make in the resource allocation system. The extended CP-net in Fig. 2 shows how the basic CP-net can be extended with such auxiliary information. First of all, the colour set U has been extended to the product colour set UxI to include an integer for the counter in every process token. In addition, the arc inscriptions have been modified to pass on and to update the cycle counters. The cycle counters for the p and q processes are increased each time a p or q token passes the transition T5. The initial marking has also been modified to include the initial values of the cycle counters. Using this extended CP-net it is possible to determine the number of cycles a process has completed by inspecting the counter of the corresponding tokens.

The version extended with the cycle counters is not useful for all kinds of analysis. This is due to the fact that the cycle counters for the p and q processes increase each time the p and q tokens pass

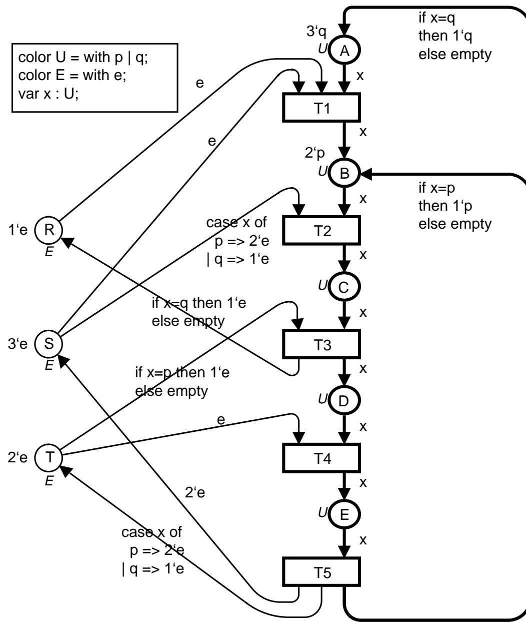


Fig. 1. The basic CP-net for the resource allocation system.

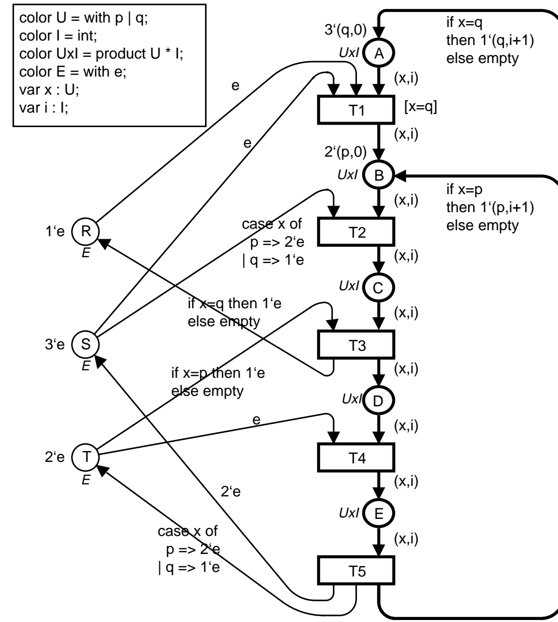


Fig. 2. The CP-net for the resource allocation system extended with a counter.

transition T5, thus resulting in an infinite state space. Therefore, the extended version with cycle counters may be inappropriate for certain kinds of state space analysis. The effect of the cycle counters on the state space can be factored out using equivalence classes, however, it may be annoying to have to remember to manually take care of such auxiliary information before doing state space analysis. In contrast, the state space for the basic CP-net without the cycle counters is finite. This means that the full state space can be generated and analysed, e.g. to prove that the system never reaches a deadlocked state.

Analysing the performance of the resource allocation system is another kind of analysis that requires auxiliary information to be maintained for the tokens in the CP-net. The timed CP-net from [9] could be used to measure the average processing times for each of the two processes. This timed CP-net can be created by modifying the CP-net in Fig. 2 by changing the colour set `UxI` to a timed colour set, and by adding an auxiliary component to the colour set to be used for recording the time when a process restarts a cycle,[1] i.e the time at which a `q` process is removed from place A or the time at which a `p` process is removed from place B. This value can then be used to calculate the processing time for a given process when it passes the T5 transition. If the timed CP-net should be used for a purpose where the auxiliary information should be ignored, it should often be removed. In the tool Design/CPN, it is easy to disable time, i.e. to consider a timed CP-net as an untimed CP-net. However, auxiliary components that have been added to the colour sets also need to be removed by manually modifying the colour sets and arc inscriptions.

From the examples presented above it should now be clear that when using CP-nets for different purposes it is often necessary to maintain different versions of a CP-net with slightly different behaviour. The reason for maintaining different versions is, as mentioned, that it may be necessary to be able to include auxiliary information in tokens. However, the auxiliary information may be extraneous or even disastrous for other uses, e.g. consider the effects of the cycle counters on the size of the state space. Including extra information in a CP-net often requires modification of colour sets, arc expressions, and initialisation expressions.

[1] The colour set `U` could be modified to consist of pairs (u,t) where u∈U is a process and t∈TIME is the time at which the process started processing.

3 Informal Introduction to Annotated CP-nets

In this section we will informally present a method for augmenting tokens in a CP-net with extra or auxiliary information that affects the behaviour of the CP-net in a very limited and predictable manner. To do this we introduce the concept of an *annotation* which is very similar to a token colour in that an annotation is an additional data value that can be attached to a token. An *annotation layer* is used to define annotations and how these annotations are to be associated with tokens in a particular CP-net. An annotation layer cannot be defined independently from a specific CP-net. Therefore, it is always well-defined to refer to the unique CP-net for which an annotation layer is defined. We will refer to this unique CP-net as the *underlying* CP-net of an annotation layer. An *annotated CP-net* is a pair consisting of an annotation layer and its underlying CP-net. We define the semantics of annotations by describing a translation from an annotated CP-net to a CP-net without annotations, referred to as the *matching CP-net*. In practice, the annotations are integrated into the matching CP-net when the translation is made. Section 3.1 gives an informal introduction to annotations and annotation layers. Section 3.2 describes the intuition of how to translate an annotated CP-net to a matching CP-net. Section 3.3 discusses the behaviour of the matching CP-net, and it discusses how the behaviour of the matching CP-net is similar to the behaviour of the underlying CP-net. The formal definition of annotated CP-nets follows in Sect. 5.

3.1 Annotation Layer

To get an intuitive understanding of how annotations can be used, let us see how the cycle counters that were discussed in Sect. 2 can be added as annotations. Recall that Fig. 2 shows how the CP-net from Fig. 1 can be modified to include the cycle counters as part of the token colours.

Figure 3 contains an annotated CP-net for the basic CP-net for the resource allocation system from Fig. 1. In Fig. 3 the elements from the annotation layer are shown in black, whereas the underlying CP-net is shown in grey. The annotation layer contains *auxiliary declarations* and *auxiliary net inscriptions*, where the auxiliary net inscriptions consist of auxiliary arc expressions, auxiliary colour sets, and auxiliary initialisation expressions.

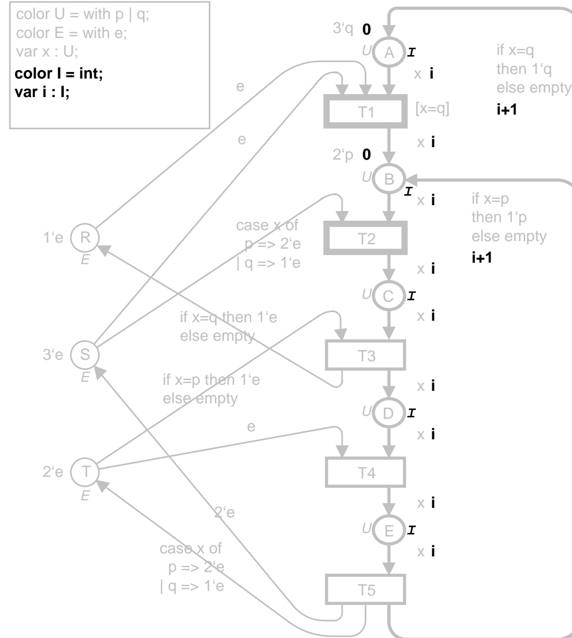


Fig. 3. Annotated CP-net for the resource allocation system.

The colour set I is declared in the first line of the auxiliary declarations. Places A, B, C, D and E have auxiliary colour set I which means that tokens on these places will be annotated tokens that carry integer annotations. A token that carries the annotation n has completed the cycle n times. Places with auxiliary colour sets are called *annotated places*. The token value for a token on an annotated place has both a token colour and an annotation. Not all places will contain annotated tokens, therefore, some places will not have an associated auxiliary colour set.

All of the annotated places that have an initialisation expression in the underlying CP-net must also have an *auxiliary initialisation expression* in the annotation layer. Places A and B have the auxiliary initial expression 0. This expression means that all tokens on places A and B will have annotation 0 in the initial marking.

All arcs that are connected to annotated places have an *auxiliary arc expression*. In Fig. 3, most auxiliary arc expressions consist of the variable i which has type I. Variable i is declared in the annotation layer, therefore, it may only be used within the annotation layer, i.e. it cannot be used in the underlying CP-net. In contrast, variables, colour sets, functions, etc. that are declared in the underlying CP-net may be used both in the underlying CP-net and in the annotation layer. However, certain conditions must be fulfilled in order to ensure that using the same elements in both the annotation layer and the underlying CP-net does not affect the behaviour of the underlying CP-net. These conditions will be discussed further in Sect. 5.2.

Let us consider the intuition behind the auxiliary arc expressions on the arcs surrounding transition T5. In the underlying CP-net, T5 can occur whenever there is one token on place E, and this must still be true in an annotated version of the CP-net. Informally, the interpretation of the two types of arc expressions surrounding T5 is that when transition T5 occurs with, e.g. binding <x=q, i=5>, one token with colour q and annotation 5 will be removed from place E. One token with colour q and annotation 5+1=6 will be added to place A, and the empty multi-set of annotated tokens will be added to place B. On the other hand, if T5 occurs with binding <x=p, i=3>, then one token with colour p and annotation 3+1=4 will be added to place B, and no tokens will be added to place A. In both bindings, multi-sets of (non-annotated) e tokens are also added to places T and S, which are non-annotated places.

The intuition behind the auxiliary inscriptions that have been discussed until now is fairly straightforward. There are, however, some restrictions on the kinds of auxiliary arc expressions that are allowed in order to ensure that annotations have only limited influence on the behaviour of the underlying CP-net. All of the auxiliary arc expressions on arcs from annotated places to transitions consist only of variables, and this is not accidental. For example, the auxiliary arc expression i on the arc from C to T3 must not be replaced with, e.g. the constant 4, which would require that when removing a token from C the annotation must be 4. Allowing such an auxiliary arc expression would mean that the behaviour of the matching CP-net and the underlying CP-net would no longer be similar. Sections 5.2 and 5.3 discuss the restrictions about which kinds of auxiliary arc expressions are allowed.

3.2 Translating an Annotated CP-net to a Matching CP-net

Rather than defining the semantics for annotated CP-nets, we will define the semantics of annotations by describing how an annotated CP-net can be translated to an ordinary CP-net, which is referred to as the *matching* CP-net. The discussion above should have provided a sense of what kinds of annotations the tokens should have and of how an annotated CP-net for the resource allocation system should behave. The annotation layer (referred to as $\mathcal{A}$ and shown in black in Fig. 3) and the underlying CP-net (referred to as CPN and shown in Fig. 1) constitute an annotated CP-net for the resource allocation system. In this section, we will show how the various auxiliary inscriptions from $\mathcal{A}$ and the inscriptions from CPN are translated to inscriptions in the matching CP-net (referred to as CPN* and shown in Fig. 4). The general rules for translating an arbitrary annotation layer and its underlying CP-net are presented in Sect. 5.3. In the following, we shall say that a place/arc is annotated/non-annotated in a matching CP-net (like Fig. 4) if it is annotated/non-annotated in the corresponding annotated CP-net (like Fig. 3).

The rules are simple for translating an annotated CP-net to a matching CP-net. CPN and CPN* have the same net structure. The colour sets for non-annotated places in CPN* are unchanged with respect to CPN; in the example, the colour sets for places R, S and T are unchanged. The colour sets for the annotated places are now product colour sets which are products of the original colour sets and the auxiliary colour sets. The colour set for annotated places A-E in CPN* is UxI which is a product of colour set U (the colour

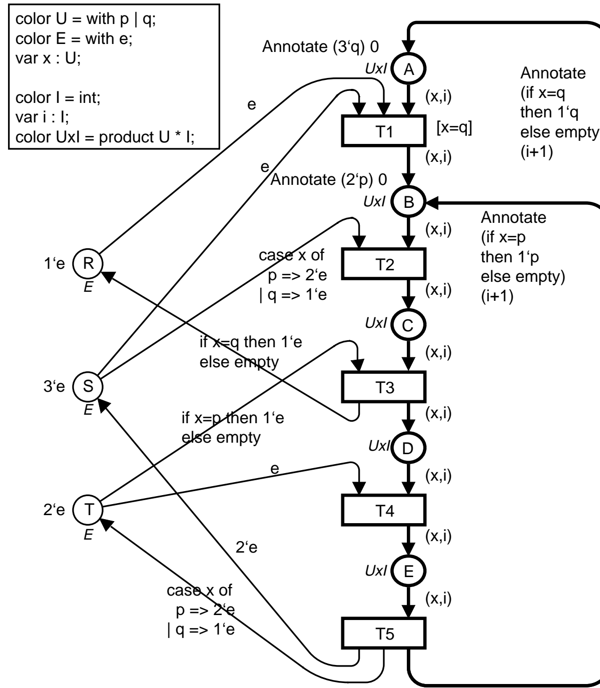


Fig. 4. Matching CP-net for the resource allocation system.

set of places A-E in CPN) and auxiliary colour set I (the auxiliary colour sets from $\mathcal{A}$). The tokens on an annotated place p in CPN* are said to have an annotated colour (c,a), where c is a colour (from the colour set of p in CPN), and a is an annotation (from the auxiliary colour set of p in $\mathcal{A}$). The set of colour sets for CPN* is the union of the set of colour sets from CPN, the set of auxiliary colour sets from $\mathcal{A}$, and the set of product colour sets for annotated places.

In the previous section, the intuitive meaning of several auxiliary expressions was that a given annotation should be added to all elements in a multi-set of colours. This is the meaning of, for example, all of the auxiliary initial expressions. Let us define a function Annotate that, given an arbitrary multi-set and an arbitrary annotation, will annotate all of the elements in the multi-set with the annotation. This function is used in several net inscriptions in CPN*.

Let us consider how the initialisation expressions are created for CPN*. The initialisation expressions for non-annotated places (R, S and T) are unchanged, and evaluating these expressions yields non-annotated multi-sets (1`e, 3`e, and 2`e respectively). The initial markings for annotated places in CPN* must be multi-sets of annotated colours. Place A has the initialisation expression Annotate (3`q) 0, which evaluates to 3`(q,0) tokens which correspond to the desired multi-set of three annotated tokens, each with colour q and annotation 0. Note, in particular, that if the annotations are removed from the multi-set 3`(q,0), then we obtain the multi-set 3`q which is exactly the multi-set that is obtained when evaluating the initialisation expression for A in the underlying CP-net. Similarly, place B has an initial marking of two tokens, each with colour p and annotation 0. The initial markings of the remaining places are empty.

The arc expressions of CPN and the auxiliary arc expressions of $\mathcal{A}$ are combined in a similar manner to create arc expressions for CPN*. If the type of an arc expression in CPN, expr, is a single colour, then the arc expression in CPN* is the pair (expr, aexpr), where aexpr is the auxiliary arc expression in $\mathcal{A}$. The arc expressions for most annotated arcs in Fig. 4 have this form. When the type of an arc expression is a multi-set of colours, then the arc expression for CPN* is Annotate expr aexpr. The arc expressions for the arcs from transition T5 to places A and B were created in this manner. The next section discusses how the behaviour of the matching CP-net is similar to the behaviour of the underlying CP-net.

3.3 Behaviour of Matching CP-nets

In a matching CP-net some places contain annotated tokens, other places contain non-annotated tokens, and occurrences of binding elements can remove and add both regular, non-annotated tokens and annotated tokens. Figure 5 shows a marking of the matching CP-net, CPN*. The marking of place A contains two tokens – one with colour (q,4), the other with colour (q,5). This corresponds to a marking in the annotated CP-net of Fig. 3 where A has one token with colour q and annotation 4 and another token with colour q and annotation 5. Similarly, place B contains three tokens with colours (p,5), (p,6) and (q,1). This corresponds to a marking in the annotated net where B has one token with colour p and annotation 5, another token with colour p and annotation 6, and a third token with colour q and annotation 1. Finally, places S and T each contain two non-annotated tokens with colour e.

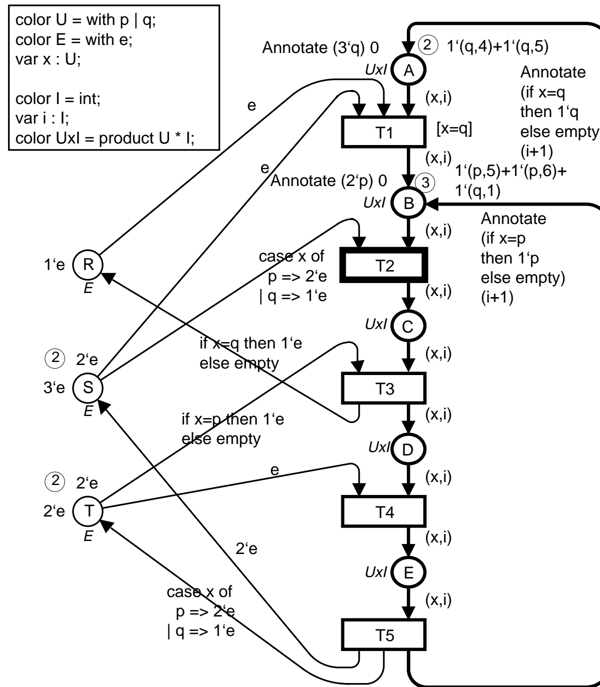


Fig. 5. Marking of the matching CP-net for the resource allocation system.

We say that the behaviour of the matching CP-net *matches* the behaviour of its underlying CP-net. Informally this means that every occurrence sequence in the matching CP-net is also an occurrence sequence in the underlying CP-net if annotations are ignored. Furthermore, for every occurrence sequence in the underlying CP-net, it is possible to find at least one matching occurrence sequence in the matching CP-net which is identical to the occurrence sequence from the underlying CP-net when annotations are ignored. Consider, for example, a marking, M, of the basic CP-net from Fig. 1, in which there are two e tokens on places S and T, two q tokens on place A, and two p tokens and one q token on place B. The binding element (T2, $\langle$x=p$\rangle$) is enabled in M. The marking of the matching CP-net in Fig. 5 is the same as marking M if annotations are ignored. The occurrence of either (T2, $\langle$x=p, i=5$\rangle$) or (T2, $\langle$x=p, i=6$\rangle$) in the matching CP-net will result in markings that are equal M′ where M[(T2, $\langle$x=p$\rangle$)$\rangle$M′ when annotations are ignored. A formal definition of matching behaviour can be found in Sect. 5.4.

4 Using Annotation Layers in Practice

This section discusses how to use several annotation layers for the basic CP-net of the resource allocation system presented in Fig. 1 in Sect. 2. The purpose of this section is to illustrate that multiple annotation layers can be added on top of each other without changing the original model, and to illustrate some of the uses of annotations. We will discuss an example of how annotations can be used for visualising simulation results. In particular we will consider how message sequence charts (MSCs) can be created using annotations.

A MSC can be used, e.g. to visualise the use of resources. Figure 6 depicts a MSC for the basic CP-net of the resource allocation system. The MSC contains two vertical lines which represent the activities of allocating and deallocating resources in the resource allocation system. An arrow represents the dependency between the allocation and deallocation of an S or an R resource.

The MSC in Fig. 6 visualises a sequence of allocations of resources by the p and q processes. The arrows for p processes are dashed. The MSC shows that first the q process makes a full cycle where it first allocates an R and an S resource when T1 occurs, and an additional S resource when T2 occurs. The R resource is deallocated when T3 occurs, and the two S resources are deallocated when T5 occurs. The last five arrows show a situation where two p processes interleave with a q process. First the q process allocates an R and an S resource, but then two cycles of p processes appear (the two dashed arrows) before the q process continues the cycle. This interleaving is explicitly visualised by the arrows started by T1 and ended by T3 and T5, and crossing the arrows representing the two p processes.

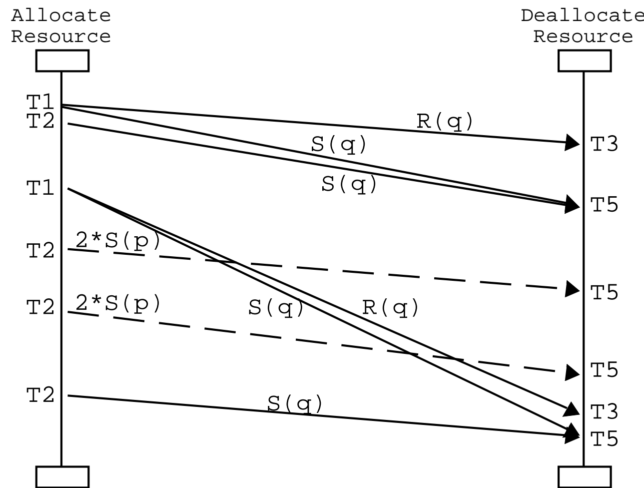


Fig. 6. Message sequence chart for the resource allocation system.

A MSC can be generated automatically from a simulation of a CP-net. However, first it is necessary to specify which occurrences of binding elements in the CP-net should generate which arrows in the MSC. Normally, an arrow in a MSC is created when a single transition occurs. However, arrows as illustrated above correspond to two events: one for creating the start-point and one for creating the end-point of an arrow. Such arrows are defined by means of these two points. First the start-point of the arrow is given. Then some other events may appear, and then the event leading to ending the arrow is given. For CP-nets this means that the occurrence of one transition may define the start-point of an arrow while the occurrence of another transition may define the end-point of an arrow. We call such arrows *two-event arrows*.

When using two-event arrows, it is often necessary to annotate a token to hold information of which arrows have been started but not ended yet. In other words, a token must hold the arrow-id of the start-point of the arrow when the first transition occurs and keep it until a transition supposed to end the arrow consumes the token. To avoid modifying the colours of the the CP-net, annotations can be used. Figure 7

depicts how the basic CP-net for the resource allocation system can be annotated to generate the MSC in Fig. 6. The contents of the annotation layer are shown in black, while the underlying CP-net is shown in grey. The annotation colour `MSC` is an integer which has the purpose of holding the start-point id of an arrow, while the annotation colour `MSCs` is a list of MSC ids. We do not give the details of the functions `msc_start` and `msc_stop` here, however, they are used to set the start-point and end-point of each arrow. In addition each of the functions return a list of arrow-ids of the non-stopped arrows. This list becomes an annotation for the underlying colour. As this example illustrates, we allow auxiliary arc expressions to have side effects. However, the side effects may not affect the behaviour of the underlying CP-net.

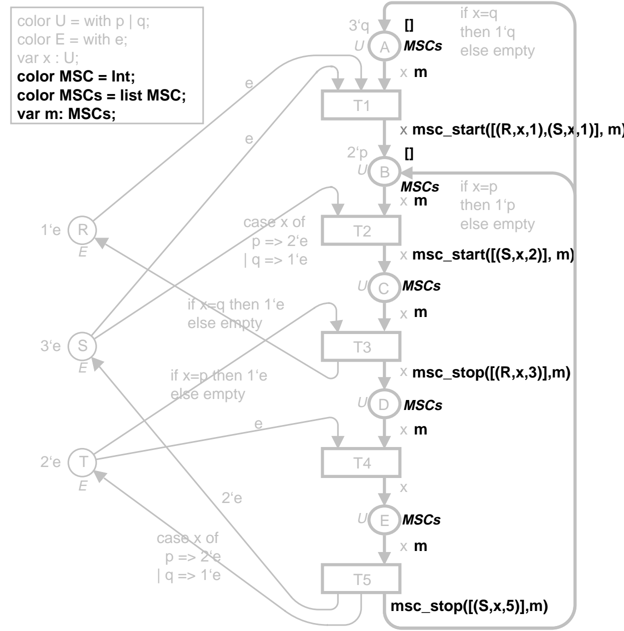


Fig. 7. Resource allocation system with MSC annotation layer.

The annotation layer in Fig. 3 from Sect. 3.1 adds a cycle counter to the CP-net. The value of the cycle counter could be included on the arrows in Fig. 6 in addition to the type of resource and process. To obtain this, an annotation layer that resembles the MSC Annotation Layer in Fig. 7 can be added on top of the cycle counter annotation layer in Fig. 3. We will refer to this new annotation layer as Cycle MSC Annotation Layer. In the Cycle MSC Annotation Layer it is possible to refer to the annotations of the Cycle Counter Annotation Layer from the MSC Annotation Layer.

Figure 8 depicts some of the possible ways to add annotation layers on top of each other. Notice that an alternative to adding the Cycle MSC Annotation Layer on top of the Cycle Counter Annotation Layer, is to add the original MSC Annotation Layer on top of the Cycle Counter Annotation Layer. This makes sense even though the annotations in Cycle Counter Annotation Layer are not used in the MSC Annotation Layer.

If we had not been able to use an annotation layer for creating the MSC, we would have had to create a new CP-net by adding and modifying the colours of the basic CP-net. For example, the colour sets of the places A, B,C, D, and E should also hold the `MSCs` colour set. In addition, so-called code-segments possibly had to be added to execute the the `msc_start` and `msc_stop` function calls, and the arc-expressions had to be modified to include the MSC variable `m`. In other words, we had to modify the CP-net model itself to generate the MSCs. If the information for updating MSCs is included directly in the CP-net, then it would be difficult to disable the updating of the MSCs, and there is no guarantee that the modifications would not affect the behaviour of the underlying CP-net in unexpected ways.

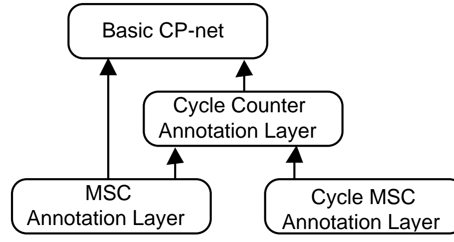


Fig. 8. Structure of annotation layers for a basic CP-net.

5 Formal Definition of Annotated CP-nets

In this section, we will formally define annotated CP-nets. We will start by introducing some new terminology. We will then define annotation layers, and we will discuss how an annotation layer and a CP-net can be translated into a matching CP-net. We want to define the annotation rules so that they are straightforward to use and understand. To achieve this it turns out to be convenient only to allow annotation of those input arcs of transitions where the arc expressions are uniform with multiplicity one (i.e. always evaluate to a single token colour). For output arcs there are no similar restrictions. If an input arc expression is uniform with multiplicity larger than one, it is usually easy to split the arc into a number of arcs that each have multiplicity one. The requirement can be formally expressed as:

Requirement 1. Let CPN=(Σ, P, T, A, N, C, G, E, I) be a CP-net as defined in Def. 2.5 in [8]. Let $P_{\mathcal{A}} \subseteq P$ be the set of *annotated places*. The following must hold in order to be able to annotate CPN:

$\forall\ p \in P_{\mathcal{A}}: \forall\ a \in A$ such that N(a)=(p,t), E(a) must be uniform with multiplicity 1.

This requirement may seem very restrictive. However, in our experience, the kinds of arc inscriptions that are currently not possible to annotate are rarely used in practice. Therefore, the definitions presented here should prove to be useful for annotating many of the CP-nets that are used in practice.

Section 5.1 introduces terminology regarding multi-sets of annotated colours. Section 5.2 defines annotation layers by describing the auxiliary net inscriptions that are allowed in the annotation layer. Section 5.3 presents rules for translating a CP-net and an annotation layer into the matching CP-net, and it discusses the relationship between markings, binding elements, and steps in a matching CP-net and its underlying CP-net. Section 5.4 defines matching behaviour. Finally, Sect. 5.5 discusses the use of multiple annotation layers.

5.1 Multi-sets of Annotated Colours

In the previous section we used expressions such as: `` 1`(p,5)+1`(p,6)+1`(q,1) `` to denote the marking of annotated places[2] in a matching CP-net. This indicates that the marking consists of three tokens with colours `(p,5)`, `(p,6)` and `(q,1)`. However, within the context of annotated CP-nets, this marking can also be interpreted to represent a multi-set of annotated tokens: two tokens with colour `p` and annotations `5` and `6`, and one token with colour `q` and annotation `1`. Multi-sets of annotated elements are ordinary multi-sets[3] of so-called *annotated elements*.

[2] The first paragraph in Sect. 3.2 explains what we mean when we refer to an annotated place in a matching CP-net.
[3] Multi-sets as defined in Def. 2.1 in [8]

Definition 1. For a non-empty set of elements, S, and a non-empty set of annotations, AN, an *annotated element* (from S) is a pair (s,a), where s∈S and a∈AN. π((s,a))=s is the projection of the annotated element (s,a) onto the non-annotated element s.

A *multi-set of annotated elements* over S×AN is a multi-set over S×AN.

If am is a multi-set of annotated elements (of S), then am *determines* an ordinary (non-annotated) multi-set am_π over S, where $\text{am}_\pi(\text{s}) = (\sum_{a\in AN} \text{am}(s,a))\grave{}\, s$. $\pi(\text{am})$ is the *projection* of am onto the non-annotated multi-set determined by am.

If am is multi-set over S×AN, m is a multi-set over S, and π(am)=m, then am is said to *cover* m, and we say that m is *covered* by am.

In Sect. 3.2 we informally defined the function `Annotate` that will add a given annotation to all elements in a given multi-set. Let us now formally define `Annotate`.

Definition 2. Given an annotation a∈AN and a multi-set $\text{m}=\sum_{s\in S} \text{m}(s)\grave{}\, s$ over a set S, the function `Annotate` is defined to be:

$$\texttt{Annotate}\,\texttt{m}\,\texttt{a} = \sum_{s\in S} \text{m}(s)\grave{}\,(s,\text{a})$$

which is a multi-set of annotated elements of S, i.e. a multi-set with type $(\text{S}\times\text{AN})_{MS}$.

As a consequence of Defs. 1 and 2, $\pi(\texttt{Annotate}\,\texttt{m}\,\texttt{a}) = \texttt{m}$, for all multi-sets m and all annotations a.

5.2 Annotation Layer

We are now ready to define an annotation layer. An annotation layer is used solely to determine how to add annotations to tokens for a subset of the places in a CP-net. An annotation layer consists of elements that are similar to their counterparts in CP-nets. An annotation layer contains auxiliary net inscriptions, and each auxiliary net inscription is associated with an element of the net structure of the underlying CP-net. When translating an annotation layer and its underlying CP-net to the matching CP-net, these auxiliary net expressions will be combined with their counterparts from the underlying CP-net to create colour sets, initialisation expressions and arc expressions for the matching CP-net. There are, however, additional requirements for each of the concepts. An explanation of each item in the definition is given immediately below the definition. A similar remark applies for many of the other definitions in this paper.

Definition 3. Let CPN=(Σ, P, T, A, N, C, G, E, I) be a CP-net. An *annotation layer* for CPN is a tuple $\mathcal{A}=(\Sigma_\mathcal{A}, P_\mathcal{A}, A_\mathcal{A}, C_\mathcal{A}, E_\mathcal{A}, I_\mathcal{A})$ where

i. $\Sigma_\mathcal{A}$ is a finite set of non-empty sets, called *auxiliary colour sets*, where $\Sigma\subseteq\Sigma_\mathcal{A}$.
ii. $P_\mathcal{A}\subseteq P$ is a finite set of *annotated places*.
iii. $A_\mathcal{A}\subseteq A$ is the finite set of *annotated arcs*, where $A_\mathcal{A} = A(P_\mathcal{A})$.
iv. $C_\mathcal{A}$ is an *auxiliary colour* function. It is a function from $P_\mathcal{A}$ into $\Sigma_\mathcal{A}$.
v. $E_\mathcal{A}$ is an *auxiliary arc expression* function. It is defined from $A_\mathcal{A}$ into expressions such that:
$\forall a\in A_\mathcal{A}$: $\text{Type}(E_\mathcal{A}(a))=C_\mathcal{A}(p(a)) \wedge \text{Type}(\text{Var}(E_\mathcal{A}(a)))\subseteq\Sigma_\mathcal{A}$
vi. $I_\mathcal{A}$ is an *auxiliary initialisation* function. It is a function from $P_\mathcal{A}$ into closed expressions such that:
$\forall p\in P_\mathcal{A}$: $\text{Type}(I_\mathcal{A}(p))=C_\mathcal{A}(p)$.

i. The set of *auxiliary colour sets* is the set of colour sets that determine the types, operations and functions that can be used in the auxiliary net inscriptions. The auxiliary colour sets determine the type of annotations that the tokens on the annotated places carry. All colour sets from the underlying CP-net can be used as auxiliary colour sets. Additional auxiliary colour sets may be declared within an annotation layer.

ii. The set of annotated places are the only places that are allowed to contain annotated tokens.

iii. The annotated arcs are exactly the surrounding arcs for the places in $P_\mathcal{A}$.

iv. The *auxiliary colour function*, $C_{\mathcal{A}}$, is a function from $P_{\mathcal{A}}$ into $\Sigma_{\mathcal{A}}$, and is defined analogously to the colour function for CP-nets. Thus, for all $p \in P_{\mathcal{A}}$, $C_{\mathcal{A}}(p)$ is the auxiliary colour set of p.

v. Auxiliary arc expressions are only allowed to evaluate to a single annotation of the correct type. If the arc expression of an arc is missing in CPN, then we require that its auxiliary arc expression is also missing in $\mathcal{A}$.

vi. The auxiliary initialisation function maps each annotated place, p, into a closed expression which must be of type $C_{\mathcal{A}}(p)$, i.e. a single annotation from $C_{\mathcal{A}}(p)$. If the initial expression of place p is missing in CPN, then we require that its auxiliary initial expression is also missing in $\mathcal{A}$.

5.3 Translating Annotated CP-nets to Matching CP-nets

We will now define how to translate an annotated CP-net, (CPN, $\mathcal{A}$), to a new CP-net, CPN^*, which is called a matching CP-net. CPN^* and CPN have the same net structure. Net inscriptions for non-annotated places, non-annotated arcs, and transitions in CPN^* are unchanged with respect to CPN. In contrast, net inscriptions for annotated places and annotated arcs in CPN^* are obtained by combining net inscriptions from CPN with their counterpart auxiliary net inscriptions in $\mathcal{A}$. A matching CP-net is defined below.

Definition 4. Let (CPN, $\mathcal{A}$) be an annotated CP-net, where CPN=(Σ, P, T, A, N, C, G, E, I) and $\mathcal{A}$=($\Sigma_{\mathcal{A}}$, $P_{\mathcal{A}}$, $A_{\mathcal{A}}$, $C_{\mathcal{A}}$, $E_{\mathcal{A}}$, $I_{\mathcal{A}}$) is a annotation layer. We define the *matching CP-net* to be CPN^*=(Σ^*, P^*, T^*, A^*, N^*, C^*, G^*, E^*, I^*) where

i. $\Sigma^* = \Sigma_{\mathcal{A}} \cup \{C(p) \times C_{\mathcal{A}}(p) \mid p \in P_{\mathcal{A}}\}$.
ii. $P^* = P$
iii. $T^* = T$
iv. $A^* = A$
v. $N^* = N$
vi. $C^*(p) = \begin{cases} C(p) & \text{if } p \notin P_{\mathcal{A}} \\ C(p) \times C_{\mathcal{A}}(p) & \text{if } p \in P_{\mathcal{A}} \end{cases}$
vii. $G^* = G$
viii. $E^*(a) = \begin{cases} E(a) & \text{if } a \notin A_{\mathcal{A}} \\ \text{Annotate } E(a)\ E_{\mathcal{A}}(a) & \text{if } a \in A_{\mathcal{A}} \end{cases}$
ix. $I^*(p) = \begin{cases} I(p) & \text{if } p \notin P_{\mathcal{A}} \\ \text{Annotate } I(p)\ I_{\mathcal{A}}(p) & \text{if } p \in P_{\mathcal{A}} \end{cases}$

i. $\{C(p) \times C_{\mathcal{A}}(p) \mid p \in P_{\mathcal{A}}\}$ is the set of product colour sets for the annotated places in CPN^*.

ii. + iii. + iv. + v. The places, transitions, arcs, and node function in CPN^* are unchanged with respect to CPN.

vi. Defining the colour function C^* is straightforward. The colour set for a non-annotated place in CPN^* is the same as its colour set in CPN. The colour set for an annotated place p in CPN^* is $C(p) \times C_{\mathcal{A}}(p)$.

vii. The guard function in CPN^* is unchanged with respect to CPN.

viii. The arc expression for a non-annotated arc a in CPN^* is the same as the arc expression for a in CPN. If the arc expression for a is missing in CPN, then its arc expression will also be missing in CPN^*. This is shorthand for empty, as usual for CP-nets. The arc expression for an annotated arc a in CPN^* is derived from the arc expression for a in CPN and the auxiliary arc expression for a in $\mathcal{A}$. The expression `Annotate (E(a)) (`$E_{\mathcal{A}}$`(a))` will yield a multi-set with type $(C(p(a)) \times Type(E_{\mathcal{A}}(p(a))))_{MS}$ which is exactly $(C(p(a)) \times C_{\mathcal{A}}(p(a)))_{MS}$, as required. If an arc expression (for an annotated arc) evaluates to a single colour in CPN, then we allow the arc expression for CPN^* to be the pair `(E(a),` $E_{\mathcal{A}}$`(a))` where the first element is the arc expression from CPN, and the second element is the auxiliary arc expression from $\mathcal{A}$. This is shorthand for the multi-set ``1`(E(a),`` $E_{\mathcal{A}}$`(a))`.

ix. If p is not an annotated place, then the initial expression of p in CPN* is unchanged with respect to CPN. If the initial expression of a place is missing in CPN, then its initial expression will also be missing in CPN*. A missing initial expression is shorthand for the empty. For an annotated place p, the expression `Annotate (I(a)) (`$I_{\mathcal{A}}$`(a))` will yield a multi-set with type $(C(p(a))\times Type(I_{\mathcal{A}}(p(a))))_{MS}$ which is exactly $(C(p(a))\times C_{\mathcal{A}}(p(a)))_{MS}$, as required. $I^*(p)$ is a closed expression for all p, since I(p) is a closed expression for all p, and $I_{\mathcal{A}}(p)$ is closed for all $p\in P_{\mathcal{A}}$. When the type of the initial expression for an annotated place in the underlying CP-net is a single colour, then we allow the the initial expression in CPN* to be the pair `(I(p),`$I_{\mathcal{A}}$`(p))` that is uniquely determined by the initial expression of p in CPN and the auxiliary initial expression of p in $\mathcal{A}$. This is shorthand for `` 1`(I(p), ``$I_{\mathcal{A}}$`(p))`.

