Editors: Michael Duvigneau and
Daniel Moldt and
Kunihiko Hiraishi

Proceedings of the International Workshop on

P etri
N ets and
S oftware
E ngineering
PNSE'11

University of Hamburg
Department of Informatics

Preface

These are the proceedings of the International Workshop on *Petru Nets and Software Engineering* (PNSE'11) in Newcastle upon Tyne, United Kingdom, June 20–21, 2011. It is a co-located event of *Petri Nets 2011*, the 32nd international conference on Applications and Theory of Petri Nets and Concurrency, and *ACSD 2011*, the 11th International Conference on Application of Concurrency to System Design.

More information about the workshop can be found at

`http://www.informatik.uni-hamburg.de/TGI/events/pnse11/`

For the successful realisation of complex systems of interacting and reactive software and hardware components the use of a precise language at different stages of the development process is of crucial importance. Petri nets are becoming increasingly popular in this area, as they provide a uniform language supporting the tasks of modelling, validation, and verification. Their popularity is due to the fact that Petri nets capture fundamental aspects of causality, concurrency and choice in a natural and mathematically precise way without compromising readability.

The use of Petri nets (P/T-nets, coloured Petri nets and extensions) in the formal process of software engineering, covering modelling, validation, and verification, is presented as well as their application and tools supporting the disciplines mentioned above.

The program committee consists of:

Kamel Barkaoui (France)
Piotr Chrzastowski-Wachtel (Poland)
José-Manuel Colom (Spain)
Michael Duvigneau (Germany) (Chair)
Giuliana Franceschinis (Italy)
Guy Gallasch (Australia)
Xudong He (USA)
Kunihiko Hiraishi (Japan) (Chair)
Gabriel Juhás (Slovakia)
Peter Kemper (USA)
Astrid Kiehn (India)
Hanna Klaudel (France)
Lars Kristensen (Norway)
ZhiWu Li (China)
Robert Lorenz (Germany)
Daniel Moldt (Germany) (Chair)
Atsushi Ohta (Japan)
Wojciech Penczek (Poland)

Laure Petrucci (France)
Lucia Pomello (Italy)
Yann Thierry-Mieg (France)
Naoshi Uchihira (Japan)
H.M.W. (Eric) Verbeek (Netherlands)
Manuel Wimmer (Austria)
Karsten Wolf (Germany)
Shingo Yamaguchi (Japan)
Satoshi Yamane (Japan)

We received 18 high-quality contributions. The program committee has accepted five of them for full presentation. Furthermore the committee accepted six papers as short presentations. Two contributions were submitted and accepted as posters.

The international program committee was supported by the valued work of Luca Bernardinello, Kent Inge Fagerland Simonsen, Elisabetta Mangioni, Artur Męski, Maciej Szreter, and Samir Tata as additional reviewers. Their work is highly appreciated.

Furthermore, we would like to thank the organizational teams of the Japan Advanced Institute of Science and Technology, Kanazawa, Japan and the University of Newcastle, Newcastle upon Tyne, U.K., for their general organizational support.

Without the enormous efforts of authors, reviewers, PC members and the organizational teams this workshop wouldn't provide such an interesting booklet.

Thanks!

Michael Duvigneau, Daniel Moldt, and Kunihiko Hiraishi
Newcastle, June 2011

Contents

Part I Invited Talks

Part II Long Presentations

Part III Short Presentations

Part IV Poster Abstracts

Part I

Invited Talks

Unfolding Models of Asynchronous Systems: Applications to Analysis and Synthesis

Victor Khomenko

Newcastle University
School of Computing
victor.khomenko@ncl.ac.uk

Abstract. Analysis and synthesis of concurrent systems suffers from combinatorial state space explosion. That is, even a relatively small system specification can (and often does) yield a very large state space. One of the prominent techniques for alleviating this problem is based on complete prefixes of Petri net unfoldings. It relies on the partial order view of concurrent computation, and represents system states implicitly, using an acyclic Petri net. This talk describes applications of the unfolding technique to analysis of concurrent systems in general, and to verification and synthesis of asynchronous circuits in particular.

Design, Modelling and Analysis of a Workflow Reconfiguration

Manuel Mazzara[1], Faisal Abouzaid[2],
Nicola Dragoni[3], and Anirban Bhattacharyya[1]

[1] Newcastle University, Newcastle upon Tyne, UK
{Manuel.Mazzara, Anirban.Bhattacharyya}@ncl.ac.uk
[2] École Polytechnique de Montréal, Canada
m.abouzaid@polymtl.ca
[3] Technical University of Denmark (DTU), Copenhagen
ndra@imm.dtu.dk

Abstract. This paper describes a case study involving the reconfiguration of an office workflow. We state the requirements on a system implementing the workflow and its reconfiguration, and describe the system's design in BPMN. We then use an asynchronous π-calculus and $Web\pi_\infty$ to model the design and to verify whether or not it will meet the requirements. In the process, we evaluate the formalisms for their suitability for the modelling and analysis of dynamic reconfiguration of dependable systems.

1 Introduction

Competition drives technological development, and the development of dependable systems is no exception. Thus, modern dependable systems are required to be more flexible, available and dependable than their predecessors, and dynamic reconfiguration is one way of achieving these requirements.

A significant amount of research has been performed on hardware reconfiguration (see [5] and [9]), but little has been done for reconfiguration of services, especially regarding computational models, formalisms and methods appropriate to the service domain. Furthermore, much of the current research assumes that reconfiguration can be instantaneous, or that the environment can wait during reconfiguration for a service to become available (see [14] and [13]). These assumptions are unrealistic in the service domain. For example, instantaneous mode change in a distributed system is generally not possible, because the system usually has no well-defined global state at a specific instant (due to significant communication delays). Also, waiting for the reconfiguration to complete is not acceptable if (as a result) the environment becomes dangerously unstable or the service provider loses revenue by the environment aborting the service request.

These observations lead to the conclusion that further research is required on dynamic reconfiguration of dependable services, and especially on its formal foundations, modelling and verification. In a preliminary paper [16], we examined a number of well-known formalisms for their suitability for reconfigurable

The invited speaker is Manuel Mazzara.

dependable systems. In this paper, we focus on one of the formalisms ($Web\pi_\infty$) and compare it to a π-calculus in order to perform a deeper analysis than was possible in [16]. We use a more complex case study involving the reconfiguration of an office workflow for order processing, define the requirements on a system implementing the workflow and its reconfiguration, and describe the design of a system in BPMN (see section 2). We then use an asynchronous π-calculus with summation (in section 3) and $Web\pi_\infty$ [18] (in section 4) to model the design and to verify whether or not the design will meet the reconfiguration requirements. We chose process algebras because they are designed to model interaction between concurrent activities. An asynchronous π-calculus was selected because π-calculi are designed to model link reconfiguration, and asynchrony is suitable for modelling communication in distributed systems. $Web\pi_\infty$ was selected because it is designed to model composition of web services.

Thus, the contribution of this paper is to identify strengths and weaknesses of an asynchronous π-calculus with summation and $Web\pi_\infty$ for modelling dynamic reconfiguration and verifying requirements (discussed in section 5). This evaluation may be useful to system designers intending to use formalisms to design dynamically reconfigurable systems, and also to researchers intending to design better formalisms for the design of dynamically reconfigurable systems.

2 Office Workflow: Requirements and Design

This case study describes dynamic reconfiguration of an office workflow for order processing that is commonly found in large and medium-sized organizations [7]. These workflows typically handle large numbers of orders. Furthermore, the organizational environment of a workflow can change in structure, procedures, policies and legal obligations in a manner unforseen by the original designers of the workflow. Therefore, it is necessary to support the unplanned change of these workflows. Furthermore, the state of an order in the old configuration may not correspond to any state of the order in the new configuration. These factors, taken in combination, imply that instantaneous reconfiguration of a workflow is not always possible; neither is it practical to delay or abort large numbers of orders because the workflow is being reconfigured. The only other possibility is to allow overlapping modes for the workflow during its reconfiguration.

2.1 Requirements

A given organization handles its orders from existing customers using a number of activities arranged according to the following procedure:

1. `Order Receipt`: an order for a product is received from a customer. The order includes customer identity and product identity information.
2. `Evaluation`: the product identity is used to perform an inventory check on the availability of the product. The customer identity is used to perform a credit check on the customer using an external service. If both the checks are positive, the order is accepted for processing; otherwise the order is rejected.

3. `Rejection`: if the order is rejected, a notification of rejection is sent to the customer and the workflow terminates.
4. If the order is to be processed, the following two activities are performed concurrently:
   (a) `Billing`: the customer is billed for the total cost of the goods ordered plus shipping costs.
   (b) `Shipping`: the goods are shipped to the customer.
5. `Archiving`: the order is archived for future reference.
6. `Confirmation`: a notification of successful completion of the order is sent to the customer.

In addition, for any given order, `Order Receipt` must precede `Evaluation`, which must precede `Rejection` or `Billing` and `Shipping`.

After some time, managers notice that lack of synchronisation between the Billing and Shipping activities is causing delays between the receipt of bills and the receipt of goods that are unacceptable to customers. Therefore, the managers decide to change the order processing procedure, so that `Billing` is performed before `Shipping` (instead of performing the two activities concurrently). During the transition interval from one procedure to the other, the following requirements must be met:

1. The result of the `Evaluation` activity for any given order should not be affected by the change in procedure.
2. All accepted orders must be billed and shipped exactly once, then archived, then confirmed.
3. All orders accepted after the change in procedure must be processed according to the new procedure.

2.2 Design

We designed the system implementing the office workflow using the Business Process Modeling Notation (BPMN) [4]. We chose BPMN because it is a widely used graphical tool for designing business processes. In fact, BPMN is a standard for business process modelling, and is maintained by the Object Management Group (see http://www.omg.org/).

The system is designed as a collection of eight pools: Office Workflow, Order Generator, Credit Check, Inventory Check, Reconf. Region, Bill&Ship1, Bill&Ship2 and Archive. The different pools represent different functional entities, and each pool can be implemented as a separate concurrent task (see Figure 1). Office Workflow coordinates the entire workflow: it receives a request from a customer, and makes a synchronous call to Order Generator to create an order. It then calls Credit Check (with the order) to check the creditworthiness of the customer, and tests the returned value using an Exclusive Data-Based Gateway. If the test is positive, Office Workflow calls Inventory Check (with the order) to check the availability of the ordered item, and tests the returned value. If either of the two tests is negative, the customer is notified of the rejected order

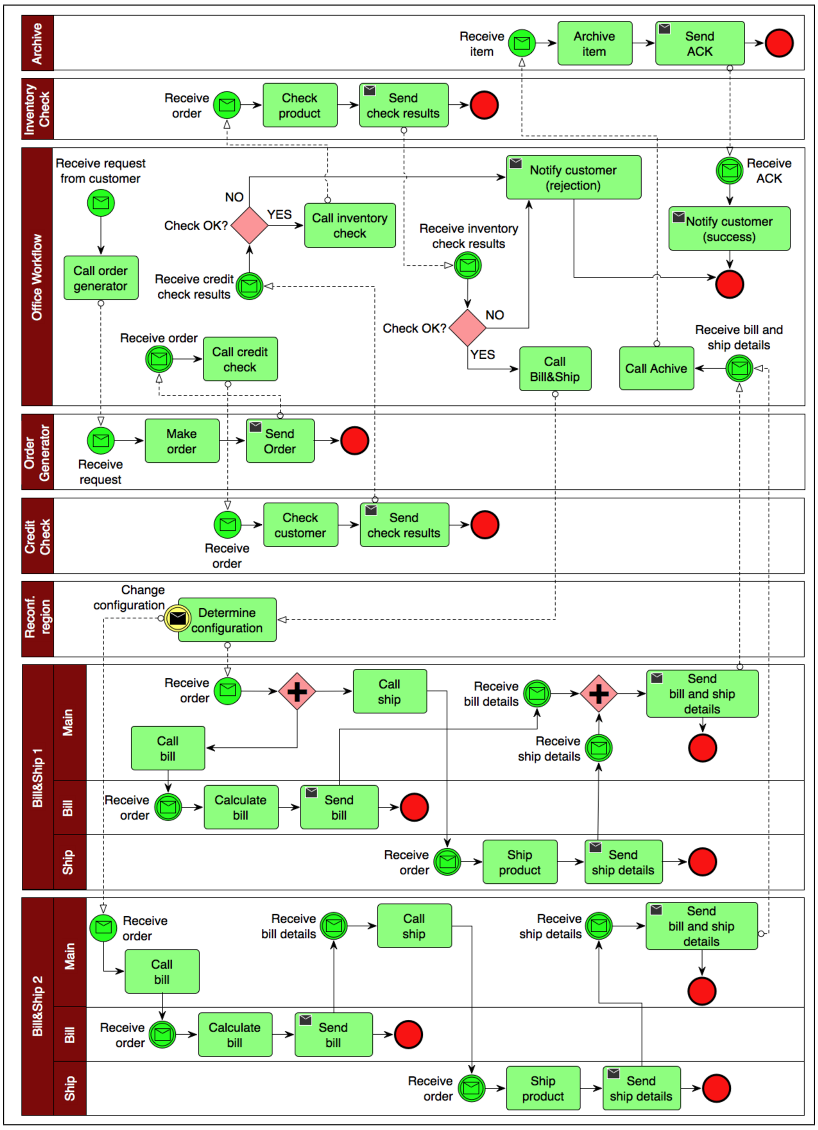


Fig. 1. Office workflow - BPMN diagram of the reconfiguration

and the workflow terminates. If both tests are positive, Office Workflow calls Reconf. Region, which acts as a switch between configuration 1 and configuration 2 of the workflow, and thereby handles the reconfiguration of the workflow.

Reconf. Region calls Bill&Ship1 by default: it makes an asynchronous call to the Main pool within Bill&Ship1, which uses a Parallel Gateway to call Bill and Ship concurrently and merge their respective results, and then returns these results to Office Workflow. The Office Workflow then calls Archive to store the order, then notifies the customer of the successful completion of the order, and then terminates the workflow. However, if Reconf. Region receives a change configuration message, it calls the Main pool within Bill&Ship2 instead, which makes sequential a call to Bill and then to Ship, and then returns the results to Office Workflow.

Notice that for the sake of simplicity, we assume neither Bill nor Ship produces a negative result. Furthermore, the Bill and Ship pools are identical in both configurations, which suggests their code is replicated (rather than shared) in the two configurations. Finally, we assume the reconfiguration is planned rather than unplanned.

3 Asynchronous π-Calculus

The asynchronous π-calculus ([10], [3]) is a subset of Milner's π-calculus [20], and it is known to be more suitable for distributed implementation. It is considered a rich paradigm for asynchronous communication, although it is not as expressive as Milner's π-calculus in representing mixed-choice constructs, such as $\overline{a}.P+b.P'$ (see [22]).

We recall the (monadic) asynchronous π-calculus. Let $\mathcal{N}$ be a set of names (e.g. $a, b, c, ...$) and $\mathcal{V}$ be a set of variables (e.g. $x, y, z, ...$). The set of the asynchronous π-calculus processes is generated by the following grammar:

$$P ::= \bar{x}z \mid G \mid P|P \mid [a=b]P \mid (\nu x)P \mid A(x_1, ..., x_n)$$

where guards G are defined as follows:

$$G ::= 0 \mid x(y).P \mid \tau.P \mid G+G$$

Intuitively, an output $\bar{x}z$ represents a message z tagged with a name x indicating that it can be received (or consumed) by an input process $x(y).P$ which behaves as $P\{z/y\}$ upon receiving z. Furthermore, $x(y).P$ binds the name y in P and the restriction $(\nu x)P$ declares a name x private to P and thus binds x. Outputs are non-blocking.

The parallel composition $P|Q$ means P and Q running in parallel. $G+G$ is the non-deterministic choice that is restricted to τ and input prefixes.

$[a=b]P$ behaves like P if a and b are identical.

$A(y_1, ..., y_n)$ is an *identifier* (also *call*, or *invocation*) of arity n. It represents the instantiation of a defined agent. We assume that every such identifier has a unique, possibly recursive, definition $A(x_1, ..., x_n) \overset{def}{=} P$ where the x_is are pairwise distinct, and the intuition is that $A(y_1, ..., y_n)$ behaves like P with each y_i replacing x_i.

Furthermore, for each $A(x_1, ..., x_n) \overset{def}{=} P$ we require: $fn(P) \subseteq \{x_1, ..., x_n\}$, where $fn(P)$ stands for the set of free names in P, and $bn(P)$ for the set of bound names in P. The *input prefix* and the ν *operator* bind the names. For example, in a process $x(y).P$, the name y is bound. In $(\nu x)P$, x is considered to be bound. Every other occurrences of a name like x in $x(y).P$ and x, y in $\bar{x}\langle y\rangle.P$ are free.

Due to lack of space we omit to give details on structural congruence and operational semantics for the asynchronous π-calculus. They can be found in [1] for the version of the calculus we use in this paper.

The Model in Asynchronous $\boldsymbol{\pi}$-Calculus The model in asynchronous π-calculus needs to keep the synchronization between actions in sequence coherent with the workflow definition. So sequence is implemented by using parallel composition with prefix and postfix on the same channel. Channel names are not restricted since the full system is not described here and has to be put in parallel with the detailed implementation of the environment process described (that will be omitted here).

The entire model is expressed in asynchronous π-calculus as follows:

Entire Model

Let $\mathit{params} =$
$\{\mathit{customer}, \mathit{item}, \mathit{Archive}, \mathit{ArchiveReply}, \mathit{Bill}, \mathit{BillReply}, \mathit{BillShip}, Confirm,$
$\mathit{CreditCheck}, \mathit{CreditOk}, \mathit{CreditReject}, InventoryCheck,$
$InventoryOk, InventoryReject, OrderGenerator,$
$\mathit{OrderGeneratorReply}, \mathit{OrderReceipt}, \mathit{Reject}, \mathit{Ship}, \mathit{ShipReply}, \mathit{reco}, \mathit{recn}\}$

We can define the $Workflow$ process as follows:

$$\begin{array}{l}
Workflow(\mathit{params}) \triangleq \\
(\nu\, \mathit{order})\ (\mathit{OrderReceipt}(\mathit{customer}, \mathit{item}).\overline{\mathit{OrderGenerator}}\, \mathit{customer}, \mathit{item} \\
\mid \mathit{OrderGeneratorReply}(\mathit{order}).\overline{CreditCheck}\, \mathit{customer} \\
\mid (\mathit{creditOk}().\overline{InventoryCheck}\, \mathit{item} + \mathit{CreditReject}().\overline{\mathit{Reject}}\, \mathit{order}) \\
\mid (InventoryOk().\overline{\mathit{BillShip}} + InventoryReject().\overline{\mathit{Reject}}\, \mathit{order}) \\
\mid \mathit{rec}_o().\mathit{BillShip}().(\overline{\mathit{Bill}}\, \mathit{customer}, \mathit{item}, \mathit{order} \mid \overline{\mathit{Ship}}\, \mathit{customer}, \mathit{item}, \mathit{order}) \\
\mid \mathit{BillReply}(\mathit{order}).\mathit{ShipReply}(\mathit{order}).\overline{\mathit{Archive}}\, \mathit{order} \\
+\mathit{rec}_n().\mathit{BillShip}().(\overline{\mathit{Bill}}\, \mathit{customer}, \mathit{item}, \mathit{order} \\
\mid \mathit{BillReply}(\mathit{order}).\overline{\mathit{Ship}}\, \mathit{customer}, \mathit{item}, \mathit{order})\ \mid \mathit{ShipReply}(\mathit{order}).\overline{\mathit{Archive}}\, \mathit{order} \\
\mid \mathit{ArchiveReply}(\mathit{order}).\overline{Confirm}\, \mathit{order})\ \mid Workflow(\mathit{params})
\end{array}$$

In the model, the old region is identified as follows:

$$\begin{array}{l}
\mathit{rec}_o().\mathit{BillShip}().(\overline{\mathit{Bill}}\, \mathit{customer}, \mathit{item}, \mathit{order} \mid \overline{\mathit{Ship}}\, \mathit{customer}, \mathit{item}, \mathit{order}) \\
\mid \mathit{BillReply}(\mathit{order}).\mathit{ShipReply}(\mathit{order}).\overline{\mathit{Archive}}\, \mathit{order}
\end{array}$$

And the new region is:

$$\begin{array}{l}
\mathit{rec}_n().\mathit{BillShip}().(\overline{\mathit{Bill}}\, \mathit{customer}, \mathit{item}, \mathit{order} \\
\mid \mathit{BillReply}(\mathit{order}).\overline{\mathit{Ship}}\, \mathit{customer}, \mathit{item}, \mathit{order})\ \mid \mathit{ShipReply}(\mathit{order}).\overline{\mathit{Archive}}\, \mathit{order}
\end{array}$$

In the asynchronous π-calculus, two outputs cannot be in sequence. In order to impose ordering between $\overline{\mathit{Bill}}$ and $\overline{\mathit{Ship}}$, in the new region, it is necessary to put a guard on *Ship*, which requires enlarging the boundary of the old region to include the processes in the environment of the workflow that synchronize with $\overline{Bill}$ and $\overline{Ship}$. We did not model these processes because they are outside the system being designed, but the limitations of the asynchronous π-calculus imply that we must be able to access the logic of external services for which we know only the interfaces. For a more detailed description of this problem, please see [12].