Covering Markings, Bindings and Steps We will now define what it means for markings, bindings and steps of a matching CP-net to cover the markings, bindings and steps of its underlying CP-net.

Definition 5. Let (CPN, $\mathcal{A}$) be an annotated CP-net with matching CP-net CPN*. We then define three projection functions π that map a marking M* of CPN* into a marking M of CPN, a binding b* of a transition t in CPN* into a binding b of t in CPN, and a step Y* of CPN* into a step Y of CPN, respectively.

i. $\forall p\in P^*: (\pi(M^*))(p) = \begin{cases} M^*(p) & \text{if } p\notin P_{\mathcal{A}} \\ \pi(M^*(p)) & \text{if } p\in P_{\mathcal{A}} \end{cases}$

ii. $\forall\, v\in Var(t): (\pi(b^*))(v)=b^*(v)$, where Var(t) are the variables of t in CPN.

iii. $(\pi(Y^*)) = \sum_{(t,b^*)\in Y^*} (Y^*(t,b^*))`(t,\pi(b^*))$

If $\pi(M^*)=M$, $\pi(b^*)=b$, and $\pi(Y^*)=Y$, then we say that M*, b*, and Y* *cover* M, b, and Y, respectively. We also say that M, b, and Y are *covered* by M*, b*, and Y*, respectively.

i. Given a marking of a matching CP-net, π will remove the annotations from the tokens on annotated places, and it will leave the markings of non-annotated places unchanged. A marking of a matching CP-net covers a marking of its underlying CP-net, if the two markings are equal when annotations in the first marking are ignored.

ii. Given a binding of a transition in CPN*, π removes the bindings of the variables in $Var^*(t)\setminus Var(t)$, i.e. π removes the bindings of the variables of t that are not found in CPN. A binding of a transition in CPN* covers a binding of the corresponding transition in CPN when the variables that are found in both CP-nets are bound to the same value.

iii. For each binding element (t,b^*) in Y*, π removes the bindings of the variables of t that are not found in CPN.

We define similar functions that map the set of markings ($\mathbb{M}^*$), the set of steps ($\mathbb{Y}^*$), the set of token elements (TE*), and the set of binding elements (BE*) of CPN* into the corresponding sets in CPN:

$\pi(\mathbb{M}^*)=\{\pi(M^*): M^*\in\mathbb{M}^*\}$
$\pi(TE^*)=\{(p, c^*) \mid p\notin P_{\mathcal{A}} \text{ and } c^*\in C^*(p)\} \cup \{(p, \pi(c^*)) \mid p\in P_{\mathcal{A}} \text{ and } c^*\in C^*(p)\}$
$\pi(\mathbb{Y}^*)=\{\pi(Y^*): Y^*\in\mathbb{Y}^*\}$
$\pi(BE^*) = \{(t,\pi(b^*)) \mid (t,b^*)\in BE^*\}$

Sound Annotation Layers Definition 3 defines the syntax for elements in annotation layers, but it does not guarantee that annotations do not affect the behaviour of the underlying CP-net. Instead of specifying which kinds of auxiliary arc inscriptions are allowed, we will define a more general property that has to be satisfied.

Definition 6. Let (CPN, $\mathcal{A}$) be an annotated CP-net with matching CP-net CPN*. $\mathcal{A}$ is a *sound* annotation layer if the following property is satisfied:

$\forall M \in \mathbb{M}, \forall Y \in \mathbb{Y}, \forall M^* \in \mathbb{M}^*: M[Y\rangle \wedge \pi(M^*)=M \Rightarrow \exists Y^* \in \mathbb{Y}^*: \pi(Y^*)=Y \wedge M^*[Y^*\rangle$

where $\mathbb{M}$ and $\mathbb{Y}$ are the set of markings and the set of steps, respectively, for CPN, and $\mathbb{M}^*$ and $\mathbb{Y}^*$ are the analogous sets for CPN*.

Assume that the step Y is enabled in the marking M in the underlying CP-net. Let M^* be a marking of CPN* that covers M. Definition 6 states that it must be possible to find a step Y^* that covers Y, and Y^* must be enabled in M^*. The soundness of an annotation layer is essential for showing that for every occurrence sequence in the underlying CP-net, there is at least one matching occurrence sequence in the matching CP-net which is identical to the occurrence sequence from the underlying CP-net when annotations are ignored.

The auxiliary arc expressions on input arcs to a transition will be used, in part, to determine if the transition is enabled in a given state of the matching CP-net. By limiting the kinds of auxiliary arc expressions that are allowed on input arcs to transitions, it is possible to guarantee that annotations cannot restrict the enabling of a transition in the matching CP-net with respect to what is allowed in the underlying CP-net. Exactly which kinds of auxiliary arc expressions should be allowed may be decided by the implementors of tools supporting CP-nets. It is also the responsibility of tool implementors to prove that their allowable set of auxiliary arc expressions fulfil Def. 6. An example of an allowable auxiliary arc expression for arc a is a single variable v. However, v must also fulfil the following: v may not found in any arc expressions for the arcs surrounding t(a), and v may not be found in any other auxiliary arc expression for input arcs to t(a).

5.4 Matching Behaviour

In the previous sections we have stated that the behaviour of a matching CP-net *matches* the behaviour of its underlying CP-net. Informally this means that every occurrence sequence in a matching CP-net corresponds to an occurrence sequence in the underlying CP-net, and for every occurrence sequence in the underlying CP-net, it is possible to find at least one corresponding occurrence sequence in the matching CP-net. If a matching CP-net is derived from a CP-net and a *sound* annotation layer, then the following theorem shows how the behaviour of the matching CP-net matches the behaviour of its underlying CP-net.

Theorem 1. *Let (CPN, $\mathcal{A}$) be an annotated CP-net with a sound annotation layer. Let CPN* be the matching CP-net derived from (CPN, $\mathcal{A}$). Let M_0, $\mathbb{M}$, and $\mathbb{Y}$ denote the initial marking, the set of all markings, and the set of all steps, respectively, for CPN. Similarly, let M_0^*, $\mathbb{M}^*$, and $\mathbb{Y}^*$ denote the same concepts for CPN*. Then we have the following properties:*

i. $\pi(\mathbb{M}^*)=\mathbb{M} \wedge \pi(M_0^*)=M_0$.
ii. $\pi(\mathbb{Y}^*)=\mathbb{Y}$.
iii. $\forall M_1^*, M_2^* \in \mathbb{M}^*, \forall Y^* \in \mathbb{Y}^*: M_1^*[Y^*\rangle M_2^* \Rightarrow \pi(M_1^*)[\pi(Y^*)\rangle \pi(M_2^*)$
iv. $\forall M_1, M_2 \in \mathbb{M}, \forall Y \in \mathbb{Y}, \forall M_1^*, M_2^* \in \mathbb{M}^*:$
$M_1[Y\rangle M_2 \wedge \pi(M_1^*)=M_1 \Rightarrow \exists Y^* \in \mathbb{Y}^*: \pi(Y^*)=Y \wedge M_1^*[Y^*\rangle M_2^* \wedge \pi(M_2^*)=M_2$

i. The markings of a matching CP-net cover the markings of its underlying CP-net. The markings of the underlying CP-net are covered by the markings of the matching CP-net. The initial marking of a matching CP-net covers the initial marking of its underlying CP-net.

ii. The steps of a matching CP-net cover the steps of its underlying CP-net. The steps of the underlying CP-net are covered by the steps of the matching CP-net.

iii. An occurrence sequence of length one in the matching CP-net covers an occurrence sequence of length one in its underlying CP-net. In other words, if marking M_2^* is reached by the occurrence of Y^* in marking

M_1^* in CPN*, then $\pi(M_2^*)$ will be reached by the occurrence of $\pi(Y^*)$ in $\pi(M_1^*)$ in CPN.

iv. An occurrence sequence of length one in the underlying CP-net can be covered by an occurrence sequence of length one in the matching CP-net. If M_2 is reached by the occurrence of Y in M_1 in CPN, and if marking M_1^* in CPN* covers M_1, then it is always possible to find a step Y^* in CPN*, such that Y^* covers Y and is enabled in M_1^*. If the occurrence of Y^* in M_1^* yields the marking M_2^*, then M_2^*will cover M_2.

The proof for Theorem 1 can be found in Appendix A.

5.5 Multiple Annotation Layers

The previous sections have discussed how to create a single annotation layer for a CP-net. The purpose of introducing an annotation layer is to make it possible to separate annotations from the CP-net, and to annotate a CP-net for several different purposes like, e.g. performance analysis and MSCs. However, if only one annotation layer exists, then it is not possible to easily disable, e.g. only the annotations for performance analysis, while still using the annotations for MSCs. The reason is, that all annotations have to be written in the one and only annotation layer. This motivates the need for multiple layers of annotations. When multiple annotation layers are allowed, then independent annotations can be written in separate annotation layers, and thereby making it easy to enable and disable each of the independent annotation layers.

Definition 7 defines multiple annotation layers. Multiple annotation layers are defined using the fact that a single annotation layer, $\mathcal{A}_1$, and a CP-net, CPN, is translated to another CP-net, CPN$_1^*$. Seen from another annotation layer, $\mathcal{A}_2$, CPN$_1^*$ is essentially the same as CPN aside from the added annotations, and can therefore be annotated with an annotation layer $\mathcal{A}_2$. The consequence of this definition is that $\mathcal{A}_2$ can refer to annotations in $\mathcal{A}_1$. In general, annotations in annotation layer $\mathcal{A}_i$ can refer to annotations in annotation layer $\mathcal{A}_j$ when $j \leq i$.

Definition 7. Let CPN be a CP-net and let $\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_n$ be annotation layers for CPN. Let τ be the translation from an annotation layer $\mathcal{A}$ and a corresponding CP-net CPN to CPN*, as defined in Sect. 5.3. Then CPN* with multiple annotation layers is defined by:

$$CPN^* = \tau(\ldots\tau(\tau(\text{CPN},\mathcal{A}_1),\mathcal{A}_2),\mathcal{A}_n)$$

6 Conclusion

In this paper we have discussed annotations for CP-nets where annotations are used to add auxiliary information to tokens. Auxiliary information is needed to support different uses of a single CP-net, such as for performance analysis and visualisation, thus the information should not have influence on the dynamic behaviour of a CPN model. One of the advantages of using annotations instead of manually extending the colour sets in a CPN model is that annotations are specified separately from the colour sets and arc inscriptions. That means that it is easy to enable and disable annotations from being part of the simulation. This is a great advantage when using a model for several purposes such as functional analysis, performance analysis, and visualisation. In addition, it is a great advantage that the behaviour of the matching CP-net matches the behaviour of the underlying CP-net in a very specific and predictable way.

Related work is considered in, e.g. Lakos' work on abstraction [11], where behaviour-respecting abstractions of CP-nets have been investigated, and a so-called colour refinement is proposed. This colour refinement is used to specify more detailed behaviour in sub-modules by extending colour sets to larger domains. The refined colours are only visible in the sub-modules, and the refined colours will typically contain information that is necessary for modelling the behaviour of the system in question. This colour refinement somewhat corresponds to our way of extending colour sets by adding annotations to colours. We are not aware of any other work that addresses the problem of introducing *auxiliary* information into a CP-net (or any other type of simulation model) while at the same time preserving the behaviour of the

CP-net. Nor do we know of any other method that can be used to automatically enable or disable different kinds of instrumentation when analysing different aspects of one particular model.

Ex*Spect* [1] is another tool for CP-nets. The tool provides libraries of so-called building blocks that provide support for, e.g., creating message sequence charts and performance analysis. Each building block is similar to a substitution transition and its subpage in Design/CPN. In Ex*Spect* all information that is necessary for updating a MSC or for collecting performance data is included in token colours. Reading the relevant data from token values and processing it is also encoded directly into the model via the building blocks. For example, the building block that can be used to calculate performance measures contains a place which holds the current result. When a certain transition occurs, a new value can be read from a binding element, and the result on this place is updated accordingly. While the building blocks are very easy to use, no attempt is made to separate auxiliary information from a CP-net, and the behaviour of the CP-net also reflects behaviour that is probably not found in the system being modelled.

There are many issues that can be addressed in future work regarding annotations. The techniques that have been presented here have not yet been used in practice. Clearly, it is important that support for annotations be implemented in a CPN tool in order to investigate the practicality and usefulness of the proposed method. Future work includes additional research on dealing with arc inscriptions that do not evaluate to a single colour on input arcs to transitions. In addition, further work is required to improve our proposal of how to add annotations to multi-sets of tokens. The definition of annotation layers states that it is only possible to add one particular annotation to all elements in a multi-set that is obtained by evaluating either an initial expression or an arc expression on an output arc from a transition. This is unnecessarily restrictive, and it should be generalised to make it possible to add different annotations to different elements in a multi-set. Practical experience with annotations may also show that the definition of annotation layers should be extended to include the possibility of defining guards in annotation layers.

In this paper we have only considered how to add annotations to existing arcs expressions, and thereby only considered how to annotate existing tokens. However, it might be useful also to be able to add net structure to the annotation layers. As an example, a place could be added only to the annotation layer with a token to hold a counter with the number of occurrences of a transition. Allowing additional net structure at the annotation layers would make it possible to take advantage of the powerfulness of the graphical notation of CP-nets when encoding the logics of the annotations.

We have only discussed separating the auxiliary annotations and the CP-net from each other. This could be generalised to also allow splitting a CP-net into layers where more layers can be combined to specify the full behaviour of a CP-net. In other words, the specification of the behaviour in a CP-net could be split in more layers. As an example, reconsider the resource allocation CP-net in Fig. 1 in Sect. 2. The loop handling the resource on the place R (R, T1, B, T2, C, and T3) is to some extent independent from the remaining model (even though it has impact on the behaviour). This loop could be separated from the remaining CPN model into a new layer to emphasise the fact that the loop is an extra requirement that can be added to the system. This facility could turn out to be very useful when a modeller is simplifying a CP-net to, e.g. be able to generate a sufficiently small state space to be able to analyse it. It would be a matter of moving the parts of the net structure that should not be included when generating the state space to another layer, and then only conduct the analysis on the remaining parts of the CP-net. This could be obtained by disabling the layer with the unneeded behaviour, and the state space could be generated. The advantage is that now a single model exists with layers specifying different behaviour which can be enabled or disabled – instead of having several similar models. Finally, such layers can also make it easier to develop tools where more people can work on a model concurrently, when they operate on different layers.

Acknowledgements We would like to thank Kurt Jensen who has read several drafts of this paper and has provided invaluable comments and feedback. We would also like to thank Søren Christensen, Louise Elgaard and Thomas Mailund who have also read and commented on previous drafts of this paper. Finally, we would also like to thank the anonymous reviewers for their comments and suggestions.

References

1. W. v. d. Aalst, P. d. Crom, R. Goverde, K. v. Hee, W. Hofmann, H. Reijers, and R. v. d. Toorn. ExSpect 6.4 – An Executable Specification Tool for Hierarchical Coloured Petri Nets. In M. Nielsen and D. Simpson, editors,

Proceedings of the 21st International Conference on Application and Theory of Petri Nets, volume 1825 of *LNCS*, pages 455–464. Springer-Verlag, 2000.
2. CAPLAN Project, Online: http://www.daimi.au.dk/CPnets/CAPLAN/.
3. S. Christensen, J. Jørgensen, and L. Kristensen. Design/CPN – A Computer Tool for Coloured Petri Nets. In E. Brinksma, editor, *Proceedings of TACAS'97*, volume 1217, pages 209–223. Springer-Verlag, 1997.
4. CPN Tools, Online: http://www.daimi.au.dk/CPnets/CPN2000/.
5. Design/CPN, Online: http://www.daimi.au.dk/designCPN/.
6. G. Gallasch and L. Kristensen. Comms/CPN: A Communication Infrastructure for External Communication with Design/CPN. In K. Jensen, editor, *Third Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 79–93. University of Aarhus, Department of Computer Science, 2001. Online: http://www.daimi.au.dk/CPnets/workshop01/.
7. ITU-T Recommendation Z.120, Message Sequence Chart (MSC), 1996.
8. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997. 2nd corrected printing.
9. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. 2, Analysis Methods*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997. 2nd corrected printing.
10. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. 3, Practical Use*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997.
11. C. Lakos. On the Abstraction of Coloured Petri Nets. In *Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, volume 1248 of *LNCS*, pages 42–61. Springer-Verlag, 1997.
12. B. Lindstrøm and L. Wells. *Design/CPN Performance Tool Manual*. Department of Computer Science, University of Aarhus, Denmark, 1999. Online: http://www.daimi.au.dk/designCPN/man/.
13. B. Lindstrøm and L. Wells. Towards a Monitoring Framework for Discrete-Event System Simulations. In *To appear in Proceeding of Workshop on Discrete Event Systems*, October 2002.
14. Message Sequence Charts in Design/CPN, Online: http://www.daimi.au.dk/designCPN/libs/mscharts/.
15. J. L. Rasmussen and M. Singh. *Mimic/CPN: A Graphic Animation Utility for Design/CPN*. Department of Computer Science, University of Aarhus, Denmark. Online: http://www.daimi.au.dk/designCPN/libs/mimic/.

A Proof of Matching Behaviour

Theorem 1 (same as in Sect. 5.4) *Let (CPN, $\mathcal{A}$) be an annotated CP-net with a sound annotation layer. Let CPN* be the matching CP-net derived from (CPN, $\mathcal{A}$). Let M_0, $\mathbb{M}$, and $\mathbb{Y}$ denote the initial marking, the set of all markings, and the set of all steps, respectively, for CPN. Similarly, let M_0^*, $\mathbb{M}^*$, and $\mathbb{Y}^*$ denote the same concepts for CPN*. Then we have the following properties:*

i. $\pi(\mathbb{M}^*)=\mathbb{M} \wedge \pi(M_0^*)=M_0$.
ii. $\pi(\mathbb{Y}^*)=\mathbb{Y}$.
iii. $\forall\, M_1^*, M_2^*\in\mathbb{M}^*, \forall\, Y^*\in\mathbb{Y}^*: M_1^*[Y^*\rangle M_2^* \Rightarrow \pi(M_1^*)[\pi(Y^*)\rangle\pi(M_2^*)$
iv. $\forall\, M_1,M_2\in\mathbb{M}, \forall\, Y\in\mathbb{Y}, \forall\, M_1^*,M_2^*\in\mathbb{M}^*$:
$M_1[Y\rangle M_2 \wedge \pi(M_1^*)=M_1 \Rightarrow \exists Y^*\in\mathbb{Y}^*: \pi(Y^*)=Y \wedge M_1^*[Y^*\rangle M_2^* \wedge \pi(M_2^*)=M_2$

Proof: The proof is a simple consequence of earlier definitions and Jensen's definitions for CP-nets [8]. Let TE, (t,b), and BE denote the set of all token elements, a binding element, and the set of all binding elements, respectively, for CPN. Similarly, let TE*, (t,b*), BE* denote the same concepts for CPN*.

Before showing that the above properties hold, we will show that the following holds for all annotated arcs:

$$\forall\, (t,b^*), \forall\, a\in A_{\mathcal{A}}\cap A(t): \pi\big(E^*(a)\langle b^*\rangle\big)=E(a)\langle\pi(b^*)\rangle. \qquad (\dagger)$$

Let (t,b*) and $a\in A_{\mathcal{A}}\cap A(t)$ be given.

$$\pi\big(E^*(a)\langle b^*\rangle\big) \overset{Def.\ 4.viii}{=} \pi\big((\text{Annotate } E(a)\ E_{\mathcal{A}}(a))\langle b^*\rangle\big) \overset{Defs.\ 1\&2}{=} E(a)\langle b^*\rangle \overset{Def.\ 5.ii}{=} E(a)\langle\pi(b^*)\rangle$$

Property i. We will show that $\mathbb{M}=\pi(\mathbb{M}^*)$. It is straightforward to show that $\pi(\mathbb{M}^*)=(\pi(TE^*))_{MS}$, and the proof is therefore omitted. From Def. 2.7 in [8] we have that $\mathbb{M}=TE_{MS}$. Thus it is sufficient to show that $TE=\pi(TE^*)$. The definition of $\pi(TE^*)$ gives us:

$$\pi(TE^*)=\big\{(p, c^*) \mid p\notin P_{\mathcal{A}} \text{ and } c^*\in C^*(p)\big\} \cup \big\{(p, \pi(c^*)) \mid p\in P_{\mathcal{A}} \text{ and } c^*\in C^*(p)\big\}$$

which by the definition of C* (Def. 4.vi) is equivalent to:

$\pi(TE^*)=\{(p, c) \mid p\notin P_{\mathcal{A}} \text{ and } c\in C(p)\} \cup \{(p, \pi(c^*)) \mid p\in P_{\mathcal{A}} \text{ and } c^*\in C(p)\times C_{\mathcal{A}}(p)\}$

which by the definition of the projection of annotated elements (Def. 1) is equivalent to:

$\pi(TE^*)=\{(p, c) \mid p\notin P_{\mathcal{A}} \text{ and } c\in C(p)\} \cup \{(p, c) \mid p\in P_{\mathcal{A}} \text{ and } c\in C(p)\}$

the two sets can be combined and we have:

$\pi(TE^*)=\{(p, c) \mid p\in P \text{ and } c\in C(p)\} \overset{Def.\ 2.7 in\ [8]}{=} TE$

To show that $\pi(M_0^*)=M_0$, we will show that $\forall p\in P^*$: $(\pi(M_0^*))(p)=M_0(p)$.
Consider non-annotated places:

$\forall p\notin P_{\mathcal{A}}$: $(\pi(M_0^*))(p) \overset{Def.\ 5.i}{=} M_0^*(p) = I^*(p) \overset{Def.\ 4.ix}{=} I(p) = M_0(p)$

Consider annotated places:

$\forall p\in P_{\mathcal{A}}$: $(\pi(M_0^*))(p) \overset{Def.\ 5.i}{=} \pi(M_0^*(p)) = \pi(I^*(p)) \overset{Def.\ 4.ix}{=} \pi(\text{Annotate } I(p)\ I_{\mathcal{A}}(p)) \overset{Defs.\ 1\&2}{=} I(p) = M_0(p)$

Property ii. We must show that $\mathbb{Y}=\pi(\mathbb{Y}^*)$. It is straightforward to show that $\pi(\mathbb{Y}^*)= (\pi(BE^*))_{MS}$, therefore the proof is omitted. From Def. 2.7 in [8] we have that $\mathbb{Y}=BE_{MS}$, therefore it is sufficient to show that $BE=\pi(BE^*)$, which we will do by showing: $(t,b')\in\pi(BE^*)\Leftrightarrow (t,b')\in BE$.

Let us show $\Rightarrow$: Let $(t, b')\in\pi(BE^*)$ be given. There exists $(t,b^*)\in BE^*$ such that $(t, \pi(b^*))=(t, b')$ (by definition of $\pi(BE^*)$). b^* is a binding of t in CPN*, therefore for all $v\in Var^*(t)$, where $Var^*(t)$ is the set of variables for t in CPN*, $b^*(v)\in Type(v)$, and b^* fulfils the guard of t in CPN*, i.e. $G^*(t)\langle b^*\rangle$.

From Def. 5.ii we have that for all $v\in Var(t)$, $(\pi(b^*))(v)=b^*(v)$, and we know that $b^*(v)\in Type(v)$. Since $G^*(t)=G(t)$ (Def. 4.vii) and $Var(G(t))\subseteq Var(t)$, we can conclude that $G(t)\langle\pi(b^*)\rangle$, i.e. $\pi(b^*)$ fulfills the guard of t in CPN. From the definition of a binding (Def. 2.6 in [8]), we have that $\pi(b^*)$ is a binding for t in CPN, therefore $(t, \pi(b^*))=(t,b')$ is a binding element for CPN, i.e. $(t, b')\in BE$.

Let us show $\Leftarrow$: Let $(t, b')\in BE$ be given. Using arguments that are similar to the above it is straightforward to show that b' fulfills the guard for t in CPN*, i.e. $G^*(t)\langle b'\rangle$. The binding b' does not bind the variables in $Var^*(t)\setminus Var(t)$. Define a new function b^* on $Var^*(t)$:

$$b^*(v) = \begin{cases} b'(v) & \text{if } v\in Var(t) \\ \text{an arbitrary value from } Type(v) & \text{if } v\in Var^*(t)\setminus Var(t) \end{cases}$$

According to Def. 2.6 in [8], b^* is a binding for t in CPN*. Therefore, (t, b^*) is a binding element for CPN*. By definition of b^*, we have that $\pi(b^*)=b'$, and as a result, $(t, b')\in\pi(BE^*)$.

Property iii. We must show that $\forall M_1^*, M_2^*\in\mathbb{M}^*, \forall Y^*\in\mathbb{Y}^*: M_1^*[Y^*\rangle M_2^* \Rightarrow \pi(M_1^*)[\pi(Y^*)\rangle\pi(M_2^*)$

We will first show that $\pi(M_1^*)[\pi(Y^*)\rangle$. By the *enabling rule* (Def. 2.8 in [8]) we have that:

$$\forall p\in P^*: \sum_{(t,b^*)\in Y^*} \sum_{a\in A(p,t)} E^*(a)\langle b^*\rangle \leq M_1^*(p) \qquad (*)$$

Consider non-annotated places and non-annotated arcs. Since $E^*=E$ for all non-annotated arcs (by Def. 4.viii), and $M_1^*=\pi(M_1^*)$ for all non-annotated places (by Def. 5.i), it follows from $(*)$ that:

$$\forall p\notin P_{\mathcal{A}}: \sum_{(t,b^*)\in Y^*} \sum_{a\in A(p,t)} E(a)\langle b^*\rangle \leq (\pi(M_1^*))(p)$$

which by the fact that $\pi(b^*)=b^*$ for all variables in $Var(E(a))$ (by Def. 5.ii) and the definition of $\pi(Y^*)$ (Def. 5.iii) is equivalent to:

$$\forall p\notin P_{\mathcal{A}}: \sum_{(t,\pi(b^*))\in\pi(Y^*)} \sum_{a\in A(p,t)} E(a)\langle\pi(b^*)\rangle \leq (\pi(M_1^*))(p) \qquad (**)$$

Consider annotated places and annotated arcs. From Def. 1 and $(*)$, it follows that:

$$\forall p\in P_{\mathcal{A}}: \pi\Big(\sum_{(t,b^*)\in Y^*} \sum_{a\in A(p,t)} E^*(a)\langle b^*\rangle\Big) \leq \pi(M_1^*(p))$$

which by Defs. 1 and 5.i is equivalent to:

$\forall p \in P_{\mathcal{A}}: \sum_{(t,b^*)\in Y^*} \sum_{a\in A(p,t)} \pi\big(E^*(a)\langle b^*\rangle\big) \leq (\pi(M_1^*))(p)$

which by (†) and the definitions of $\pi(b^*)$ and $\pi(Y^*)$ is equivalent to:

$\forall p \in P_{\mathcal{A}}: \sum_{(t,\pi(b^*))\in \pi(Y^*)} \sum_{a\in A(p,t)} E(a)\langle \pi(b^*)\rangle \leq (\pi(M_1^*))(p)$

which together with (∗∗) and the enabling rule gives us that $\pi(M_1^*)[\pi(Y^*)\rangle$.

Next we have to prove that the marking reached when Y^* occurs in M_1^* covers the marking that is reached when $\pi(Y^*)$ occurs in $\pi(M_1^*)$, i.e. that $\pi(M_1^*)[\pi(Y^*)\rangle\pi(M_2^*)$. A proof similar to the above can be used to show this, and the proof is therefore omitted.

Property iv. We must show that $\forall\ M_1,M_2 \in \mathbb{M}$, $\forall\ Y \in \mathbb{Y}$, $\forall\ M_1^*,M_2^* \in \mathbb{M}^*$:

$M_1[Y\rangle M_2 \wedge \pi(M_1^*)=M_1 \Rightarrow \exists Y^* \in \mathbb{Y}^*: \pi(Y^*)=Y \wedge M_1^*[Y^*\rangle M_2^* \wedge \pi(M_2^*)=M_2$

Let $M_1[Y\rangle M_2$ in CPN be given. It is straightforward to show that it is always possible to find $M_1^* \in \mathbb{M}^*$ such that $\pi(M_1^*)=M_1$, thus the proof is omitted. Since CPN* is a matching CP-net that is derived from an annotated CP-net with a sound annotation layer, and $\pi(M_1^*)=M_1$, Def. 6 tells us that there exists $Y^* \in \mathbb{Y}^*$ such that $\pi(Y^*)=Y$ and $M_1^*[Y^*\rangle$.

We have only left to show that the marking reached after Y occurs in M_1 is covered by the marking reached when Y^* occurs in M^*. Since $M_1[Y\rangle M_2$ in CPN, the *occurrence rule* (Def. 2.9 in [8]) gives us that:

$$\forall\ p \in P: M_2(p) = \Big(M_1(p) - \sum_{(t,b)\in Y} \sum_{a\in A(p,t)} E(a)\langle b\rangle\Big) + \sum_{(t,b)\in Y} \sum_{a\in A(t,p)} E(a)\langle b\rangle \qquad (\diamond)$$

Since $M_1^*[Y^*\rangle$ in CPN*, the occurrence rule gives us that:

$$\forall p \in P^*: M_2^*(p) = \Big(M_1^*(p) - \sum_{(t,b^*)\in Y^*} \sum_{a\in A(p,t)} E^*(a)\langle b^*\rangle\Big) + \sum_{(t,b^*)\in Y^*} \sum_{a\in A(t,p)} E^*(a)\langle b^*\rangle \qquad (\diamond\diamond)$$

In other words, $M_1^*[Y^*\rangle M_2^*$. We must now show that $\pi(M_2^*)=M_2$.

We will show that $\pi(M_2^*)=M_2$ for non-annotated places. We have found M_1^*, such that $\pi(M_1^*)=M_1$. We have that $M_1^*=\pi(M_1^*)$ and $M_2^*=\pi(M_2^*)$ for non-annotated places (by Def. 5.i). For all non-annotated arcs $E^*=E$ (by Def. 4.viii). It follows from these facts and (◇◇) that:

$\forall p \notin P_{\mathcal{A}}: (\pi(M_2^*))(p) = \Big(M_1(p) - \sum_{(t,b^*)\in Y^*} \sum_{a\in A(p,t)} E(a)\langle b^*\rangle\Big) + \sum_{(t,b^*)\in Y^*} \sum_{a\in A(t,p)} E(a)\langle b^*\rangle$

which by the fact that $\pi(b^*)=b^*$ for all variables in Var(E(a)) (by Def. 5.ii) and the fact that $\pi(Y^*)=Y$ is equivalent to:

$$\forall p \notin P_{\mathcal{A}}: (\pi(M_2^*))(p) = \Big(M_1(p) - \sum_{(t,b)\in Y} \sum_{a\in A(p,t)} E(a)\langle b\rangle\Big) + \sum_{(t,b)\in Y} \sum_{a\in A(t,p)} E(a)\langle b\rangle \qquad (\diamond\diamond\diamond)$$

We will show that $\pi(M_2^*)=M_2$ for annotated places. From the definition of π for multi-sets and markings (Defs. 1 and 5.i) and from (◇◇), it follows that :

$\forall p \in P_{\mathcal{A}}: (\pi(M_2^*))(p) = \Big((\pi(M_1^*))(p) - \sum_{(t,b^*)\in Y^*} \sum_{a\in A(p,t)} \pi\big(E^*(a)\langle b^*\rangle\big)\Big) + \sum_{(t,b^*)\in Y^*} \sum_{a\in A(t,p)} \pi\big(E^*(a)\langle b^*\rangle\big)$

which by (†) and the fact that $\pi(M_1^*)=M_1$ is equivalent to:

$\forall p \in P_{\mathcal{A}}: (\pi(M_2^*))(p) = \Big(M_1(p) - \sum_{(t,b^*)\in Y^*} \sum_{a\in A(p,t)} E(a)\langle b^*\rangle\Big) + \sum_{(t,b^*)\in Y^*} \sum_{a\in A(t,p)} E(a)\langle b^*\rangle$

which by the definitions of $\pi(b^*)$ and $\pi(Y^*)$, and the fact that all variables in E(a) are bound by $\pi(b^*)$ is equivalent to:

$\forall p \in P_{\mathcal{A}}: (\pi(M_2^*))(p) = \Big(M_1(p) - \sum_{(t,b)\in Y} \sum_{a\in A(p,t)} E(a)\langle b\rangle\Big) + \sum_{(t,b)\in Y} \sum_{a\in A(t,p)} E(a)\langle b\rangle$

which together with (◇) and (◇◇◇) gives us that $\pi(M_2^*)=M_2$.

□

Model-based Operational Planning Using Coloured Petri Nets

Lin Zhang
Systems Simulation and Assessment Group
Command and Control Division
Defence Science and Technology Organisation
PO Box 1500, Edinburgh SA 5111
Australia

Abstract

This talk describes a model-based approach towards the development of software tools to support operational planning in the Australian Defence Force. An overview of the military planning process is provided for the purpose of identifying key concepts in operational planning. Models of the key concepts are then developed and formalised by the construction of a CPN model of an abstract operational task. The CPN model specifies the execution of tasks in a Course of Action (COA). During planning, the CPN model is instantiated with concrete tasks for execution and analysis. The model-based approach is illustrated by a prototype COA Scheduling Tool (COAST) that supports sequencing and scheduling of tasks in planning. COAST has a client-server architecture. The client provides a graphical user interface for task instantiation and COA analysis. The server uses the instantiated CPN model to conduct state space analysis for generating and analysing sequences of tasks in a COA.

Coloured Petri Nets in UML-Based Software Development – Designing Middleware for Pervasive Healthcare

Jens Bæk Jørgensen
Centre for Pervasive Computing
Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
email: jbj@daimi.au.dk

Abstract

Nowadays, the Unified Modeling Language, UML, is almost universally accepted by the software industry as *the* modelling language. However, the language has severe shortcomings. While UML is well suited to model the static aspects of software systems, the language as it is currently standardised strongly needs improvements with respect to modelling behaviour. Thus, for development of software components with complex behaviours, UML often cannot stand alone. The main contribution of this paper is to position and discuss promotion of Coloured Petri Nets, or more generally high-level Petri nets, as a supplement in UML-based software development. We make the case on a specific example, development of middleware to support what is termed pervasive healthcare, but the observations hold in general for many systems with complex behaviours.

Topics: Relationship between CPN and UML, promotion of CPN in the software industry, system design, application of CPN to pervasive and mobile systems.

1 Introduction

This paper considers design of a system to support pervasive and mobile computing [7, 17, 21] at hospitals, the *pervasive healthcare middleware system, PHM* [2]. The system is being developed in a joint project by the hospitals in Aarhus County, Denmark, the software company Systematic Software Engineering [50], and the Centre for Pervasive Computing [42] at the University of Aarhus.

In the development of PHM, the *Unified Modeling Language, UML* [32, 35, 13, 12], is applied. By many of its proponents, UML is touted as a complete modelling language in the sense that it supports all modelling needs in all software development phases, from requirements specification through analysis, design, and implementation to the final system test. Indeed, UML has many advantages, e.g., it is general-purpose and extensible, it is standardised, a large selection of tools and text books is available, industrial acceptance is overwhelming, and success stories can be counted in hundreds or more [51]. However, the

language is certainly not perfect [37], and use of UML as the one and only modelling language in the software industry can of course, from an academic point of view, be challenged. On the other hand, seen from the point of view of an industrial software developer, or his manager, an advocate of an alternative to UML has the burden of proof. The language is a de facto industrial standard, and with this status, it is the default choice of a modelling language in industrial software projects, much in the same way as Java is today's default choice of a programming language within many organisations.

In this paper, we challenge the exclusive use of UML in the design phase of the development of PHM. Here, we will suggest applying *Coloured Petri Nets, CPN* [23, 25, 43], in conjunction with UML. The proposal to combine Petri nets and UML in software development is not new, see, e.g., [39, 40]. However, the combination is often discussed in general terms only. The main contribution of this paper is to demonstrate in detail that CPN is a powerful supplement to UML, by pinpointing a number of specific, important PHM design issues that can be addressed properly in CPN, but not at all or not as easily in UML. Another contribution is considerations on the general positioning of UML and CPN in the software industry, and a discussion of how to better promote CPN. With this paper, we hope to help answering the frequently asked question: Why don't you just use UML?

A deliberate restriction of scope is that this paper does not consider formal verification. In many industrial software development projects, formal verification is not seen as an option at all, for reasons like needs, traditions, time schedules, shortcomings in verification methods and tools, and insufficiently trained software developers. Ensuring software quality is often done by systematic and thorough testing alone (whether this is wise, is a discussion beyond the scope of this paper).

This paper is structured as follows: Section 2 introduces the domain of our case study, healthcare information technology in general, pervasive healthcare in particular, and the PHM system itself. The design of a central component of PHM, the so-called session manager, by means of CPN is presented and discussed in Section 3. Section 4 provides a primer on UML, while Section 5 discusses a UML design of the session manager and compares the CPN and UML design approaches. Section 6 considers the general positioning of UML and CPN, and promotion of the latter, in the software industry. The conclusions, including a discussion of related work, are drawn in Section 7.

2 Healthcare Information Technology

The application area for PHM, the healthcare sector, is a hot topic in the public debate in Denmark, and probably also in many other countries. Everybody agrees that patients should be given the best treatment possible with the available resources. Therefore, rationalisation of hospital work processes is highly desired seen from the perspective of any stakeholder, from the minister of healthcare who wants to keep his budget, down to the nurses who have a strong need for alleviation in their busy work days.

An essential element in the rationalisation is to apply information technology to a much larger extent than has been done previously. A first step in this direction is the introduction of electronic patient records, which is going on in

many places right now [5]. One of these places is Aarhus County, whose new electronic patient record *EPR* [41] will soon replace today's paper-based records. EPR is a comprehensive, general-purpose hospital IT system with a budget of approximately 15 million US dollars. It will be initially fielded later this year and put into full operation in 2004. The first version of EPR is made available from desktop PCs placed in hospital offices. However, the EPR users, i.e., the nurses and doctors, are away from their offices and on the move a lot of the time, e.g., seeing patients in the wards. Therefore, stationary desktop PCs in hospital offices are insufficient to support the hospital work processes in the best way, and taking advantage of the emerging possibilities for pervasive and mobile computing is crucial for the next version of EPR.

In the remainder of this section, we describe the notion of pervasive healthcare, and we introduce the pervasive healthcare middleware, PHM, the software system we are considering, and the session manager component, whose design will be the focus for our discussion.

2.1 Pervasive Healthcare

In the ideal situation, the healthcare professionals should bring the patient records with them, wherever they go. Thus, up-to-date information about patients, diagnoses, treatment plans, medication etc. would always be available at any time, in any given situation. The challenge seen from an IT perspective is to develop computer support for such new ways of working, and one potential approach is to exploit pervasive and mobile computing.