The entire model represents a specific instance of the workflow that spawn concurrently another instance with fresh customer and item which here are assumed to be fresh names but in reality will be user entered (but it is not relevant to our purposes). We have to assume the existence of a "higher level" process (at the level of the BPEL engine) that activates the entire workflow and bounds the names that are free in the above π-calculus process. In this model channels $creditOK$, $creditReject$, $InventoryOK$ and $InventoryReject$ are used to receive the result of the credit check and inventory check, respectively. The old/new region is externally triggered using specific channels rec_o and rec_n chosen according to the value x received on channel $region$:

$$(\nu\, x)Workflow(param) \ \mid \ region(x).([x = new]\overline{rec_n} \ \mid \ [x = old]\overline{rec_o})$$

In section 4 we show a more efficient solution using Webπ_∞.

Analysis in π-logic Logics have long been used to reason about complex systems, because they provide abstract specifications that can be used to describe system properties of concurrent and distributed systems. Verification frameworks can support checking of functional properties of such systems by abstracting away from the computational contexts in which they are operating.

In the context of π-calculi, one can use the π-logic with the HAL Toolkit model-checker [8]. The π-logic has been introduced in [8] to specify the behavior of systems in a formal and unambiguous manner by expressing temporal properties of π-processes.

Syntax of the π-logic The logic integrates modalities defined by Milner ([21]) with $EF\phi$ and $EF\{\chi\}\phi$ modalities on possible future. The π-logic syntax is:

$$\phi ::= true \mid\ \sim \phi \mid \phi \ \wedge \ \phi' \mid EX\{\mu\}\phi \mid EF\phi \mid EF\{\chi\}\phi$$

where μ is a π-calculus action and χ could be μ, $\sim \mu$, or $\bigvee_{i\in I}\mu_i$ and where I is a finite set.

Semantics of π-formulae is given below:

- $P \models true$ for any process P;
- $P \models\sim \phi$ iff $P \not\models \phi$;
- $P \models \phi \ \wedge \ \phi'$ iff $P \models \phi$ and $P \models \phi'$;
- $P \models EX\{\mu\}\phi$ iff there exists P' such as $P \xrightarrow{\mu} P'$ and $P' \models \phi$ (**strong next**);

- $P \models EF\phi$ iff there exists $P_0, ..., P_n$ and $\mu_1, ..., \mu_n$, with $n \geq 0$, such as $P = P_0 \xrightarrow{\mu_1} P_1 ... \xrightarrow{\mu_n} P_n$ and $P_n \models \phi$. The meaning of $EF\phi$ is that ϕ must be true sometimes in a possible future.
- $P \models EF\{\chi\}\phi$ if and only if there exists $P_0, ..., P_n$ and $\nu_1, ..., \nu_n$, with $n \geq 0$, such that $P = P_0 \xrightarrow{\nu_1} P_1 ... \xrightarrow{\nu_n} P_n$ and $P_n \models \phi$ with:
  - $\chi = \mu$ for all $1 \leq j \leq n,\ \nu_j = \mu\ or\ \nu_j = \tau$;
  - $\chi = \sim \mu$ for all $1 \leq j \leq n,\ \nu_j \neq \mu\ or\ \nu_j = \tau$;
  - $\chi = \bigvee_{i \in I} \mu_i$: for all $1 \leq j \leq n, \nu_j = \mu_i$ for some $i \in I\ or\ \nu_j = \tau$.

  The meaning of $EF\{\chi\}\phi$ is that the truth of ϕ must be preceded by the occurrence of a sequence of actions χ.

Some useful dual operators are defined as usual:
$false$, $\phi \vee \phi$, $AX\{\mu\}\phi$ ($\sim EX\{\mu\} \sim \phi$), $< \mu > \phi$ (weak next), $[\mu]\phi$ (Dual of weak next), $AG\phi$ ($AG\{\chi\}$) (always).

Properties of the dynamic reconfiguration model
We need to verify that during the reconfiguration interval the requirements given in section 2.1 hold. For this purpose, we need to express the requirements formally, if possible, using the π-logic.

The result of the `Evaluation` activity for any given order should not be affected by the change in procedure. The following formula means whatever the chosen path (old or new region), an order will be billed, shipped and archived or refused:

$AG\{EF\{OrderReceipt()\}true\}$
$AG\{\big(EF\{\overline{Bill}\,customer, item, order\}true \wedge EF\{\overline{Ship}\,customer, item, order\}true \wedge$
$EF\{\overline{Archive}\,order\}true\big) \vee EF\{\overline{Reject}\,\}true\}$

All accepted orders must be billed and shipped exactly once, then archived, then confirmed. The following formula means that after an order is billed and shipped, it is archived and confirmed, and cannot be billed nor shipped again:

$AG\{EF\{BillShip()\}true\}$
$AG\{EF\{\overline{Bill}\,customer, item, order\}true \wedge EF\{\overline{Ship}\,customer, item, order\}true \wedge$
$EF\{\overline{Archive}\,order\}true\} \wedge EF\{\overline{Confirm}\,order\}true\}$
$AG\{\{\overline{Bill}\,customer, item, order\}false \wedge \{\overline{Ship}\,customer, item, order\}false\}$

All orders accepted after the change in procedure must be processed according to the new procedure We can express in the π-logic the following requirement: "after a reception on the channel rec_n, no other reception on channel rec_0 will be accepted". This meets the desired requirement since it is obvious from the model that, if a signal is received on channel rec_n, the order will be processed according to the new procedure.

$AG\{\{rec_n()\}true\ AG\{rec_0()\}false\}$

However, since the choice between the old procedure and the new one is non-deterministic, this formula will not be true, although it is an essential requirement for the model. This result illustrates the difficulty of the asynchronous π-calculus to model the dynamic reconfiguration properly. A first attempt to answer this problem is presented in the next section.

4 Webπ_∞

Webπ_∞ is a conservative extension of the π-calculus developed for modelling and analysis of Web services and Service Oriented Architectures. The basic theory has been developed in [18] and [15], whilst its applicability has been shown in other work: [12] gives a BPEL semantics in term of Webπ_∞, [6] clarifies some aspects of the Recovery Framework of BPEL, and [17] exploits a web transaction case study (a toy example has also been discussed in [16]).

Syntax and Semantics The syntax of $\mathtt{web}\pi_\infty$ *processes* relies on a countable set of *names*, ranged over by $x, y, z, u, \cdots$. Tuples of names are written $\widetilde{u}$. We intend $i \in I$ with I a finite non-empty set of indexes.

$$P \ ::= \ 0 \ \mid \ \overline{x}\,\widetilde{u} \ \mid \ \sum_{i \in I} x_i(\widetilde{u_i}).P_i \ \mid \ (x)P \ \mid \ P|P \ \mid \ !x(\widetilde{u}).P \ \mid \ \langle\!| P \ ; \ P |\!\rangle_x$$

It is worth noting that the syntax of $\mathtt{web}\pi_\infty$ simply augments the asynchronous π-calculus with a workunit process. A workunit $\langle\!| P \ ; \ Q |\!\rangle_x$ behaves as the *body* P until an abort $\overline{x}$ is received, and then it behaves as the *event handler* Q.

We give the semantics of $\mathtt{web}\pi_\infty$ in two steps, following the approach of Milner [19], separating the laws that govern the static relations between processes from the laws that rule their interactions. The static relations between processes are governed by the *structural congruence* $\equiv$, the least congruence satisfying the Abelian monoid laws for parallel and summation (associativity, commutativity and $\mathbf{0}$ as identity) and closed with respect to α-renaming and the axioms shown in table 1.

The scope laws are standard while novelties regard workunit and floating laws. The law $\langle\!| \mathbf{0} \ ; \ Q |\!\rangle_x \equiv \mathbf{0}$ defines committed workunit, namely workunit with $\mathbf{0}$ as body. These ones, being committed, are equivalent to $\mathbf{0}$ and, therefore, cannot fail anymore. The law $\langle\!| \langle\!| P \ ; \ Q |\!\rangle_y \,|\, R \ ; \ R' |\!\rangle_x \equiv \langle\!| P \ ; \ Q |\!\rangle_y \,|\, \langle\!| R \ ; \ R' |\!\rangle_x$ moves workunit outside parents, thus flattening the nesting. Notwithstanding this flattening, parent workunits may still affect the children by means of names. The law $\langle\!| \overline{z}\,\widetilde{u} \,|\, P \ ; \ Q |\!\rangle_x \equiv \overline{z}\,\widetilde{u} \,|\, \langle\!| P \ ; \ Q |\!\rangle_x$ floats messages outside workunit boundaries. By this law, messages are particles that independently move towards their

Scope laws	$(u)\mathbf{0} \equiv \mathbf{0}, \qquad (u)(v)P \equiv (v)(u)P$
	$P \,\|\, (u)Q \equiv (u)(P \,\|\, Q)\ , \quad \textit{if}\ u \notin \mathrm{fn}(P)$
	$\langle\!\|(z)P\ ;\ Q\|\!\rangle_x \equiv (z)\langle\!\|P\ ;\ Q\|\!\rangle_x\ , \quad \textit{if}\ z \notin \{x\} \cup \mathrm{fn}(Q)$
Workunit laws	$\langle\!\|\mathbf{0}\ ;\ Q\|\!\rangle_x \equiv \mathbf{0}$
	$\langle\!\|\langle\!\|P\ ;\ Q\|\!\rangle_y \,\|\, R\ ;\ R'\|\!\rangle_x \equiv \langle\!\|P\ ;\ Q\|\!\rangle_y \,\|\, \langle\!\|R\ ;\ R'\|\!\rangle_x$
	$\langle\!\|(z)P\ ;\ Q\|\!\rangle_x \equiv (z)\langle\!\|P\ ;\ Q\|\!\rangle_x\ , \quad \textit{if}\ z \notin \{x\} \cup \mathrm{fn}(Q)$
Floating law	$\langle\!\|\overline{z}\,\widetilde{u} \,\|\, P\ ;\ Q\|\!\rangle_x \equiv \overline{z}\,\widetilde{u} \,\|\, \langle\!\|P\ ;\ Q\|\!\rangle_x$

Table 1. $\mathtt{web}\pi_\infty$ Structural Congruence

inputs. The intended semantics is the following: if a process emits a message, this message traverses the surrounding workunit boundaries until it reaches the corresponding input. In case an outer workunit fails, recoveries for this message may be detailed inside the handler processes.

The dynamic behavior of processes is instead defined by the reduction relation $\rightarrow$ which is the least relation satisfying the axioms and rules shown in table 2 and closed with respect to $\equiv$, $(x)\_$, $\_ \mid \_$, and $\langle\!| \_\ ;\ Q|\!\rangle_z$. In the table we use the shortcut: $\langle\!|P\ ;\ Q|\!\rangle \stackrel{\mathrm{def}}{=} (z)\langle\!|P\ ;\ Q|\!\rangle_z$ where $z \notin \mathrm{fn}(P)\ \cup\ \mathrm{fn}(Q)$

COM	$\overline{x_i}\,\widetilde{v} \,\|\, \sum_{i\in I} x_i(\widetilde{u_i}).P_i \ \rightarrow\ P_i\{\widetilde{v}/\widetilde{u_i}\}$
REP	$\overline{x}\,\widetilde{v} \,\|\, !x(\widetilde{u}).P \ \rightarrow\ P\{\widetilde{v}/\widetilde{u}\} \,\|\, !x(\widetilde{u}).P$
FAIL	$\overline{x} \,\|\, \langle\!\|\, \prod_{i\in I} \sum_{s\in S} x_{is}(\widetilde{u_{is}}).P_{is} \,\|\, \prod_{j\in J} !x_j(\widetilde{u_j}).P_j\ ;\ Q\|\!\rangle_x \ \rightarrow\ \langle\!\|Q\ ;\ \mathbf{0}\|\!\rangle$ $\textit{where}\ J \neq \emptyset \vee (I \neq \emptyset \wedge S \neq \emptyset)$

Table 2. $\mathtt{web}\pi_\infty$ Reduction Semantics

Rules (COM) and (REP) are standard in process calculi and model input-output interaction and lazy replication. Rule (FAIL) models workunit failures: when a unit abort (a message on a unit name) is emitted, the corresponding body is terminated and the handler activated. On the contrary, aborts are not possible if the transaction is already terminated (namely every thread in the body has completed its own work), for this reason we close the workunit restricting its name.

The model in Webπ_∞ For the modelling purposes of this work, the idea of workunit and event handler turn out to be particularly useful. Webπ_∞ uses the mechanism of workunit to bound the identified regions, and event raising is exploited to operate the non immediate change (reconfiguration). The model can be expressed as follows (as a shortcut we will use here process invocation):

$Workflow(customer, item) \triangleq$

$$\begin{array}{l}
(\nu\, order)\ \mathit{OrderReceipt}(customer, item).\overline{\mathit{OrderGenerator}}\, customer, item \\
\mid \mathit{OrderGeneratorReply}(order).\overline{\mathit{CreditCheck}}\, customer \\
\mid (\mathit{CreditCheckReply}_t(order).\overline{\mathit{InventoryCheck}}\, item \\
+\mathit{CreditCheckReply}_f(order).\overline{\mathit{Reject}}\, order) \\
\mid (\mathit{InventoryCheckReply}_t(order).\overline{\mathit{BillShip}} \\
+\mathit{InventoryCheckReply}_f(order).\overline{\mathit{Reject}}\, order) \\
\mid \langle\!|\mathit{BillShip}().(\overline{\mathit{Bill}}\, customer, item, order \mid \overline{\mathit{Ship}}\, customer, item, order \\
\mid (\nu\, customer)(\nu\, item)\, Workflow(customer, item)) \\
;\ (\nu\, customer)(\nu\, item)\, Workflow_n(customer, item)|\!\rangle_{rec} \\
\mid \mathit{BillReply}(order).\mathit{ShipReply}(order).\overline{\mathit{Archive}}\, order \\
\mid \mathit{ArchiveReply}(order).\overline{\mathit{Confirm}}\, order
\end{array}$$

Webπ_∞ shows here a subtle feature which is important for modelling reconfigurable systems. Since the floating laws of structural congruence allow the asynchronous outputs in a workunit to freely escape, once the region to reconfigure has been entered and the $\overline{\mathit{BillShip}}$ has been triggered, $\overline{\mathit{Bill}}\, customer, item, order$ and $\overline{\mathit{Ship}}\, customer, item, order$ will not be killed by any incoming rec signal. This means that, once the region has been entered by an order, that order will go through without being interrupted by reconfiguration events and the old order will be processed according to the old procedure, not the new one. Future orders will find instead only the new procedure $Workflow_n$ waiting for orders:

$$\begin{array}{l}
Workflow_n(customer, item) \triangleq \\
(\nu\, order)\ \mathit{OrderReceipt}(customer, item).\overline{\mathit{OrderGenerator}}\, customer, item \\
\mid \mathit{OrderGeneratorReply}(order).\overline{\mathit{CreditCheck}}\, customer \\
\mid (\mathit{CreditCheckReply}_t(order).\overline{\mathit{InventoryCheck}}\, item\, + \\
\mathit{CreditCheckReply}_f(order).\overline{\mathit{Reject}}\, order) \\
\mid (\mathit{InventoryCheckReply}_t(order).\overline{\mathit{BillShip}}\, + \\
\mathit{InventoryCheckReply}_f(order).\overline{\mathit{Reject}}\, order) \\
\mid \mathit{BillShip}().(\overline{\mathit{Bill}}\, customer, item, order \mid \mathit{BillReply}(order).\overline{\mathit{Ship}}\, customer, item, order) \\
\mid \mathit{ShipReply}(order).\overline{\mathit{Archive}}\, order\ \mid \mathit{ArchiveReply}(order).\overline{\mathit{Confirm}}\, order \\
\mid (\nu\, customer)(\nu\, item)\, Workflow_n(customer, item)
\end{array}$$

As in the π-calculus model, we have to assume the existence of a top level process activating the entire workflow and bounding all the names appearing free in the above π-calculus process. The change in procedure will be activated when the channel t is triggered.

$$(\nu\, customer)(\nu\, item)(\nu\, rec)\, Workflow(customer, item) \mid t().\overline{rec}$$

This process is also responsible for triggering the reconfiguration.

Analysis in Webπ_∞ Analysis in Webπ_∞ is intended as equational reasoning. At the moment, one severe weakness of Webπ_∞ is its lack of tool support, i.e. automatic system verification. However, it is clearly possible to encode Webπ_∞ into the π-calculus, being the only technical complication the encoding of the workunit and its asynchronous interrupt. Once the compilation into the π-calculus has been done, we can proceed using HAL. From one side, Webπ_∞ simplifies the modelling of dependable systems expressing with its workunit the recovery behavior. On the other side, it makes the verification more difficult. Luckily, there is an optimal solution using Webπ_∞ as *modelling language* and the π-calculus as *intermediate language*, i.e. a *verification bytecode*. We can then offer a practical modelling suite to the designer and still use the tool support for the π-calculus. At the moment our research has not gone so far, so we will just discuss the three requirements here. We will analyse the requirements in terms of equational reasoning (see [18] and [15]). The case study of this paper is interesting at showing both the modelling power of Webπ_∞ and the weaknesses of its reasoning system.

The result of the `Evaluation` activity for any given order should not be affected by the change in procedure. The acceptability of an order (`Evaluation` activity) is computed *outside* the region to be reconfigured, and there is no interaction between `Evaluation` and the region. That means that the `Evaluation` in the old procedure $workflow$ is exactly the same as in the new procedure $workflow_n$, i.e. the checks are performed in the same exact order. We can formally express it, in term of equational reasoning, stating that the `Evaluation` activity in the old procedure $workflow$ is bisimilar to the `Evaluation` activity in the new procedure $workflow_n$ which is trivially true.

All accepted orders must be billed and shipped exactly once, then archived, then confirmed. The presence of a workunit does not affect how the order itself is processed. The workflow of actions described by the requirement can be formally expressed as follows:

$$\begin{array}{l}(\nu\, x)(\nu\, y)\ (\overline{Bill}\, customer, item, order \mid \overline{Ship}\, customer, item, order \\ \mid BillReply(order).\overline{x} \mid ShipReply(order).\overline{y}\ \ \mid x().y().\overline{Archive}\, order \\ \mid ArchiveReply(order).\overline{Confirm}\, order)\end{array}$$

In plain words this process describes billing and shipping happening in any order but both before archiving and confirming. The channels x and y are there precisely to work as a joint for billing and shipping. If we want to express the requirements in term of equational reasoning, we can require that both the old and the new regions have to be bisimilar with the above process. However, this is too strict since the above process allows a set of traces which is a superset of both the set of traces of the old configuration and the new one. In this case similarity could be considered instead of bisimilarity.

All orders accepted after the change in procedure must be processed according to the new procedure To show this requirements has been implemented in the model semantic reasoning is not necessary, structural congruence is sufficient. The change in procedure is here modelled by triggering the *rec* channel and spawning the workunit handler. The handler then activates a new instance of the workflow based on the new procedure scheme which has been called $workflow_n$. The floating laws of structural congruence of Webπ_∞ (definition 1) allow the asynchronous outputs in a workunit to freely escape the workunit itself. Thus, once the region to reconfigure has been already entered and the $\overline{BillShip}$ has been triggered, $\overline{Bill}\, customer, item, order$ and $\overline{Ship}\, customer, item, order$ will not be killed by any incoming rec signal. Thus, once the region has been entered by an order, that order will be not interrupted by reconfiguration events so that old order will be processed according to the old procedure and not the new one.

5 Discussion

In this section, we discuss three issues which arose during design and modelling: how the modelling influenced our design, how the π-calculus and Webπ_∞ compare with respect to modelling, and correctness criteria for verification of the workflow reconfiguration.