The term *pervasive healthcare* [47] covers the healthcare services that can be offered when the hospital staff has access to relevant data and functionality wherever they are. With pervasive healthcare, there may be computers "everywhere", e.g., in the room where the nurses prepare the prescribed medicine for patients, and in the trolleys used to transport trays with medicine. There may be large-screen computers in the wards so it is easy to show pictures like X-rays to the patients. Doctors and nurses may carry small personal digital assistants, PDAs, with which they can, e.g., access EPR and control other devices such as TV sets.

A specific example of a future pervasive healthcare scenario is the following: A nurse is about to prepare a round to the wards to give medicine to patients. The nurse wears a badge that emits a signal enabling various computers to sense her location. While she is approaching the room where the medicine is kept, a computer in this room automatically accesses EPR and fetches the medication plans for the patients that the nurse is responsible for. The computer presents these plans on the screen. The nurse prepares the medicine trays and uses the computer to acknowledge for each patient when the medicine has been poured correctly. Finally, all trays are put on a trolley. The trolley is equipped with a location-sensitive computer, and when the trolley approaches a ward, the medication plans for the patients in that ward are automatically fetched and presented on the screen. In this way, it is fast and easy for the nurse to acknowledge to the system when a patient has taken the medicine. If a patient has a question regarding the medicine, the nurse is able to answer supported by accessing medicine handbooks on-line using her PDA and perhaps using the TV set in the ward to display information.

2.2 The Pervasive Healthcare Middleware – PHM

The *pervasive healthcare middleware, PHM,* is a new computer system designed to support pervasive healthcare. PHM is a distributed system consisting of a number of components running in parallel on various mobile and stationary computing devices, and communicating over a wireless network. Some components run on a central background server, while others are deployed on the mobile devices. An early, overall design for PHM is proposed in [2]. Figure 1 shows selected components of the PHM architecture – the names hint their functionality.

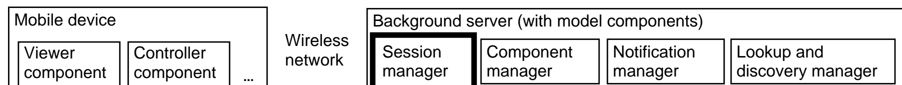


Figure 1: Selected components of PHM.

The scope of this paper is restricted to discussing design of the session manager component, shown with thick border in Figure 1.

A *session* comprises a number of devices that are joined together, sharing data, and communicating in support of some specific work process. A session is appropriate, e.g., if a nurse wants to use her personal digital assistant, PDA, to control a TV set in a ward in order to show an X-ray picture to a patient. In this case, the TV and the PDA must be joined in a session. Another example is a nurse who wishes to discuss some data, e.g., electronic patient record data, or audio and video in a conference setting, with doctors who are in remote locations. Here, the relevant data must be shown simultaneously on a number of devices joined in a session, one device for the nurse and one device for each doctor.

In general, session data is viewed and possibly edited by the users through their devices. The PHM architecture is based on the Model-View-Controller pattern [8]. The model part administers the actual data being shared and manipulated in a session. Each participating device has both a viewer and a controller component which are used as, respectively, interface to and manipulator of the session data. Model, viewer, and controller components communicate over the wireless network.

2.3 The Session Manager

Sessions are managed by the *session manager*, which is one of the most central and most complex components of PHM. The session manager manages zero to any number of sessions, and a session comprises one or more devices. Seen from the point of view of the session manager, a device is either inactive, i.e., not currently participating in any session, or active, i.e., participating in some session. A device participates in at most one session at a time.

The operations that the session manager must provide can be grouped into three main functional areas:

1. *Configuration management*: Initiation, reconfiguration (i.e., supporting devices dynamically joining and leaving), and termination of sessions.

2. *Lock management*: Locking of session data. Session data is shared and must be locked by a device, which wants to edit it.

3. *Viewer/controller management*: Change of viewers and controllers for active devices, e.g., if a nurse enters a room containing a TV, she may wish to view something on the large TV screen instead of on her small PDA display. In this case, viewer and controller replacement on the PDA and the TV is needed.

Devices interact with the session manager by invoking its operations. One interaction scenario is shown in Fig. 2, which illustrates the communication between the session manager and two devices, `d1` and `d2`.

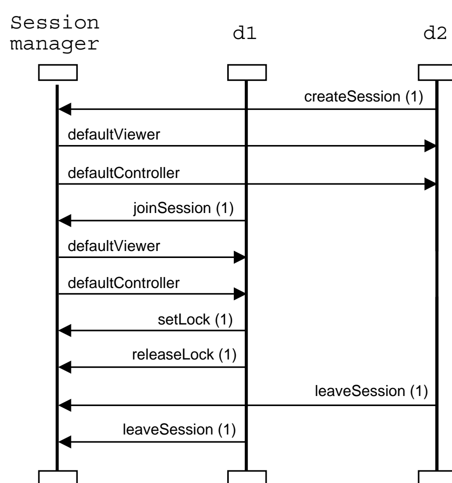


Figure 2: Session manager / device communication.

First, `d2` creates a session, which gets the session identifier 1. The session manager responds to the creation request by providing `d2` with a default viewer and a default controller for the new session. Then, `d1` joins the session, and also gets a default viewer and a default controller from the session manager. At some point, `d1` locks the session, probably does some editing, commits, and later releases the lock. Finally, `d2` and then `d1` leave the session.

3 Session Manager Design in CPN

Figure 2 is an example illustrating one single possible sequence of interactions between the session manager and some devices. In the session manager design, of course much more is needed. It is crucial to be able to specify and investigate the general behavioural properties of session management.

We now present a CPN model, whose purpose is to constitute an initial design proposal for the session manager, by identifying and giving an overall characterisation of the operations to be provided, with focus both on their individual behaviour and their interplay. The static architecture of the session manager in terms of classes and relationships between classes is outside the

scope of the use of CPN (here, a fine job can be done with UML). The CPN model is created with Design/CPN [44].

It should be noted that the intention of this paper is not to present a large and complex CPN model. The aim is to create a model, which in the first place serves its purpose in the PHM development by being a design of the session manager, but secondly and equally important, is sufficiently tractable to constitute the foundation for the comparison of CPN and UML to be made in Section 5. In fact, the moderate size of the model to be presented is an advantage for the purpose to promote CPN. As we will argue, even for a modelling task of this size, UML is not sufficient, and the problems encountered in UML are exacerbated for larger models.

3.1 Net Structure and Declarations

The net structure of the CPN model consists of four modules (pages). The top-level of the model is the `SessionManager` module, shown in Figure 3, where the three main functional areas are represented by means of substitution transitions. In this way, each main functional area is modelled by an appropriately named sub-module, `ConfigurationManager`, `LockManager`, and `ViewCtrManager`, respectively.

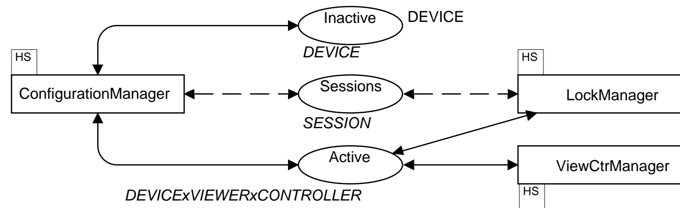


Figure 3: `SessionManager` module.

Figure 4 shows the declaration of constants, colour sets, and variables.

It can be seen that the model contains declarations of simple index colour sets for devices (`DEVICE`), viewers (`VIEWER`), and controllers (`CONTROLLER`). In addition, there are Cartesian product colour sets used to model when devices are associated with viewers and controllers. Sessions are modelled using the `SESSION` colour set, whose elements are triples `(s,dl,l)`, where `s` is a session identifier, `dl` is a list of devices, and `l` is a lock indicator.

The operations of the session manager correspond to the transitions of the CPN model. The detailed behaviour of the operations may immediately be derived from the arc expressions and guards. The latter corresponds to checks that the session manager must carry out before allowing the corresponding operation to be executed. We have chosen to use function calls consistently as arc expressions and guards, e.g., a function call like "`createSession s d`" instead of the expression "`(s,[d],NO_LOCK)`". In this way, the sub-routines of the session manager operations are explicitly identified.

In the following, we describe the model modules corresponding to the three main functional areas of the session manager. For each module, the corresponding session manager operation sub-routines are listed in terms of signatures for the functions used in inscriptions on that module.

```
------------------------------------------------------------
(* Constant declarations *)
val NO_OF_DEVICES = ...;
val NO_OF_VIEWERS = ...;
val NO_OF_CONTROLLERS = ...;

(* Colour set declarations *)
color DEVICE = index de with 1..NO_OF_DEVICES declare ms;
color INDEX = int with 1..NO_OF_DEVICES;
color VIEWER = index vi with 1..NO_OF_VIEWERS;
color CONTROLLER = index co with 1..NO_OF_CONTROLLERS;
color DEVICExVIEWERxCONTROLLER =
           product DEVICE * VIEWER * CONTROLLER;
color DEVICExVIEWER = product DEVICE * VIEWER;
color DEVICExCONTROLLER = product DEVICE * CONTROLLER;
color SESSION_ID = int;
color DEVICELIST = list DEVICE;
color LOCK = union L: DEVICE + NO_LOCK;
color SESSION = product SESSION_ID * DEVICELIST * LOCK;

(* Variable declarations *)
var d,d1,d2:  DEVICE;
var i:  INDEX;
var v:  VIEWER;
var c:  CONTROLLER;
var s:  SESSION_ID;
var dl:  DEVICELIST;
var l:  LOCK;
------------------------------------------------------------
```

Figure 4: Declaration of constants, colour sets, and variables.

3.2 Configuration Management

The `ConfigurationManager` module is shown in Figure 5. It has four places, `Inactive`, `Active`, `Sessions`, and `Next id`. Initially, all devices are inactive, corresponding to all the `DEVICE` tokens initially being at the `Inactive` place. The only transition which is enabled in the initial marking is `Create session`. When it occurs, triggered by an initiating device `d`, a new session is created. The session gets a fresh session id from the `Next id` place, starts in unlocked mode, and a token corresponding to the new session is put on the `Sessions` place. The token corresponding to the initiating device is augmented with a default viewer and controller, and that triple is put on the `Active` place.

The transition `Join session` adds devices to sessions. Any join must be preceeded by a permission check, modelled by the guard function `joinOK`. When a device `d` joins a session `(s,dl,l)`, the appropriate token residing on the `Sessions` place is updated accordingly, by use of the `joinSession` function. Moreover, the `d` token is removed from the `Inactive` place, augmented with a default viewer and controller, and put on the `Active` place. The transition `Leave session` works similarly, but reversely, to `joinSession`. Upon leaving,

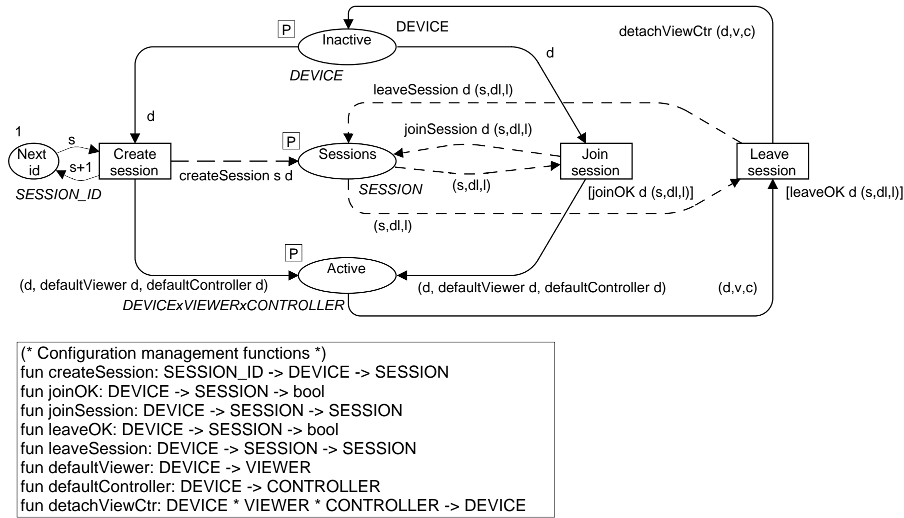


Figure 5: `ConfigurationManager` module.

the applicable viewer and controller for the device are detached.

3.3 Lock Management

The `LockManager` module is shown in Figure 6.

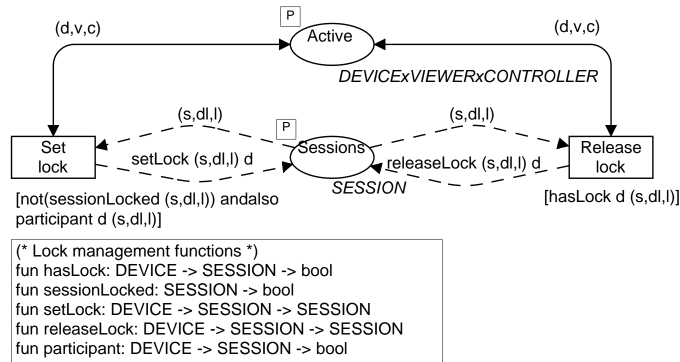


Figure 6: `LockManager` module.

`LockManager` contains the two places `Active` and `Sessions`, which are already described. In addition, the module contains the two transitions `Set lock` and `Release lock`. When `Set lock` occurs, the selected session `(s,dl,l)` is locked by the requesting device, which is identified by the `d` part of the `(d,v,c)` token. The session can only be locked if it is not locked already and if the requesting device is currently a participant in that session. These two conditions are checked by the guard of `Set lock`. The effect of the transition `Release lock` is to release the lock of the current session. This is only possible if the

requesting device is the actual lock holder, modelled by the `hasLock` function of the guard.

3.4 Viewer/Controller Management

The `ViewCtrManager` module is shown in Figure 7.

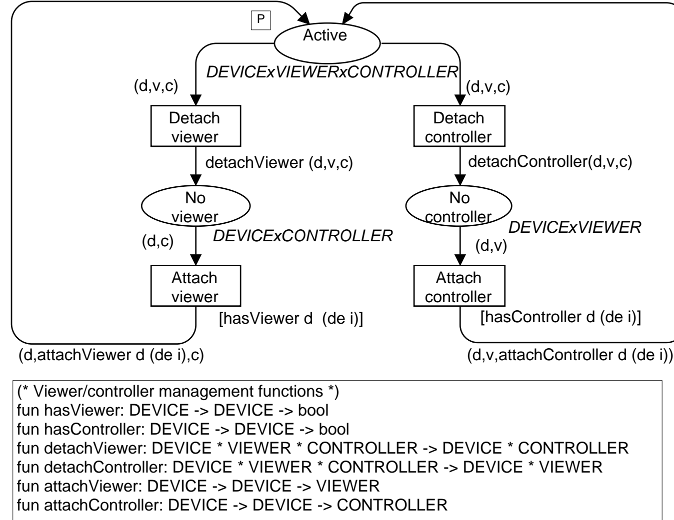


Figure 7: `ViewCtrManager` module.

In addition to the place `Active` already described, the `ViewCtrManager` module contains the two places `No viewer` and `No controller`. A token on either `No viewer` or `No controller` models that the corresponding device is suspended – even though the device is participating in a session. This means that the device is temporarily not able to read and write data (only those devices whose tokens are currently on the `Active` place are able to read and, if an appropriate lock is set, write data).

Viewers and controllers are not replaced in one atomic action. The replacement mechanism is modelled by four transitions. The two `Detach` transitions do, as the names indicate, detach the viewer or controller, respectively. There is no precondition for allowing this. The `Attach viewer` transition checks that the viewer that some device requests actually can be provided. The request is checked by the guard "`hasviewer d (de i)`", which evaluates to true if it is possible to equip device `d` with a viewer for device number i, `de i` (as can be seen from Figure 4, the devices are indexed and `i` is a free variable over the index range). The guard function `hasViewer` enforces that some viewers fit with some devices and not with others. The `Attach controller` transition works in a similar fashion.

3.5 Module Dependencies

The interplay between the three main functional areas of the session manager is reflected in dependencies between the three corresponding modules of the CPN

model, e.g., there is a dependency between lock management and configuration management, because a device in the process of editing session data is not allowed to abruptly leave the session. We require that the lock is released before permission to leave can be granted. Similarly, there is a dependency between viewer/controller management and configuration management. A temporarily suspended device in the process of replacing its viewer or controller is not allowed to leave the session, because the device might have the session lock set.

3.6 Modelling Decisions

The CPN model is an abstract view of the session manager. Some modelling decisions have been made in order to keep the model relatively simple, but still serving its purpose as constituting an initial design of the session manager. In particular, we have assumed reliable communication and that devices do not crash, i.e., errors in the communication between devices and the session manager are not modelled, and a question like what should happen if the device that is currently having a lock on session data crashes cannot be answered from the CPN model.

In a hectic and busy work day with the PHM system at the hospitals, communication errors will happen on the wireless network, and devices will crash, e.g., be turned off accidentally or run out of battery. To investigate such problems and strategies for coping with them, the CPN model could be extended. However, for the purpose of keeping a clear focus when we make comparative discussions of CPN and UML, we do not discuss how to model these issues in this paper.

4 The Unified Modeling Language – UML

We want to compare the proposed CPN design of the session manager with a design made exclusively in UML. For that purpose, and in order to make this paper relatively self-contained, in this section, we provide a primer on essential elements of the *Unified Modeling Language, UML*. The section may be skipped by readers already familiar with UML.

4.1 UML Background

UML supports object-oriented software development and offers a number of diagram types to model various views or perspectives of systems. Some views capture static aspects and other views describe behavioural aspects. UML as a whole is very big with an abundance of concepts and features, and the interested reader is referred to, e.g., [35, 13] for a more thorough description. The authoritative language documentation is the standard [32], but this massive and complex document cannot serve as an introduction.

The first version of UML appeared in 1997 as a result of an effort to unify a number of different, older object-oriented modelling methods. The current standard is UML 1.4 [32] dated May 2001. A major revision, UML 2.0, is expected to be approved late this year (2002). The main organisational body for the development and standardisation of UML is the Object Management Group, OMG [46]. UML is supported by many commercial tools, e.g., the

market-dominant Rational Rose suite from Rational Software Corporation [48] and the Rhapsody suite from I-Logix [45].

4.2 Diagram Types

UML offers the following diagram types:

- Class diagrams
- Object diagrams
- Use case diagrams
- Component diagrams
- Deployment diagrams
- State machines
- Activity diagrams
- Sequence diagrams
- Collaboration diagrams

A *class diagram* is used to describe the architectural, structural composition of a system by identifying classes and their interrelations, and an *object diagram* is a structural description on the object level. A *use case diagram* is applied to capture and describe the future users' requirements for a system to be built. A *component diagram* reflects the actual implementation of a system, and a *deployment diagram* covers the physical architecture in terms of the hardware and software that make up a system.

The behaviour of a system is modelled using state machines, activity diagrams, sequence diagrams, and collaboration diagrams. A *state machine* is used to describe the behaviour of objects instantiated from a certain class, and captures the states and the events that may happen in each state. State machines may communicate with each other, thus allowing modelling of the combined behaviour of a number of interacting objects. State machines are derived from statecharts [18, 20]. In general, statecharts augment conventional state-transition diagrams with notions of hierarchy, concurrency, and a special kind of broadcast communication. In particular, a state of a state machine may be a compound state comprising other states and events. In this way, a state may to some extent correspond to a substitution transition of a CPN model. An *activity diagram* is a special kind of state machine, where there are slightly different rules for trigging and execution of events than for a pure state machine.

Both sequence diagrams and collaboration diagrams show selected examples of communication between the objects of a distributed system. A *sequence diagram* resembles a message sequence chart [22] and focus is on time (an example of a sequence diagram is shown in Figure 2). A *collaboration diagram* is a kind of annotated object diagram, and focus is on objects and their relations, together with their communication. Sequence numbers are attached to arrows between objects to describe a certain chain of communications.

5 Comparative Session Manager Design in UML

In this section, we consider design of the session manager using UML exclusively. As with CPN, we focus on the behaviour of the session manager rather than its static architecture. However, in UML, the two aspects are not separated, and both must be dealt with. Application of UML's behavioural diagrams assumes the existence of well-defined classes, e.g., a state machine always specifies the behaviour of a certain class, and a sequence diagram always shows the communication between objects instantiated from certain classes. Therefore, before we model behaviour, we must define classes. For this reason, Figure 8 shows the main classes and relationships of concern for session management.

Figure 8: Session management class diagram.

Based on this class diagram, it was attempted to model the general behavioural properties of session management. This requires specification of both the individual behaviour of device and session manager objects, and the combined behaviour when these objects interact. The only UML behavioural diagram types which are candidates to be used in the general design are state machines and activity diagrams. Sequence and collaboration diagrams can only be used to show specific scenarios, i.e., they do not describe the general behaviour.

Therefore, it was attempted to create communicating state machines for the `SessionManager` and `Device` classes, and subsequently investigate their individual and combined behaviour. In this process, a number of severe problems were encountered. The problems that will be discussed in this paper are described below, and are all instances of more general shortcomings in UML behavioural modelling. The problems may be alleviated by using CPN as a supplement to UML, as we will argue below.

5.1 Executable Models

The first shortcoming in UML is lack of executable models. Without executable models, it is in practice impossible to investigate behavioural consequences of various design proposals for session management, prior to implementation. Executable models presume a well-defined formal execution semantics, which UML is currently lacking. None of the diagrams, in particular none of the behavioural diagrams, of the current UML standard [32] have a well-defined formal semantics, whereas the original statecharts of [18] do, defined in later papers, e.g., [19]. However, because of alterations made from statecharts to UML state machines, this well-defined semantics has been obscured – or, some would say, deliberately relaxed to become more "user-friendly".

If the lack of executable models was the only problem encountered in UML, it might be alleviated by using UML tools from, e.g., I-Logix's Rhapsody suite [45] or Rational's Rose RealTime [48] that do offer execution of UML behavioural

diagrams – with execution algorithms which are, by necessity, based on proprietary semantic decisions. A formal execution semantics will most likely sooner or later be part of the UML standard (but we need it now for the PHM project). State machines already have an informal, textually described semantics in the current standard [32].

As we know very well, CPN has formal execution semantics in terms of the enabling and occurrence rules, and consequently, CPN does indeed offer executable models. Therefore, CPN models allow us to investigate the behavioural consequences of alternative design choices, and, thus, there is a sound and convenient foundation for pursuing improvements. In the PHM project, the CPN model of Section 3 reflects a number of design decisions for the session manager, many of which may be argued, e.g., would it be better to allow a device to have more than one viewer and one controller at a time instead of just one of each; or should a session always be explicitly terminated by the initiating device instead of being implicitly terminated when the last participating device leaves. Alternatives can be modelled and investigated by making modifications and simulations of the CPN model.

5.2 Modelling of Dependencies

The UML state machines for the `SessionManager` and `Device` classes are closely interrelated. For both classes, all state changes of concern are consequences of devices invoking operations in the session manager. We have had difficulties in describing individual state machines for the two classes, while at the same time properly capturing their communicating behaviour.

The difficulties are caused by the interplay between the three main functional areas of the session manager. The interplay materialises as dependencies between various model entities, as discussed in Section 3.5. It is difficult to capture such dependencies in a proper way with state machines. Undesired interferences between the three main functional areas must be precluded, e.g., that a device loses its lock on session data during viewer/controller replacement. We have tried to use various advanced features of state machines, e.g., concurrent and-states and the history mechanism (the latter is controversial [37]), but have not been able to describe the dependencies between the three main functional areas in a satisfactory way.

In theory, it is possible to create state machines that capture all dependencies, simply by introducing a sufficient number of states, e.g., instead of two individual states like `Has lock` and `Is replacing controller`, introduce states with more complex interpretations like `Has lock and is replacing controller`. However, the approach does not scale well – the size of the state machines grows quickly with the number of dependencies to be modelled. As an example, state machines are not feasible to describe a more fine-grained locking scheme than the current coarse-grained one. Allowing locking of subsets of session data requires simultaneous management of several locks and introduces many dependencies.

Functional area dependencies can be properly described in CPN because of the fine-grained nature of CPN models. In the CPN model of the session manager, e.g., lock management and viewer/controller management cannot interfere with each other in an undesired way. Whether a session is locked or not is captured by a value in the `SESSION` token on the `Sessions` place. As can

be seen from Figure 3, replacement of viewers and controllers (modelled by the substitution transition `ViewCtrManager`) does not involve the `Sessions` place at all.

In contrast to state machines, CPN models scale well, e.g., a more fine-grained locking scheme, can be modelled based on letting `SESSION` tokens comprise lists of locks, instead of just one single lock. Moreover, it would be straightforward to extend the CPN model to do deal with unreliable communication and device crashes, as discussed in Section 3.6. It would be much harder to model these issues in UML, because they introduce more dependencies to be handled.

5.3 Modelling of Bookkeeping

A key task of session management is bookkeeping by tracking which devices are currently joined in sessions. Bookkeeping records must be updated each time a device creates, joins, or leaves a session. Proper investigation of session bookkeeping requires container-like data structure such as sets or lists to be supported in the session management behavioural models, e.g., to describe that in the current state, there are two sessions, one with devices {`d1,d2,d3`} and one with devices {`d4,d5`}. The state notion offered by UML state machines does not allow us to express this in a feasible way.

In the CPN model of the session manager, the place `Sessions` has a structured colour set defined and used with the purpose to do the desired bookkeeping, i.e., tracking which devices are in sessions together.

5.4 UML State Machines Versus Petri Nets

UML state machines have some resemblance with low-level Petri nets. The modelling of dependencies and of bookkeeping would also have caused problems if we had chosen low-level Petri nets as our modelling language to describe the behaviour of the session manager. These issues are dealt with smoothly in CPN, exactly because of the increased modelling convenience that characterises high-level Petri nets in general compared to low-level Petri nets.

As an aside, the original statecharts paper [18] sketches a rough idea about parameterised states, which has some similarity with high-level Petri nets. Also, in [18] Petri nets are recognised as a powerful means to describe behaviour, but it is noted as a main problem that Petri nets cannot be hierarchically decomposed. Since the publication of [18] in 1987, as we know, this problem has indeed been solved [23].

6 UML and CPN in the Software Industry

As demonstrated above, CPN may be used to alleviate general and severe problems encountered in UML. Therefore, one might argue that CPN often should be an obvious choice for industrial software developers engaged in modelling behaviour. Why this is not the case anyway is discussed in this section, where we state general observations regarding use of UML and CPN in the software industry, and where we also consider how to better promote CPN.

First of all, UML had a head start in comparison with CPN in the software industry, because the two modelling languages had very different starting

points. UML came out of the object-oriented programming community, which set the dominating trend for industrial software development over more than the last decade. Therefore, from its advent, UML has enjoyed much attention in the software industry. In contrast, Petri nets and CPN emerged from mathematics and theoretical computer science as a model for concurrency, and have gained attention mostly within academia. Moreover, from the very beginning, UML became the subject for an attractive commercial market for tool vendors, consultants, etc., which all contributed to the boost of the language. We have not seen anything similar for CPN, or for Petri nets in general.

6.1 UML in the Software Industry

Many software companies are continuously applying UML, typically for the description of static, architectural properties of systems, by creation of primarily class diagrams. A software developer's incentive to use class diagrams is high, both because they are a very usable and convenient abstraction, but also because it is common functionality of UML tools to be able to generate source code level class skeletons automatically from class diagrams. In this way, the artifacts produced during design, i.e., the class diagrams, save the developer for some amounts of work during implementation.

Often, class diagrams suffices, e.g., for traditional administrative systems with the major part of the functionality being database access, such as the first version of EPR, Aarhus County's electronic patient record. For such systems, complex behavioural issues, e.g., involving communication, synchronisation and resource sharing are handled within standard off-the-shelf software components like database management systems and various middleware components such as application servers. Consequently, application-level software developers do not have to deal directly with the complexities themselves. However, when focus moves from database access to developing systems with complex behaviour, UML sometimes needs a supplement.

There are many speculations, not the least in the academic UML community itself, see, e.g., [34], that in many cases, the UML behavioural diagram types are not sufficient, and, as a consequence, not that widely used. In particular, UML state machines and activity diagrams are the only options to describe general behaviour, and they are not always feasible. There are a number of reasons for this. A main reason is the lack of executable models, which, together with the accompanying issue of formal semantics, are hot research topics within the academic UML community, see, e.g., [10]. Many proposals to define formal semantics for various kinds of UML diagrams are published, e.g. [4, 14, 26], and some implementations exist, but none are currently widely accepted and certainly none are standardised. Other reasons which may render UML behavioural diagrams insufficient include the dependencies and bookkeeping issues discussed in Section 5, which both are of a general, often encountered, and severe nature.

6.2 CPN in the Software Industry

A number of industrial CPN projects have taken place, see, e.g., [43] or [24]. Typically, the initiators of a project is a group of CPN experts from a university or another research institution, who establishes a cooperation with a small number of software engineers from some company. Often, such a project happens in

isolation, in the sense that it is carried out, some experiences are gained, and a report or paper is written. However, when the project ends and the CPN experts walk away, the use of CPN within the company is in many cases discontinued. CPN seems to have a problem with manifesting itself broadly in the software industry.

If we want to advocate wide-spread and long-term use of CPN in some company, we have to convince not only the software developers, but also some higher-level decision makers like business managers or project managers. In conversations with the latter, we should stress the key business question: How does my company save time and money by using CPN? Sometimes, we should perhaps talk about reducing time to market, increasing return of investment, and limitation of risks, instead of about, e.g., nice theoretical properties like formal semantics. Moreover, in many cases, the arguments should stress CPN as a powerful vehicle to investigate behaviour and dynamic properties by simulation, much more than CPN as a means for formal verification. The latter is used and appreciated in a number of specialised application areas, but is far beyond interest and reach in very many software companies.

One approach to ensure continuous use of CPN in a company could be to define a step in the company's software development process, where complex components are identified and afterwards modelled and simulated using CPN. A written software development process is demanded on higher maturity levels of the Capability Maturity Model, CMM [33], which is gaining momentum in the industry right now. However, a proposal to add a step to a software development process has a high risk of being received with hesitation and eventually be rejected. Real-life software development projects are often characterised with extremely tight budgets and very hard deadlines, and decision makers and company management may think that introducing an additional step in the development process will do exactly the opposite of saving time and money.

A potential, genuine disadvantage of CPN in otherwise UML-based software engineering is that it indeed is a deviation from an established standard. However, UML itself allows various kinds of tailoring (in terms of what is known as constraints, tagged values, and stereotypes) to accommodate the language to situations where the standardised version is not applicable. In many cases, it may well be that the use of CPN is not a more dramatic deviation than the tailoring approved by the UML standard.

A severe drawback of using CPN is that the step from software design, i.e., CPN models, to implementation in an object-oriented programming language typically must be done manually. In the PHM project, the implementation of the CPN design models will involve manual coding. This is a general drawback of CPN, whose elaborated data type concept often is an advantage when creating models, but on the other hand makes automatic code generation from CPN models more complicated than, e.g., code generation from various versions of state machines and statecharts, for which there are many code generators available, e.g., [38]. Therefore, when use of CPN is considered in a software development project, a trade-off must be made between the desire to have strong, executable design models on one side, and ease of implementation on the other. With this remark, it is not said that automatic code generation from CPN models is impossible. CPN and Design/CPN have been used for automatic code generation to other target languages than object-oriented ones, even in industrial settings [29].

7 Conclusions

The use of UML in conjunction with Petri nets has been studied intensively in recent years. Much of the work done is concerned with automatic translation from certain types of UML diagrams into Petri nets, often aimed at formal verification, e.g., [36]. Also, Petri nets have been used to give precise execution semantics to different classes of UML diagrams, e.g., [3]. A small number of examples of combined use of UML and CPN in software development have appeared in the literature, e.g., design of user interfaces is described in [11]. High-level Petri nets in general in conjunction with mobile computing has been investigated in [31], where, again, the focus is on formal verification. Much research has addressed development of concepts and theories that combine the ideas of object-orientation in general (not just UML) and Petri nets [1]. Specifically in [27], Object Petri Nets are defined, which extend CPN with object-oriented features like inheritance and polymorphism. Other examples on work in this area are [6, 15, 28].

In this paper, we have taken the practitioner's pragmatic attitude, by suggesting to apply both UML and CPN in the design of the PHM system, by taking immediate advantage of the best of both worlds. More specifically, we propose using UML to describe static aspects and using CPN as a supplement to model behavioural aspects of complex components. We believe that this proposal carries over to projects, where there really is a need for executable models and a fine-grained investigation of behaviour. In [12], it is stated that UML lacks facilities to model the interaction between system components in a sufficiently fine-grained way. This general-term observations is sustained in this paper, where we have pointed out issues that are addressed better with CPN than with UML. The problems with modelling of dependencies and bookkeeping in UML arise exactly because the UML behavioural diagrams are too coarse-grained. Moreover, in this paper, we have discussed how to position CPN in the UML-dominated world of software engineering. In particular, we have addressed a question posed in [39], where it is recognised that one of the key challenges in promoting Petri nets is to find the right projects and the right development phases to apply Petri nets in otherwise UML-dominated software engineering. One type of projects where Petri nets may be used successfully could be design of systems to support pervasive and mobile computing. Such systems are characterised by classical and well known complications that apply to many distributed systems [9], plus a number of new behavioural problems to be tackled, e.g., regarding mobility. Moreover, off-the-shelf standard middleware components supporting pervasive and mobile computing architectures are not well-established on the market yet.

With the success of UML, the software industry has in large scale adopted modelling as such as a valuable discipline, and today, modelling is generally accepted as a natural ingredient in everyday software development. Many software developers appreciate UML class diagrams as a productive asset to help them in their work. Those who have tried also to model behavioural aspects in UML might have encountered problems such as the ones discussed in this paper. Therefore, for many developers, the motivation to use a supplementary modelling language together with UML may be quite high. In this way, the success of UML can be seen as a good chance to establish CPN more broadly in the software industry.

References

[1] G. Agha, F.D. Cindio, and G. Rozenberg, editors. *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[2] J. Bardram and H.B. Christensen. Middleware for Pervasive Healthcare, A White Paper. In *Workshop on Middleware for Mobile Computing*, Heidelberg, Germany, 2001.

[3] L. Baresi and M. Pezzé. On Formalizing UML with High-Level Petri Nets. In Agha et al. [1].

[4] M.v.d. Beeck. Formalization of UML-Statecharts. In Gogolla and Kobryn [16].

[5] M. Berg. Accumulating and Cordinating: Occasions for Information Technologies in Medical Work. In *Computer Supported Cooperative Work*, volume 8, 1999.

[6] O. Biberstein, D. Buchs, and N. Guelfi. Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 Formalism. In Agha et al. [1].

[7] J. Burkhardt, H. Henn, S. Hepper, K. Rintdorff, and T. Schäck. *Pervasive Computing – Technology and Architecture of Mobile Internet Applications*. Addison-Wesley, 2002.

[8] F. Buschmann, R. Meunier, H. Rohnert, and P. Sommerlad. *Pattern-Oriented Software Architecture*. John Wiley and Sons, 1996.

[9] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems – Concepts and Design*. Addison-Wesley, 2001.

[10] W. Damm. Understanding UML, Pains and Rewards. In Gogolla and Kobryn [16].

[11] M. Elkoutbi and R.K. Keller. User Interface Prototyping Based on UML Scenarios and High-Level Petri Nets. In Nielsen and Simpson [30].

[12] G. Engels, R. Heckel, and S. Sauer. UML – A Universal Modeling Language? In Nielsen and Simpson [30].

[13] H.E. Eriksson and M. Penker. *UML Toolkit*. John Wiley and Sons, 1998.

[14] R. Eshuis and R. Wieringa. An Execution Algorithm for UML Activity Graphs. In Gogolla and Kobryn [16].

[15] H. Giese, J. Graf, and G. Wirtz. Closing the Gap Between Object-Oriented Modeling of Structure and Behaviour. In R. France and B. Rumpe, editors, *≪UML≫ 1999 - The Unified Modeling Language, 2th International Conference*, volume 1723 of *Lecture Notes in Computer Science*, Fort Collins, Colorado, 1999. Springer-Verlag.

[16] M. Gogolla and C. Kobryn, editors. *≪UML≫ 2001 - The Unified Modeling Language, 4th International Conference*, volume 2185 of *Lecture Notes in Computer Science*, Toronto, Canada, 2001. Springer-Verlag.

[17] U. Hansmann, L. Merk, M.S. Nicklous, and T. Stober. *Pervasive Computing Handbook*. Springer Verlag, 2001.

[18] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8, 1987.

[19] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering*, 5(4), 1996.

[20] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts:The STATEMATE Approach*. McGraw-Hill, 1998.

[21] A. Helal, B. Haskell, J.L. Carter, R. Brice, D. Woelk, and M. Rusinkiewicz. *Any Time, Anywhere Computing – Mobile Computing Concepts and Technology*. Kluwer Academic Publishers, 1999.

[22] ITU-T Recommendation Z.120: Message Sequence Chart. International Telecommunication Union; Telecommunication Standardization Sector (ITU-T), 1999.

[23] K. Jensen. *Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts*. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 1992.

[24] K. Jensen. *Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use. Volume 3, Practical Use*. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 1997.

[25] L.M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2), 1998.

[26] S. Kuske. A Formal Semantics of UML State Machines Based on Structured Graph Transformations. In Gogolla and Kobryn [16].

[27] C.A. Lakos. Object-Oriented Modelling with Object Petri Nets. In Agha et al. [1].

[28] C. Maier and D. Moldt. Object Coloured Petri Nets – A Formal Technique for Object Oriented Modelling. In Agha et al. [1].

[29] K.H. Mortensen. Automatic Code Generation Method Based on Coloured Petri Net Models Applied on an Access Control System. In Nielsen and Simpson [30].

[30] M. Nielsen and D. Simpson, editors. *21st International Conference on Application and Theory of Petri Nets 2000*, volume 1825 of *Lecture Notes in Computer Science*, Aarhus, Denmark, 2000. Springer-Verlag.

[31] L. Ojala, N. Husberg, and T. Tynjälä. Modelling and Analysing a Distributed Dynamic Channel Allocation Algorithm for Mobile Computing Using High-Level Net Methods. In K. Jensen, editor, *Workshop on Practical Use of High-level Petri Nets*, Aarhus, Denmark, 2000.

[32] OMG Unified Modeling Language Specification, Version 1.4. Object Management Group (OMG); UML Revision Taskforce, 2001.

[33] M.C. Paulk, C.V. Weber, and B. Curtis. *The Capability Maturity Model: Guidelines for Improving the Software Process.* Addison Wesley, 1995.

[34] J. Rumbaugh. The Preacher at Arrakeen. In Gogolla and Kobryn [16].

[35] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison Wesley, 1999.

[36] J. Saldhana and S.M. Shatz. UML Diagrams to Object Petri Net Models: An Approach for Modeling and Analysis. In *International Conference on Software Engineering and Knowledge Engineering*, Chicago, Illinois, 2000.

[37] A.J.H. Simons and I. Graham. 30 Things That Go Wrong in Object Modelling with UML 1.3. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems.* Kluwer Academic Publishers, 1999.

[38] J. Staunstrup. *IAR visualSTATE Concept Guide, version 4.* IAR Systems, 1999. www.iar.com.

[39] G. Wirtz. Application of Petri Nets in Modelling Distributed Software Systems. In D. Moldt, editor, *Workshop on Modelling of Objects, Components, and Agents*, Aarhus, Denmark, 2001.

[40] J. Xu and J. Kuusela. Analyzing the Execution Architecture of Mobile Phone Software with Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2), 1998.

[41] Aarhus Amt Electronic Patient Record. www.epj.aaa.dk.

[42] Centre for Pervasive Computing. www.pervasive.dk.

[43] Coloured Petri Nets at the University of Aarhus. www.daimi.au.dk/CPnets.

[44] Design/CPN. www.daimi.au.dk/designCPN.

[45] I-Logix. www.ilogix.com.

[46] Object Management Group. www.omg.org.

[47] Pervasive Healthcare. www.healthcare.pervasive.dk.

[48] Rational Software Corporation. www.rational.com.

[49] Systematic Software Engineering A/S. www.systematic.dk.

[50] Unified Modeling Language. www.uml.org.

Performance Study of Distributed Generation of State Spaces Using Colored Petri Nets

W.M. Zuberek

Department of Computer Science
Memorial University
St.John's, Canada A1B 3X5

Abstract

The performance of many distributed applications depends upon the ratio of computation to communication times. In the case of distributed generation of state spaces for timed Petri nets, this ratio is determined by the partitioning function which assigns generated states to classes associated with processors; if the partition classes correspond to clusters of states with only a few connections between clusters, the required communication is minimized, and the performance is maximized. The effects of state clustering are analyzed by simulating a colored timed Petri net modeling the distributed state space generation.

1. Introduction

Actual implementation of complex, real-world systems is usually preceded by thorough studies, performed on a formal model of the original system. For systems which exhibit concurrent activities, Petri nets are a popular choice of the modeling formalism, because of their ability to express concurrency, synchronization, precedence constraints and nondeterminism. Moreover, Petri nets "with time" (stochastic or timed) include the durations of modeled activities and this allows to study the performance aspects of the modeled system [1, 14, 21].

The basic analysis of net models, based on exhaustive generation of all reachable states, is known as reachability analysis. In reachability analysis, the states of the model and the transitions between states are organized in the so called reachability graph which is used for verifying the required qualitative properties (such as absence of deadlocks or liveness). For timed and stochastic Petri nets (with deterministic or exponentially distributed times), this graph is a Markov chain, so its stationary probabilities of states can be determined using known numerical methods [19]. These stationary probabilities are used to derive performance measures of the model [1, 6, 21].

For large net models, the state space can easily exceed the resources of a single computer system. The availability of clusters of (inexpensive) workstations and portable libraries for distributed computing make distributed generation of the state space quite an attractive alternative to the traditional sequential approach.

In distributed generation of the reachability graph, the (yet unknown) state space is partitioned into n disjoint regions, $R_1, R_2, ..., R_k$, and these regions are constructed independently by k identical processes running concurrently on different machines. At the end, the regions can be integrated in one state graph if needed.

There are several approaches to the distributed generation of reachability graphs. Many results and implementation details for parallel reachability graph generation on shared–memory multiprocessors are given in [2, 3, 4, 5]. In shared–memory systems, however, the

processors are tightly connected, so inter–processor communication is quite efficient, which makes such systems significantly different from clusters of workstations or PCs. Distributed state space generation described in [12] is organized into a sequence of phases, and each phase contains processing of currently available states followed by communication in which non-local states are sent to their regions (i.e., processors). The next phase begins only after completion of all operations of the previous phase. Reasonable values of speedup were reported for small numbers of processors (2 to 4) and for large state spaces.

The purpose of this paper is to present a simple colored Petri net model of a generation of the reachability graphs for timed Petri nets using a cluster of processors connected by a switch, and to illustrate the effects of state space partitioning on the performance of distributed generation. In particular, it has been observed that for some types of timed Petri nets, the straightforward partitioning algorithm [16, 17] results in poor speedups, so a more sophisticated partitioning algorithm, taking into account structural properties of the model, may be required to improve the performance of distributed state space generation.

Section 2 provides a brief overview of basic concepts of timed Petri nets which are relevant to this paper. Section 3 outlines the distributed generation of the state space for timed Petri nets. A simple colored timed Petri net model of a cluster of processors connected by a switch is presented in Section 4, while Section 5 discusses the results obtained from the presented model. A simplified approach to performance evaluation is outlined in Section 6. Section 7 contains several concluding remarks.

2. Timed Petri Nets

A timed Petri net is a triple $\mathcal{T} = (\mathcal{M}, c, f)$ where $\mathcal{M}$ is a marked (place/transition) net, $\mathcal{M} = (P, T, A, m_0)$, with P denoting the set of places, T – the set of transitions, A – the set of directed arcs, $A \subseteq P \times T \cup T \times P$, and m_0 – the initial marking function, $m_0 : P \to \{0, 1, 2, ...\}$. $\mathcal{M}$ can be extended in a number of ways if needed; for example, it can include a set of inhibitor arcs $B \subset P \times T$, $\mathcal{M} = (P, T, A, B, m_0)$, $A \cap B = \emptyset$, or multiple arcs described by a weight function $w : A \to \{1, 2, ...\}$, $\mathcal{M} = (P, T, A, w, m_0)$, and so on. c is a conflict-resolution function which assigns probabilities to conflicting transitions, $c : T \to [0, 1]$, in such a way that for each free-choice class of transitions the sum of assigned probabilities is equal to 1. For more general conflicts, the probabilities assigned to transitions represent relative frequencies of transition firings, which are used to determine the probabilities of state transitions [10]. f is the firing time function which assigns the firing times to transitions, $f : T \to \mathbf{R}^+$, where $\mathbf{R}^+$ denotes the set of nonnegative real numbers; if firing times are constant, the nets are called D–timed; if they are random variables with (negative) exponential distributions (described by their average values), the nets are called M–timed (or Markovian); other distributions can also be used.

In timed nets, it is assumed that each transition which can fire, initiates its firing in the same instant of time in which it becomes enabled, and the firings of transitions occur in "real time", i.e., the tokens are removed from the input places at the beginning of the firing interval, and they are deposited into the output places at the end of this interval. Consequently, the behavioral description of a timed net must take into account the places (i.e., the remaining tokens) as well as the firing transitions of a net [21]. For D–timed nets, a state of a net, s, is a triple, $s = (m, n, r)$, where m is a marking function, $m : P \to \{0, 1, 2, ...\}$, n is a firing function, $n : T \to \{0, 1, 2, ...\}$, which indicates, for each transition, the number of its firings occurring in this state, and r is a (partial) remaining-firing-time function, $r : T \to (\mathbf{R}^+)^*$, which indicates, for each firing of each transition, the remaining time of firing (D–timed nets do not have the memoryless property, so the r

components of state descriptions represents a “history” of firings; for M–timed nets, this last component of state descriptions is not used at all).

For example, let the firing times associated with the transitions of the D–timed net shown in Fig.2.1 be $f(t_1) = 1.0$, $f(t_2) = 0.9$, $f(t_3) = f(t_4) = 0.1$, and $f(t_5) = 0.8$.

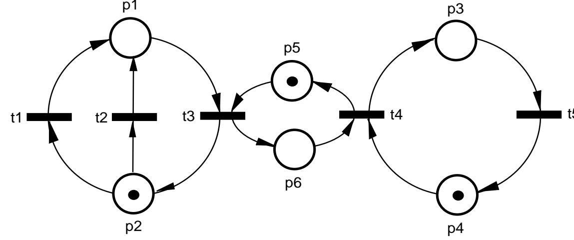


Fig.2.1. D–timed net.

For the initial marking shown in Fig.2.1, the net has two initial states, one corresponding to t_1's firing and the second corresponding to t_2's firing; these two initial states are as follows (the m, n and r components of states are separated by semicolons; undefined values of r are indicated by '–'):

$$s_1 = (0,0,0,1,1,0;1,0,0,0,0;1.0,-,-,-,-),$$
$$s_2 = (0,0,0,1,1,0;0,1,0,0,0;-,0.9,-,-,-).$$

The holding times (or durations) of s_1 and s_2 are equal to 1.0 and 0.9 (time units) respectively; at the end of the firing time, a token is deposited in p_1, so t_3 can initiate its firing, which is represented by s_3:

$$s_3 = (0,0,0,1,0,0;0,0,1,0,0;-,-,0.1,-,-).$$

The holding time of s_3 is 0.1, at the end of which tokes are deposited in p_2 and p_6, enabling t_1, t_2 and t_4, so there are two possible next states, s_4 with t_1 and t_4 firing concurrently, and s_5 with t_2 and t_4 firing concurrently:

$$s_4 = (0,0,0,0,0,0;1,0,0,1,0;1.0,-,-,0.1,-),$$
$$s_5 = (0,0,0,0,0,0;0,1,0,1,0;-,0.9,-,0.1,-).$$

Both these states have holding times equal to 0.1, at the end of which the firing of t_4 ends (while t_1 or t_2 continues its firing, with the remaining firing time equal to 0.9 and 0.8, respectively), and t_5 initiates its firing, creating states s_6 (from s_4) and s_7 (from s_5):

$$s_6 = (0,0,0,0,1,0;1,0,0,0,1;0.9,-,-,-,0.8),$$
$$s_7 = (0,0,0,0,1,0;0,1,0,0,1;-,0.8,-,-,0.8).$$

The holding times of s_6 and s_7 are both 0.8. At the end of s_6, t_1 continues to fire (with the remaining firing time equal to 0.1) creating state s_8:

$$s_8 = (0,0,0,1,1,0;1,0,0,0,0;0.1,-,-,-,-),$$

while the end of s_7 corresponds to the termination of its both firings, tokens are deposited in p_1 and p_4, and t_3 can initiate its firing; this is s_3 again.

The state graph (or the reachability graph) for this example is shown in Fig.2.2 with the state holding times shown in parentheses).

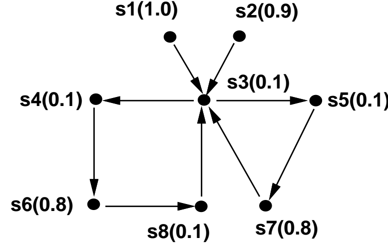


Fig.2.2. State graph for the net shown in Fig.2.1.

It should be noticed that for different values of firing times, the reachability graph for the net shown in Fig.2.1 may be significantly different from the one shown in Fig.2.2.

Timed colored Petri nets are extensions of place/transitions timed nets; a timed colored Petri net is also a triple $\mathcal{T} = (\mathcal{M}, c, f)$ where $\mathcal{M}$ is a colored net, $\mathcal{M} = (P, T, C, A, e, m_0)$, with P, T and A as before, C denoting a set of attributes called "colors", e assigning expressions (over colors and color variables) to arcs of the set A, the initial marking function extended to $m_0 : P \times C \rightarrow \{0, 1, 2, ...\}$, and the choice and firing time functions extended to: $c : T \times B \rightarrow [0, 1]$ and $f : T \times B \rightarrow \mathbf{R}^+$, where B denotes the set of bindings, i.e., assignments of colors from the set C to variables used in arc expressions.

An outline of a "standard" approach to the (sequential) generation of state space (just the states, without arcs) can be as follows:

```
 1. State space generation:
 2. var States := ∅;          (* set of states *)
 3.     unexplored := ∅;   (* queue of states *)
 4. begin
 5.     for each s in IntitalStates(m0) do
 6.         States := States ∪ {s};
 7.         insert(unexplored, s)
 8.     endfor;
 9.     while nonempty(unexplored) do
10.         state := remove(unexplored);
11.         for each s in NextStates(state) do
12.             if s ∉ States then
13.                 States := States ∪ {s};
14.                 insert(unexplored, s)
15.             endif
16.         endfor
17.     endwhile
18. end.
```

where function *IntialStates(m)* determines the set of initial states corresponding to the marking function m, and function *NextStates(s)* determines the set of successor states of s.

3. Distributed State Space Generation

In distributed generation of the state space, the partitioning function assigns each state to the region to which it belongs. The number of disjoint regions is usually equal to the number of available processors, k_p.

A straightforward partitioning function is based on the definition of the state, similarly to the one used in [12]. For timed nets, however, the partitioning function also takes into account the firing transitions:

$$region(s) = [\sum_{i=0}^{|P|} \alpha_i \ m(p_i) + \sum_{i=0}^{|T|} \beta_i \ n(t_i)] \bmod (k_p)$$

where $|P|$ is the cardinality of the set of places P, $|T|$ is the cardinality of the set of transitions T, the coefficients α_i and β_i are integer numbers, and m and n are marking and firing components of a state s [21].

Three kinds of (logical) processes are used in distributed generation of the state space [16], as shown in Fig.3.1: a process starting the distributed system and initiating the computations, called *Spawner*; several processes constructing the regions of the state space, called *Generators*, and a process collecting and integrating the results, called *Collector*.

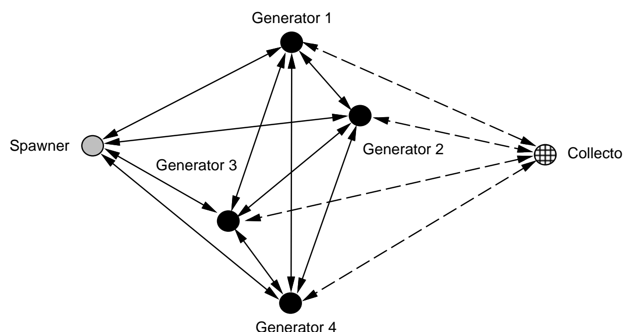


Fig.3.1. Distributed state space generation system.

The execution begins with the *Spawner* which creates the *Collector* and spawns k_p *Generators* on the hosts of the cluster; it also collects the identifiers of all created processes and broadcasts them to all processes so each process can send messages to any other one [16]. In its final phase, the *Spawner* sends the initial states to the appropriate *Generators*.

An outline of the *Spawner* process is as follows [16]:

```
 1. Spawner:
 2. var m0;                  (* initial marking *)
 3.     k;                   (* the number of hosts *)
 4.     proc_table[];        (* processor identifiers *)
 5. begin
 6.     input virtual machine and model descriptions;
 7.     spawn Collector on this_host;
 8.     for i := 1 to k do
 9.         proc_table[i] :=spawn Generator on host[i]
10.     endfor;
11.     broadcast(proc_table);
12.     for each s in InitialStates(m0) do
13.         send(proc_table[region(s)], s)
14.     endfor
15. end.
```

For each state belonging to region R_i, the $Generator_i$ determines all successor states. A successor state can be in the same region (in which case it is called a *local* state) or in a different region (in which case it is called an *external* state). Each $Generator_i$ sends all external states to the *Generators* determined by the partitioning function.

In order to perform state processing concurrently with communication, each *Generator* is composed of three processes: the *Analyzer*, responsible for processing the states, the

Sender, responsible for sending messages to other processes, and the *Receiver*, responsible for receiving messages from other processes and for the termination detection. When the *Spawner* creates the *Generators*, it actually creates *Analyzer* processes. As its first step, each *Analyzer* creates its *Receiver* and *Sender* processes [16].

The *Analyzer*, *Receiver*, and *Sender* processes of each *Generator* reside on the same processor. Their communication is based on shared variables, as shown fin Fig.3.2.

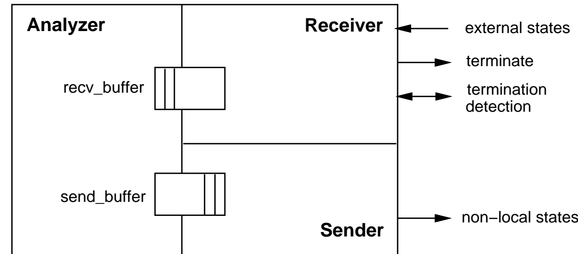


Fig.3.2. The structure of a *Generator*.

Each *Generator* processes the states from the internal queue *unexplored* (local states) and from *recv_buffer* (non–local states), with non–local states taking priority over local ones.

```
 1. Analyzer_i:
 2. var States_i := ∅;              (* set of states *)
 3.         unexplored := ∅;        (* queue of states *)
 4.         cont := true;           (* continuation flag *)
 5. begin
 6.         spawn Receiver, Sender on this_host;
 7.         while cont do
 8.               if empty(recv_buffer) ∧ nonempty(unexplored) then
 9.                     state := remove(unexplored);
10.                     new := true
11.               else
12.                     state :=get(recv_buffer);
13.                     if state = null then
14.                           cont :=false
15.                     else
16.                           new := state ∉ States_i;
17.                           if new then
18.                                 States_i := States_i ∪ {state}
19.                           endif
20.                     endif
21.               endif;
22.               if cont ∧ new then
23.                     for each s in NextStates(state) do
24.                           if region(s) = i then
25.                                 if not s ∉ States_i then
26.                                       States_i := States_i ∪ {s};
27.                                       insert(unexplored, s)
28.                                 endif
29.                           else
30.                                 put(send_buffer, s)
31.                           endif
32.                     endfor
33.               endif
34.         endwhile
35. end.
```

An important aspect of distributed applications is the termination condition which identifies the situation when no processor can continue its execution. The completion of all current tasks by any one of processors does not imply the overall termination, as some other processors may still continue and generate new states for processing. On the other hand, the processors cannot just wait for each other forever. A global termination detection algorithm [8] is interleaved with the computations, repeatedly checking if all processors have finished their tasks [16]. When global termination is detected (i.e., when all *Analyzer* processes are waiting on *get* operation – line 12), a special *null* state is sent to all *Analyzer* processes to terminate their operation (lines 13, 14).

Processes residing on different hosts constitute a "virtual machine"; they communicate by exchanging messages using the popular PVM (Parallel Virtual Machine) message passing library [9].

4. Petri Net Model

An outline of a cluster of (four) workstations connected to a switch is shown in Fig.4.1 (typically clusters contain more workstations, e.g., 16 or 32).

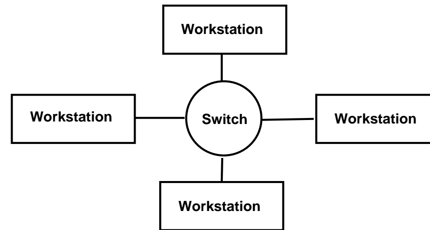


Fig.4.1. An outline of a cluster of 4 workstations.

A Petri net model of a single processor connected to a switch, with independent processes for state generation and for message passing to other processors, is shown in Fig.4.2.

The processing of states is represented by transition t_{ip} with the average time of processing a single state assigned to it as the firing time. The queue of states waiting for processing is represented by place p_{ir}, which, in Fig.3.2, contains 2 tokens (i.e., 2 states waiting for processing).

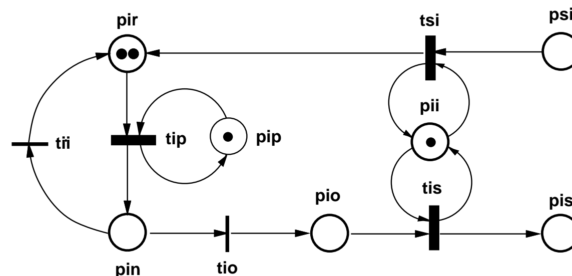


Fig.4.2. Petri net model of a processor and its link.

Transitions t_{is} and t_{si} represent message passing to and from the switch, respectively. Since the messages share the same link, place p_{ii} is a shared resource (the link) which can be used either for sending (t_{is}) or for receiving (t_{si}) a message.

For a 4–machine cluster (Fig.4.1), the switch connecting the processors is outlined in Fig.4.3; each of messages incoming from four directions (p_{1s}, p_{2s}, p_{3s} and p_{4s}) has a free–choice structure which forwards the message to one of the other processors connected to

the switch (more precisely, the messages are first sent to p_{si}, which represents a queue of messages waiting for forwarding to processor i; the messages are forwarded from p_{si} to the processor, p_{ir}, one message at a time, as shown in Fig.4.2). If the states are uniformly distributed over the regions, all free–choice probabilities associated with selections within the switch are equal to $1/(k_p - 1)$, so, in Fig.4.3, are equal to 1/3.

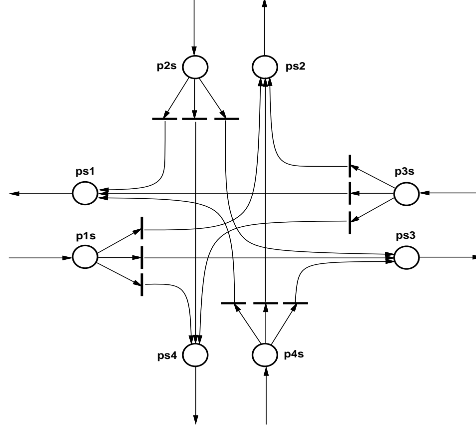


Fig.4.3. Petri net model of a switch.

In Fig.4.2, the result of processing a state is a new state, which, after termination of t_{ip}'s firing, is inserted into place p_{in}. If this new state is local, it is forwarded, by firing t_{ii}, to the waiting queue p_{ir} for further processing; if the new state is external, it is sent to p_{io} by firing t_{io}, and then to its target host by firing first t_{is} and then t_{sj} of the selected host j (the selection is made within the switch).

Place p_{in} is a free–choice place, and the selection of local or external state is described by free–choice probabilities associated with t_{ii} and t_{io}; for a cluster of k_p machines, assuming uniform distribution of states over regions, the free–choice probabilities are $1/k_p$ for t_{ii} and $(k_p - 1)/k_p$ for t_{io}. Since, within the switch, the free–choice transitions connected to places p_{is} have the free–choice probabilities equal to $1/(k_p - 1)$, all hosts are selected with the same probabilities equal to $1/k_p$.

The probability associated with t_{ii}, denoted $c(t_{ii})$, is an important parameter for the performance analysis of distributed state space generation. Its value greater than $1/k_p$ (the value corresponding to uniform distribution of states over the hosts) indicates "locality" of the partitioning function, created by clusters of states within regions.

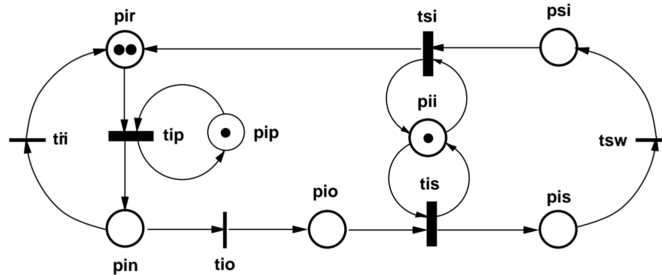


Fig.4.4. Colored Petri net model of a cluster.

Since the model of a complete cluster contains many identical (sub)models of processors, colored Petri nets [11] can be used to "fold" all these replications into a compact model of the cluster, as shown in Fig.4.4, in which the number of colors is equal to k_p, the number

of processors, and the switch is represented by a single transition t_{sw} with the number of occurrences equal to $k_p * (k_p - 1)$, grouped into k_p free–choice structures corresponding to the outline shown in Fig.4.3.

5. Performance Results

The performance of distributed applications depends upon several factors, which include the balancing of the workload among the processors, and also the amount of communication that is needed for the execution of distributed tasks.

The net model shown in Fig.4.4 contains two timing parameters, the firing time of t_{ip}, representing the (average) time required to process a single state, and the firing times of t_{is} and t_{si}, that represent the (average) time required for sending a single state description between a processor and the switch. Since the performance of the model depends upon the ratio of these two parameters rather than on their specific values, it is convenient to use just one parameter which characterizes model's temporal properties. The computation–to–communication ratio, $r_{comp/comm}$, is the ratio of the (average) state processing time to the (average) time needed for sending a single state description between two processors (which is assumed to be the same for all pairs of processors in a cluster).

In order to represent the behavior of the state generation process, some simplifying assumptions are made. It is assumed that a finite (but large) state space is generated in a "steady" manner, i.e., that the size of the *unexplored* queue in the scheme outlined at the end of Section 2, is approximately constant for the most part of the generation process. This assumption is not always valid, but is reasonable for many net models (in particular for models which contain buffers with large capacities, in which case the state space generation basically repeats similar groups of states for different "states" of the buffer(s)).

A practical consequence of the "steady" generation assumption is that each analyzed state generates one new state (i.e., a state can generate several next states, but, on average, only one of these states is new).

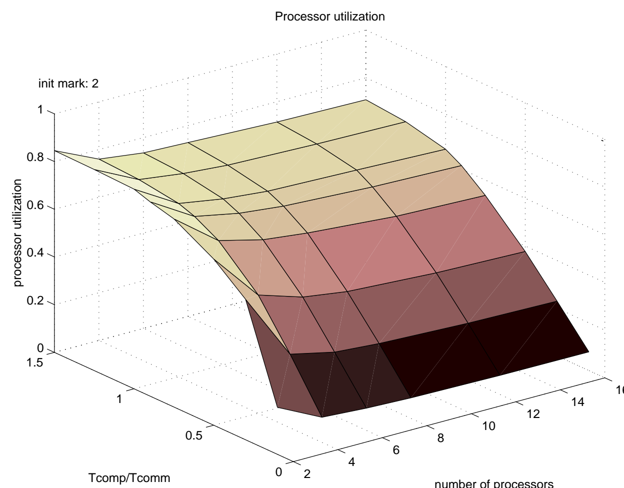


Fig.5.1. Utilization of processors for $m_0(p_{ir}) = 2$.

The workload of the system is controlled by the initial distribution of states among the processors (the initial marking of p_{ir} in Fig.4.2 and Fig.4.4). Two cases are analyzed: (i) when the same initial conditions are used for all processors, irrespective of their number (this is called "proportional load" as the total load is proportional to the number of processors), and (ii) the "shared load", when the (total) initial marking is the same for different numbers

of processors, so as the number of processors increases, the load assigned to each processor decreases.

For proportional load, the utilization of processors, for different values of $r_{comp/comm}$ (from 0.1 to 1.5) and for several different numbers of processors (from 2 to 16), is shown in Fig.5.1 for small workload (the initial marking of p_{ir} equal to 2), and in Fig.5.2 for high workload (the initial marking of p_{ir} equal to 8).

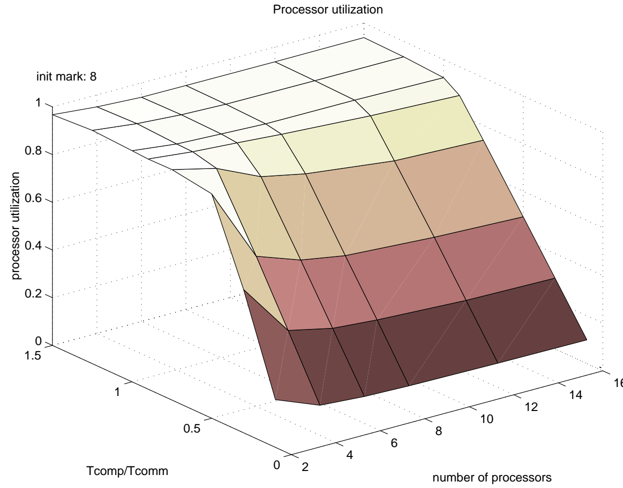


Fig.5.2. Utilization of processors for $m_0(p_{ir}) = 8$.

The results have similar character, and it can be observed that the utilization increases with the increase of the workload; Fig.5.2 shows some "saturation effects" for values of $r_{comp/comm}$ greater than 1. Moreover, for communication times significantly greater than the computation time (i.e., for $r_{comp/comm}$ less than 0.5), processor's utilization is consistently poor, below 50%, and practically linearly tends to 0 with the value of $r_{comp/comm}$.

For the case of shared load, similar results are shown in Fig.5.3 for small workload (the initial marking of 8 is shared among all the processors), and in Fig.5.4 for high workload (the initial marking of 32 is shared among the processors, so the plots for $k_p = 16$ in Fig.5.4 and Fig.5.1 are the same; both correspond to the initial marking of p_{ir} equal to 2).

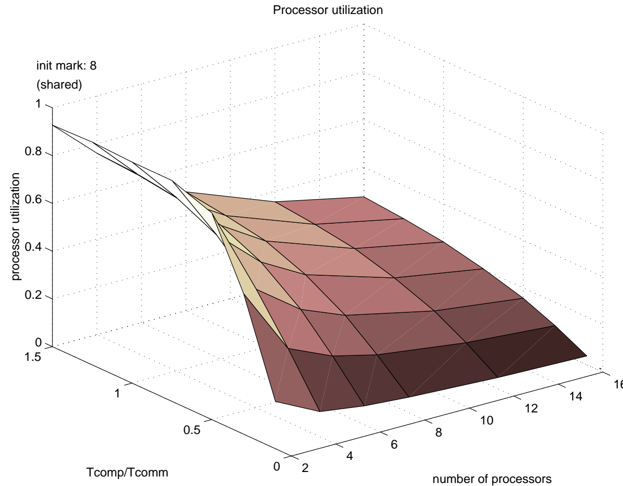


Fig.5.3. Utilization of processors, low shared load ($m_0 = 8$).

In the model shown in Fig.4.2, the (free–choice) probability associated with t_{ii}, $c(t_{ii})$,

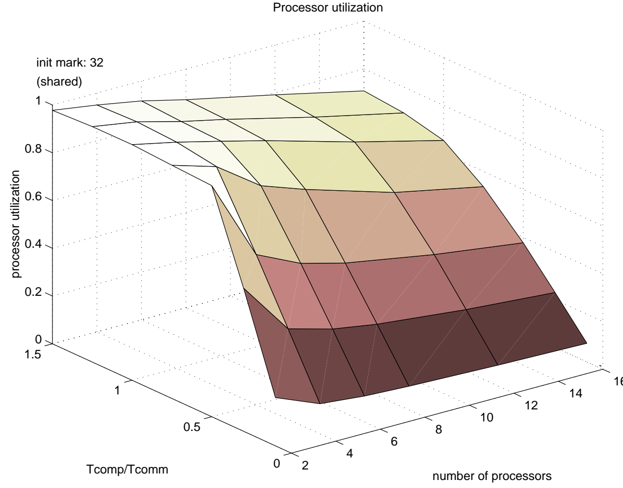


Fig.5.4. Utilization of processors, high shared load ($m_0 = 32$).

represents the probability that a new, generated state is a local one. Its default value, for uniform distribution of states over the nodes of the cluster of workstations, is $1/k_p$.

Fig.5.5 and Fig.5.6 correspond to Fig.5.1 and Fig.5.2 but assume that 50% of new states in each region are local states, while the remaining 50% are uniformly distributed over the other regions.

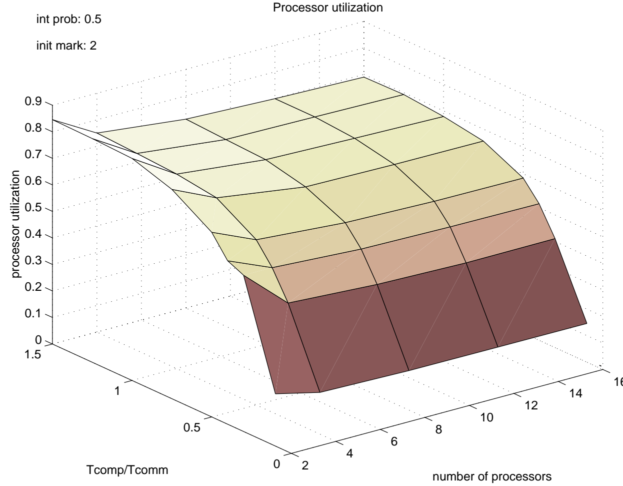


Fig.5.5. Utilization of processors for $m_0(p_{ir}) = 2$ and $c(t_{ii}) = 0.5$.

Increased values of $c(t_{ii})$ correspond to the clustering of states in the regions, i.e., the situations when many states generated by each processor are local states, so they do not require any communication, and this improves the utilization of processors. Increased utilization of processors results in increased speedup.

The speedup $S(k_p)$ of a k_p–processor system is usually defined as the ratio of execution time of an application on one processor, $T(1)$, to the application's execution time on k_p processors, $T(k_p)$ [20]:

$$S(k_p) = \frac{T(1)}{T(k_p)}.$$

For the ideal, uniform distribution of workload among the processors, the execution time on k_p processors can be expressed as:

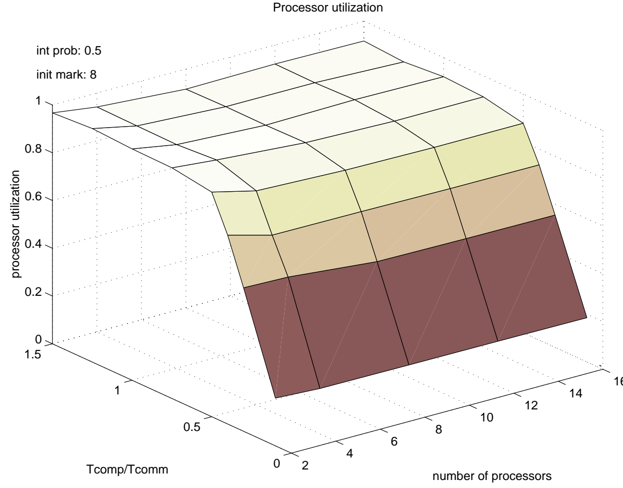


Fig.5.6. Utilization of processors for $m_0(p_{ir}) = 8$ and $c(t_{ii}) = 0.5$.

$$T(k_p) = \frac{T(1)}{k_p} \frac{1}{u_p(k_p)}$$

where $u_p(k_p)$ is the utilization of each processor in a k_p–processor system, and then the speedup is simply:

$$S(k_p) = k_p u_p(k_p).$$

Fig.5.7 and Fig.5.8 show the speedup as a function of the number of processors, k_p, for medium (proportional) workload ($m_0(p_{ir}) = 4$) and several values of $c(t_{ii})$, and for two different values of $r_{comp/comm}$; in Fig.5.7 $r_{comp/comm} = 1$, and in Fig.5.8 $r_{comp/comm} = 0.25$.

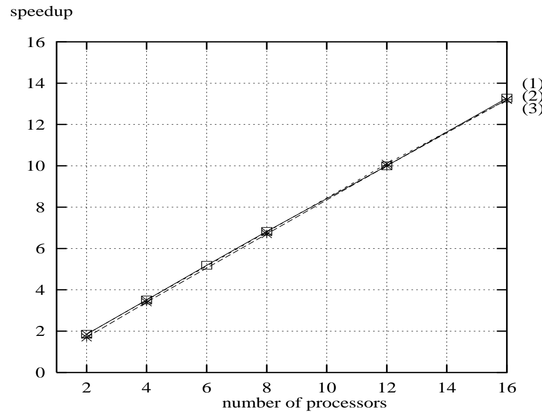


Fig.5.7. Speedup as a function of k_p; $r_{comp/comm} = 1.0$,
(1) $c(t_{ii}) = 1/k_p$, (2) $c(t_{ii}) = 0.5$, (3) $c(t_{ii}) = 0.75$.

It can be observed that, by increasing the value of $c(t_{ii})$, a significant improvement of the speedup can be obtained in cases when the communication is critical (i.e., $r_{comp/comm} < 0.5$), as shown in Fig.5.8; for $r_{comp/comm} \geq 1$ the effect of state clustering is quite negligible (Fig.5.7); in this case, the communication is not critical to the performance of the system. The value of $c(t_{ii})$, the probability that a new, generated state is local, depends primarily upon the partitioning function which assigns states to the regions of the distributed state

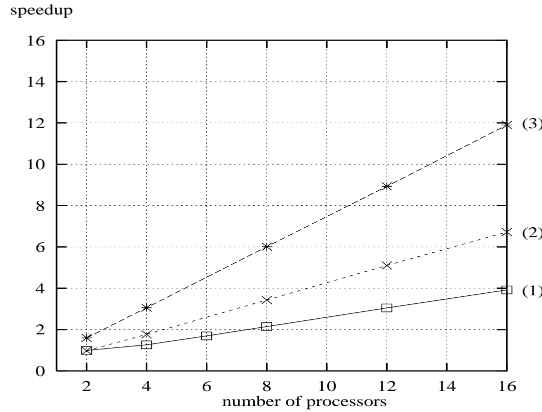


Fig.5.8. Speedup as a function of k_p; $r_{comp/comm} = 0.25$,
(1) $c(t_{ii}) = 1/k_p$, (2) $c(t_{ii}) = 0.5$, (3) $c(t_{ii}) = 0.75$.

space. Finding a partitioning function which takes the properties of the net model into account in a way that maximizes the value of $c(t_{ii})$, i.e., maximizes the clustering of states within regions, is an interesting area of research.

6. Simplified Performance Evaluation

In simulation–based performance evaluation, the simulation time depends more than linearly upon the size of the modeled cluster. A natural question is then if there is a need to model all the processors of the cluster to obtain the performance results.

For the shared workload, if the total token count of the initial marking is m_0 tokens, and the cluster is composed of k_p processors, the average load per processor is equal to m_0/k_p. The same average load per processor exists in a cluster with k'_p processors and with the initial marking containing $m_0 * k'_p/k_p$ tokens. Consequently, instead of simulating a 16–processor cluster with the total workload of 32 tokens, a 4-processor cluster can be simulated with the load of 8 tokens. Fig.6.1 and Fig.6.2 show the results obtained for a 4–processor cluster with the load adjusted to the case shown in Fig.5.3 and Fig.5.4.

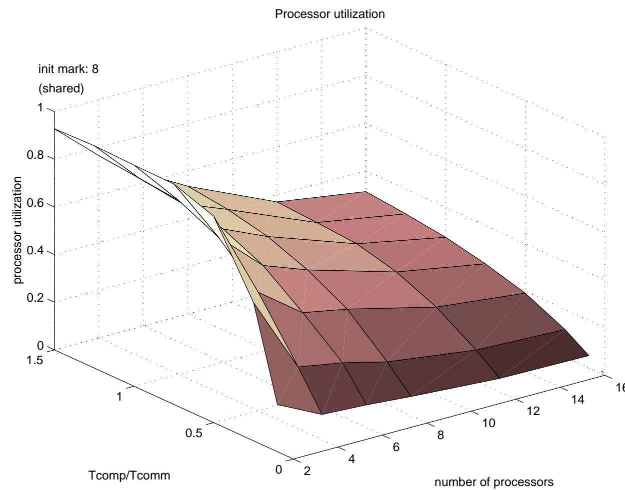


Fig.6.1. Utilization of processors, low shared load ($m_0 = 8$).