Modelling and Design Different formalisms have different biases on design because of their different perspectives. In one of the alternative designs we considered, the Bill and Ship pools were outside the reconfiguration region, so that their code was shared between the two configurations. Thus, the boundary of the reconfiguration region was different. We chose the design in section 2.2 because it is easier to model. It is the job of a formalist to model what the system designers produce, and ask them to change the design if it cannot be modelled or is unverifiable. Our experience with asynchronous π-calculi and Webπ_∞ suggested that extending the boundary of the reconfiguration region to include billing and shipping was a practical choice. This is because in the asynchronous π-calculus (and consequently in Webπ_∞), two outputs cannot be in sequence. So, in order to impose ordering between `Bill` and `Ship`, we had to enlarge the boundary of the reconfiguration region to include the processes in the environment of the workflow that synchronize with them. The negative side of this solution is that we have been forced to include in the region parts of the system that were not intended to be changed. Here the asynchronous π-calculus shows its weakness in terms of reconfiguring processes dynamically.

Comparison of $\boldsymbol{\pi}$-calculus and Web$\boldsymbol{\pi_\infty}$ This paper has shown the Webπ_∞ workunit as being able to offer a more efficient solution to the problem of modelling the case study. In particular, by means of the Webπ_∞ floating laws, reconfiguration activities can be better handled. However, at the moment, one weakness of Webπ_∞ is its lack of tool support, whereas the π-calculus is supported by verification tools (e.g. TyPiCal [11] and HAL [8]). Therefore, Webπ_∞ has to be intended as a a front end for modelling with the the π-calculus as the *verification bytecode*. As mentioned above, neither the asynchronous π-calculus nor Webπ_∞ can have two outputs in sequence, and this leads to the specific design choice.

Correctness Criteria The standard notion of correctness used in process algebras is congruence based on bisimulation. However, our requirements are not all expressible as congruences between processes. The first and third requirements can be expressed as congruences, and so bisimulation can be used in the reasoning. The second requirement cannot be expressed as a congruence because the old and new configurations are not behaviourally congruent. So, we have used reasoning based on simulation instead. Thus, we found that congruence as it has been used in section 4 is not always applicable for verifying the correctness of our models. Therefore, in section 3 we have investigated model checking.

The discussion leads us to the following:

1. It is easier to model workflow reconfiguration in Webπ_∞ than in the asynchronous π-calculus. However, modelling would be even easier in a synchronous version of Webπ_∞.
2. Model checking is more widely applicable than equational reasoning based on congruences for verifying workflow reconfiguration.

These two conclusions seem to have wider applicability than just reconfiguration of workflows; but this needs to be verified.

Future Work We intend to proceed with a deeper analysis of alternative designs for this case study, and evaluate other formalisms, such as VDM [2] and Petri nets [23]. We are also working on a BPEL implementation of the system. We also need larger industrial case studies to help us to design and evaluate formalisms for the modelling and analysis of dynamic reconfiguration.

Acknowledgments

This work is partly funded by the EPSRC under the terms of a graduate studentship. The paper has been improved by conversations with John Fitzgerald, Cliff Jones, Alexander Romanovsky, Jeremy Bryans, Gudmund Grov, Mario Bravetti, Massimo Strano, Michele Mazzucco, Paolo Missier and Mu Zhou. We also want to thank members of the Reconfiguration Interest Group (in particular, Kamarul Abdul Basit, Carl Gamble and Richard Payne), the Dependability Group (at Newcastle University) and the EU FP7 DEPLOY Project (Industrial deployment of system engineering methods providing high dependability and productivity).

References

1. R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291 – 324, 1998.
2. D. Bjorner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer, 1978.
3. G. Boudol. Asynchrony and the π-calculus. rapport de recherche 1702. Technical report, INRIA, Sophia-Antipolis, 1992.
4. BPMN. Bpmn - business process modeling notation. ‘http://www.bpmn.org/.

5. A. Carter. Using dynamically reconfigurable hardware in real-time communications systems: Literature survey. Technical report, Computer Laboratory, University of Cambridge, November 2001.
6. N. Dragoni and M. Mazzara. A formal semantics for the ws-bpel recovery framework - the pi-calculus way. In *WS-FM'09, Springer Verlag*, 2009.
7. C. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In *Proceedings of the Conference on Organizational Computing Systems (COOCS 1995).* ACM, 1995.
8. G. L. Ferrari, S. Gnesi, U. Montanari, and M. Pistore. A model-checking verification environment for mobile processes. *ACM Transactions on Software Engineering and Methodology*, 12(4):440–473, 2003.
9. P. Garcia, K. Compton, M. Schulte, E. Blem, and W. Fu. An overview of reconfigurable hardware in embedded systems. *EURASIP J. Embedded Syst.*, 2006, January 2006.
10. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In P. America, editor, *European Conference on Object-Oriented Programming (ECOOP)*, pages 133–147. Lecture Notes in Computer Science 512, 1991.
11. N. Kobayashi. Typical: Type-based static analyzer for the pi-calculus. `http://www.kb.ecei.tohoku.ac.jp/ koba/typical/`.
12. R. Lucchi and M. Mazzara. A pi-calculus based semantics for ws-bpel. *Journal of Logic and Algebraic Programming*, 70(1):96–118, 2007.
13. J. Magee, N. Dulay, and J. Kramer. Structuring parallel and distributed programs. *Software Engineering Journal (Special Issue)*, 8(2):73–82, 1993.
14. J. Magee, J. Kramer, and M. Sloman. Constructing distributed systems in conic. *IEEE Transactions on Software Engineering*, 15(6):663–675, 1989.
15. M. Mazzara. *Towards Abstractions for Web Services Composition.* PhD thesis, Department of Computer Science, University of Bologna, 2006.
16. M. Mazzara and A. Bhattacharyya. On modelling and analysis of dynamic reconfiguration of dependable real-time systems. In *DEPEND, International Conference on Dependability*, 2010.
17. M. Mazzara and S. Govoni. A case study of web services orchestration. In *COORDINATION*, pages 1–16, 2005.
18. M. Mazzara and I. Lanese. Towards a unifying theory for web services composition. In *WS-FM*, pages 257–272, 2006.
19. R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
20. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, 1999.
21. R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 1993.
22. C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In *Mathematical Structures in Computer Science*, pages 256–265. ACM, 1997.
23. C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Fakultät Matematik und Physik, Technische Universität Darmstadt, 1962.

Part II

Long Presentations

Efficient Implementation of Prioritized Transitions for High-level Petri Nets

Michael Westergaard* and H.M.W. (Eric) Verbeek

Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands
{m.westergaard,h.m.w.verbeek}@tue.nl

Abstract. Transition priorities can be a useful mechanism when modeling using Petri nets. For example, high-priority transitions can be used to model exception handling and low-priority transitions can be used to model background tasks that should only be executed when no other transition is enabled. Transition priorities can be simulated in Petri nets using, e. g., inhibitor arcs, but such constructs tend to unnecessarily clutter models, making it useful to support priorities directly.

Computing the enabling of transitions in high-level Petri nets is an expensive operation and should be avoided. As transition priorities introduce a nonlocal enabling condition, at first sight this forces us to compute enabling for all transitions in a highest-priority-first order, but it is possible to do better. Here we describe our implementation of transition priorities in CPN Tools 3.0, where we minimize the number of enabling computations. We describe algorithms for executing transitions at random, useful for automatic simulation without user interactions, and for maintaining a set of known enabled transitions, useful for interactive user-guided simulation. Experiments show that using our algorithms we can execute $4-7$ million transitions a minute for real-life models and more than 20 million transitions a minute for other models, a significant improvement over the $1-5$ million transitions a minute possible for simpler algorithms.

1 Introduction

Prioritized transitions can be of use when modeling using Petri nets. For example, one can give a transition high priority to force it occur before other transitions if it is enabled, which is useful for handling exceptions, by letting the exception handler have higher priority than transitions handling usual cases. One can assign a transition a lower priority to prevent it from occurring unless no other transitions are enabled, which is useful for implementing a scheduler that should only be executed when all interesting tasks are unable to proceed.

In this paper we are concerned with efficient implementation of simulation of high-level Petri net models with transitions with priorities as well as efficient enabling updates. The described algorithms are implemented in CPN Tools 3.0 [4].

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

Priorities can be implemented using inhibitor arcs or any construction which serves the same purpose (by adding inhibitor arcs from places which have arcs to transitions with higher priority), but it is beneficial to support them directly in an implementation to reduce clutter in models. Furthermore, a direct implementation makes it possible to make enabling computation more efficient than implementations relying on general constructs.

Enabling computation of high-level Petri nets, such as coloured Petri nets (CPNs) supported by CPN Tools, is computationally expensive. To alleviate this, tools can implement algorithms to avoid having to compute the enabling of transitions too often. For example, if the goal is just to randomly execute transitions, there is no need to compute the enabling for all transitions – as soon as an enabled transition is found, it can be executed. By using caching of enabling status and structural properties of the model, the number of enabling computations can be reduced even further. We extend such an algorithm to handle prioritized transitions by modifying the step where transitions are picked at random to instead pick transitions at random in a highest-priority-first order, so enabled transitions with higher priority are executed before transitions with lower priority. We present an algorithm and data structures supporting this.

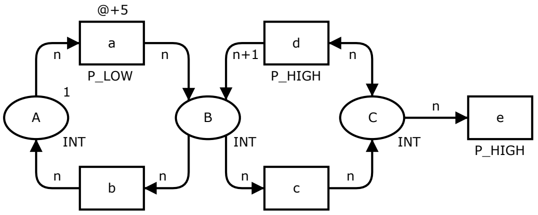
When a tool shows a model during simulation in a graphical user interface, the enabling status of transitions is typically shown to allow users to pick between enabled transitions for guided simulation. To do this, the enabling state of all transitions must be computed. It is not necessary to recompute the enabling status of all transitions after each execution of a transition, though. We only need to recompute the enabling of transitions for which it has potentially changed, and we can give a static over-approximation of this which roughly says that if a transition is connected to a place also connected to the executed transition, its enabling may have changed. We present an even better approximation in Sect. 2. This approximation is not good enough if allowing priorities, as the execution of a transition may enable or disable a transition with the highest priority, thereby causing unconnected transitions to be disabled or enabled. We present an algorithm for over-approximating the set of transitions influenced by this.

The remainder of this paper is structured as follows: in the next section, we present background material and in Sect. 3 we present algorithms for efficiently finding a random enabled transition taking priorities into account, and for efficiently updating the enabling status of all transitions. In Sect. 4, we conclude and provide directions for future work.

2 Background

In this section we briefly introduce coloured Petri nets using an example and describe an efficient algorithm for enabling computations. The algorithm is described in further detail in [6, 11].

A Petri net is a bipartite graph, where the nodes are partitioned into *places* and *transitions*. Places are usually drawn as circles or ellipses and transitions



	Dependency Set	Disable Set
a	{b, c}	{a}
b	{a}	{b, c}
c	{d, e}	{b, c}
d	{b, c}	∅
e	∅	{e}

Fig. 1: A simple coloured Petri net.

are typically drawn as rectangles or lines. In Fig. 1, we see a Petri net with 3 places (A–C) and 5 transitions (a–e). Places can contain *tokens* and represent the state of the system. In coloured Petri nets tokens are distinguishable and can have a value from the type of the place they reside on. In Fig. 1, all places have type INT (integer) and the only token is a single one with the value 1 residing on place A. Places and transitions are connected using directed arcs. Arcs describe preconditions and postconditions for transitions and are inscribed with *expressions* which may contain typed *variables*. For example, the arc from the place A to the transition a has inscription n, which is a variable of type INT. We allow double arcs as an abbreviation of an arc in both directions with the same expression. In the example, we have a double arc between C and d.

A transition of a CPN model is *enabled* if there exists a *binding* of values to all variables on arcs surrounding it so all *input places* (places with arcs to the transition) contain all tokens dictated by evaluation of the corresponding arc expressions. A transition with a binding is called a *binding element*. In Fig. 1, the transition a is enabled in the binding $n = 1$ as A contains a single token with value 1. An enabled binding element can be *executed*, consuming tokens on input places, and producing new tokens on *output places* (places with an arc from the transition). When a is executed in the binding $n = 1$, it consumes the single token 1 from A and produces a new token on the place B.

Tokens can have an attached time stamp, and are only available when a global clock reaches a value larger than or equal to their associated time stamp. Transitions can have execution times, shown as @+ annotations. In Fig. 1 only transition a has an execution time, namely 5. If a transition with an execution time is executed, all produced tokens shall have a time stamp that is the current global time plus the execution time of the transition. For example, if a is executed at time 2 in the binding $n = 1$, the token on A is consumed and a new token with value 1 and a time stamp of 7 $(2 + 5)$ is produced on B. Transitions b and c are not enabled before the global time reaches 7.

We can at any time partition transitions into enabled and *disabled* (i. e., not enabled) transitions. Computing enabling is a complex task, so CPN Tools implements an algorithm which uses heuristics to find bindings in a way that is fast in practice (see [6, 11] for details), but even using this technique, computation takes considerable time. If we just want to execute a random transition, there is no need to compute the enabled state of all transitions; we randomly pick a transition, check whether it is enabled, and if it is we execute it in a random binding. If the transition is not enabled, we cannot execute it and just continue with the next transition. This strategy, although better than computing the enabled state of all transitions, throws away information, namely that a transition is known to be disabled. As we execute transitions, transitions may move from disabled to enabled and vice versa, but only some transitions can move when certain other transitions are executed. For example, executing transition a in Fig. 1 can never alter the enabled state of e as the places they are connected do not intersect. We can exploit this to do a more efficient enabling computation. For example, during an execution of the model in Fig. 1, we first try executing transition e, and find it is disabled. We then try executing a and succeed. Now, there is no need to recompute the enabling of transition e as the enabling of this transition cannot be altered by the execution of a. The *dependency set* of a transition t captures this and is the set of all transitions that may be enabled by executing t. This can be computed as all transitions for which an output place of t is an input place (not counting places connected with double arcs to t). Similarly, the *disable set* is the set of transitions that can become disabled by executing a transition. We have summarized the dependency sets and disable sets of transitions of Fig. 1 in the table in the right side of the figure.

When we deal with timed models, we can have an additional state for each transition: it is not enabled right now, but may become enabled at a later stage when time has increased. This leads us to partitioning transitions into three sets: the Disabled, the Unknown, and the MaybeReady. The first are transitions known to be disabled, the second are transitions for which the enabled state is not yet known, and the last is for transitions that are not enabled but may become so at a later point in time. We note, we do not have a set for enabled transitions, as we immediately execute a transition if it is found to be enabled.

An algorithm for random execution of transitions is shown as Algorithm 1. The algorithm works in time epochs, where Unknown contains all transitions that are possibly enabled in the current epoch and MaybeReady transitions that may become enabled in a later epoch. We start with all transitions in the MaybeReady set. We start an epoch by increasing the time of the epoch to the least time stamp any transition of MaybeReady can be enabled (l. 5) and move all transitions that can be enabled at that time to Unknown (l. 6). As long as transitions can be enabled at the current epoch (l. 7), we pick one randomly (l. 8). We assume the existence of a function Enabled which returns one of three values: enabled, disabled, and maybe_ready_at(n), where the last value not only indicates that the transition is not enabled now, but also provides an estimate (n) of when the transition may be enabled. If a transition is not enabled it is moved to either Disabled or MaybeReady. If the picked transition is enabled (l. 9), we execute it, add its dependency set to Unknown and remove its dependency set from Disabled and MaybeReady (ll. 10–13). If a transition is disabled or maybe_ready_at(n), we move it from Unknown to either Disabled or MaybeReady. The inner while loop (ll. 7–19) executes all transitions enabled in a single epoch, and the outer loop

Algorithm 1 Algorithm for enabling computation for timed models.

```
 1: Unknown $\leftarrow \emptyset$
 2: Disabled $\leftarrow \emptyset$
 3: MaybeReady $\leftarrow \{0\} \times Transitions.all$
 4: while MaybeReady $\neq \emptyset$ do
 5:    $IncreaseTime$(MaybeReady)
 6:    Unknown $\leftarrow RemoveLeast$(MaybeReady)
 7:    while Unknown $\neq \emptyset$ do
 8:       Pick any $t \in$ Unknown
 9:       if $Enabled(t) =$ enabled then
10:          $Execute(t)$
11:          Unknown $\leftarrow$ Unknown $\cup\ DependencySet(t)$
12:          Disabled $\leftarrow$ Disabled $\setminus\ DependencySet(t)$
13:          MaybeReady $\leftarrow$ MaybeReady $\setminus\ DependencySet(t)$
14:       else if $Enabled(t) =$ disabled then
15:          Unknown $\leftarrow$ Unknown $\setminus \{t\}$
16:          Disabled $\leftarrow$ Disabled $\cup \{t\}$
17:       else if $Enabled(t) =$ maybe_ready_at($n$) then
18:          Unknown $\leftarrow$ Unknown $\setminus \{t\}$
19:          MaybeReady $\leftarrow$ MaybeReady $\cup \{(n, t)\}$
```

(ll. 4–19) executes all epochs. The algorithm terminates when (if) there are no more transitions in MaybeReady and Unknown, so all transitions are in Disabled.

The operations needed for Unknown are to add all transitions, pick a random element, add a set of elements not already contained, and remove a particular element. This can be efficiently implemented by enumerating all transitions from $0, 1, \ldots, |\mathit{Transitions}| - 1$, storing them in an array A of size $|\mathit{Transitions}|$ and adding a pointer *last* pointing to the position after the last element of Unknown. Add all transitions can be performed by setting all entries of the array to their index ($A[i] := i$) and setting the last pointer to $|\mathit{Transitions}|$, picking a random element corresponds to drawing a random number $r \in \{0, 1, \ldots, \mathit{last} - 1\}$ and returning the value $A[r]$. Adding a set of not already contained elements consists of adding the elements to positions $\mathit{last}, \mathit{last} + 1, \ldots$ and incrementing *last* accordingly. Removal of an element consists of swapping the element with the last one and decrementing the *last* counter. By combining the get random element and remove operations (this is possible by moving lines 15 and 18 up after line 8 in algorithm 1 and adding any transition to its own dependency set) we can perform picking in constant time and insertion in time linear in the number of elements we insert. We call this data-structure a *RandomSet* and use it to implement Unknown. For MaybeReady we insert each transition with a weight, namely the time at which it is earliest enabled, and only remove elements with the least weight, which naturally makes us implement MaybeReady as a priority queue, allowing us to add and remove elements in time $\log |\mathit{Transitions}|$ for each element. Storing the position of elements in the priority queue also allows us to remove internal elements (needed to remove the dependency set of a transition) in the same time. We never read from the Disabled set, and hence do not need to explicitly represent it. It is only shown to make the algorithm clearer (and can be computed as the complement of Unknown and MaybeReady anyway).

3 Algorithm

In this section we develop an algorithm for fast random execution of transitions for timed coloured Petri net models using priorities. We also develop algorithms for operations useful for graphical tool support for simulation and modification of such models. We also present experimental performance data of the algorithms on both toy examples and several real-life models [5, 10, 12] developed in other contexts.

When we talk about coloured Petri nets with priorities, we assign to each transition an expression evaluating to a nonnegative integer indicating the priority of the transition. Priorities considered here are global and cannot depend on the binding of the transition; we later discuss other priority concepts. We can think of the priority as a function assigning to each transition t a numeric priority, $Priority(t)$[1]. At any point in time, a transition is *preenabled* if all tokens required for executing the transition are available (also taking time into account). Only the transitions with the highest priority among the preenabled transitions are actually enabled. In the model in Fig. 1, we have assigned priorities to a, d, and e, namely P_LOW, P_HIGH, and P_HIGH respectively. We assume we have defined constants such that P_LOW < P_NORMAL < P_HIGH and that transitions without a priority inscription have priority P_NORMAL. Here we just use three levels of priorities, but our algorithm handles an arbitrary number, p.

3.1 Random Execution

Our goal is to randomly execute transitions quickly, adhering to the priorities. We use algorithm 1 as a basis. Extending this algorithm to handle priorities is simple: instead of picking transitions completely randomly in line 8, we pick them randomly among the transitions with the highest priority.