It can be observed that the results shown in Fig.6.1 and Fig.5.3 as well as Fig.6.2 and Fig.5.4 are practically the same.

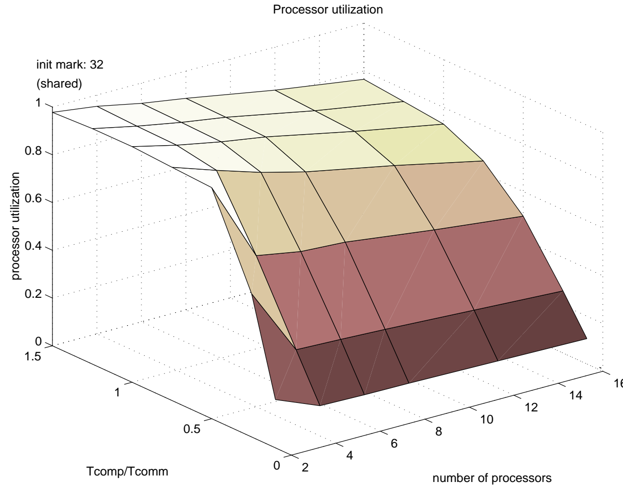


Fig.6.2. Utilization of processors, high shared load ($m_0 = 32$).

For proportional load, the performance is practically independent of the number of processors for k_p greater than 3, as can be observed in Fig.5.1 and Fig.5.2. This is due to the model of the switch which does not introduce any delays of forwarded messages (all delays are associated with the links).

7. Concluding Remarks

The paper shows that the performance of distributed state space generation can be improved, sometimes quite significantly, if the partitioning function assigns clusters of states to regions associated with independent processors. More specifically, if the communication is the bottleneck of a distributed application, i.e., if the computation to communication ratio is less than 1, any reduction of the amount of required communication improves the performance of the distributed application. However, designing a partitioning function with good locality properties is not an obvious task.

For example, the net model of the bounded buffer communication scheme, shown in Fig.7.1 (both producer and consumer are represented by simple free–choice structures), has a reachability graph with systematic structure, in which a section of the graph is repeated for each additional token in p_5 and/or p_6 (i.e., another unit of space in the buffer).

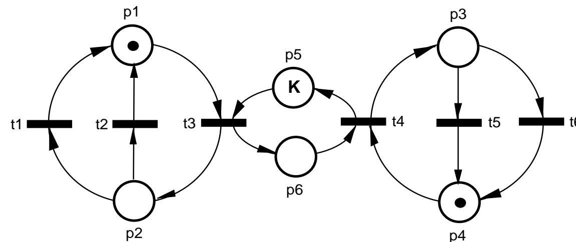


Fig.7.1. Net model of a bounded–buffer synchronization.

Fig.7.2 shows the initial part of the reachability graph for the model shown in Fig.7.1, with two different partitioning functions: (a) groups the nodes into regions with only a very few connections between the regions while (b) shows regions in which only a few nodes are connected within the same region, so there are very few local nodes. Case (a) has

significantly less communication than case (b), and the partitioning function in this case should take into account only the marking of place p_5 (or place p_6).

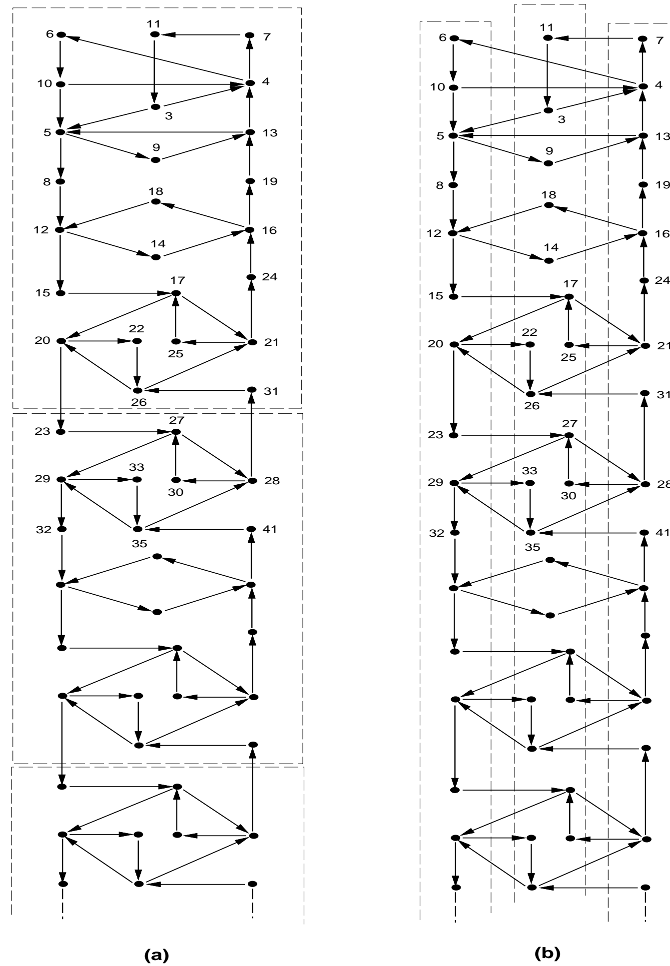


Fig.7.2. Part of the reachability graph for the net in Fig.7.1
with two different partitioning functions.

The very simple model of distributed state space generation, described in this paper, assumes that the temporal behavior can be described by constant times, i.e., that the analysis of each state requires the same amount of time, and that sending the state description from one processor to another also requires the same amount of time for each state. The first assumption (that the state processing time is constant) is not very realistic because the amount of (processing) time required for the generation of all "next states" depends upon the topology of the net (i.e., whether the current state is a conflict–free, free–choice or more general conflict state). However, it appears that the effects of variable state processing times are not very significant. Fig.7.3 shows the utilization of the processors for the case of shared load when the state processing time is exponentially distributed with the average value that is equal to the (constant) value used in Fig.5.3. It can be observed that the utilization shown in Fig.5.3 is slightly better than in Fig.7.3, but the differences are not significant.

Similarly, Fig.7.4 shows the utilization of processors for high shared load with random state processing times (also exponentially distributed), and the average value equal to that used for Fig.5.4. Again, the results shown in Fig.7.4 and Fig.5.4 are quite similar.

The assumption that the time required for sending a state description from one processor

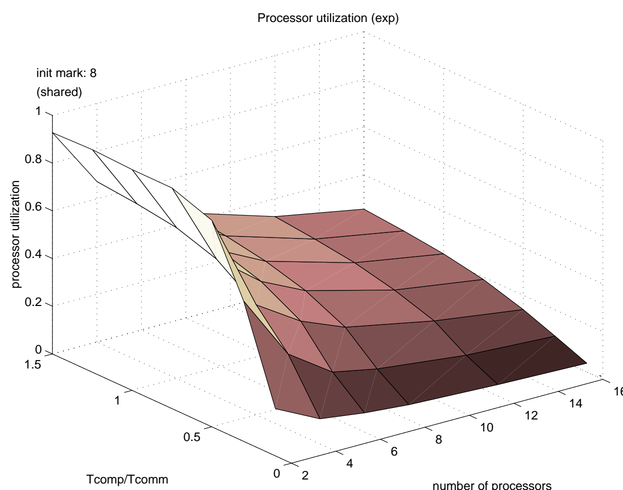


Fig.7.3. Utilization of processors, low shared load ($m_0 = 8$).

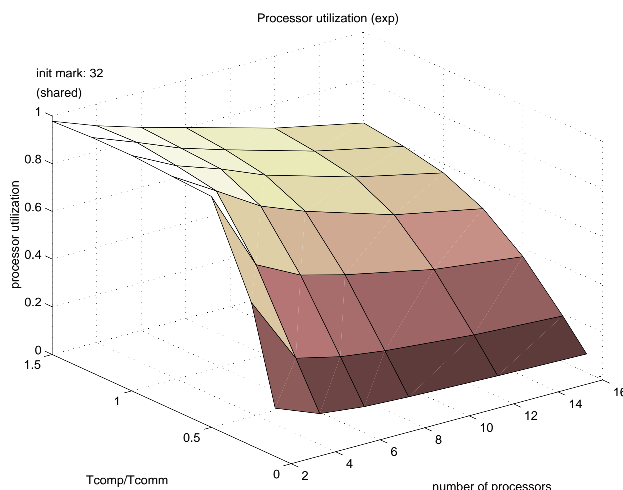


Fig.7.4. Utilization of processors, high shared load ($m_0 = 32$).

to another is the same for all states is based on typical characteristics of the message sending time as a function of message length which, for typical message–passing libraries (such as PVM or MPI) practically does not depend upon the message length for short (i.e., in the range of Kbytes) messages [18]; a representation of a state with 100 marked places and 100 firing transitions (which would typically correspond to a model with thousands of places and thousands of transitions) requires, without any "compaction", about 1.6 Kbytes assuming the "standard" 4-byte representation of integers.

In order to capture the steady–state behavior of distributed generation of the state space, it was assumed that each analyzed state generates one new state. For many net models the state space is not generated in such a steady way, and the number of "unexplored" states changes in a very broad range during the generation. It is expected that the effects of such variations can be represented by an extension of the presented mode, and then the (average) performance can be studied in a similar way.

The presented approach is not restricted to the discussed application; it can be used to represent the behavior of many other applications which use reachability analysis, such as model checking [7] or automated verification of discrete–event systems [13].

All simulations of net models were performed using the TPN-tools package [22], [23].

The simulation runs corresponded to processing at least 100,000 states (for the shared load cases). Although the confidence intervals were not determined for these experiments, the results are presented without any "smoothing" operations, which indicates rather small variances of the obtained results.

Acknowledgements

The Natural Sciences and Engineering Research Council of Canada partially supported this research through grant RGPIN-8222.

Several constructive and insightful comments and remarks of four anonymous reviewers, an expecially reviewer A, are gratefully acknowledged.

References

[1] M. Ajmone Marsan, G. Balbo, and G. Conte, 1984. "A class of generalized stochastic Petri nets for the performance evaluation of systems"; *ACM Transactions on Computer Systems*, vol.2, no.2, pp.93–122.

[2] S.C. Allmaier and G. Horton, 1997. "Parallel shared-memory state-space exploration in stochastic modeling"; *Solving Irregularly structured Problems in Parallel (IRREGULAR'97), Lecture Notes in Computer Science*, vol.1253, pp.207–218, Springer-Verlag.

[3] S.C. Allmaier, S. Dalibor, and D. Kreische, 1997. "Parallel graph generation algorithms for shared and distributed memory machines"; *Parallel Computing: Fundamentals, Applications and New Directions (Proc. of the Conference ParCo'97), Advances in Parallel Computing*, vol.12, pp.581–588, Elsevier, North-Holland.

[4] S.C. Allmaier, M. Kowarschik, and G. Horton, 1997. "State space construction and steady state solution of GPSN on a shared memory multiprocessor"; Proc. IEEE Int. Workshop Petri Nets and Performance Models (PNPM '97), pp.112–121.

[5] S.C. Allmaier and D. Kreische, 1999. "Parallel approaches to the numerical transient analysis of stochastic reward nets"; in *Application and Theory of Petri Nets 1999 (Proc. 20th International Conference, IACTPN'99)*, (Lecture Notes in Computer Science 1639), pp.147–167, Springer-Verlag.

[6] F. Bause and P. Krinzinger, 1996. *Stochastic Petri Nets - An Introduction to the Theory.* Vieweg.

[7] E.M. Clarke, O. Grumberg, D. Peled, 1999. *Model checking.* MIT Press.

[8] E. Dijkstra, W. Feijen, and A. van Gasteren, 1983. "Derivation of a termination detection algorithm for distributed computations"; *Information Processing Letters*, vol.16, no.5, pp.217–219.

[9] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, 1994. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial.* MIT Press.

[10] M.A. Holliday, M.K. Vernon, "Exact performance estimates for multiprocessor memory and bus interference"; *IEEE Trans. on Computers*, vol.36, no.1, pp.76–85, 1987.

[11] K. Jensen, 1987. "Coloured Petri nets"; in *Advanced Course on Petri Nets 1986* (Lecture Notes in Computer Science 254), pp.248–299, Springer–Verlag.

[12] P. Marenzoni, S. Caselli, and G. Conte, 1997. "Analysis of large GSPN models: a distributed solution tool"; Proc. IEEE Int. Workshop on Petri Nets and Performance Models (PNPM'97), pp.122–131.

[13] K.L. McMillan, 2000. "A methodology for hardware verification using compositional model checking", *Science of Computer Programming*, vol.37, no.1-3, pp.279-309.

[14] T. Murata, 1989. "Petri nets: properties, analysis, and applications"; *Proceedings of the IEEE*, vol.77, no.4, pp.541–580.

[15] J. Peterson, 1981. *Petri Net Theory and the Modeling of Systems.* Prentice Hall.

[16] I. Rada, 2000. "Distributed generation of state space for timed Petri nets"; M.Sc. Thesis, Department of Computer Science, Memorial University of Newfoundland, St.John's, Canada. o

[17] I. Rada, W.M. Zuberek, "Distributed generation of state space for timed Petri nets"; Proc. High Performance Computing Symposium 2001, Seattle, WA, pp.219–227, 2001.

[18] M.R. Steed, M.J. Clement, 1996. "Performance prediction of PVM programs"; Proc. 10-th Int. Parallel Processing Symposium (IPPS-96), pp.803-807.

[19] W. Stewart, 1994. *Introduction to the Numerical Solution of Markov Chains.* Princeton University Press.

[20] B. Wilkinson, 1996. *Computer Architecture – Design and Performance* (2-nd ed.). Prentice Hall.

[21] W.M. Zuberek, 1991. "Timed Petri nets, definitions, properties, and applications"; *Microelectronics and Reliability*, vol.31, no.4, pp.627–644.

[22] W.M. Zuberek, 1996. "Modeling using timed Petri nets – model description and representation"; Technical Report #9601, Department of Computer Science, Memorial University, St.John's, Canada A1B 3X5.

[23] W.M. Zuberek, 1996. "Modeling using timed Petri nets - event-driven simulation"; Techical Report #9602, Department of Computer Science, Memorial University, St.John's, Canada A1B 3X5,

Verification of Timed and Hybrid Systems

Kim G Larsen
BRICS
Aalborg University
Denmark

Abstract

UPPAAL is a tool for modelling, simulating and verifying real-time and hybrid systems, developed collaboratively by BRICS at Aalborg University and Department of Computer Systems at Uppsala University since the beginning of 1995 (see www.uppaal.com). The theoretical foundation of UPPAAL is that of timed automata and the early decidability results provided by Alur and Dill in 1990.

The presentation will contain a description and demonstration of the main components of newest release of UPPAAL. Particular focus will be on the evolution of the algorithms and datastructures underlying the tool and their dramatic impact on performance. The presentation will enclude recent work on exact acceleration and other abstraction techniques for tackling the problem of fragmentation experienced during model checking of real-time systems with hughly varying time-scales, e.g. in verification of the run-time behaviour of LEGO Mindstorms programs.

Sweep-Line State Space Exploration for Coloured Petri Nets*

Guy Edward Gallasch[1], Lars Michael Kristensen[1], and Thomas Mailund[2]

[1] Computer Systems Engineering Centre
University of South Australia
Mawson Lakes Campus, SA 5095, AUSTRALIA
Email: guy.gallasch@postgrads.unisa.edu.au, lars.kristensen@unisa.edu.au
[2] Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N, DENMARK
Email: mailund@daimi.au.dk

Abstract. The basic idea of the sweep-line state space method is to exploit a formal notion of progress found in many concurrent and distributed systems. Exploiting progress makes it possible to sweep through the state space of a CP-net while storing only a small fragment of the states in memory at any time. Properties of the system can then be verified on-the-fly during the sweep of the state space. This can lead to significant savings in peak memory usage and computation time. Examples of systems possessing progress are transport protocols, transactions protocols, workflow models, and systems modelled with Timed CP-nets. We present SWEEP/CPN, a library extension to DESIGN/CPN supporting the sweep-line method, and demonstrate its use.

Keywords: State space methods, State explosion problem, extensions to DESIGN/CPN, Verification and validation.

1 Introduction

State space exploration and analysis is a powerful way to investigate the correctness of distributed and concurrent systems, and is one of the main analysis methods for Coloured Petri Nets (CP-nets or CPNs) [11, 12]. The basic idea behind state space exploration is to construct a directed graph (called the state space, or the reachability/occurrence graph) representing all reachable states of the system and the transitions between these states. From this graph a large number of dynamic properties of the system can be analysed algorithmically. The main drawback of using state spaces for analysis of systems is the *state explosion problem:* even for systems of moderate complexity the number of reachable states can be astronomical, and storing the state space in the available computer memory might not be feasible.

In an attempt to alleviate the state explosion problem a number of state space reduction methods and techniques have been developed. These techniques can be split into three main classes. The first are those methods that represent the state space in a compact or condensed form. Symmetry reduction [4, 5, 13] is an example of this, where a representative state is used to represent a set of symmetric states. The second class of methods represent only a subset of the full state space. Such methods include partial order reduction methods [17, 20, 21]

* Supported by an Australian Research Council (ARC) Discovery Grant (DP0210524).

where only some orderings of independent events and not all orderings are stored. The reduction is done in such a way that the answers to verification questions can still be determined from the reduced state space.

The third class of reduction techniques involve deleting states or state information during state space exploration. Such methods include hash-compaction [18,22] and bit-state hashing [8,9] in which a hash-value is calculated from the state and only the hash-value – not the entire state – is stored. The state space caching method [6,10], is another method based on deleting information during exploration. State space caching exploits the fact that during a depth-first exploration of the state space, only the states on the depth-first stack need to be stored to ensure termination. States not on the stack can be deleted once memory becomes scarce without compromising the termination of the state space exploration.

The *sweep-line method* [3, 15] is a state space method belonging to the third category. It is aimed at systems for which the notion of progress can be formalised as a *progress measure* mapping from markings to a set of ordered progress values. The sweep-line method uses the progress values on markings to determine which states can safely be deleted. Progress measures provide a conservative estimate of reachability: For a marking to be reachable from the set of unprocessed markings, it would have to have a progress value larger than at least one of the unprocessed markings. Markings with progress value smaller than all unprocessed markings can therefore safely be deleted while still guaranteeing termination of the state space exploration. Progress measures will often be specific to the system under consideration, such as sequence numbers and retransmission counters in communication protocols, and the control flow of the system. However in some instances a progress measure can be defined based on the modelling language itself, and such progress measures then apply to all models of systems constructed using this modelling language. The global clock representing time in Timed Coloured Petri Nets [12] is an example of one such progress measure. The sweep-line method was used in [7] for verification of transactions in the Wireless Application Protocol (WAP) with a reduction in peak memory usage to 20%. The use of the sweep-line method on Timed CP-nets was studied in [2]. The sweep-line method is not tied to CP-nets, but can be applied to all modelling languages where state space methods are applicable.

The contribution of this paper is to study the sweep-line method in the context of CP-nets, and to present SWEEP/CPN [16], a library extension to the DESIGN/CPN state space tool [1] implementing the basic sweep-line method from [3] for CP-nets. Consequently there is a shift in the focus of the paper, from a theoretical introduction to the sweep-line method to a description of the SWEEP/CPN implementation, which could be regarded as a high level manual for practitioners.

The rest of this paper is organised as follows. Section 2 introduces the stop-and-wait protocol which we will use as a running example throughout this paper. Section 3 gives the necessary background information on state spaces for CP-nets, while Sect. 4 and 5 present the sweep-line method and the application of SWEEP/CPN to the stop-and-wait protocol. Section 6 presents the set of generic state space exploration functions available for verification using

SWEEP/CPN. Section 7 shows how the generic state space exploration functions can be specialised for the verification of standard dynamic properties of CP-nets. Finally, in Sect. 8 we sum up with a conclusion and discuss future work.

2 The Stop-and-wait Communication Protocol

To introduce the sweep-line method and demonstrate the use of SWEEP/CPN, we will use a simple communication protocol as a running example. The protocol under consideration is a stop-and-wait communication protocol from the datalink control layer of the OSI (Open Systems Interconnection) network architecture. The CPN model of the stop-and-wait protocol is taken from [14]. The stop-and-wait protocol is not a sophisticated protocol, however it is interesting enough to deserve closer investigation, and complex enough to demonstrate the use of SWEEP/CPN. In this paper we only explain the part of the CPN model necessary to understand the application of SWEEP/CPN. A complete description of the CPN model can be found in [14].

Figure 1 shows the prime page of the CPN model. The system consists of a Sender (left) transmitting data packets to a Receiver (right) across a bi-directional Communication Channel (bottom). The packets to be transmitted are stored in a Send buffer at the sender side, and the packets received by the receiver are stored in a Received buffer. The Communication Channel is unreliable, which means that loss and overtaking is possible, however for simplicity only loss is considered in this example. The data packets in the Send buffer have to be delivered over the communication channel exactly once, and in the correct order, to the Received buffer. A stop-and-wait strategy is employed to achieve this, whereby the sender will send a data packet and continue to retransmit the same data packet until a matching acknowledgement is received, at which point the next data packet can be transmitted. The number of retransmissions of a data packet allowed by this stop-and-wait protocol is unbounded for simplicity. In order to make the CPN model tractable for state space analysis, the buffers between the sender, receiver, and communication channel (TransmitData, ReceiveAck, TransmitAck, ReceiveData) have been modelled with a maximum capacity of one packet each. This still allows for unbounded retransmissions, whilst ensuring the state space will be finite.

3 State Spaces of CP-nets

In this section we briefly recall the basic ideas of full state space exploration for CP-nets [12]. The state space of a CP-net can be characterised as a directed graph $G = (V, E)$, where $V = [M_0\rangle$ (the set of nodes or markings reachable from the initial marking M_0), and $E = \{(M, (t, b), M') \in V \times BE \times V \mid M[(t, b)\rangle M'\}$, i.e., there is an arc $(M, (t, b), M')$ in the state space if the binding element (t, b) (belonging to the set of all binding elements BE) is enabled in the marking M and its occurrence leads to the marking M'.

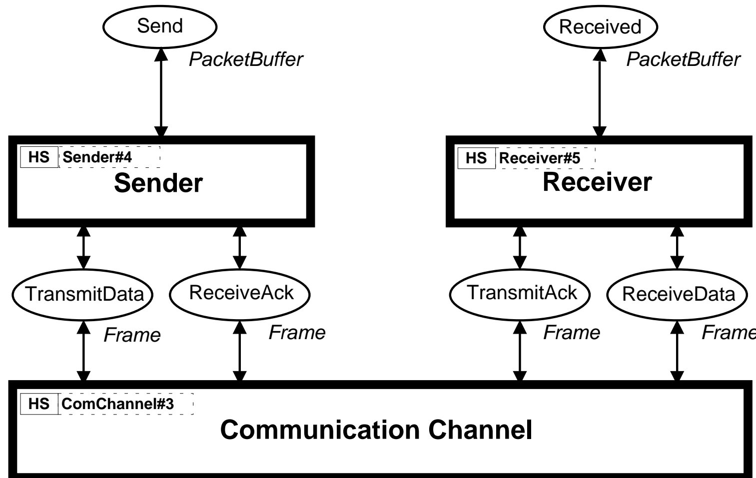


Fig. 1. Stop-and-wait Communication Protocol.

Figure 2 shows a variant of the classical algorithm for generating the state space of a CP-net. The algorithm operates on two sets: NODES and UNPROCESSED. The set NODES holds all markings generated so far, and the set UNPROCESSED holds the markings for which successor markings have not yet been calculated. Initially (line 1) UNPROCESSED contains only M_0. As long as there are unprocessed markings, the algorithm selects one marking among the unprocessed (line 4) and calculates all its successors (line 5). If a successor has not been processed and is not contained in NODES, it is added to NODES and UNPROCESSED (lines 6-9). When the algorithm terminates, NODES contains all reachable markings.

```
 1: Unprocessed ← {M0}
 2: Nodes ← {M0}
 3: while ¬ Unprocessed.Empty() do
 4:   M ← Unprocessed.GetNextElement()
 5:   for all ((t,b), M') such that M[(t,b)⟩M' do
 6:     if ¬(Nodes.Contains(M')) then
 7:       Nodes.Add(M')
 8:       Unprocessed.Add(M')
 9:     end if
10:   end for
11: end while
```

Fig. 2. Full state space exploration algorithm.

Figure 3 shows a snapshot of the state space generation for the stop-and-wait protocol. Dashed nodes are *fully processed* markings (i.e. markings that are stored in memory and all their successor markings have been calculated). Nodes with a thick solid black border are *unprocessed* nodes (i.e. nodes that are stored in memory, but their successor markings have not yet been calculated). Nodes with a thin solid black border are markings that have not yet been calculated. Node 1 corresponds to the initial marking. In order to make the state space fragment comprehendable, the situations where packets are lost on the com-

munication channel have been ignored. The states in this fragment have been arranged into two *layers*: Layer 1, where no packets have yet been received by the Receiver, and Layer 2, where one packet has been received. Such an ordering is useful for explaining the concept of progress in the next section. Arcs have been labelled with their corresponding action in the CPN model. Accept Packet 0 indicates that the Sender has accepted the first packet for sending. Send Packet 0 indicates that the Sender has placed the packet in the Communication Channel. Transmit Packet 0 indicates that the Communication Channel has (successfully) transmitted the packet from the sender to the receiver. Receive Packet 0 shows the Receiver receiving the packet from the Communication Channel. Send Ack indicates that the Receiver has placed an acknowledgement into the Communication Channel. TimeOut shows an occurrence of the Sender timing out (i.e. expiration of a timer) because it has not yet received an acknowledgement.

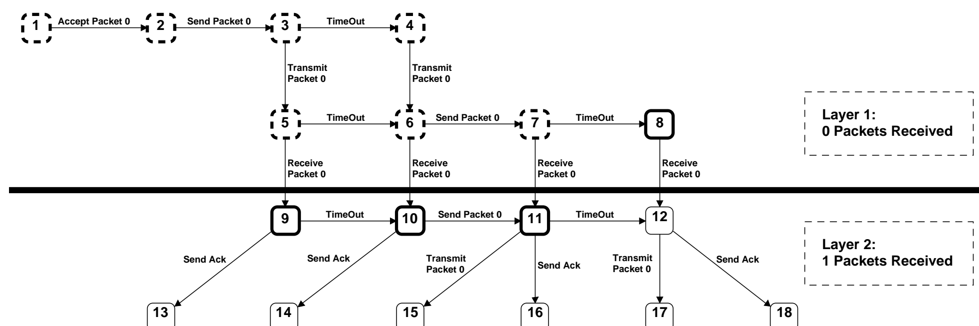


Fig. 3. Initial fragment of state space.

If we analyse states on-the-fly, the set NODES in Fig. 2 is used solely for determining whether successor states of a given state have already been examined, or whether they should be considered unprocessed. However, at any time some states in NODES are no longer reachable from the states in UNPROCESSED, and will not be used for this test in the further processing. These states could be removed from NODES, and the algorithm would still visit each reachable state. The problem with this approach is, of course, determining whether a state is reachable from one of the unprocessed states. The sweep-line method uses a notion of *progress* to obtain a conservative estimate of the reachability relation, and uses this estimate to reduce NODES.

4 The Sweep-Line Method

The concept of progress exploited by the sweep-line method can be thought of in terms of how far we have moved towards a desired end-state or a predefined goal, or towards the termination of execution of a system. To give an example of each of these, consider a farmer who has a number of fields to plow. If we consider the goal of the farmer to have all fields plowed, we can then consider the progress made by the farmer towards achieving this goal. Progress could be measured in terms of number of fields plowed. In this case, we have a predefined

goal, which is to have all fields plowed. The farmer would start with no fields plowed, and plow field after field until all fields had been plowed. Progress could also be measured in terms of total time spent plowing. We can see that at any given time during the plowing, the farmer would have plowed the same amount or more than at any given time previous to this. In this case we do not know how long the plowing will take to complete, so we cannot specify a predefined goal, but we can work towards the termination of the system whereby the farmer has completed plowing.

There is an intuitive presence of progress in the stop-and-wait protocol as more and more packets are being transferred from the sender to the receiver. This progress is also reflected in the state space of the CPN model, an initial fragment of which was shown in Fig. 3. The key observation to make is that progress in the stop-and-wait protocol manifests itself by the property that a marking in a given layer has successor markings either in the same layer or in some layer with a higher number, but never in a layer with a smaller number. The idea underlying the sweep-line method is to exploit such progress by deleting markings on-the-fly during state space exploration. The deletion is done such that the state space exploration will eventually terminate and upon termination all reachable markings will have been explored exactly once.

Examining the initial state space fragment shown in Fig. 3, if the state space exploration algorithm processes markings according to the progress of the protocol they correspond to, node 8 will be the marking among the unprocessed markings that will be selected for processing next. This will add node 12 to the set of stored markings and mark it as unprocessed. At this point it can be observed that it is not possible for any of the unprocessed markings to reach one of the markings 1-8. The reason is that nodes 1-8 represent markings where the protocol has not progressed as far as in any of the unprocessed markings, i.e. no packets have been received by the receiver. Hence, it is safe to delete nodes 1-8, as they cannot possibly be needed for comparison with newly generated markings when checking (during the state space exploration) whether a marking has already been visited. In a similar way, once all the markings in the second layer have been fully processed, these nodes can be deleted from the set of nodes stored in memory. Intuitively, one can think of a *sweep-line* being aligned with the layer currently being processed, i.e. the layer that contains unprocessed markings. During state space exploration, unprocessed markings are selected for processing in a least-progress-first order causing the sweep-line to move forwards. Markings will thereby be added in front of the sweep-line and deleted behind the sweep-line.

The progress exploited by the sweep-line method is formally captured by a *progress measure*. A progress measure consists of a *progress mapping* assigning a *progress value* to each marking, and a *partial order* on the progress values.[1] Moreover, the partial ordering of the progress values is required to preserve the reachability relation between markings of the CP-net. The definition of progress measure below is identical to Def. 1 in [3] except that we give the definition in

[1] A partial order $(O, \sqsubseteq)$ consists of a set O and a relation $\sqsubseteq \subseteq O \times O$ which is reflexive, transitive, and antisymmetric.

a CP-net formulation. In the definition $\mathbb{M}$ denotes the set of all markings. If a marking M' is reachable from a marking M via some occurrence sequence, we write this as $M' \in [M\rangle$. In particular $M \in [M\rangle$ for all markings $M \in \mathbb{M}$, since the empty occurrence sequence leads from M to M.

Definition 1. *(Def. 1 in [3]) A **progress measure** is a tuple $\mathcal{P} = (O, \sqsubseteq, \psi)$ such that $(O, \sqsubseteq)$ is a partial order and $\psi : \mathbb{M} \rightarrow O$ is a progress mapping from markings into O satisfying: $\forall M, M' \in [M_0\rangle : M' \in [M\rangle \Rightarrow \psi(M) \sqsubseteq \psi(M')$.*

Exploring the state space using the sweep-line method is based on the algorithm used for conventional state space construction. Figure 4 shows the state space exploration algorithm for the sweep-line method. It is derived from the standard algorithm by including deletion of states that can no longer be reached by states still to be explored, and by exploring the states in UNPROCESSED according to their progress values. In each iteration (lines 3-15) a new unprocessed marking is selected (line 4) such that this node has a minimal progress value among the states in UNPROCESSED. After a marking has been processed, states with a progress value strictly smaller than the minimal progress value among the markings in UNPROCESSED can be deleted (line 14). In the case of a total order, there will only be one minimal progress value. In the more general case of a partial order, it is possible for all unprocessed states to have different, incomparable progress measures. Hence, in the worst case, the progress measure of a state needs to be compared to each of them in order to determine whether it can be deleted or not. The condition that the progress measure preserves the reachability relation is checked on-the-fly when successors of a marking are being calculated (lines 6-8). State space exploration is terminated if the progress measure is not valid, and a triple $(M, (t, b), M')$ is given to demonstrate why the progress measure was not valid.

```
 1: UNPROCESSED ← {M0}
 2: NODES ← {M0}
 3: while ¬ UNPROCESSED.EMPTY() do
 4:    M ← UNPROCESSED.GETMINELEMENT()
 5:    for all ((t,b), M') such that M[(t,b)⟩M' do
 6:       if ψ(M) ⋢ ψ(M') then
 7:          STOP("Progress measure rejected:",(M, (t,b), M'))
 8:       end if
 9:       if ¬(NODES.CONTAINS(M')) then
10:          NODES.ADD(M')
11:          UNPROCESSED.ADD(M')
12:       end if
13:    end for
14:    NODES.DELETE(min{ψ(M) | M ∈ UNPROCESSED})
15: end while
```

Fig. 4. The sweep-line state space exploration algorithm.

In order to maximise the number of states that can be deleted in each *garbage collection* (or *memory reclaim*), the unprocessed state chosen by the

GetMinElement() method (line 4) should always return (as the name implies) a state with a minimal progress measure among the states in Unprocessed. If the progress measure maps to a total order (as has been the case for all our applications) then Unprocessed can be implemented as a priority queue using the progress measure as the priority, and $\sqsubseteq$ as the ordering. A breadth-first generation (with respect to progress measure) of the state space will result, ensuring that the number of states that can be deleted is maximised.

5 Sweep-Line Exploration of the Stop-and-Wait Protocol

The Sweep/CPN library supports progress measures based on total orderings, and consists of a set of Standard ML [19] files which can be loaded into the Design/CPN state space tool [1]. Currently there is no graphical user interface (GUI) support associated with Sweep/CPN, and all interaction with the library is via evaluation of auxiliary boxes containing SML code. The GUI of the Design/CPN state space tool can however be used to draw nodes and arcs of the state space.

The user specifies the progress measure to be used by writing an SML function mapping from the markings into unbounded integers, that is, from the type SLMark into IntInf.int. Verification of the user specified progress measure is done on-the-fly as described in the previous section, whereby validity is checked upon the calculation of the successors of a marking. The progress measure for the stop-and-wait protocol based on the number of packets received by the receiver can be implemented as follows. An explanation of the function is given below.

```
fun SWPM M = IntInf.fromInt
     (List.length (ms_to_col (SLMark.SWProtocol'Received 1 M)));
```

For a marking M, the function works by extracting the marking of place Received on instance 1 of the page SWProtocol. (Page SWProtocol is the name of the prime page of the CPN model shown in Fig. 1.) This is done by using the accordingly named function of the SLMark structure. The marking of place Received is a multi-set containing a single token which is a list of the packets received until now. The list of received packets is extracted from the multi-set using the function ms_to_col. Finally, the function List.length is used to obtain the length of the list, and the function IntInf.fromInt is used to convert the integer denoting the length of the list into an infinite (unbounded) integer.

A number of functions are available in Sweep/CPN for state space exploration with the sweep-line method. These will be discussed in more detail in the following section. However, in order to provide the complete picture for the stop-and-wait protocol, the simplest of these functions will be described. This is the SL.ExploreStateSpace function. For the stop-and-wait protocol example, the SL.ExploreStateSpace function is used as follows:

```
SL.ExploreStateSpace{PM=SWPM,
                     GC = 100,
```

```
Secs = 300,
Logfile = (SOME "/tmp/SW.dat")
};
```

The function takes a record with four fields, the *sweep options*, as an argument and conducts sweep-line state space exploration according to the values of these fields. The first field of the sweep options, PM, specifies the progress measure to use. For this stop-and-wait example, it is the progress measure defined by the SWPM function specified above. The GC field specifies the *garbage collection* threshold. This is the number of additional nodes that have to be explored before a deletion of markings is initiated in line 14 of the algorithm in Fig. 4. In this case, garbage collection of nodes will occur every time 100 new nodes are explored. The Secs field specifies an upper bound (in seconds) on the time that the state space exploration will run for. Once this time limit is reached the state space exploration will terminate regardless of whether the exploration is complete. The file specified in the Logfile field will contain a log generated by SWEEP/CPN during state space exploration. The log file will contain statistical information about the state space exploration.

Table 1 lists statistics for the application of the sweep-line method for different configurations of the stop-and-wait communication protocol. The results were obtained when garbage collecting after each 2000 new nodes (markings). The table consists of four columns. The Packets column gives the configuration under consideration, i.e. the number of packets that must be delivered in order and without loss to the receiver from the sender. The Full State Spaces column gives the number of markings in the full state space and the amount of CPU time needed to generate it. The third column, Sweep-Line Method, gives the maximum number of markings stored in memory at any given instance during the sweep-line state space exploration, and the time it took to sweep through the entire state space. The Reduction gives the reduction in the number of markings required to be stored (and thus the reduction in memory consumption) as well as the reduction in time taken to generate the full state space, when using the conventional DESIGN/CPN state space tool and when using the SWEEP/CPN library. All results were obtained using an Athlon XP 2000+ with 1Gb DDR memory.

Table 1. Experimental results – Stop-and-Wait Communication Protocol.

	Full State Spaces		Sweep-Line Method		Reduction	
Packets	Markings	Time	Markings	Time	Markings	Time
10	2,576	0:00:01	2,001	0:00:01	22.3%	0.0%
50	13,416	0:00:05	2,280	0:00:04	83.0%	20.0%
100	26,966	0:00:14	2,280	0:00:07	91.5%	50.0%
200	54,066	0:00:42	2,280	0:00:14	95.8%	66.7%
500	135,366	0:03:11	2,287	0:00:38	98.3%	80.1%
1000	270,866	0:11:21	2,287	0:01:21	99.2%	88.1%
2000	541,866	0:47:10	2,287	0:03:05	99.6%	93.5%

Table 1 shows that the use of Sweep/CPN saves both memory and time. For example, when there are 50 packets to be delivered from sender to receiver, the reduction in the number of markings that must be stored in memory at one time is 83%, while the reduction in time is 20%. In addition to this, the savings in memory consumption and time grow larger as the size of the full state space grows larger. This can be seen when examining the situation where 2000 packets must be delivered from the sender to the receiver. The saving in memory consumption is 99.6%, and the saving in time is 93.5%.

Savings in memory consumption were expected since markings are being deleted on-the-fly during state space exploration. The saving in runtime is perhaps more surprising. The explanation can be found in the test for whether successor states are new or have previously been processed (in line 9 of the algorithm in Fig. 4). By keeping the set Nodes small we avoid this runtime performance penalty. For this example at least, Table 1 indicates that using Sweep/CPN and the sweep-line method is faster than generating the full state space by conventional means. The time overhead required to perform garbage collection (delete states on-the-fly) is more than compensated for by having significantly fewer states to compare with when determining whether a newly generated marking has already been visited.

6 State Space Exploration

The general sweep-line algorithm was introduced in Section 4. This algorithm demonstrates how a state space can be explored using the sweep-line method. Using conventional state space exploration, analysis can be done by examining all states in the state space once the entire state space has been constructed. However, since states are deleted during the state space exploration when using the sweep-line method, properties must be verified *on-the-fly*. The general algorithm from Section 4 must be augmented so that it provides not just a sweep through the state space, but also the possibility for verification and analysis of properties of the system.

Figure 5 shows the sweep-line state space exploration algorithm augmented with hooks for analysing *state-related properties* (e.g. boundedness, other model-specific properties such as duplication of packets), *arc-related properties* (e.g dead transition instances) and *dead markings*. The analysis is done through three procedures: ExploreNode, ExploreArc, and ExploreDead, operating on the variables NodeResult, ArcResult, and DeadResult respectively. The three variables, NodeResult, ArcResult and DeadResult along with the set Nodes are assumed to have a scope that includes these procedures. The analysis procedures are shown in Fig. 6, Fig. 7, and Fig. 8. As seen, they are all more or less of the same form: A predicate, e.g. NodePred, selects whether a node or arc should be analysed. If it should, then an *evaluation function*, e.g. NodeEval, extracts a value from the node or arc. This value is then combined, via a *combine function* e.g. NodeComb, with the result so far. Predicate, evaluation functions, and combination functions are all user specified. The value so far is stored in the respective result variable, NodeResult, ArcResult, and DeadResult, which are all initialised with user provided values

NodeInit, ArcInit, and DeadInit. Additional predicates, NodeStore and DeadStore, once again user specified, are used to prevent garbage collection for selected nodes.

```
1: Unprocessed ← {M_0}
2: Nodes ← {M_0}
3: NodeResult ← NodeInit
4: ArcResult ← ArcInit
5: DeadResult ← DeadInit
6: while ¬ Unprocessed.Empty() do
7:   M ← Unprocessed.GetMinElement()
8:   ExploreNode(M)
9:   if Dead(M) then
10:    ExploreDead(M)
11:  end if
12:  for all ((t,b),M') such that M[(t,b)⟩M' do
13:    ExploreArc(M,(t,b),M')
14:    if ψ(M) ⋢ ψ(M') then
15:      Stop("Progress measure rejected:",(M,(t,b),M'))
16:    end if
17:    if ¬(Nodes.Contains(M')) then
18:      Nodes.Add(M')
19:      Unprocessed.Add(M')
20:    end if
21:  end for
22:  Nodes.GarbageCollect(min{ψ(M) M ∈ Unprocessed})
23: end while
```

Fig. 5. The augmented sweep-line state space exploration algorithm.

The first procedure, ExploreNode, is invoked for every marking M encountered during the sweep. An example of how this can be used would be to inspect all markings M to determine the best upper integer bound on a particular place. The best upper bound of a place is the maximum number of tokens that may reside on a place in any reachable marking. In this case, NodeResult would be of type integer, NodePred would always return true, and the number of tokens for this particular place in M would be returned by NodeEval. NodeComb would then determine whether the number of tokens as returned by NodeEval is larger than the best upper integer bound computed until now (in NodeResult) and update NodeResult if necessary. The second procedure, ExploreArc, is called for every arc (edge) encountered during the sweep. It works in a very similar way to ExploreNodes, except that it works on arcs rather than nodes, and there is no provision for the storage of arcs. The third procedure, ExploreDead, is only evaluated for markings that have no sucessor markings, but otherwise works in a similar way to ExploreNodes.

The augmented sweep-line algorithm has been implemented in Sweep/CPN as the exploration function called SL.Explore. The function is parameterised with 15 arguments. To provide for more convenient use of Sweep/CPN, three specialisations to SL.Explore have been added to the library. These are SL.ExploreStates, SL.ExploreArcs, and SL.ExploreDead, and are specialisations to examine all nodes, all arcs, and all dead markings of the state space respectively. Dead

1: **procedure** ExploreNode(M) **do**
2: **if** NodePred(M) **then**
3: NodeResult $\leftarrow$ NodeComb(NodeResult, NodeEval(M))
4: **end if**
5: **if** NodeStoreM) **then**
6: Nodes.MarkPersistent(M)
7: **end if**
8: **end procedure**

Fig. 6. The ExploreNode procedure.

1: **procedure** ExploreArc($M, (t, b), M'$) **do**
2: **if** ArcPred($M, (t, b), M'$) **then**
3: ArcResult $\leftarrow$ ArcComb(ArcResult, ArcEval($M, (t, b), M'$))
4: **end if**
5: **end procedure**

Fig. 7. The ExploreArc procedure.

markings are treated as a separate case to ordinary nodes/markings, because of the underlying structure and implementation of the existing state space tool. As an example, the function SL.ExploreStates has the following type:

```
SL.ExploreStates : {PM      : SLMark -> IntInf.int,
                    GC      : int,
                    Secs    : int,
                    Logfile : string option,
                    Init    : 'a,
                    Comb    : 'a * 'b -> 'a,
                    Eval    : SLMark -> 'b,
                    Pred    : SLMark -> bool,
                    Store   : SLMark -> bool} -> 'a
```

The first four fields in the record (PM, GC, Secs, Logfile) correspond to the four sweep options discussed in Section 5 and are the progress measure, garbage collection threshold, exploration time-out and logfile location respectively. Init allows the user to provide the initial value for the Comb funtion. The Comb is the combination function on nodes, Eval is the evaluation function on nodes, and Pred is the predicate on nodes. Store is a function which determines if the nodes should be made persistent. The types of SL.ExploreArcs and SL.ExploreDead are

1: **procedure** ExploreDead(M) **do**
2: **if** DeadPred(M) **then**
3: DeadResult $\leftarrow$ DeadComb(DeadResult, DeadEval(M))
4: **end if**
5: **if** DeadStore(M) **then**
6: Nodes.MarkPersistent(M)
7: **end if**
8: **end procedure**

Fig. 8. The ExploreDead procedure.

similar to SL.ExploreStates. We give examples of the use of these functions in the next section.

7 Standard Query Functions

We now present the set of standard query functions available in Sweep/CPN for verifying a subset of the standard dynamic properties of CP-nets as defined in [12]. The standard dynamic properties of CP-nets are often the first set of properties investigated for the system under consideration. In addition to presenting the available standard query functions, we also show how some of these have been implemented by means of the generic state space exploration functions presented in Sect. 6. The structure of Sweep/CPN (i.e. as a set of query functions) has been inspired by the structure of the existing state space exploration tool, which was in turn partially inspired by functional programming concepts such as mapping and folding [19]. The standard dynamic properties of CP-nets are informally introduced throughout this section. The reader interested in the full and formal definition of the standard dynamic properties of CP-nets is referred to [12].

7.1 Reachability Properties

A marking is said to be *reachable*, if it is reachable via some occurrence sequence starting in the initial marking. The function Reachable is available for determining whether a reachable marking exists satisfying a given predicate:

```
SLReachable : SweepSpec * (SLMark -> bool) * bool -> bool
```

The first parameter (of type SweepSpec) is a record which provides the parameters for the sweep-line exploration. This is the same record as shown in Section 6, where the four arguments (progress measure, garbage collection threshold, upper bound on the time for state space exploration, and the log file name and location) are given. The second argument is the predicate on markings. The third argument determines whether the encountered markings which satisfy the marking predicate should be available upon the completion of the sweep.

To illustrate the use of SLReachable, we will show how it can be used to investigate whether the stop-and-wait protocol may duplicate packets. If the stop-and-wait protocol can duplicate packets, a reachable marking will exist in which some packet occurs twice or more in the list on place Received. The following marking predicate DupPackets determines whether this is the case in a marking M. The function remdupl removes duplicates from a list.

```
fun DupPackets M =
    let
       val packets = (ms_to_col (SLMark.Receiver'Received 1 M))
    in
       List.length (remdupl packets) <> (List.length packets)
    end;
```

The DupPackets predicate can now be used in an invocation of the SLReachable query function as follows:

```
SLReachable ({PM=SWPM,GC=100,Secs=300,Logfile=NONE},
             DupPackets, true);
```

In this case, markings satisfying DupPackets will not be deleted during the sweep. Reachability of a specific marking M can be checked by using a predicate as argument to SLReachable which evaluates to true only on M.

The SLReachable function has been implemented using the SL.ExploreStates function as follows:

```
fun SLReachable ({PM=pm,GC=gc,Secs=secs,Logfile=logfile},
                 markpred,keep) =
    (SL.ExploreStates
        {PM=pm,GC=gc,Secs=secs,Logfile=logfile,
         Pred = markpred,
         Store = keep,
         Eval = (fn _ => 1),
         Comb = (fn (a,b) => a+b),
         Init = 0}) > 0;
```

The sweep specification is given by the sweep specification provided as the first argument to SLReachable. The evaluation (Eval) and combination (Comb) functions are used to count the number of times the marking predicate evaluates to true. If it evaluates to true at least once, a marking exists satisfying the predicate.

7.2 Boundedness Properties

The boundedness properties are concerned with the number of tokens on places and their possible colours. We first consider integer bounds. The *best upper integer bound* of a place is the maximum number of tokens present on the place in any reachable marking. Similarly, the *best lower integer bound* is the minimum number of tokens present on the place in any reachable marking. Two functions are available for determining the integer bounds of a place:

```
SLBestUpperInteger : SweepSpec * (SLMark -> int) -> int
SLBestLowerInteger : SweepSpec * (SLMark -> int) -> int
```

Both functions take the sweep specification as their first argument. The second argument is a function mapping from markings into integers. The following shows how the functions can be used to obtain the best upper and best lower integer bound of the place TransmitData (see Fig. 1). The function TransmitDataCount counts the number of tokens on the place TransmitData in a marking M. The function mssize returns the number of tokens in a multi-set.

```
fun TransmitDataCount M = mssize (SLMark.Sender'TransmitData 1 M);
```

```
SLBestUpperInteger ({PM=SWPN,GC=100,Secs=300,Logfile=NONE},
                     TransmitDataCount);
SLBestLowerInteger ({PM=SWPN,GC=100,Secs=300,Logfile=NONE},
                     TransmitDataCount);
```

The SLBestUpperInteger and SLBestLowerInteger functions are both implemented using the SL.ExploreStates function. Below we give the implementation of SLBestUpperInteger. The implementation of SLBestLowerInteger is similar.

```
fun SLBestUpperInteger ({PM=pm,GC=gc,Secs=secs,Logfile=logfile},
                         evalfun) =
        (SL.ExploreStates
            {PM=pm,GC=gc,Secs=secs,Logfile=logfile,
             Pred = (fn _ => true),
             Store = (fn _ => false),
             Eval = evalfun,
             Comb = (fn (a,b) => Int.max(a,b)),
             Init = 0})
```

The predicate function (Pred) ensures that the evaluation function is invoked on each marking encountered. The maximum function on integers (Int.max) is used to obtain the maximum value resulting from the invocation of the evaluation function.

Multi-set boundedness is similar to integer boundedness, except we are no longer dealing with the size of the multi-set of tokens on a particular place, but rather with the multi-set of tokens itself. For a particular place, it is possible to find the smallest multi-set of tokens that is larger than any of the multi-sets of tokens found on this place in any reachable marking. This multi-set is called the *best upper multi-set bound* for this particular place. Finding the upper multi-set bound can be done by examining the marking of this place for every reachable marking, and by using multi-set operations to determine the multi-set comprising the best upper multi-set bound. Similarly, the *best lower multi-set bound* can also be found. This is the largest multi-set that is smaller than all multi-sets found on the particular place, for every reachable marking.

Sweep/CPN provides two query functions to find multi-set bounds. The first is SLBestUpperMultiSet, to find the best upper multi-set bound, and the second is SLBestLowerMultiSet, to find the best lower multi-set bound:

```
SLBestUpperMultiSet : SweepSpec -> (SLMark -> ''a ms) -> ''a ms
SLBestLowerMultiSet : SweepSpec -> (SLMark -> ''a ms) -> ''a ms
```

These are similar to the functions for integer bounds except that they work on the level of multi-sets, not integers. The argument of type SweepSpec is again a record that gives the sweep-line parameters (progress measure, garbage collection threshold, exploration time-out, log file.) The second argument is again an evaluation function, but it returns the multi-set of tokens on the specified place upon evaluation, not an integer. Below we show how functions can be used to find the best upper and best lower multi-set bound of the place TransmitData.

```
SLBestUpperMultiSet ({PM=SWPM,GC=100,Secs=300,Logfile=NONE},
                     (SLMark.Sender'TransmitData 1));
SLBestLowerMultiSet ({PM=SWPM,GC=100,Secs=300,Logfile=NONE},
                     (SLMark.Sender'TransmitData 1));
```

The functions SLBestUpperMultiSet and SLBestLowerMultiSet have been implemented based on the SL.ExploreStates function in a similar way to the query functions for integer bounds.

7.3 Liveness Properties

Dead markings are the reachable markings of the CPN model without enabled binding elements. These correspond to the states of the system where it has terminated. Finding such dead markings is often an important step in the analysis of a system. SWEEP/CPN provides a function SLListDeadMarkings which performs a sweep of the state space and records all of the dead markings found in the process. Upon termination of the sweep, the list of dead markings/nodes is returned.

```
SLListDeadMarkings : SweepSpec -> Node list
```

The argument of type SweepSpec is the same as in previous functions, it is a record containing the sweep parameters. The return type of this function is Node list, a list of the dead markings found during the sweep. For the stop-and-wait communication protocol, the SLListDeadMarkings function would be invoked as follows:

```
SLListDeadMarkings {PM=RNPM,GC=100,Secs=300,Logfile=NONE};
```

The returned list of nodes can then be examined further in any way desired by the user, e.g., by drawing the nodes using the state space tool, and then by inspecting the descriptors of the nodes which give details about the marking of each place. The SLListDeadMarkings function has been implemented using the SL.ExploreDeadStates function as follows:

```
fun SLListDeadMarkings {PM=pm,GC=gc,Secs=secs,Logfile=logfile} =
    SL.ExploreDeadStates {PM=pm,GC=gc,Secs=secs,Logfile=logfile,
                          Pred = (fn _ => true),
                          Store = (fn _ => true),
                          Eval = GetNodeNo,
                          Comb = (fn (a,b) => b::a),
                          Init = []};
```

The function GetNodeNo maps from markings into node numbers. The store functions ensure that the dead markings are available after the sweep, and the combination function is used to accumulate the node numbers of the dead markings encountered.

In a manner similar to obtaining dead markings, SWEEP/CPN provides a function for obtaining *dead transition instances*. A dead transition instance is a

transition instance that is not enabled in any marking reachable from the initial marking. The function is called SLListDeadTIs and has the type shown below:

```
SLListDeadTIs : SweepSpec -> TI.TransInst list
```

As has been the case for all sweep-line state space exploration functions so far, this function performs a sweep of the state space and thus the sweep parameters are supplied by the argument of type SweepSpec. The return value is of type `TI.TransInst list`, a list of transition instances representing the dead transition instances. To find the dead transition instances of the stop-and-wait communication protocol example, SLListDeadTIs would be invoked as follows:

```
SLListDeadTIs {PM=RNPM,GC=100,Secs=300,Logfile=NONE};
```

The function has been implemented based on the SL.ExploreArcs function. The implementation is shown below. The function SLArcToTI maps an arc of the state space into the corresponding transition instance. The function RemoveTI removes a transition instance (t) from a list of transitions instances (ts). The constant TI.All is the list of all transition instances of the CPN model.

```
fun SLArcToTI (_,b,_) = BEToTI b;

fun RemoveTI (ts,t) = List.filter (fn t' => t' <> t) ts;

fun SLListDeadTIs {PM=pm,GC=gc,Secs=secs,Logfile=logfile}) =
      SL.ExploreArcs
        {PM=pm,GC=gc,Secs=secs,Logfile=logfile,
         Pred = (fn _ => true),
         Eval = SLArcToTI,
         Comb = RemoveTI,
         Init = TI.All};
```

8 Conclusions and Future Work

We have presented Sweep/CPN, an extension to the Design/CPN state space analysis tool which supports state space exploration with the basic sweep-line method as given in [3]. Moreover, we have demonstrated the use of Sweep/CPN on a small example of a communication protocol.

Future work on Sweep/CPN will consider the further development of Sweep/CPN to support also the generalised sweep-line method presented in [15]. The basic sweep-line method is only useful in situations where the strongly connected component graph is non-trivial (i.e. has multiple nodes) and cannot be used on fully reactive systems. The generalised sweep-line method in [15] supports a relaxed notion of progress compared to the monotone notion of progress considered in this paper. With the relaxed notion of progress, it is possible to have arcs in the state space leading from states with high progress values to states with lower progress values. This is achieved by detecting such

cases and conducting multiple sweeps of parts of the state space. The generalised sweep-line method can be used on fully reactive systems.

In this paper we have shown how the sweep-line method can be used to determine standard dynamic properties of CP-nets such as reachability and boundedness properties, and dead markings. Future work will be concerned with extending the set of properties that can be verified. It follows from the definition of progress measures that all markings in a strongly connected component of the state space will have the same progress value. This means that markings belonging to a given strongly connected component will be stored simultaneously in memory at some point during the sweep. This observation holds the key to extending the properties that can be verified to home markings, home spaces, and fairness properties.

Another topic of future work is also the generation of counter examples (error-traces). With the sweep-line method, part of the path leading from the initial marking to a marking satisfying, e.g., a given marking predicate might have been deleted, and needs to be reconstructed to obtain an error-trace. Combining the sweep-line method with disk-based storage seems a promising approach to obtain error-traces with the sweep-line method. With the sweep-line method, markings can be written to disk as they are deleted from main memory, and there is no need to search for markings on disk. In this way, the usual run-time penalty encountered in disk-based searching is avoided altogether. To obtain the error trace one can work backwards (on disk) from the marking satisfying the marking predicate to the initial marking. This can be done efficiently by storing information about predecessor markings on the disk instead of successor markings. This approach may not give the shortest error-trace in terms of occurrences of transitions, but it can be used to obtain a shortest error-trace in terms of how far the system has progressed according to the progress measure.

References

1. S. Christensen, K. Jensen, and L.M. Kristensen. *Design/CPN Occurrence Graph Manual.* Department of Computer Science, University of Aarhus, Denmark. On-line version: `http://www.daimi.au.dk/designCPN/`.
2. S. Christensen, K. Jensen, T. Mailund, and L. M. Kristensen. State Space Methods for Timed Coloured Petri Nets. In *Proceedings of 2nd International Colloquium on Petri Net Technologies for Modelling Communication Based Systems*, Berlin Germany, September 2001.
3. S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proceedings of TACAS 2001*, volume 2031 of *LNCS*, pages 450–464. Springer-Verlag, 2001.
4. E.M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting Symmetries in Temporal Logic Model Checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996.
5. E.A. Emerson and A.P. Sistla. Symmetry and Model Checking. *Formal Methods in System Design*, 9(1/2):105–131, 1996.
6. P. Godefroid, G.J. Holzmann, and D. Pirottin. State-Space Caching Revisited. *Formal Methods in System Design*, 7(3):227–241, 1995.
7. S. Gordon, L.M. Kristensen, and J. Billington. Verification of a Revised WAP Wireless Transaction Protocol. In *Proceedings of ICATPN'02*, volume 2360 of *LNCS*, pages 182–202. Springer-Verlag, 2002.

8. G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International Editions, 1991.
9. G.J. Holzmann. An Analysis of Bitstate Hashing. *Formal Methods in System Design*, 13(3):287–305, 1998.
10. C. Jard and T. Jeron. Bounded-memory Algorithms for Verification On-the-fly. In *Proceedings of CAV'91*, volume 575 of *LNCS*, pages 192–202. Springer-Verlag, 1991.
11. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* Monographs in Theoretical Computer Science. Springer-Verlag, 1992.
12. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods.* Monographs in Theoretical Computer Science. Springer-Verlag, 1994.
13. K. Jensen. Condensed State Spaces for Symmetrical Coloured Petri Nets. *Formal Methods in System Design*, 9(1/2):7–40, 1996.
14. L.M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.
15. L.M. Kristensen and T. Mailund. A Generalised Sweep-Line Method for Safety Properties. In *Proceedings of FME'02*, volume 2391 of *LNCS*, pages 549–567. Springer-Verlag, 2002.
16. L.M. Kristensen, T. Mailund, and G. Gallasch. SWEEP/CPN. Department of Computer Science, University of Aarhus, 2002. www.daimi.au.dk/designCPN/libs/sweepcpn/.
17. D. Peled. All from One, One for All: On Model Checking Using Representatives. In *Proceedings of CAV'93*, volume 697 of *LNCS*, pages 409–423. Springer-Verlag, 1993.
18. U. Stern and D.L. Dill. Improved Probabilistic Verification by Hash Compaction. In P.E. Camurati and H. Eveking, editors, *Correct Hardware Design and Verification Methods*, volume 987, pages 206–224. Springer-Verlag, 1995.
19. J.D. Ullman. *Elements of ML Programming.* Prentice-Hall, 1998.
20. A. Valmari. A Stubborn Attack on State Explosion. In *Proceedings of CAV'90*, volume 531 of *Lecture Notes in Computer Scienc*, pages 156–165. Springer-Verlag, 1990.
21. P. Wolper and P. Godefroid. Partial Order Methods for Temporal Verification. In *Proceedings of CONCUR'93*, volume 715 of *LNCS*, pages 233–246. Springer-Verlag, 1993.
22. P. Wolper and D. Leroy. Reliable Hashing withoug Collision Detection. In *Proceddings of CAV'93*, volume 697 of *LNCS*, pages 59–70. Springer-Verlag, 1993.

Coloured Petri Nets and State Space Generation with the Symmetry Method

Louise Lorentsen

Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N, DENMARK.
louisel@daimi.au.dk

Abstract. This paper discusses state space generation with the symmetry method in the context of Coloured Petri Nets (CP-nets). The paper presents the development of the Design/CPN OPS tool which, together with the Design/CPN OE/OS tool, provides fully automatic generation of symmetry reduced state spaces for CP-nets with consistent symmetry specifications. Practical experiments show that the practical applicability of the symmetry method is highly depended on efficient algorithms for determining whether two states are symmetric. We present two techniques to obtain an efficient symmetry check between markings of CP-nets: a technique that improves the generation time and a technique that reduces the memory required to handle the symmetries during calculation. The presented algorithms are implemented in the Design/CPN OPS tool and their applicability is evaluated based on practical experiments.

1 Introduction

The state space of a system is a directed graph with a node for each reachable state of the system and an arc for each state change. From the state space it is possible to verify whether the system possesses a set of desired properties, e.g., the absence of deadlocks, the possibility to always reenter the system's initial state, etc. However, the practical use of state spaces for formal analysis and verification of systems is often limited by the *state explosion problem* [17]: even small systems may have a large (or infinite) number of states, thus making it impossible to construct the full state space of the system. Several reduction techniques have been suggested to alleviate the state explosion problem. An example of such a reduction technique is the *symmetry method* [3, 9, 4, 6]. The symmetry method is not restricted to a specific modelling language. In this paper we work with the symmetry method for Coloured Petri Nets (CP-nets or CPN) [8, 9]. The basic observation behind the symmetry method is that many concurrent and distributed systems posses a degree of symmetry which is also reflected in the state space. The idea behind the symmetry method is to factor out this symmetry and obtain a *condensed state space* which typically is much smaller than the full state space, but from which the same properties can be verified without unfolding to the full state space. The symmetries in such systems can be described by algebraic groups of permutations. For CP-nets the symmetries used for the reduction are induced by algebraic groups of permutations on the atomic colour sets of the CP-net. Hence, we will also use the term *state spaces with permutation symmetries* (SSPSs) to denote the condensed state spaces obtained by using the symmetry method.

In the context of CP-nets the theory of the symmetry method is well developed [9, 8] and a computer tool (the Design/CPN OE/OS tool [11, 10]) that supports state space generation with the symmetry method has been developed. However, the symmetry method in the context of CP-nets has only few applications in practice, e.g. [14, 5]. One of the drawbacks of the Design/CPN OE/OS tool is that it requires the user to implement two predicates determining whether two states/actions are symmetric or not. This requires both programming skills and a deep knowledge of the symmetry method. This is especially the case if the predicates are required to be efficient. However, when constructing SSPSs for CP-nets, it can be observed that the predicates can be automatically deduced [8] provided the algebraic groups of permutations used for the reduction

have been specified[1]. The above observation has motivated the construction of a tool (the Design/CPN OPS tool [13]) which, given an assignment of algebraic groups of permutations to the atomic colour sets of the CPN model, generates the predicates for the Design/CPN OE/OS tool expressing whether two states/actions are symmetric.

The problem of determining whether two states/actions are symmetric is in literature also referred to as *the orbit problem* and an efficient solution to this problem is a central issue for the applicability of the symmetry method. The computational complexity of the orbit problem has been investigated in [3] showing that in general it is at least as hard as *the graph isomorphism problem* for which no polynomial time algorithm is known. However, in the context of CP-nets symmetry can be determined efficiently in a number of special cases, e.g., when the CP-net only contains atomic colour sets [1, 8]. This is, however, not true in general; the problem of determining symmetry between states/actions is complicated by the fact that colour sets can contain arbitrary structural dependencies.

During development of the Design/CPN OPS tool a number of practical experiments have been performed with different strategies for the implementation of the predicates. The practical experiments show that the chosen strategy for the implementation of the predicates greatly influences whether the symmetry method for CP-nets is applicable in practice. The algebraic groups of permutations used for the reduction potentially becomes very large as the system parameters grow. The number of symmetries used for the reduction is potentially $\prod_{A \in \Sigma_A} |A|!$ where Σ_A denotes the atomic colour sets of the CPN model. Hence, efficient handling of the symmetries used for the reduction becomes an important aspect when developing algorithms for the predicates used in the symmetry method.

In this paper we present techniques and algorithms which implements an efficient solution to the orbit problem in SSPS generation for CP-nets. The algorithms presented in this paper are based on general techniques which can be applied independently of model specific details. Hence, the predicates can be automatically constructed by the tool.

The paper is structured as follows. Section 2 presents the symmetry method for CP-nets by means of an example. Section 3 presents the basic generation algorithm for SSPSs. Section 4 introduces a basic solution to the orbit problem for CP-nets that will be used for reference purposes. Section 5 presents algorithms that improve the run-time of the basic algorithm. Section 6 presents algorithms that ensure an compact representation of the symmetries throughout calculation of the SSPSs. Finally, Sect. 7 contains the conclusions.

2 The Symmetry Method for CP-nets

In this section we introduce the symmetry method for CP-nets by means of an example. We will use the example of a distributed database from Sect. 1.3 in [7]. Section 2.1 presents the CPN model of the distributed database and show how the symmetry method can be used to reduce the size of the state space. Section 2.2 explains how the symmetries used in the symmetry method are specified as permutations of atomic colours. Finally, Sect. 2.3 presents a data structure which can be used to represent sets of symmetries in a CP-net.

2.1 Example: Distributed Database

The CP-net for the distributed database is shown in Fig. 1. The CP-net models a simple distributed database with n different sites. Each site contains a copy of all data and this copy is handled by a database manager. Each database manager can change its own copy of the database and send a message to all other database managers requesting them to update their copy of the database.

[1] There are two main approaches in the literature: either the permutations can be automatically deduced from the model, e.g., [2, 16], or explicitly specified by the modeller [8, 9]. The latter approach is based on the belief that the modeller, who constructs the model is familiar with the system modelled and has an intuitive idea of the symmetries present in the model [8].

The distributed database system uses the indexed colour set `DBM` to model the database managers, the enumeration colour set `E` to model whether the protocol is active, and the product colour set `MES` to model the messages. The content of the database and the messages are not modelled. Only header information (the sender and the receiver) is contained in a message.

The distributed database system possesses a degree of symmetry. The database managers are treated similarly, only their identities differ. This symmetry is also reflected in the state space of the distributed database system. The state space for the CPN model with three database managers is shown in the left-hand side of Fig. 2. The idea behind SSPSs is to factor out this symmetry and obtain a smaller state space from which the properties of the distributed database system can be verified without unfolding to the full state space. When constructing the SSPS for the database system we consider two markings/binding elements to be symmetric if they are equal except for a bijective renaming of the database managers. This kind of symmetry (based on bijective renamings) induces two equivalence relations; one on the set of markings and one on the set of binding elements [8]. The basic idea when constructing the SSPS is to lump together symmetric markings/binding elements into one node/arc, i.e., only store one representative from each equivalence class. The right-hand side of Fig. 2 shows the SSPS for the distributed database. The nodes in the full state space (in the left-hand side of the figure) are coloured such that nodes corresponding to symmetric markings have the same colour. The same colours are used in the SSPS (in the right hand side of the figure). From the figure it can be seen that the SSPS only contains one node per equivalence class of symmetric markings.

2.2 Symmetry Specification

The symmetries used for the reduction are obtained from permutations of the atomic colours in the CPN model. Let Σ_A denote the set of atomic colour sets of the CPN model. For each atomic colour set in the CPN model, $A \in \Sigma_A$, we define an algebraic group of permutations Φ_A, i.e., a subgroup of $[A \to A]$. A symmetry ϕ of the system is a set of permutations of the atomic colour sets of the model, i.e., $\phi = \{\phi_A \in \Phi_A\}_{A \in \Sigma_A}$. In the rest of the paper we will use the term *permutation symmetry* to denote a set of permutations of the atomic colour sets of a CPN model.

The symmetry considered in the distributed database system is a bijective renaming of the database managers. This is obtained by allowing all permutations of the atomic colour set `DBM`. Hence a permutation symmetry in the distributed database system is a set $\phi = \{\phi_E \in \Phi_E, \phi_{DBM} \in \Phi_{DBM}\}$, where $\Phi_{DBM} = [\mathrm{DBM} \to \mathrm{DBM}]$ and $\Phi_E = \{\phi_{id}\}$ (where ϕ_{id} is the identity permutation, i.e., $\phi_{id}(\mathtt{e})=\mathtt{e}$). From the permutation symmetries of the CPN model we derive permutations of the structured colour sets, multi-sets, markings and binding elements as described in [8].

A *symmetry specification* of a CP-net is an assignment of algebraic groups of permutations to each of the atomic colour sets of the CP-net and hence determines a group of permutation

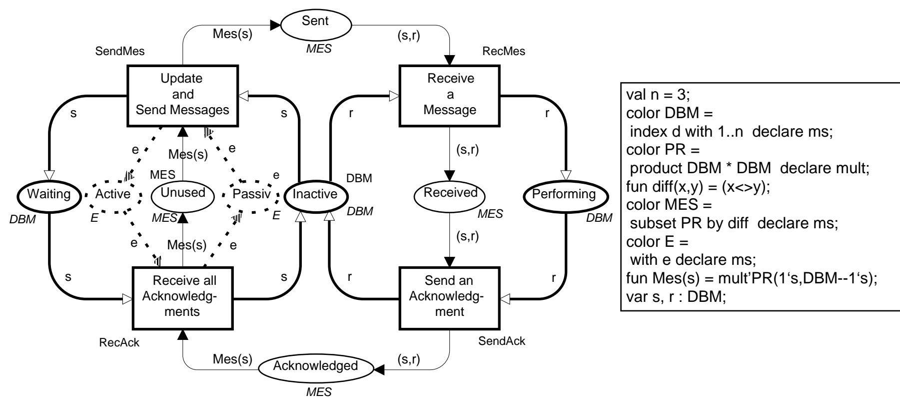


Fig. 1. CP-net for the Distributed Database example.

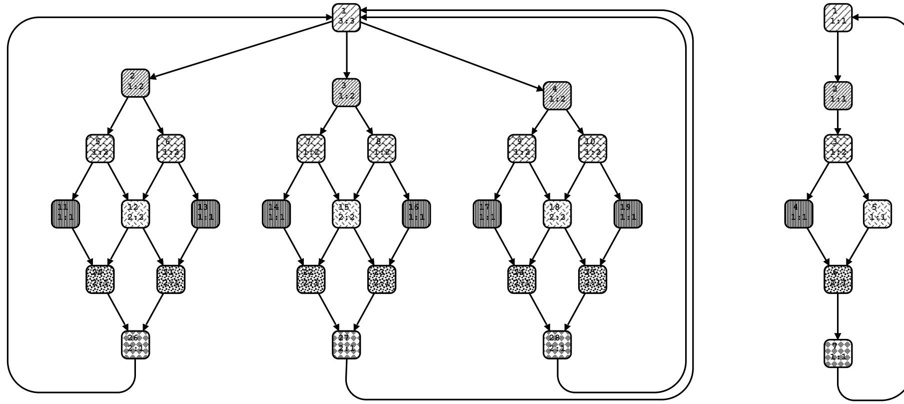


Fig. 2. The full state space (the left-hand side) and SSPS (the right-hand side) of the CP-net for the distributed database example with 3 symmetric database managers($n = 3$).

symmetries. The symmetry specification is required to be consistent [8] which means that it is required to only express symmetries that are actually present in the system. We will use Φ_{SG} to denote the group of permutation symmetries given by a consistent symmetry specification SG. In the rest of the paper we assume that a CP-net with places $P = \{p_1, p_2, ..., p_n\}$ is given together with a consistent symmetry specification SG which determines a group Φ_{SG} of permutation symmetries.

2.3 Restriction Sets

A consistent symmetry specification SG determines a group of permutation symmetries Φ_{SG}. During generation of the SSPS we need some kind of representation of Φ_{SG}. One possibility is to list the permutation symmetries. Since the symmetry groups used for the reduction can be very large, this is not a feasible solution.

A set of permutations of an atomic colour set can instead be represented as a *restriction set*. Restriction sets are introduced in [8] and formally defined in [1]. Here we will introduce restriction sets by means of an example. Below we use a restriction set to represent a subset of [`DBM` $\rightarrow$ `DBM`]. The set of permutations mapping `d(1)` to `d(2)` and the set {`d(2)`,`d(3)`} to the set {`d(1)`,`d(3)`} can be represented by the following restriction set:

d(1)	d(2)
d(2) d(3)	d(1) d(3)

Each row in the restriction set introduces a requirement for the set of permutations represented by the restriction set. The individual restrictions (rows) express that the colours on the left-hand side must be mapped into the colours of the right-hand side. In [1] it is proven that restriction sets can be efficiently intersected (while maintaining the compact representation) and that an arbitrary set of permutations can be represented by a set of restriction sets. Hence, restriction sets provide a potentially compact representation of sets of permutations. In the rest of the paper we will use restriction sets to represent sets of permutations of atomic colour sets. Hence, a symmetry specification can be represented by a set of restriction sets for each atomic colour set in the CP-net.

3 Condensed State Space Generation

In this section we give an introduction to the standard algorithm GENERATE-SSPS for construction of the SSPS of a CP-net [8]. *Nodes* and *Arcs* are sets of states (markings) and actions (binding elements), respectively, and it contains the states and actions that are included in the SSPS. *Unprocessed* is a set of states, and contains the states for which we have not yet calculated the successor

states. M_0 denotes the initial state. NEXT(M) is a function calculating the set of possible next moves (an action and the resulting state) from the state M. NODE(M) is a function that checks whether a node symmetric to M is already included in the SSPS. If not, M is added to *Nodes* and *Unprocessed*. Similarly, ARC(M_1,b,M_2) is a function that checks whether a symmetric arc is already included in the SSPS, i.e., an arc consisting of a binding element symmetric to b from a marking symmetric to M_1 to a marking symmetric to M_2. If not, (M_1,b,M_2) is added to *Arcs*.

Algorithm: GENERATESSPS () =

1: $\mathit{Nodes} \leftarrow \{M_0\}$
2: $\mathit{Arcs} \leftarrow \emptyset$
3: $\mathit{Unprocessed} \leftarrow \{M_0\}$
4: **repeat**
5:    select $M_1 \in \mathit{Unprocessed}$
6:    **for all** $(b,M_2) \in \mathsf{Next}(M_1)$ **do**
7:       NODE(M_2)
8:       ARC(M_1,b,M_2)
9:    **end for**
10:    $\mathit{Unprocessed} := \mathit{Unprocessed} \setminus \{M_1\}$
11: **until** $\mathit{Unprocessed} = \emptyset$

The algorithm proceeds in a number of iterations. In each iteration a state (M_1) is selected from *Unprocessed* and the successor states (and actions) are calculated using the NEXT function. For each of the successor states, M_2, it is checked whether *Nodes* already contains a state symmetric to M_2. If not M_2 is added to both *Nodes* and *Unprocessed*. Similar checks are made for the actions. The check for symmetric states and symmetric actions are instances of the *orbit problem*. From the basic generation algorithm it can be seen that efficient generation of the SSPS is highly dependent on the efficiency of the algorithms for determining the following two problems: 1) When reaching a new marking M during generation of the SSPS, is there a marking symmetric to M already included in the SSPS? And 2) When reaching a new arc (M_1, b, M_2) during generation of the SSPS, is there a symmetric arc already included in the SSPS?

GENERATESSPS is implemented in the Design/CPN OE/OS tool and used when calculating SSPSs for CP-nets. In Design/CPN a hash function is used when storing the markings during generation of the SSPS. When reaching a new marking during generation of the SSPS each marking stored with the same hash value is checked to see if it is symmetric to the newly reached marking, i.e., symmetry checks are performed locally between markings in the collision lists. The user of the tool is free to use his own hash function. The only requirement is that the hash function used is symmetry respecting, i.e., symmetric states are mapped to the same hash value. This is the case for the default hash function used in Design/CPN. Hence, when using the Design/CPN OE/OS tool for the generation of SSPSs efficient generation is dependent on the efficiency of the two predicates, P_M and P_{BE}, determining symmetry between markings and binding elements, respectively.

P_M: Given $M_1, M_2 \in \mathbb{M}$ determine whether $\exists \phi \in \Phi$ s.t. $\phi(M_1) = M_2$.
P_{BE}: Given $(t_1, b_1), (t_2, b_2) \in$ BE determine whether $\exists \phi \in \Phi$ s.t. $\phi(t_1, b_1) = (t_2, b_2)$.

The Design/CPN OE/OS tool requires P_M and P_{BE} to be implemented by the user. Implementing such predicates is error-prone for large CPN models and requires both programming skills and a detailed knowledge of the symmetry method. This is especially the case if the predicates are required to be efficient. The required user implementation of P_M and P_{BE} in the Design/CPN OE/OS tool has motivated the development of the Design/CPN OPS tool which given a CP-net and a consistent symmetry specification automatically generates the two predicates, P_M and P_{BE}, needed by the Design/CPN OE/OS tool. In the rest of the paper we will present techniques and algorithms to obtain implementations of P_M and P_{BE} in the Design/CPN OPS tool which are

efficient in practice. The algorithms are independent of the specific CP-net. Hence, the predicates can be automatically generated.

In the following discussions we will concentrate on the markings since the symmetry check between binding elements can be viewed as a special case of symmetry checks between markings: Given a transition t with variables $v_1, v_2, \ldots, v_m$, a binding b of t can be viewed as a vector of singleton multi-sets $(1`b(v_1), 1`b(v_2), \ldots, 1`b(v_m))$, where $b(v)$ denotes the value assigned to v in the binding b. Since transitions cannot be permuted by permutation symmetries in CP-nets determining symmetry between binding elements is the same as determining symmetry between markings. Hence, in the rest of the paper we will present techniques and algorithms to obtain an efficient implementation of P_M, i.e., given $M_1, M_2 \in \mathbb{M}$ determine whether $\exists \phi \in \Phi$ such that $\phi_{\mathbb{M}}(M_2) = M_1$.