A way to implement this efficiently is to use a priority queue of *RandomSet*s for Unknown. That is, for each priority, we have a *RandomSet* like earlier. We can get nearly the same time guarantees for this implementation as for the simple *RandomSet*. We can get and remove an element with the lowest priority in time $\log p$ where p is the number of different priorities used (3 in the example). This extra cost (compared with constant time previously) is incurred as we may have to rebalance the priority queue. The time required to add elements to Unknown depends on the implementation. If we use no auxiliary data structure, we may need to search the priority queue for the correct *RandomSet* to insert into, i. e., insertion takes time p for each element. We can keep a search tree mapping priorities to *RandomSet*s, lowering the insertion time to $\log b$ for each element. We

[1] In our implementation we actually use a low number as high priority, but our explanation shall not reflect that for improved readability.

can also maintain an array mapping priorities to *RandomSet*s, bringing down insertion time for each element to constant time. This, however, comes at the cost of using memory linear in the highest numeric value of a priority. Finally, we could store the *RandomSet*s in a hash-map mapping priorities to the corresponding *RandomSet*, which allows constant time look-up and using space linear in p but using a larger constant than using the array. Unless p is large, which one we use in practice has little influence on the speed of the algorithm. We do not expect p to be larger than 10 in practice. We call any such implementation a *PriorityRandomSet* and obtain an algorithm for random execution of transitions adhering to priorities by using algorithm 1 with a *PriorityRandomSet* implementation for Unknown. In CPN Tools we use the implementation using an array as index into the priority queue to impose as little overhead in execution time as possible (as we do not have to traverse a pointer-based data-structure, but just look up a value in an array). We notice that if $p = 1$ all representations collapse to the same as the implementation not taking priorities into account, as we never have to rebalance the priority queue and search in the auxiliary data-structure pointing into the priority queue.

3.2 Random Enabling Computation

If we want to compute enabling for all transitions, this is easily done: sort the transitions according to priority and compute enabling highest-priority first. When an enabled transition is found, we stop computing enabling for transitions with lower priority.

Sometimes this may not be desired, however. For example, if a user is only looking at part of a model, the tool may only need to compute enabling for parts of the transitions (the visible ones) to show enough information to the user. Furthermore, we wish our algorithm to also efficiently handle maintenance of a set of enabled transitions, which can be done without recomputing enabling for all transitions. Hence, we seek an algorithm for computing the enabling of a random transition as efficiently as possible but still adhering to priorities. Furthermore, we want the algorithm to efficiently compute enabling of subsequent transitions, i.e., the main focus is on amortized running time.

When we want to compute enabling for a transition, we need to know whether any transition with higher priority is enabled. If we are computing enabling for more than one transition, part of this work may be reusable. For example, in Fig. 1, if we want to compute the enabling for a, b and c, we first need to establish the enabling of d and e as their priorities are higher. Naturally, this computation only needs to be done once, even if we first compute enabling for a and b and in a subsequent call (without executing any transition) for c.

The idea of our enabling computation algorithm is to use the data-structures Unknown, Disabled, and MaybeReady from algorithm 1. Ignoring priorities for the time being, we update the data-structures as in the inner loop in lines 8–19 of algorithm 1, except we do not execute transitions, and hence do not do operations based on dependency sets (ll. 11–13). The adapted algorithm is algorithm 2. The algorithm only checks if transitions are enabled at the current time, and needs

Algorithm 2 Algorithm for checking enabling without priority.

```
 1: proc $CheckEnabling(t)$ is
 2:   if $t \notin$ Unknown then
 3:     return false
 4:   else
 5:     if $Enabled(t) =$ enabled then
 6:       return true
 7:     else if $Enabled(t) =$ disabled then
 8:       Unknown $\leftarrow$ Unknown $\setminus \{t\}$
 9:       Disabled $\leftarrow$ Disabled $\cup \{t\}$
10:       return false
11:     else if $Enabled(t) =$ maybe_ready_at$(n)$ then
12:       Unknown $\leftarrow$ Unknown $\setminus \{t\}$
13:       MaybeReady $\leftarrow$ MaybeReady $\cup \{(n, t)\}$
14:       return false
```

Algorithm 3 Simple algorithm for checking enabling with priority.

```
1: SortedTransitions $\leftarrow PrioritySort(Transitions.all)$
2: proc $CheckEnablingPriority(t)$ is
3:   for all $t' \in$ SortedTransitions do
4:     if $Priority(t') > Priority(t)$ then
5:       if $CheckEnabling(t')$ then
6:         return false
7:     else
8:       return $CheckEnabling(t)$
9:   return $CheckEnabling(t)$
```

somebody external to increase time and move elements from MaybeReady to Unknown when Unknown becomes empty. We note that we could use a bit-array of entries in Unknown to retain constant time look-up in line 2 and maintain the performance of all other operations.

To also handle priorities, we can use algorithm 2 as a subprocedure to compute preenabledness, i. e., whether a transition is enabled when ignoring priorities. A simple way to do this is shown as algorithm 3; we sort all transitions according to priority and process them highest-priority-first until we reach t. If we find a preenabled transitions with higher priority than t, we return false. If we do not find a preenabled transition with higher priority than t we return the preenabledness of t. We assume that we traverse the transitions in a highest-priority-first order in line 3, and have introduced early termination as soon as the condition in the if statement in line 4 no longer holds. This is acceptable, as enabling of a transition with the same or lower priority cannot affect the enabling of t. If a transition is in Disabled it does not only mean it is disabled, but the stronger condition that it is not even preenabled.

We choose to compute SortedTransitions based on all transitions instead of based on Unknown (which would also work), as we then can precompute this for a given model, making the execution *CheckEnablingPriority* independent of this computation.

When this algorithm is called repeatedly, it only calls *Enabled* for each transition with higher priority than the first preenabled transition or the transition with the lowest priority (whichever is higher) plus once for each call (as soon as a transition is marked as disabled, it is no longer in Unknown). The number of calls to *CheckEnabling* is the sum of the numbers of transitions with higher priority than each of the transitions, which can be quadratic in the number of transitions (if each transition has a unique priority and only the one with the highest priority is enabled). A call to *CheckEnabling* is cheap as long as it does not result in a call to *Enabled*, but if we want to limit the number of calls here, we could introduce an approximation of the priority of the first enabled transition in SortedTransitions. As long as we have not found an enabled transition, this estimate is $-\infty$, and it is set to the priority of the first enabled transition as soon as one is found. We also maintain an index of the last transition checked for enabling, so we do not check transitions already verified to be disabled again, thus skipping calls to *CheckEnabling*. The resulting algorithm allows the same bound on the number of calls to *CheckEnabling*, namely one for each call plus one for each transition with priority higher than or equal to the first preenabled transition or the transition with the lowest priority (whichever is higher).

In CPN Tools we have implemented the version of the algorithm shown in algorithm 3, i. e., without estimation of the priority of the first enabled transition. This is done because CPN models rarely have more than a few transitions (50-100), so traversing SortedTransitions imposes a very small overhead.

3.3 Enabling Set Maintenance

Often we wish to run a random simulation and show intermediate results to users. We therefore wish to merge algorithm 1 (augmented to handle priority as described earlier) and 3 into a single algorithm sharing Unknown, Disabled, and MaybeReady in a way that makes it possible to do random simulation as well as to check enabling of selected transitions with as few calls to *Enabled* as possible.

We can get by with few changes, as we do not have to change algorithm 3 as long as we faithfully maintain Unknown, Disabled, and MaybeReady. The best place to call *CheckEnablingPriority* is between lines 9 and 10 in algorithm 1, as this is the only place we know we have increased the time sufficiently that a transition is enabled.

We can call *CheckEnablingPriority* for all the transitions we are interested in, but that is not necessary. The reason we wish to avoid that in CPN Tools is that this incurs a communication overhead, as the GUI and the simulator are separate processes. This can be relevant for any tool if the number of transitions is high, as enabling checks no longer depend directly on the total number of transitions in the model.

If we disregard priority, the enabling status can only have changed for transitions in the dependency set of the last transition executed, but when taking priority into account, things are not as simple, as the enabling of a transition

Algorithm 4 Algorithm for random simulation using priority while maintaining the set of all enabled transitions.

```
1:  SortedTransitions ← $PrioritySort(Transitions.all)$
2:  Unknown ← $\emptyset$
3:  Disabled ← $\emptyset$
4:  MaybeReady ← $\{0\} \times Transitions.all$
5:  Enabled ← $\emptyset$
6:  while MaybeReady $\neq \emptyset$ do
7:     $IncreaseTime$(MaybeReady)
8:     Unknown ← $RemoveLeast$(MaybeReady)
9:     while Unknown $\neq \emptyset$ do
10:       Enabled ← $\{t' \in$ Unknown $\mid CheckEnablingPriority(t')\}$
11:       while Enabled $\neq \emptyset$ do
12:          Pick any $t \in$ Enabled
13:          $Execute(t)$
14:          Unknown ← Unknown $\cup DependencySet(t)$
15:          Disabled ← Disabled $\setminus DependencySet(t)$
16:          MaybeReady ← MaybeReady $\setminus DependencySet(t)$
17:          Enabled ← Enabled $\setminus DisableSet(t)$
18:          New ←
                $\{t' \in DependencySet(t) \cup DisableSet(t) \mid CheckEnablingPriority(t')\}$
19:          if New $\neq \emptyset$ then
20:             if $\exists t1 \in$ New$, t2 \in$ Enabled$.Priority(t1) > Priority(t2)$ then
21:                Enabled ← New
22:             else
23:                Enabled ← Enabled $\cup$ New
```

with higher priority than all currently enabled transitions will disable them. We know that all transitions that have remained in the Disabled set since last time are still there (i. e., if a transition was not preenabled before and not in the dependency set of the transition executed last, it is still not preenabled). We also know that only if new transitions become enabled do we have to disable other transitions. If we disable all enabled transitions and do not enable any with the same or higher priority, we need to consider the preenabled transitions or increase the model time. We can thus compute the enabled transitions using algorithm 4. Here, we maintain a set Enabled in addition to the ones we already maintain. This set contains all enabled transitions and aside from initialization (l. 10), which takes place initially and whenever we need to increment time because no more transitions are enabled, we only ever update it according to the dependency set and the disable set of executed transitions (ll. 17, 20, 23, 25, and 27). This algorithm can be made interactive by pausing and asking the user for a transition to execute in line 12.

3.4 Extension to Other Priority Concepts

While the priority concept detailed until now, assigning to each transition a fixed numeric priority, is in line with standard statically prioritized Petri nets [3], it is not very high-level. For example, we cannot assign higher priority to a specific task in a folded net (such as assigning d priority depending on n in Fig. 1). In [2] a dynamic priority concept is adopted. This allows priorities to depend on the entire marking of the model. In our opinion, this is way too centralized to easily comprehend and specify.