4 Basic Algorithm for P_M

In this section we will present a basic algorithm which implements the predicate P_M. Section 4.1 presents the algorithm. Section 4.2 presents experimental results obtained using the Design/CPN OPS tool where the basic algorithm presented in this section is used to determine symmetry between markings.

4.1 Presentation of the Algorithm

The algorithm is based on a simple approach where Φ_{SG}, i.e., the group of permutation symmetries allowed by the symmetry specification SG, is iterated to determine whether $\exists \phi \in \Phi_{SG}$ s.t. $\phi(M_1) = M_2$. The algorithm P_M^{Basic} is given below.

Algorithm: $P_M^{Basic}(M_1, M_2)$

1: **for all** $\phi \in \Phi_{SG}$ **do**
2:    **if** $\phi(M_1) = M_2$ **then**
3:       **return** `true`
4:    **end if**
5: **end for**
6: **return** `false`

The algorithm repeatedly selects a permutation symmetry ϕ from Φ_{SG} (line 1) and tests whether ϕ is a symmetry between the two markings, M_1 and M_2 given as input (lines 2-4). The iteration stops when a permutation symmetry ϕ for which $\phi(M_1) = M_2$ is found (line 3) or the entire Φ_{SG} has been iterated (line 6).

The algorithm P_M^{Basic} potentially tests fewer permutation symmetries than $|\Phi_{SG}|$. This is however not the case if M_1 and M_2 are not symmetric. In that case the algorithm checks the whole Φ_{SG}. Hence, P_M^{Basic} is only useful for CP-nets with few permutation symmetries. This is also supported by the experimental results presented below.

However, before we present the experimental results of the P_M^{Basic} algorithm we will briefly introduce how it is tested whether a permutation symmetry ϕ maps a marking M_1 to another marking M_2 (line 2 in P_M^{Basic}). In [1] it is shown how the set of permutation symmetries between two markings can be determined as the intersection of the sets of permutation symmetries between the markings of the individual places. Hence, to determine whether a permutation symmetry $\phi \in \Phi_{SG}$ is a symmetry between two markings M_1 and M_2, we in turn test the multi-sets constituting the markings of $p_i \in P$. Note that if a permutation symmetry is not a symmetry for the marking of a place $p_i \in P$, i.e., $\phi(M_1(p_i)) \neq M_2(p_i)$ the permutation symmetry ϕ cannot be a symmetry between M_1 and M_2 and therefore there is no need to test the remaining places in P. Using

the ideas presented in [1] we obtain an algorithm TestPermutationSymmetry which given a permutation symmetry ϕ and two markings, M_1 and M_2, tests whether $\phi(M_1) = M_2$.

Algorithm: TestPermutationSymmetry(ϕ,M_1,M_2) =

1: **for all** $p_i \in P$ **do**
2:   **if** $\phi(M_1(p_i)) \neq M_2(p_i)$ **then**
3:     **return** `false`
4:   **end if**
5: **end for**
6: **return** `true`

The algorithm repeatedly selects a place $p_i \in P$ of the CP-net (line 1) and tests whether ϕ is a symmetry between the markings of p_i in M_1 and M_2 (line 2-4). If not, ϕ is not a symmetry between M_1 and M_2, otherwise a new place is tested. The iteration proceeds until a place $p_i \in P$ for which ϕ is not a symmetry is found (line 3) or all places have been tested (line 6).

4.2 Experimental Results of the P_M^{Basic} Algorithm

This section presents experimental results obtained using the Design/CPN OPS tool. The following results are obtained using an implementation of P_M^{Basic} to determine whether two markings are symmetric. A similar approach is used for the implementation of P_{BE}.

The Design/CPN OPS tool represents Φ_{SG} as a restriction set. When checking symmetry between two markings using P_M^{Basic} Φ_{SG} is listed and the permutation symmetries from the list are removed and tested until a permutation symmetry ϕ is found for which $\phi(M_1) = M_2$ or the list is empty.

SSPSs have been generated for two different CP-nets in a number of configurations. The CP-nets used in the experiments are briefly described below. For a detailed description of the CP-nets we refer to [8, 12].

Commit [12]. A CP-net modelling a two-phase commit protocol with a coordinator and w symmetrical workers.

Distributed database [8]. The CP-net presented in Sect. 2 modelling the communication between n symmetrical database managers.

Table 1 shows the generation statistics for of the SSPS for different configurations of the two CP-nets using the P_M^{Basic} algorithm. The CP-net column gives the name (C stands for commit and D stands for distributed database) and configuration of the CP-net for which the SSPS is generated as well as the number of permutation symmetries given by the symmetry specification SG used for the reduction. The Count column gives two numbers: the total number of times the P_M predicate is called during calculation of the SSPS and the number of calls which evaluate to true, i.e., the number of those calls which determine that the two markings are symmetric. The Tests column presents statistics on the number of permutation symmetries applied to markings during generation of the SSPS: Total gives the total number of permutation symmetries applied to markings during generation of the SSPS, P_M^{Basic} gives the average number of permutation symmetries applied in each call of P_M^{Basic}, P_M^{Basic}=`true` gives the average number of permutation symmetries applied in each call of P_M^{Basic} which evaluates to true (the case where iteration of the entire Φ_{SG} is potentially avoided), and finally, % $|\Phi_{SG}|$ gives the average percentage of the permutation symmetries which are tested in a call of P_M^{Basic}. Finally, the Time column gives the number of seconds it took to generate the SSPS for the given CP-net. A '–' in an entry means that the SSPS could not be generated within 600 seconds. All experimental results presented in this paper are obtained on a 333MHz PentiumII PC running Linux. The machine is equipped with 128 Mb RAM.

From Table 1 it can be seen that when system parameters increase the number of permutation symmetries tested increase significantly. This is caused by the increasing size of Φ_{SG}. From the

CP-net		Count		P_M^{Basic}				
				Tests				Time
Con.	$\|\Phi_{SG}\|$	P_M	P_M=true	Total	P_M^{Basic}	P_M^{Basic} =true	% $\|\Phi_{SG}\|$	Secs
C_2	2	11	7	19	1.73	1.57	78.5	0
C_3	6	26	19	90	3.46	2.53	42.0	0
C_4	24	53	41	488	9.21	4.88	20.3	0
C_5	120	95	76	3,242	34.1	12.7	10.5	0
C_6	720	157	127	27,297	174	44.86	6.2	23
C_7	5,040	–	–	–	–	–	–	–
D_2	2	4	2	7	1.75	1.50	75.0	0
D_3	6	14	6	61	4.36	2.17	36.0	0
D_4	24	35	15	533	15.2	3.53	14.7	0
D_5	120	71	31	5,037	70.9	7.64	6.37	0
D_6	720	126	56	51,693	410	23.1	3.20	16
D_7	5,040	–	–	–	–	–	–	–

Table 1. Generation statistics for SSPS generation using the P_M^{Basic} algorithm.

% $|\Phi_{SG}|$ column it can be seen that the average percentage of Φ_{SG} which are tested in P_M^{Basic} decreases when the system parameters increase. However, the increasing size of Φ_{SG} makes it impossible to generate the SSPS for the two CP-nets when system parameters, i.e., the number of concurrent readers or database managers, becomes greater than 6. This is also caused by the approach where Φ_{SG} is listed before the permutation symmetries are tested. For systems of increasing size $|\Phi_{SG}|$ imply that the entire Φ_{SG} cannot be represented in memory and, thus, generation of the SSPS is not possible. It should be noted that the results presented in Table 1 depends on the order in which the permutation symmetries are applied. The order used for the experiments is the same order in each call of P_M^{Basic} based on a recursive unfolding of the restriction set.

We conclude that the experiments performed using P_M^{Basic} in generation of SSPSs show that the run-time incurred by the iteration of Φ_{SG} becomes significant when system parameters grow. Hence, in order to make the calculation of SSPSs for CP-nets applicable in practice we need to carefully consider the number of permutation symmetries tested in the generation of the SSPSs. The next section presents techniques which improve P_M^{Basic} in this direction.

5 Approximation Techniques

In this section we will present an algorithm which presents an improved implementation of the predicate P_M^{Basic}. Section 5.1 presents the algorithm. Section 5.2 presents experimental results obtained using the Design/CPN OPS tool where the improved algorithm presented in this section is used to determine symmetry between markings.

5.1 Presentation of the Algorithm

The problem when using P_M^{Basic} for the symmetry check between markings is that in the worst case $|\Phi_{SG}|$ permutation symmetries will be checked. When determining symmetry between markings a selection of simple checks can in many cases determine that two markings are not symmetric or determine a smaller set of permutation symmetries that have to be checked.

In this section we will present a new algorithm for P_M which given two markings, M_1 and M_2, calculates a set Ψ_{M_1,M_2} such that $\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\} \subseteq \Psi_{M_1,M_2} \subseteq \Phi_{SG}$. Hence, Ψ_{M_1,M_2} is a super-set of the set of permutation symmetries mapping M_1 to M_2. If $\Psi_{M_1,M_2} = \emptyset$ we can conclude that M_1 and M_2 are not symmetric. However, if Ψ_{M_1,M_2} is non-empty we have to test the individual permutation symmetries in Ψ_{M_1,M_2}. In worst case $|\Psi_{M_1,M_2}|$ permutation symmetries have to be checked. This is the case if M_1 and M_2 are not symmetric. If M_1 and M_2 are symmetric then in worst case $|\Psi_{M_1,M_2}| - |\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\}| + 1$ permutation symmetries have to be checked. Hence, the goal of the approximation technique is to construct Ψ_{M_1,M_2} as close to $\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\}$ as possible.

In [1] it was shown that if a CP-net only contains atomic colour sets then the set $\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\}$ can be determined efficiently. This is, however, not the case if the CP-net contains structured colour sets. Nevertheless, we will use the technique to efficiently obtain an approximation Ψ_{M_1,M_2} of $\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\}$, thus reducing the number of permutation symmetries which have to be checked compared to the approach used in P_M^{Basic}. This is obtained at the cost of doing the approximation. In the following we will show how such an approximation can be obtained efficiently when Φ_{SG} is represented as a restriction set. The approximation technique is based on ideas from [8, 1].

The set of permutation symmetries mapping a marking M_1 to another marking M_2 can be found as the intersection of sets of permutation symmetries mapping $M_1(p_i)$ to $M_2(p_i)$ for all $p_i \in P$. Similarly, it is shown in [8] and proved in [1] how the set of permutation symmetries between such markings of places, i.e., multi-sets, can be determined as the intersection over sets of symmetries between sets with equal coefficient in the multi-sets, i.e., for a permutation symmetry to be a symmetry between ms_1 and ms_2 it must ensure that a colour appearing with coefficient c in ms_1 must be mapped into a colour appearing with the same coefficient in ms_2. We will illustrate using the CP-net of the Distributed Database (Fig. 1) as an example.

Let ms_1=`1‘d(2)+1‘d(3)` and ms_2=`1‘d(1)+1‘d(2)` be two markings of a the place Inactive with colour set `DBM`. In ms_1 two colours (`d(2)` and `d(3)`) appear with coefficient 1 and one colour (`d(1)`) appear with coefficient 0. We can express the multi-set of coefficients as `2‘1+1‘0`. In ms_2 it is also the case that two colours (`d(1)` and `d(2)`) appear with coefficient 1 and one colour (`d(3)`) appear with coefficient 0. Hence, ms_2 has the same multi-set of coefficients as ms_1 namely `2‘1+1‘0`. A permutation ϕ_{DBM} of the colour set `DBM` is a permutation mapping ms_1 to ms_2 if ϕ_{DBM} ensures that a colour appearing with coefficient 1 in ms_1 is mapped to a colour appearing with coefficient 1 in ms_2, and similar for the rest of the coefficients (here just 0). Hence, we can construct a restriction set representing the set of permutations between ms_1 and ms_2 by constructing a restriction for each of the coefficients appearing in ms_1 and ms_2.

Coefficient 0:	d(1)	d(3)
Coefficient 1:	d(2) d(3)	d(1) d(2)

In the above example the two multi-sets had the same multi-sets of coefficients. This is a necessary requirement for the two multi-sets to be symmetric [1]. If not, the left and right-hand sides of the constructed restrictions do not contain the same number of elements, and thus does not represent a valid set of permutations. Multi-sets of coefficients are formally defined in [1]. We define multi-sets of coefficients using the notation used in this paper below and present an algorithm which calculates the set of permutation symmetries between two multi-sets over an atomic colour set.

Definition:

For a multi-set ms over a colour set C we define $\text{COEFFICIENTS}_C(ms)$ as the set of coefficients appearing in ms:

$$\text{COEFFICIENTS}_C(ms) = \{i \in \mathbb{N} | \exists c \in C \text{ such that } ms(c) = i\}$$

Let ms be a multi-set over a colour C. For $i \in \text{COEFFICIENTS}_C(ms)$ we define the *i-coefficient-class* for ms as the set of colours in C appearing with coefficient i:

$$\mathsf{C}_i(ms) = \{c \in C | ms(c) = i\}$$

We define the *multi-set of coefficients* for ms by

$$\text{CFMS}(ms) = \{ms(i)`i\}_{i \in \mathsf{Coefficients}_C(ms)}$$

Based on the above definitions we formulate an algorithm FindPermutations which given two multi-sets ms_1 and ms_2 over an atomic colour set $A \in \Sigma_A$ calculate the set $\{\phi_A \in \Phi_A \mid \phi_A(ms_1) = ms_2\}$.

Algorithm: FindPermutations$_{ms}$(ms_1,$ms2$)
1: **if** Cfms(ms_1) = Cfms(ms_2) **then**
2: **return** $\{(C_i(ms_1),C_i(ms_2)\}_{i \in \mathsf{Coefficients}(ms_1)}$
3: **else**
4: **return** $\emptyset$
5: **end if**

The algorithm tests whether Cfms(ms_1) = Cfms(ms_2) (line 1), i.e., the multi-set of coefficients are equal. If not ms_1 and ms_2 are not symmetric [1], i.e., the empty set is returned (line 4), otherwise a restriction set is constructed containing a restriction $(C_i(ms_1),C_i(ms_2))$ for each of the coefficients i in Coefficients(ms_1) (line 2).

Given two markings, M_1 and M_2, the algorithm FindPermutationSymme-tries$_M$ calculates the a set of permutation symmetries Ψ_{M_1,M_2} as the intersection of Φ_{SG} and the sets of permutations between the markings of the individual places with atomic colour sets (calculated using FindPermutations$_{ms}$).

Algorithm: FindPermutationSymmetries$_M$(M_1,M_2) =
1: $\Phi' \leftarrow \Phi_{SG}$
2: **for all** $p \in \{p' \in P \mid p'$ has an atomic colour set$\}$ **do**
3: $\Phi' \leftarrow \Phi' \cap$ FindPermutations$_{ms}$($M_1(p)$,$M_2(p)$)
4: **end for**
5: **return** Φ'

If the CP-net only contains places with atomic colour sets the set Ψ_{M_1,M_2} of permutation symmetries calculated using FindPermutationSymmetries$_M$ is equal to the set $\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\}$. If the CP-net also contains places with structured colour sets then Ψ_{M_1,M_2} is a superset of $\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\}$, i.e, $\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\} \subseteq \Psi_{M_1,M_2}$. We will use FindPermutationSymmetries$_M$ to improve the P_M^{Basic} algorithm presented in Sect. 4, i.e., to reduce the number of permutation symmetries which have to be checked. The new algorithm P_M^{Approx} is presented below.

Algorithm: $P_M^{Approx}(M_1, M_2)$
1: **for all** $\phi \in$ FindPermutationSymmetries$_M$ (M_1,M_2) **do**
2: **if** TestPermutationSymmetry' (ϕ,M_1,M_2) **then**
3: **return** `true`
4: **end if**
5: **end for**
6: **return** `false`

The algorithm repeatedly selects a permutation symmetry ϕ from the set of permutation symmetries approximated using FindPermutationSymmetries$_M$ (line 1) and tests whether ϕ is a symmetry between the two markings (lines 2-4). The iteration stops when a permutation symmetry ϕ for which $\phi(M_1) = M_2$ is found (line 3) or the entire set has been iterated (line 6). P_M^{Approx} (M_1, M_2) uses TestPermutationSymmetry' (ϕ,M_1,M_2), a modified version of the algorithm TestPermutationSymmetry presented in Sect. 4, to determine whether $\phi(M_1) = M_2$. The difference is that given a permutation symmetry ϕ and two markings, M_1 and M_2, TestPermu-

TATIONSYMMETRY' only test ϕ on the places which have a structured colour set. The markings of the places with atomic colour sets are already accounted for in the approximation and do not have to be tested again.

The complexity of the calculation of FINDPERMUTATIONSYMMETRIES$_M$ is independent of $|\Phi_{SG}|$. This is a very attractive property, since the experimental results presented in Sect. 4 showed that iterating the group of permutation symmetries is not applicable in practice when the symmetry specification determines a large set of permutation symmetries.

If the CP-net contains places with atomic colour sets P_M^{Approx} potentially tests fewer permutation symmetries than P_M^{Basic}. In P_M^{Basic} at most $|\Phi_{SG}| - |\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\}| + 1$ permutation symmetries are checked when determining whether two markings are symmetric, whereas at most $|$FINDPERMUTA-TIONSYMMETRIES$_M(M_1, M_2)| - |\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\}| + 1$ permutation symmetries are tested using P_M^{Approx}. The experimental results presented later in this section show that for the two CP-nets used in the experiments the approximation is very close (or even equal) to the exact set of permutation symmetries mapping M_1 to M_2. Hence, the number of permutation symmetries which have to be tested is very low in practice. Furthermore, if the multi-sets of coefficients are different for markings no permutation symmetries have to be tested to determine that the markings are not symmetric. It should be noted that a necessary requirement for two markings M_1 and M_2 to be symmetric is that CFMS$(M_1(p_i))$ =CFMS$(M_2(p_i))$ for all places $p_i \in P$ (also for places with structured colour sets). Hence, an obvious way to improve P_M^{Approx} is to test the equality of multi-sets of coefficients for places with structured colour sets before checking any permutation symmetries. Places with atomic colour sets are already accounted for in the approximation.

5.2 Experimental Results of the P_M^{Approx} Algorithm

In this section we will present experimental results obtained using an implementation of P_M based on the P_M^{Approx} algorithm. The approximation operated directly on the restriction sets and the approximate set Ψ_{M_1,M_2} is also represented as a restriction set. Before checking the permutation symmetries in Ψ_{M_1,M_2} the approximated set is represented as a list. The permutation symmetries from the list are removed and checked until a permutation symmetry ϕ is found for which $\phi(M_1) = M_2$ or the list is empty. The experimental results are obtained using the two CP-nets presented in Sect. 4.

Table 2 presents generation statistics for SSPSs for different configurations of the two CP-nets using the P_M^{Approx} algorithm. The first four columns are the same as the first four columns in Table 1 presenting the generation statistics using the P_M^{Basic} algorithm. The CP-net column gives the name and configuration of the CP-net for which the SSPS is generated as well as the number of permutation symmetries given by the symmetry specification. The Count column gives two numbers: the total number of times the P_M predicate is called during calculation of the SSPS and the number of calls of P_M which evaluate to true, i.e., the number of calls which determines that the two markings are symmetric. The last six columns are specific to the P_M^{approx} algorithm. The Cfms column gives the number of calls of P_M^{Approx} for which the multi-sets of coefficients are different for the two markings, i.e., the number of calls of P_M^{Approx} where no permutation symmetries are tested. The Tests column presents statistics on the number of permutation symmetries applied to markings during generation of the SSPS: Total gives the total number of permutation symmetries applied to markings during generation of the SSPS, P_M^{Approx} gives the average number of permutation symmetries applied in each call of P_M^{Approx}, P_M^{Approx}=`true` gives the average number of permutation symmetries applied in each call of P_M^{Approx} which evaluates to true (the case where iteration of the entire Φ_{SG} potentially is avoided), and finally, $\%|\Phi_{SG}|$ gives the average percentage of the permutation symmetries which are tested in a call of P_M^{Approx} during generation of the entire SSPS. Finally, the Time column gives the number of seconds it took to generate the SSPS for the CP-net in the given configuration.

From Table 2 it can be seen that checking the multi-sets of coefficients before testing any permutation symmetries in P_M^{Approx} reduces the number of permutation symmetries tested compared

to P_M^{Basic}. It is worth noticing that in all calls of P_M^{Approx} which evaluated to false no permutation symmetries are tested, i.e., in all cases the multi-sets of coefficients differ. This is of course highly dependent on the CPN model and is a question of the amount of redundancy encoded in markings of places with structured colour sets. Furthermore, all calls of P_M^{Approx} which evaluated to true only in average requires one permutation symmetry to be tested. This is not a general fact of the technique. However, it is our experience from other experiments that in practice many CP-nets contains a degree of redundancy such that the approximation based on the atomic colour sets of the CP-net often is very close (or equal) to the exact set of permutation symmetries mapping one marking to another.

Even though the number of permutation symmetries tested after approximating the set of permutation symmetries is 1 for both CP-nets in all configurations it can be seen that SSPSs could not be generated for more than 7 database managers or workers. The reason is the memory required by P_M^{Approx}: in the implementation of P_M^{Approx} used for the practical experiments the approximated set of permutation symmetries is listed before testing the permutation symmetries. Hence, even though the approximation determines the exact set of permutation symmetries in worst case $|\Phi_{SG}|$ permutation symmetries are listed. Thus, in order to make the method applicable in practice we need to carefully consider the representation of the sets of permutation symmetries during generation of the SSPSs. This is the topic of the next section.

6 Lazy Listing

In the previous sections we have used sets of restriction sets to represent sets of permutation symmetries. The approximation technique presented in Sect. 5 operates directly on the restriction sets. However, in the implementations of both P_M^{Basic} and P_M^{Approx} the permutation symmetries are listed before they are checked. The major drawback of the approach presented in the previous section is that even though the approximation is exact or very close to $\{\phi \in \Phi_{SG} \mid \phi(M_1) = M_2\}$, i.e., only few permutation symmetries have to be checked, the entire approximated set is listed. The experimental results presented in Sect. 5 also showed that this approach is not applicable in practice since the memory use becomes a serious bottleneck as system parameters grow. The main goal of this section is therefore to improve the P_M^{Approx} algorithm such that a compact representation of the approximated set of permutation symmetries is maintained during calculation of P_M^{Approx}.

In this section we will present an algorithm which is an improved implementation of the predicate P_M^{Approx}. Section 6.1 presents the algorithm. Section 6.2 presents experimental results obtained using the Design/CPN OPS tool where the improved algorithm presented in this section is used to determine symmetry between markings.

				P_M^{Approx}					
CP-net		Count		cfms	Tests				Time
Con.	$\|\Phi_{SG}\|$	P_M	P_M=true		Total	P_M^{Approx}	P_M^{Approx}=true	$\%\|\Phi_{SG}\|$	Secs
C_2	2	11	7	4	7	1	1	50.0	0
C_3	6	26	19	7	19	1	1	16.7	0
C_4	24	53	41	12	41	1	1	4.17	0
C_5	120	95	76	19	76	1	1	0.83	0
C_6	720	157	127	30	127	1	1	0.14	1
C_7	5,040	242	197	45	197	1	1	0.02	60
C_8	40,320	–	–	–	–	–	–	–	–
D_2	2	4	2	2	2	1	1	50	0
D_3	6	14	6	8	6	1	1	16.67	0
D_4	24	35	15	20	15	1	1	4.17	0
D_5	120	71	31	40	31	1	1	0.83	0
D_6	720	126	56	70	56	1	1	0.14	0
D_7	5,040	204	92	112	92	1	1	0.02	7
D_8	40,320	–	–	–	–	–	–	–	–

Table 2. Generation statistics for SSPS generation using the P_M^{Approx} algorithm.

6.1 Presentation of the Algorithm

One way of viewing a set of permutation symmetries (represented as a set of restriction sets) is as a tree. Each level in the tree corresponds to possible images of a given colour. Hence, leafs in the tree represent the permutation symmetries given by the permutation of the individual elements found by following the path from the root to the leaf. Figure 3 shows a tree representing Φ_{SG} for the CPN model of the distributed database with 3 database managers and a consistent symmetry specification SG which allow all possible permutations of the atomic colour set `DBM`. The leafs of the tree represent the 6 different permutations symmetries in Φ_{SG}. Below each leaf the permutation symmetry is represented by a restriction set.

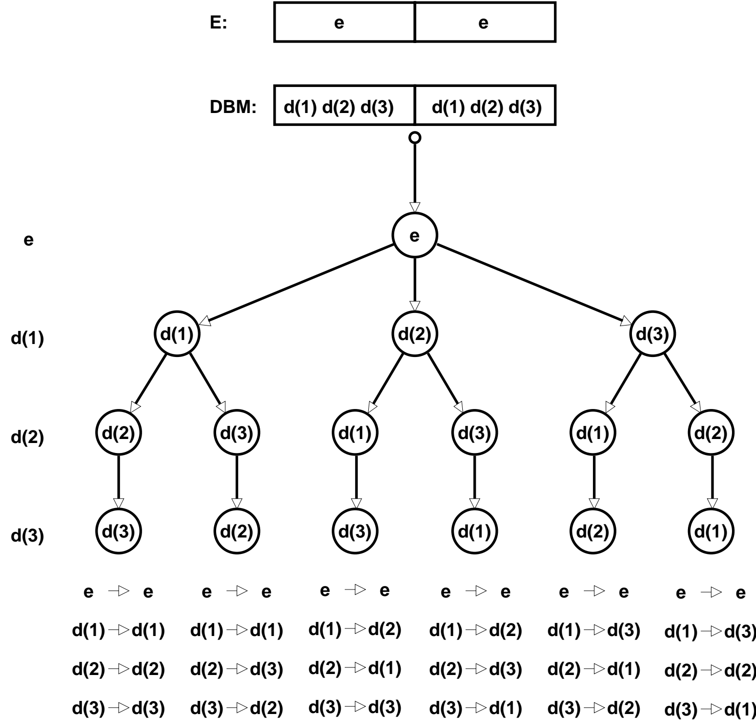


Fig. 3. All permutation symmetries in Φ_{SG} represented as a tree.

When testing a set of permutation symmetries on a marking in the P_M^{Basic} and P_M^{Approx} algorithms we first unfolded the restriction sets to a list of permutation symmetries and then applied the permutation symmetries from an end (until one was found or the entire set was checked). With the approach presented in this section we instead make recursive unfoldings of restriction sets based on a depth first generation of the 'tree view'; each node in the tree corresponds to a recursive call. Each time a leaf is reached the corresponding permutation symmetry is checked. If the permutation symmetry is a symmetry between the two markings checked we conclude that the markings are symmetric (and the iteration stops) otherwise the permutation symmetry is thrown away and the algorithm backtracks to generate the next permutation symmetry. In this way at most one permutation symmetry is contained in memory at a time. In a recursive call corresponding to the ith layer of the tree the algorithm only needs to keep track of the restriction set in the root as well as the images of the colours corresponding to the layers $1,..,(i-1)$. Hence, instead of listing potentially $\prod_{A \in \Sigma_A} |A|!$ permutation symmetries the algorithm needs to represent in the worst case images of at most $\Sigma_{A \in \Sigma_A} |A|$ colours plus the restriction set in the root. An algorithm

for such lazy listing of permutations symmetries represented by sets of restriction sets is shown below.

Algorithm: LazyList (i,Φ',M_1,M_2)

1: **if** SinglePermutationSymmetry(Φ') **then**
2: $\quad\phi \leftarrow$ GetPermutationSymmetry(Φ')
3: $\quad$**return** TestPermutationSymmetry(ϕ, M_1, M_2)
4: **else**
5: $\quad$*col* $\leftarrow$ GetColour(i)
6: $\quad$*images* $\leftarrow$ GetImages(Φ',*col*)
7: $\quad$*found* $\leftarrow$ *false*
8: $\quad$**repeat**
9: $\quad\quad$**select** $col' \in$ *images*
10: $\quad\quad$*images* $\leftarrow$ *images*$\setminus\{$*col'*$\}$
11: $\quad\quad$*found* $\leftarrow$ LazyList($i+1$,Split(Φ',*col*,*col'*,M_1,M_2))
12: $\quad$**until** *images* $= \emptyset \ \vee$ *found* $=$ true
13: **end if**
14: **return** *found*

The algorithm takes four arguments: i is the depth of the call (corresponds to the level in the tree), Φ' is a set of restriction sets representing a set of permutation symmetries, and M_1 and M_2 are the two markings which are checked. First LazyList (i,Φ',M_1,M_2) tests whether the set of restriction sets Φ' given as input represents a single permutation symmetry (line 1). If this is the case a leaf in the tree is reached and the result of applying the permutation symmetry is returned (lines 2-3). If the set of restriction sets represents more than one permutation symmetry (line 4) we have reached an internal node in the tree and a number of depth-first recursive calls are made (lines 8-12). The algorithm uses a number of functions which we will briefly describe below.

SinglePermutationSymmetry(Φ') returns true if Φ' represents a set of a single permutation symmetry and false otherwise.

GetPermutationSymmetry(Φ') returns one of the permutation symmetries in the set represented by the set of restriction sets Φ'.

TestPermutationSymmetry(ϕ, M_1, M_2) tests whether $\phi(M_1) = M_2$.

GetColour(i) returns the colour associated to the i'th level in the tree.

GetImages(Φ',*col*) returns the possible images of *col*, i.e., the right-hand side of the restriction in Φ in which *col* is contained in the left-hand side.

Split(Φ',*col*,*col'*) returns a new set of restriction sets which is similar to Φ' except that the restriction containing *col* has been split into two: one containing *col* in the left-hand side and *col'* in the right-hand side and one containing the remaining colours.

An algorithm combining approximation and lazy listing in the symmetry check between markings is given below.

Algorithm: $P_M^{Approx+Lazy}(M_1,M_2)$

1: $\Phi' \leftarrow$ FindPermutationSymmetries$_M$ (M_1,M_2)
2: **return** LazyList $(1,\Phi',M_1,M_2)$

The algorithm approximates the set of permutation symmetries using the technique presented in Sect. 5 (line 1). To avoid the lengthy listing the permutation symmetries in the approximated set are checked using the LazyList algorithm (line 2).

6.2 Experimental Results of the $P_M^{Approx+Lazy}$ Algorithm

In this section we present experimental results obtained using an implementation of P_M based on the $P_M^{Approx+Lazy}$ algorithm. The implementation represents the approximated set of permutation symmetries as a set of restriction sets. During calculation a compact representation is maintained using depth-first recursive unfoldings.

Table 3 presents the generation statistics for the generation of the SSPS for different configurations of the two CP-nets using the $P_M^{Approx+Lazy}$ algorithm. The CP-net column gives the name and configuration of the CP-net for which the SSPS is generated as well as the number of permutation symmetries given by the symmetry specification. The next three columns give the time it took to generate the corresponding SSPS using the three algorithms P_M^{Basic}, P_M^{Approx}, and $P_M^{Approx+Lazy}$, respectively.

CP-net	$\|\Phi_S G\|$	Time (secs)		
		P_M^{Basic}	P_M^{Approx}	$P_M^{Approx+Lazy}$
C_2	2	0	0	0
C_3	6	0	0	0
C_4	24	0	0	0
C_5	20	0	0	0
C_6	720	23	1	0
C_7	5,040	–	60	1
C_8	40,320	–	–	1
C_9	362,880	–	–	2
C_{10}	3,628,800	–	–	3
C_{15}	$1.3 \cdot 10^{12}$	–	–	77
D_2	2	0	0	0
D_3	6	0	0	0
D_4	24	0	0	0
D_5	120	0	0	0
D_6	720	16	0	0
D_7	5,040	–	7	1
D_8	40,320	–	–	2
D_9	362,880	–	–	4
D_{10}	3,628,800	–	–	8
D_{12}	$4.7 \cdot 10^8$	–	–	30
D_{15}	$1.3 \cdot 10^{12}$	–	–	151

Table 3. Generation statistics for SSPS generation using the $P_M^{Approx+Lazy}$ algorithm.

From Table 3 it can be seen that using the $P_M^{Approx+Lazy}$ algorithm it is possible to generate SSPSs for CP-nets with very large symmetry groups. When applying $P_M^{Approx+Lazy}$ for testing the permutation symmetries the same number of permutation symmetries is of course tested as if the permutation symmetries are listed beforehand using the P_M^{Approx} approach. However, the compact representation maintained during calculation saves space since the permutation symmetries are represented by sets of restriction sets. Hence, when combining the idea of lazy listing with the idea of approximations as presented in Sect. 5 significant speed up is gained. The reason is that in practice the approximations are often very close to or even equal to the exact set of symmetries between two markings. Thus, the number of permutation symmetries which have to be tested from large permutation groups is usually very small. Furthermore, the memory use of $P_M^{Approx+Lazy}$ caused by the size Φ_{SG} is no longer a bottleneck of the practical applicability of the method.

7 Conclusions

We have presented techniques and algorithms to determine whether two markings of CP-nets are symmetric. The algorithms presented are based on general and model independent techniques. Hence, the algorithms can be automatically generated for arbitrary CP-nets. The techniques are implemented in the Design/CPN OPS tool [13] which automatically generates the predicates P_M

and P_{BE} needed for the Design/CPN OE/OS Tool [11, 10]. The Design/CPN OPS tool has been used to conduct the experimental results presented in this paper.

The approximation technique that P_M^{Approx} is based on is introduced in [8, 1]. The contribution of this paper is to automate and implement the technique as well as integrate the technique into SSPS generation. The technique is specific to markings of CP-nets and is as such not general for the symmetry method.

The need for compact representations and avoidance of testing the entire group of permutation symmetries is, however, not specific to CP-nets. The algorithms and experimental results presented in this paper are therefore also relevant in other formalisms than CP-nets.

During SSPS generation we store an arbitrary marking from each equivalence class (the first state from the equivalence class encountered during generation of the SSPS). Another strategy is to calculate a canonical representative for each equivalence class. The symmetry check can then be reduced to a simple equivalence check. In [15] we have presented an algorithm for calculation of canonical markings of CP-nets. The algorithm requires the calculation of the minimal marking obtained as a result of applying a set of permutation symmetries. The algorithm for calculation of canonical markings of CP-nets presented in [15] encounters the same problem as the P_M^{Basic} algorithm presented in Sect. 4: applying the entire group of permutation symmetries is unfeasible in practice. In [15] we use algebraic techniques to reduce the number of iterations. Even with the use of algebraic techniques the canonicalization of markings experiences problems in practice due to the memory use required when working with large sets of permutation symmetries. The lazy listing approach presented in this paper can directly be used in the problem studied in [15] and it is envisioned that the lazy listing approach can alleviate the bottleneck caused by the memory use in [15].

It is possible to combine the use of algebraic techniques and the techniques presented in this paper to obtain a solution for P_M. However, since the approximation techniques only applies when two markings are compared the approximation techniques presented in this paper cannot be used directly in the algorithm for calculation of canonical markings of CP-nets.

References

1. R.D. Andersen, J.B. Jørgensen, and M. Pedersen. Occurrence Graphs with Equivalent Markings and Self-Symmetries. Master's thesis, Department of Computer Science, University of Aarhus, Denmark, 1991. Only available in Danish: Tilstandsgrafer med ækvivalente mærkninger og selvsymmetrier.
2. G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. On Well-Formed Coloured Nets and Their Symbolic Reachability Graph. In K. Jensen and G. Rozenberg, editors, *High-level Petri Nets*, pages 373–396. Springer-Verlag, 1991.
3. E.M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetries in Temporal Model Logic Model Checking. In Springer-Verlag, editor, *Proceedings of CAV'93*, volume 697 of *Lecture Notes in Computer Science (LNCS)*, pages 450–462. Springer-Verlag, 1993.
4. E.A. Emerson and A. Prasad Sistla. Symmetry and Model Checking. *Formal Methods in System Design*, 9, 1996.
5. D.J. Floreani, J. Billington, and A. Dadej. Designing and Verifying a Communications Gateway Using Coloured Petri Nets and Design/CPN. In J. Billington and W. Reisig, editors, *Proceedings of ICATPN'96*, volume 1091 of *Lecture Notes in Computer Science*, pages 153–171. Springer-Verlag, 1996.
6. C.N. Ip and D.L. Dill. Better Verification Through Symmetry. *Formal Methods in System Design*, 9, 1996.
7. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* Monographs in Theoretical Computer Science. Springer-Verlag, 1992.
8. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods.* Monographs in Theoretical Computer Science. Springer-Verlag, 1994.
9. K. Jensen. Condensed State Spaces for Symmetrical Coloured Petri Nets. *Formal Methods in System Design*, 9, 1996.
10. J.B. Jørgensen and L.M. Kristensen. *Design/CPN OE/OS Graph Manual.* Department of Computer Science, University of Aarhus, Denmark, 1996.
Online: `http://www.daimi.au.dk/designCPN/`.

11. J.B. Jørgensen and L.M. Kristensen. Computer Aided Verification of Lamport's Fast Mutual Exclusion Algorithm Using Coloured Petri Nets and Occurrence Graphs with Symmetries. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):714–732, July 1999.
12. L. M. Kristensen. *State Space Methods*. PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 2000.
13. L. Lorentsen. *Design/CPN OPS Graph Manual*. Department of Computer Science, University of Aarhus, Denmark, 2002.
    Online: `http://www.daimi.au.dk/~louisel/`.
14. L. Lorentsen and L.M. Kristensen. Modelling and Analysis of a Danfoss Flowmeter System. In M.Nielsen and D.Simpson, editors, *Proceedings of the 21th International Conference on Application and Theory of Petri Nets (ICATPN'2000)*, volume 1825 of *Lecture Notes in Computer Science*, pages 346–366. Springer-Verlag, 2000.
15. L. Lorentsen and L.M. Kristensen. Exploiting stabilizers and paralellism in state space generation with the symmetry method. In *Proceedings of the Second International Conference on Application of Concurrency to System Design (ICACSD'01)*, pages 211–220. IEEE, 2001.
16. K. Schmidt. How to Calculate Symmetries of Petri nets. *Actae Informaticae*, 36(7):545–590, 2000.
17. A. Valmari. The State Explosion Problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.

Implementation of Workflow Systems using Reference Nets
—
Security and Operability Aspects

Thomas Jacob[a], Olaf Kummer[b], Daniel Moldt[c], and Ulrich Ultes-Nitsche[d]

[a]Fachbereich Informatik, Universität Hamburg, Vogt-Kölln-Str. 30, 22527 Hamburg, Germany, e-mail: mail@ThomasJacob.de
[b]CoreMedia AG, Erste Brunnenstr. 1, 20459 Hamburg, e-mail: olaf.kummer@coremedia.com
[c]Fachbereich Informatik, Universität Hamburg, Vogt-Kölln-Str. 30, 22527 Hamburg, Germany, e-mail: moldt@informatik.uni-hamburg.de
[d]Department of Electronics and Computer Science, University of Southampton, Southampton, SO17 1BJ, United Kingdom, e-mail: uun@ecs.soton.ac.uk

Abstract. We present in this paper a generic approach to designing and running workflows in a flexible way. Our language of choice is reference nets, a version of high-level Petri nets that allow creation of new net instances as well as communication between net components in an object-based way.
We discuss the general approach to workflow design as well as the specifics of reference-net-specified workflow: their flexibilities and, related to this, security aspects of reference-net workflows. Reference nets allow the implementation of even elaborate security concepts for workflow systems in an elegant and simple way.

1 Introduction

To support complex business processes, workflow systems are promoted by various institutions and IT companies. Their aim is to use information technology to manage the different tasks that must be completed in order to complete a business process successfully. In addition, workflow systems must deal with incomplete tasks, missed deadlines, etc.

Various approaches have been proposed for describing workflows. We present in this paper a Petri-net-based approach, using reference nets [6]. Reference nets allow object-based design of a workflow, and the run-time environment Renew (REference NEt Workshop) [6] can be used as the basis for the workflow engine executing reference-net workflows. Some of the modelling concepts have been introduced in [10]. However, support by a tool was missing.

We discuss how the Renew tool has been extended to create a workflow engine [3]: A *task symbol* has been introduced to reference nets to allow for a more compact representation of workflows. The task symbol does not at all change

the abilities of reference nets as they can be easily compiled to a three-transition reference-net structure. In fact, tasks only occur on the graphical level of the Renew editor, but are resolved into the aforementioned reference-net structure in the internal representation of reference-net workflows. The workflow engine, as an extension to the Renew simulation engine, can handle tasks and the way how users select and perform tasks.

In addition, we present how the system, supporting the general concepts of reference nets, can be used to implement elaborate access controls to tasks and data. To each task, we propose the implementation of a task rule, which is simply a Java method that evaluates to type `boolean`, deciding whether or not access to a task will be granted (see [3] for details). This will enable us, for instance, to implement simple role-based access controls as well as sophisticated content-dependent ones [8] such as "need-to-know" access controls [1]. Having defined a generic approach, we can integrate any available information in the access control decision procedure.

Our paper is structured as follows: After a brief introduction of reference nets using an example taken from [6] in the next section, we introduce workflow specifications in the third section. Section 4 contains a discussion of the workflow engine, followed by Section 5 on workflow security. An example in section 6 illustrates some features for our workflow engine.

2 Reference Nets

Reference nets are high-level Petri nets that implement the nets-within-nets concept (see [9]) and incorporate Java annotations. In [6] a formal definition is given. The Java annotations control enabling a transition and can create side effects when a transition is fired. Firing a transition can also create a new instance of a subnet in such a way that a reference to the new net instance will be put as a token into a place. A net instance can communicate with the subnets it has created if it possesses a reference to the subnets. Communication occurs through synchronous channels associated with transitions. A net instance can also communicate with nets whose subnet it is. Subsequently, we introduce the different concepts present in reference nets using an example taken from [6]. We assume the reader's familiarity with the general notions of place/transition nets (P/T nets) and coloured Petri nets.

We start with the simple P/T net depicted in Figure 1. It describes two people who either work at home or in the office, earn money when communicating with one another while both being in the office and spend money by commuting to and from the office.

If we move in this example from P/T nets to coloured Petri nets [4], we can fold the two subnets representing `A` and `B` into a single coloured Petri net, which is presented in Figure 2.

To introduce the first concept, synchronous channels (see [5] and [2]), in this paper we extend our example slightly: We separate the "money storage" part

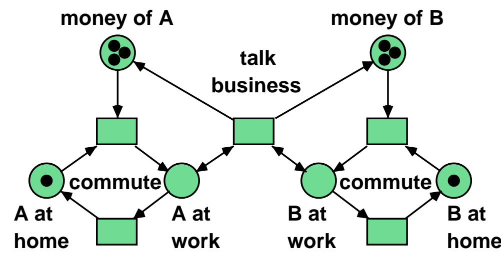


Fig. 1. P/T net example: Co-operation of two business people.

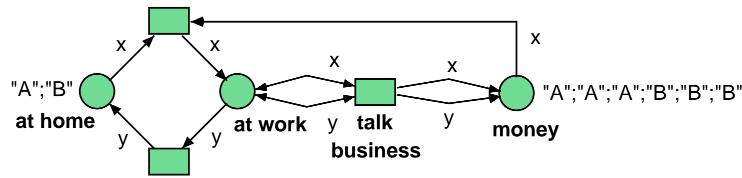


Fig. 2. Coloured Petri net version of the example.

from the "working life" part in our net by creating a simple bank account model. The model is shown in Figure 3.

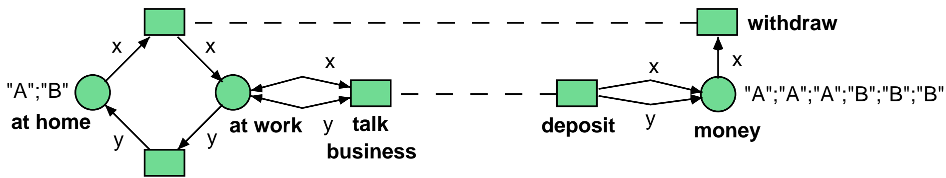


Fig. 3. Separation of "working life" and "money storage".

The model with separate account and person specifications will not provide the functionality of our initial simple model without the definition of some means of co-ordination between the two separate nets. The co-ordination concept we use is communication via parameterised synchronous channels. In this concept, one transition "owns" a channel, which is labelled with `:channelName(parameters)`. Other transitions can be synchronised with transitions using the channel by labelling them with `netName:channelName(parameters)`. If the channel is used for synchronisation within the same net, `netName` will be `this`. Otherwise `netName` is the name of the net, which contains the transition owning the channel. A transition owning a channel is enabled if and only if it is enabled in the usual sense

as well as all transitions elsewhere whose label refers to the channel are enabled in the usual sense. This includes taking into account the value bindings of variables that can occur as channel parameters. A transition using but not owning a channel will be enabled whenever the transition owning the channel is enabled. Enabled transitions sharing a channel will fire at the same time, i.e. in a synchronous fashion. Figure 4 shows the net from Figure 3 extended by channels `deposit(_,_)` and `withdraw(_)` to co-ordinate the two subnets.

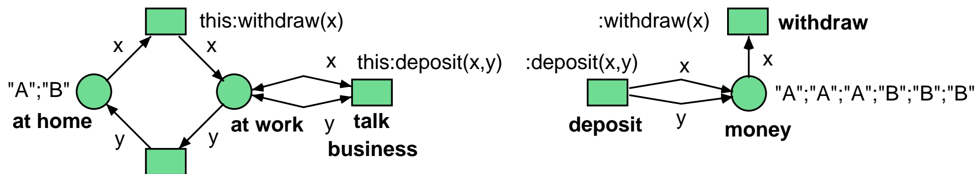


Fig. 4. Introducing synchronous channels.

By using a channel twice in a transition, we can enforce firing another transition two times as a kind of atomic event. Figure 5 shows this concept in a net, which is behaviourally equivalent to the one in Figure 4.

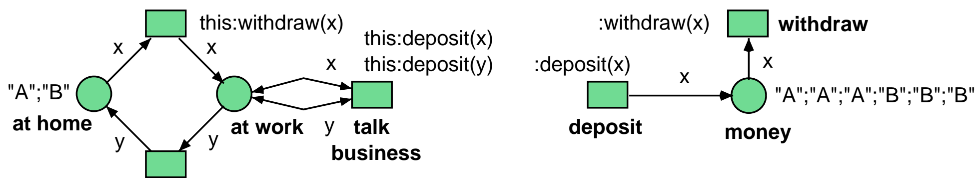


Fig. 5. Multiple channels at one transition.

Instead of dealing with the multiplicity of similarly named tokens (e.g. "`A`") to represent `A`'s credit balance, we will be dealing with integer values. This idea is introduced in Figure 6.

Having introduced synchronous channels will allow us to create instances of subnets and to communicate with subnets in an object-like way. Using this concept enables us to use different instances of the same `account` net to handle the accounts in our example separately. A new instance of a subnet is created by using the construct `name:new netName` where `new` is a keyword and `name` and `netName` are a reference to the new instance of the net and the name of the net, respectively. Figure 7 shows the `account` subnet and Figure 8 the (main) `person` subnet of our final specification.

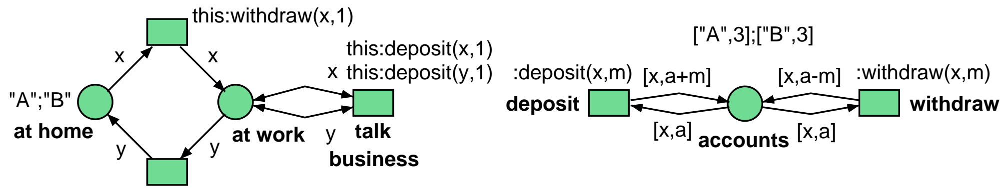


Fig. 6. Introducing integers.

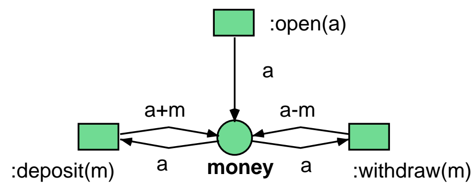


Fig. 7. Description of an account.

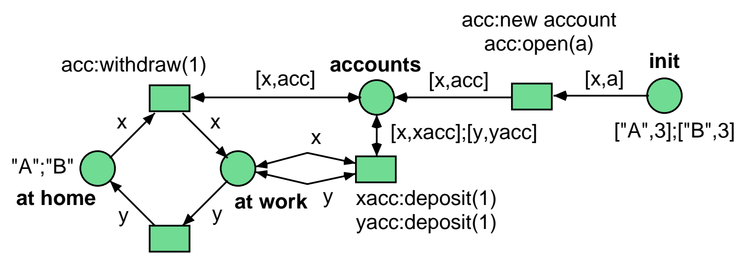


Fig. 8. Description of `person`.

Initially, two instances of net `account` will be created and references to the two instances will be stored as part of the tokens in place `accounts`. Communication with the net instances occurs as before via named, parameterised synchronous channels.

Presenting Figures 7 and 8 completes our brief overview of the capabilities of reference nets. It should be mentioned, finally, that reference nets allow for Java annotations (objects and methods) as net inscriptions, resulting in a flexible modelling language for workflows. The reference net tool Renew (REference NEt Workshop), extended in a suitable way, will then provide us with a runtime environment for workflow processes.

3 Workflow Specifications using Reference Nets

The aim of a workflow specification is to unambigously describe the *control flow* as well as the *tasks* that together implement a business process. Tasks are the atomic work items that need to be completed in order to successfully complete the business process. Control flow is the co-ordination process that controls the order of tasks to be performed, including dependencies between the tasks and their possibly various outcomes.

Petri nets, and reference nets in particular, are a very natural approach to describe the control flow in a workflow: Their strength is indeed the description of the flow of information through the net's structure and, by the firing rules of transitions, to describe dependencies between various events. Even though it is fairly evident that tasks will be somehow related to transitions, as transitions are the actions in a Petri net specification, the atomicity of transitions is slightly too strict for mapping tasks onto transitions directly. We will clarify this statement subsequently, which will lead to the definition of a task object, which we then will map onto a net structure formed by three transitions.

Before discussing its formalisation, we first have to clarify what the basic ingredients of a task are in a workflow. In crude terms: a task is an atomic unit of work that will be completed by an individual. Its support can range from completely manual (no support at all) to complete automation (no involvement of an individual but only software). The result of a task can be either positive or negative. In the positive case, the task will be completed, usually creating some kind of output which will then be provided to control the workflow and subsequent tasks. In the negative case, the task will be terminated unsuccessfully, a case in which the state of the workflow has to be reset to the one prior to the attempt of performing the task. So, even though a task is atomic within a workflow, it has some structure from the modelling point of view.

We will therefore model a task using three transitions representing *entering* the task as well as either *success* or *failure* of completing a task. Firing the entry transition will create an instance of a task-related object, which could include instantiation of a sub-net, creating e.g. a sub-workflow, instantiation of a software object needed to deal with the task, and extension of a to-do list of work items. When the task is completed, its result will tell whether the

success or failure transition has to fire. There are no time constraints imposed on the duration between entering and leaving the task. We hence have inherently solved an additional requirement for modelling tasks: performing tasks requires time. The lower part of Figure 9 shows the basic structure of a task construct in our reference-net-based modelling technique. Please note that the topmost entry places are chosen arbitrarily as well as the output places at the bottom of Figure 9.

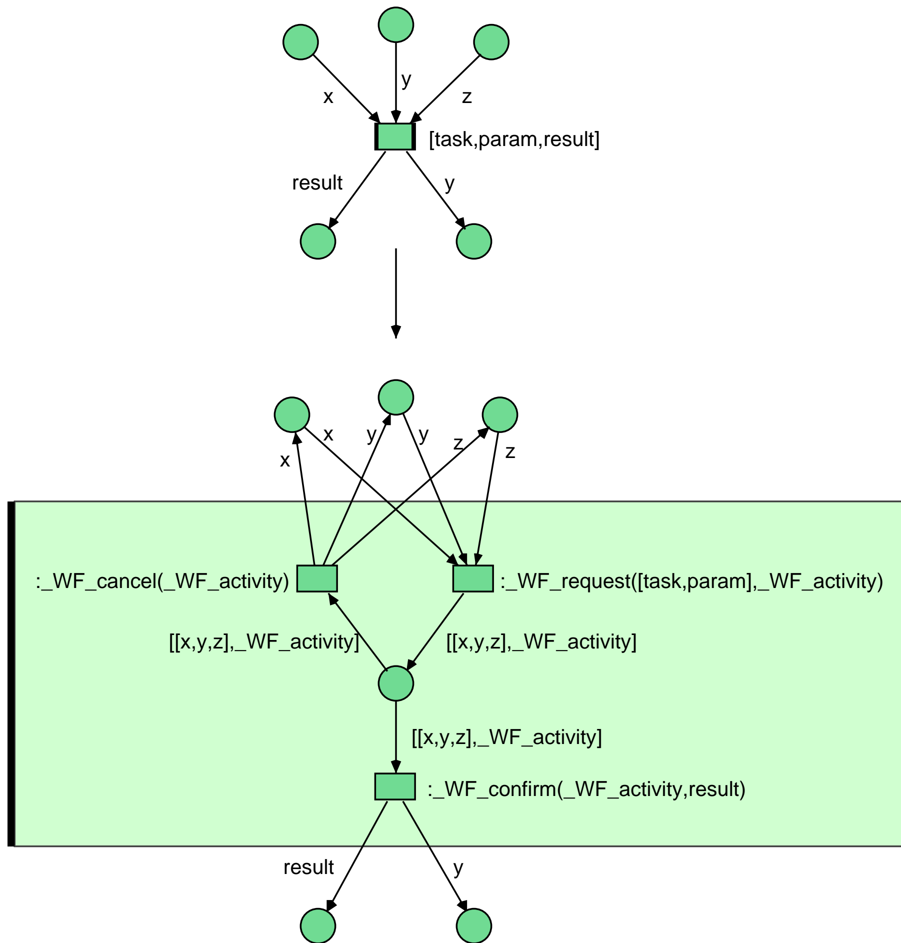


Fig. 9. Specification of a workflow task.

`_WF_request([task,param],_WF_activity)` is the transition that will be fired to invoke the task. In case of an unsuccessful task termination, transition `_WF_cancel(_WF_activity)` will fire. Note that `_WF_cancel(_WF_activity)`

will put back all tokens initially taken from the entry places when firing transition `_WF_request([task,param],_WF_activity)`. This restores the workflow state prior to entering the task. In case of a successful task completion, firing transition `_WF_confirm(_WF_activity,result)` will make the result of the task available to subsequent tasks. It can also be used for control flow purposes.

The subnet in the gray box will always have exactly the same structure. What will vary are the input and output places to the task. For the convenience of the presentation as well as the specification, we introduce therefore a new symbol that represents a task specification. As depicted, we represent the subnet shown in the lower part of Figure 9 by using the *task symbol*, which is presented in its upper part.

4 The Runtime Environment

The runtime environment is basically provided by the tool *REference NEt Workshop* (Renew). Renew provides a full simulation environment for reference nets, including the invocation of Java methods that occur as reference-net inscriptions and the handling of concurrent net processes. In particular, it handles the synchronisation of firing transitions based on their channel inscriptions. To make Renew incorporate a complete workflow engine, `Tasks`, `WorkItems`, and `Activities` as well as their storage and handling have been implemented in additional packages to the Renew system.

4.1 The Specification Language

The Renew system has been extended to deal with task symbols. They will be compiled down to reference-net structures as presented in Figure 9 for an example task. The compilation process was integrated fairly easily into Renew because of Renew's general structure: Between the level of the graphical editor, which has been extended by the task symbol, and the level of the simulator, Renew contains an additional layer called the *shadow level.* The shadow level uses an internal representation of the net drawn on the net-editor level. The translation process from the editor to the shadow level could therefore be extended by a compiler for task symbols that replaces each task symbol by the three-transition sub-net it represents (Figure 9). On the shadow level we end up with only reference nets without task symbols, which then can be fed into the simulation engine as usual. To add task-specific functionality to the simulation engine, it has been extended by a workflow engine.

4.2 The Workflow Engine

The workflow engine comprises as one of its main features an interface to workflow users: Users will be able to access their agendas, i.e. task lists, using the workflow engine. These task lists will be filtered so that a user sees only tasks she/he is authorised to work on (see next section). Users can then select a task

from their task list to deal with it and inform the system about the successful or unsuccessful completion (termination) of a task. The entire handling of tasks lies within the packages that have been added to Renew's simulation engine. This includes returning unfinished (unsuccessfully terminated) tasks to task lists, restoring state information in case of unsuccessful termination of tasks, dealing with automatic tasks (tasks that can be performed without direct user involvement), and so on. It should be noted that the additional packages provide interfaces to functionality already present in Renew, controlling, for instance, the firing of transitions according to the state of tasks. They also comprise a task database for the efficient storage of tasks and their inclusion to the aforementioned task lists.

The workflow engine fulfills therefore two tasks: the control of workflow processes via the Renew simulation engine and the management of tasks, users, and the processing of tasks by users. As mentioned above, tasks may only be processed by a user if she/he has the authority to do so. The handling of access rights is done by rules added to tasks, which form the major part of the workflow security system.

5 Workflow Security

In this section we will give a short overview of security issues in the context of workflow systems. The first subsection covers general aspects while the second one covers how access control has been integrated in our workflow concept and workflow engine.

5.1 Overview

The discussion of security in workflow systems can basically be reduced to discussing access control. In all other aspects, such as confidentiality of messages, authentication, etc., workflow systems are no different from other applications. The access-control aspect of workflow systems deals with issues like: "Who is responsible for performing a task and therefore needs access to data and software supporting working on the task?" Also delegation of work and related access rights are important issues within a workflow system. We will discuss in this section how reference-net-based workflow specification supports an elaborate access-control mechanism.

We use in our approach an extension to the role-based approach.[1] In role-based access controls, users are assigned roles. Roles are simply named collections of users, where a single user can appear in multiple roles. There is also a sub-role relation that holds true, if all users in one role (the sub-role) are also contained in another role (the super-role). For the ease of discussion, we will view users subsequently as singleton roles. They are the only "roles" than can log-in to the system.

[1] Role-based access controls are sometimes also called group-based.

In the *purely* role-based approach, rights to perform specific tasks (and therefore access rights to the relevant software) are assigned to roles. A role inherits access rights from its super-roles. This is all that is possible in a purely role-based access control. To extend it slightly by supporting delegation, to perform a single task (and using the relevant software) a role may pass on its own access rights temporarily to another role. This temporary extension of a role's access rights will expire as soon as the task is completed (either successfully or unsuccessfully).

Even though delegation of access rights makes role-based access controls in a workflow environment more flexible, it is still a quite static concept. Very frequently, access permissions to data or software will vary over time; i.e. in different workflow states, access rights be assigned differently. So more flexible access-controls have emerged. The idea of *context-dependent* access controls, for instance, takes into account that access to particular data will normally only be important to perform one task in a workflow, but not another task. So state information about the workflow is combined with role information to compute access rights. This context-dependent concept, where the context is the workflow state, can be applied to workflows in environments in which state-dependent protection of data is legally required, for instance in clinical trials.

In the approach we present in this paper we go even one step further and allow *all* available information to be part of the access control decision, i.e. we keep the role concept but combine it with information about the workflow's state, data-base information, states of running programs, states of other workflow instances, etc. The aim is to be as flexible as possible. Obviously the utmost flexibility will never be needed. So, in practice, restrictions will apply. However, in different environments, in which workflow systems can support daily work, security requirements differ. Our approach can therefore be adapted to the different environments by simply not using all its possibilities. In that sense, the approach we present in this paper is a generic one.

5.2 Access Control in a Reference-Net Workflow

As mentioned in the previous sub-section, the aim of access controls in workflows is to define who is and who is not allowed to perform a task and access relevant data for this task. The access control system is the implementation of such an access control policy. In the modelling technique we are using, we will assign an *access rule* to each task. The rule will be evaluated to check the authority of an attempt to perform a task.

The first thing to note is that by moving the access control to the task level, our access control is a context-dependent one: Aiming to perform a task will only be possible if the role trying to do so has the authority for performing the task *and* if the workflow is in a state such that the task is active. However, an access rule for a task is in principle a method of type `boolean` that checks the authority of a request to perform the task. It may use any information available in its environment, including remote information if it has network access. This offers the greatest flexibility possible.

Our approach, as a generic one, is open to where the access control is implemented in the workflow. There are basically two options: in the control flow or in the task inscription.

- (**Control flow:**) We can specify the workflow in such a way that, by using control input places to tasks, tasks will only be enabled to be performed by a role, if all other requirements constraining the access to a task are satisfied (i.e. the rule is satisfied). The satisfaction or dissatisfaction of these requirements will be reflected by the availability or unavailability of tokens in a task's control input place necessary to enable the task.
- (**Task inscription:**) As reference net's allow for the inscription of transitions with Java code, we can implement the access control method to a task by an inscription of the entry transition of the task. The task will only be enabled when its entry transition is enabled, and the entry transition can only be enabled by being in the correct workflow state plus the evaluation of the access control method to `true`.

Obviously we can mix the two concepts as we like in a reference-net workflow specification, controlling one task using the control-flow scheme and another task by a task inscription. However, in practice, combining the two approaches in one specification is probably not a good idea as it will decrease the readability of the specification and probably increase the likelihood of introducing errors.

6 An Example Workflow

This section presents an example of a workflow for a mobile-phone shop. It contains several aspects of the workflow engine. In this contribution we include only the pure nets which have been executed by our workflow engine. The environment, the data description, and some other auxilary classes are not shown here.

Figure 10 represents the main net of a mobile-phone shop. Figure 11 contains the check of some customer data and Figure 12 contains the delivery of the goods as well as the check if a lower bound for the amount of goods has been reached.

The main net starts with a transition labeled with `manual` (that is the `new customer` transition): A new order form is created. Then the `EnterOrderData`-task is displayed at all salesmen displays. After its completion a new instance of the form exists.

The larger places in the net are used to represent the existing orders for a customer. The actual order now lies in place `unchecked orders`.

For this order a `WorkflowTask` for the `check of the order` (a net instance of Figure 11) is instanciated. `:start(activity,order)` allows the access to the order within the net instance of Figure 11. Within this net a `WorkItem` will be offered to the office workers in both cases (payment by credit card or bank). Depending on the result of the check a check of the address may be performed. The main net in Figure 10 now will continue according to the result. In the

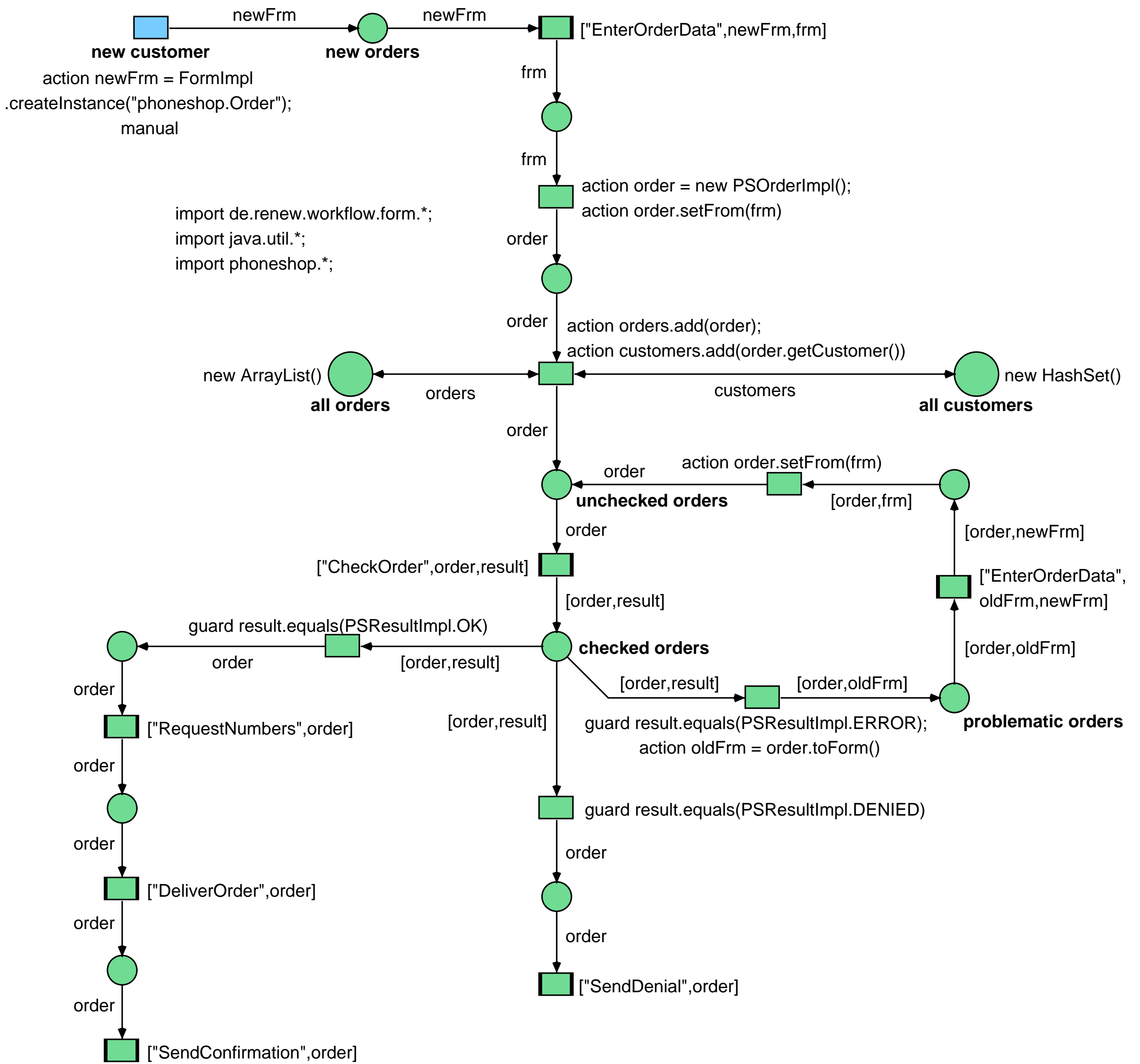


Fig. 10. Main net: Mobile-phone Shop

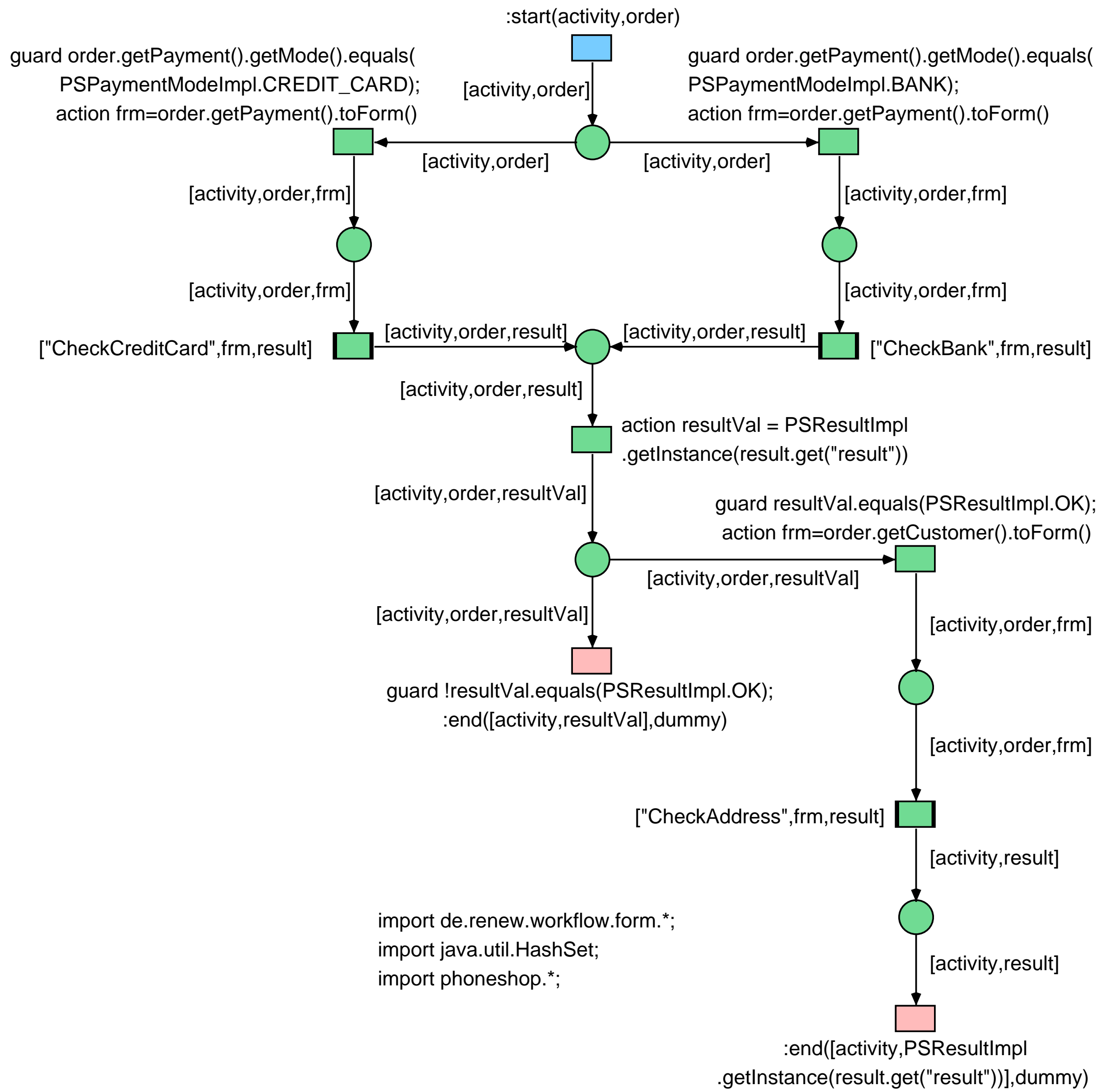


Fig. 11. Check order limit

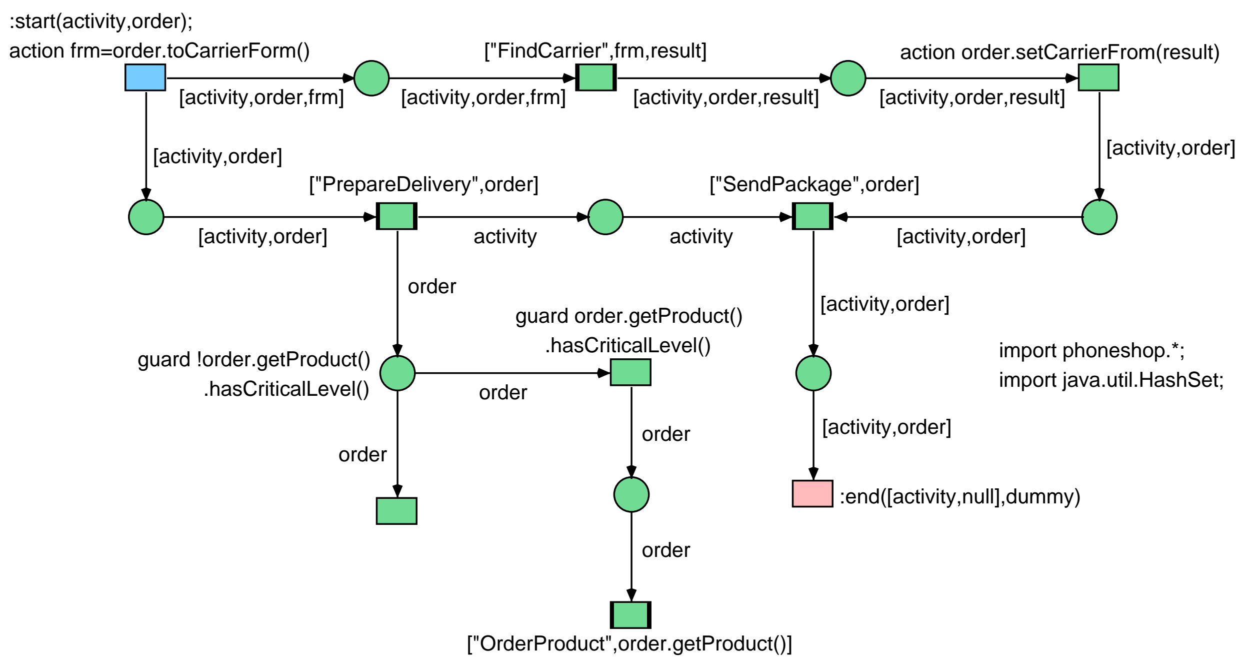


Fig. 12. Shipping of goods

positive case it will continue with the request for phone numbers and then with the sub-workflow in Figure 12 where the shipping is handled.

This last net shows the possibility to finish a workflow for the upper level by calling the `:end(activity,dummy)` transition and still having active parts within the net. Once they are finished the whole net instance is removed from the system if there is no log function in the system. This is possible due to the automatic garbage collection of the tool Renew. The designer of the net has to ensure that the nets are designed in a way that allows to use this feature, however, we will not deepen the discussion here.

This example covers only the operability, however, security issues can be added as described in the sections above. The full modelling (or programming) power of the workflow engine can be seen when looking at the possibilities of the net concepts that can be used and the inscriptions which cover nearly all kind of Java code. One main attempt of our research is now to find the right restrictions and short cuts when designing workflow systems. Following our overall approach we aim at the integration of modelling and implementation. Without tool support this does not seem to be feasible.

7 Conclusion

We have discussed in this paper how reference nets can be used to specify workflows and how the REference NEt Workshop (Renew) tool [6] can be extended to form the corresponding workflow engine. The major additional concept introduced was that one of a *task*. We were able to reduce the concept of a workflow task to a three-transition reference-net structure. Such a structure has been integrated into the three-layer architecture of Renew [3]. By adding a component to handle tasks, tasks lists of user, etc., Renew was turned into a complete run-time environment for reference-net workflows. The basic mechanism of the workflow engine has successfully been used in the context of mobile devices (see [7]).

The concept of adding *rules* to tasks enabled us to develop a generic concept for access controls in the workflow engine. As these rules are simply Java methods of type `boolean`, we can flexibly implement any existing access control scheme into our formalism, including role-based and content-dependent access controls [1, 8], based on the implemented features (see [3]). The flexibility of the concept allows for including any other kind of information to check the authority of an access request to a task, such as date and time, states of other workflows, system parameters, and so on. It should be noted that, because of the flexibility of the approach, we only have created a generic concept for the design and implementation of secure workflows. The framework will have to be filled with design strategies that will again impose constraints on the flexibility of implementing task rules. Otherwise the entire approach can become too complex when specifying concrete workflows, which can lead to undetected introduction of security holes hidden behind a too complex mechanism. It will, together with applying our approach to case studies, be part of future work to define such security design strategies.

References

1. RALPH HOLBEIN. Secure Information Exchange in Organisations. Dissertation. Institute for Computer Science, University of Zürich, Switzerland, 1996. (Published by Shaker Verlag, Aachen, Germany, 1996.)
2. SØREN CHRISTENSEN and NIELS DAMGAARD HANSEN. Coloured Petri Nets Extended with Place Capacities, Test Arcs and Inhibitor Arcs in: Marsan, Marco Ajmone (Ed.). *14th International Conference on Application and Theory of Petri Nets.* LNCS 691, Berlin Heidelberg New York: Springer Verlag, pages 186–205, 1993
3. THOMAS JACOB. Implementierung einer sicheren und rollenbasierten Workflowmanagement-Komponente für ein Petrinetzwerkzeug. Diploma Thesis. Department for Informatics, University of Hamburg, Germany, 2002.
4. KURT JENSEN. Coloured Petri Nets. EATCS Monographs on Theoretical Computer Science. 3 Volumes, Berlin Heidelberg New York: Springer Verlag, 1992, 1994, and 1997.
5. OLAF KUMMER. A Petri Net View on Synchronous Channels. In: *Petri Net Newsletter* 56: 7–11, 1999.
6. OLAF KUMMER. *Referenznetze.* Dissertation. Department for Informatics, University of Hamburg, Germany, 2002.
7. STEFAN MÜLLER-WILKEN and WINFRIED LAMERSDORF. JBSA: An Infrastructure for Seamless Mobile Systems Integration. In: Claudia Linnhoff-Popien and Heinz-Gerd Hegering (Eds.): *Proc. 3rd IFIP/GI International Conference on Trends towards a Universal Service Market (USM 2000)*, Berlin Heidelberg New York: Springer Verlag, 164-175, 2000
8. ULRICH ULTES-NITSCHE and STEPHANIE TEUFEL. Secure Internet-Access to Medical Data. In: Informatik — Journal of the Swiss Chapter of the ACM. No. 1, February 2001. Special Issue on IS and the Transformation of Health Care, pages 23–26.
9. RÜDIGER VALK. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In: *19th International Conference on Application and Theory of Petri Nets.* LNCS 1420, Berlin Heidelberg New York: Springer Verlag, pages 1–25, 1998.
10. WIL M.P. VAN DER AALST, DANIEL MOLDT, RÜDIGER VALK, and FRANK WIENBERG. Enacting Interorganizational Workflows Using Nets in Nets. In: *Proceedings of the 1999 Workflow Management Conference "Workflow-base Applications".* University of Münster, Germany, pages 117–136, 1999.