With CPNs the natural way to assign dynamic priorities to transitions is using general expressions just like guards or arc expressions. Although this is a natural priority concept for coloured Petri nets, we have chosen not to adopt it. The problem is that when the priority depends on the binding of transitions, we have to compute every preenabled binding of every transition, subsequently compute the priorities for each preenabled binding element, and finally pick one with highest priority. Although this is conceptually nice and consistent with the other inscriptions, it leads to dramatically decreased performance. The remainder of this section is dedicated to extending the static notion of priority presented hitherto while compromising performance as little as possible.

Using a Subset of Variables in Priorities CPN Tools, in addition to restricting the number of times enabling of a transition is called, also partitions all variables surrounding a transition into *binding groups*. A binding group is a subset of the variables surrounding a transition that can be assigned values independently of all other variables (i. e., if two bindings of a transition are enabled, the binding obtained by replacing the value of all variables in a binding group in the first binding by the binding of the same variables from the second binding is also enabled). Variables that occur in the same arc expression or the guard must be within the same group. By requiring that all variables occurring in the priority expression come from the same binding group, we can just compute all possible bindings of variables in that binding group instead of for all variables of the transition.

It is always correct to combine two binding groups into one, so in the worst case transitions only have one binding group, forcing us to compute all enabled bindings of all transitions anyway. We believe, though, that only a small subset of variables will be used in the priority, typically just a process ID or an independent priority on a place. In those cases, we can compute the priority for all possibilities of the binding group, schedule the transition with all resulting priorities, and execute it like before. This approach requires that we dynamically add/remove transitions to SortedTransitions and compute all partial bindings for the binding group comprising variables of the priority inscription for all transitions in $DependencySet(t) \cup DisableSet(t)$ whenever we execute t. We have not implemented this, as we believe that users may inadvertently build nets that take prohibitively long to simulate, and many interesting cases can be solved by splitting a transition into several, one for each desired priority. For example, if we want d in Fig. 1 to execute with low priority if $n > 5$, we can just make two copies of d, one with high and one with low priority, and give the highly prioritized one a guard n<=5 and the one with low priority a guard n>5.

Scoped Priorities It is often useful to be able to use scoped priorities. For coloured Petri nets with hierarchy [7], this means that we would like to say that a given transition has higher or lower priority than all other transitions on the same page (module), but it should not necessarily be considered less important than enabled transitions on other pages. This is useful for implementing multiple schedulers (e. g., for two separate but connected systems) and for handling errors in multiple places without preempting unconnected operations (e. g., handle stale messages on different communication channels). Furthermore, making priorities local makes it much easier to use modular analysis techniques.

We can implement scoped priorities by running any of the algorithms for each page in isolation (using algorithm 1 with a *PriorityRandomSet* for random simulation, algorithm 3 if we want to compute enabling, and using algorithm 4 to maintain a set of enabled transitions). We introduce a new top loop which randomly selects a page to execute a step on. We have not implemented this in CPN Tools as we have not found an elegant way of having both scoped and global priorities coexist in an easy-to-understand manner. An added advantage is that flattening of a hierarchical CPN model remains a purely syntactical operation, where we would otherwise have to consider interplay of local priorities.

3.5 Experimental Validation

We have compared the algorithm for random non-interactive simulation (algorithm 3) developed in this section with a naive algorithm just evaluating enabling in a highest-priority-first order and an algorithm computing all enabled bindings for all transitions before selecting a transition to execute. Our findings are summarized in Table 1. We have executed the algorithms with three toy examples shipping with CPN Tools: the dining philosophers, a distributed database, and a simple stop-and-wait protocol. We have also tested with three industrial examples: a protocol for routing in mobile ad-hoc networks (ERDP) [10], the DYMO protocol for route discovery in mobile ad-hoc networks [5], and a protocol for operational support for workflow execution (OS) [12]. All models have been developed independently of the implementation of priorities and hence represent natural examples and not pathological examples designed to put our algorithms in a good light. We also show extended versions of the operational support protocol, modeling more details of the system, and a version with all extensions disabled in the model which is behaviorally equivalent to the original model, but has more transitions to consider. We have made large and small versions of the OS model; the small model (OS) only has a few participants, making the size of the model suitable for state-space analysis, and the large version (OS') has many more participants and can only be analyzed using simulation. The three versions of OS use priorities while the other models do not (as the others were developed before CPN Tools supported priorities). For each model, we show the complexity as reflected by the number of modules, the number of transition instances, and the number of place instances. We also show the number of place instances after merging all places in a port/socket assignment relationship as

Table 1: Experimental results.

Model	Instances			Transitions/minute		
	Pages	Transitions	Places	All Bindings	Priority Sorted	Algorithm 3
Philosophers	1	3	3 (3)	$3.21 \cdot 10^6$	$12.39 \cdot 10^6$	$22.19 \cdot 10^6$
Database	1	5	9 (9)	$5.01 \cdot 10^6$	$12.20 \cdot 10^6$	$17.26 \cdot 10^6$
Protocol	1	5	10 (10)	$2.81 \cdot 10^6$	$7.42 \cdot 10^6$	$21.18 \cdot 10^6$
ERDP	14	16	65 (11)	$0.50 \cdot 10^6$	$1.20 \cdot 10^6$	$3.97 \cdot 10^6$
DYMO	15	25	55 (18)	$0.75 \cdot 10^6$	$2.53 \cdot 10^6$	$4.14 \cdot 10^6$
OS	25	42	134 (25)	$2.44 \cdot 10^6$	$4.01 \cdot 10^6$	$6.64 \cdot 10^6$
Extended OS 1	31	50	164 (36)	$1.68 \cdot 10^6$	$2.70 \cdot 10^6$	$6.26 \cdot 10^6$
Extended OS 2	31	50	164 (36)	$2.06 \cdot 10^6$	$3.29 \cdot 10^6$	$6.05 \cdot 10^6$
OS’	25	42	134 (25)	$0.43 \cdot 10^6$	$1.24 \cdot 10^6$	$4.21 \cdot 10^6$
Extended OS 1’	31	50	164 (36)	$0.37 \cdot 10^6$	$0.94 \cdot 10^6$	$4.11 \cdot 10^6$
Extended OS 2’	31	50	164 (36)	$0.44 \cdot 10^6$	$1.21 \cdot 10^6$	$4.32 \cdot 10^6$

well as places in a fusion group in parentheses. We show the number of transitions we can execute for each model and algorithm. These tests are performed by running CPN Tools 3.0.3 on a computer with a 2.7 GHz Core i7 Sandy Bridge dual core CPU (using one core only). All tests were run for 5 minutes and the average has been reported. The tests repeatedly execute a model and resets the scheduler structures Unknown and MaybeReady as well as the state of the model when no more transitions are enabled. We have not evaluated the performance of algorithm 4 as it incurs a large communication overhead due to the architecture of CPN Tools. We have not compared with a baseline simulator without priority for two reasons: First, we only have an implementation with an old version of the simulator, which for independent reasons is much slower, and second, the performance when a model does not use priorities is exactly the same whereas the performance of a model using priorities is incomparable, as the lack of support for priorities may cause the model to be able to reach states not reachable when using priorities, thus comparing different behavior.

We see that using our optimized algorithm, the toy examples can execute around 20 million transitions a minute. The largest gain is from not computing all bindings (though we may compute enabling for all transitions). The reason is that toy examples often have few transitions but a lot of enabled bindings for each. Thus, computing enabling of all transitions is not very expensive (as this terminates early in our implementation) but computing all bindings is. For real-life models, we see that performance of the simple algorithms significantly decreases as the number of transitions grow. The performance of our improved algorithm is roughly constant at $4 - 7$ million transitions a minute. When looking at the results for the large and small versions of OS, we see the improved algorithm handles the large model almost without any penalty whereas the simple algorithms are orders of magnitude slower. This is again because we have more enabled bindings of each transition. The penalty of the optimized algorithm for real-life models stems from the fact that each transition is much more complex (calls functions and does more advanced matching on the input tokens consumed) and therefore takes longer to execute.

4 Conclusion and Future Work

We have presented algorithms for performing fast simulation of coloured Petri nets with priorities. We have given details on performing fast random simulation of CPN models with statically prioritized transitions. We have given an algorithm for performing fast amortized enabling check of statically prioritized transitions without assuming that enabling is tested in a specific order. Additionally, we have given an algorithm which can be used for maintaining a set of enabled transitions during simulation, providing fast user-guided simulation with interactive feedback. We have implemented all these features in CPN Tools 3.0 [4], and our experiments show we are able to execute $4 - 7$ million transitions a minute for real-life models and more than 20 million transitions for other models. This is an improvement over the $1 - 5$ million transitions a minute for simple algorithms.

We have considered and sketched algorithms for extending our algorithms to handle dynamically prioritized transitions and for scoped priorities. We have chosen not to implement these, first, because dynamic priorities are prone to introducing performance bottlenecks, and, second, because we have not been able to introduce scoped priorities in a way that nicely coexists with global priorities. As scoped priorities can be useful, it would be nice to consider this in more detail.

We have not been able to find any published work concerning efficient simulation of models with priorities. We think this is because the problem only really becomes important with high-level Petri nets, where enabling computation is several orders of magnitude more computationally expensive than for low-level net classes. We have experimented with using the scheduling algorithm 1 for Place-Transition Petri nets, but have not been able to make it outperform a simple algorithm trying transitions at random without the extra book-keeping. The complexity of enabling computation for high-level nets stems from the fact that the high-level nature makes modelers more prone to generating many tokens, and that these tokens are not equal, so in the worst case we have to try all combinations of tokens. Papers treating simulation of low-level nets with priorities often translate nets with priorities to nets without, e. g., [3]. Work exists on translating high-level nets with priorities to nets without [8] or for doing distributed simulation with priorities present [9]. Here, we instead focus on efficient algorithms for direct simulation of high-level nets, which allows us to do optimizations not possible in a parallel or distributed setting.

Our algorithms can also be used for analysis by means of state-space exploration. As for simulation, analysis can be done by translating to equivalent models without priority [3, 8] and for low-level nets additionally by means of static analysis or restriction [1, 2]. This is probably because a strong feature of low-level nets is that this kind of analysis is possible. For high-level nets, analysis is usually only possible by means of state-space exploration or simulation, making fast simulation algorithms more important. Our current state-space tool implementation in CPN Tools is tuned toward breadth-first traversal, so unless we store the Disabled and MaybeReady sets for each state, we cannot make use of this information without recomputing it from scratch each time. As the state-space tool of CPN Tools comprises a lot of legacy code, we decided that instead of doing this directly, we would use the algorithm for computing enabling (algorithm 3) instead. It would be interesting to look into algorithms for improving state space analysis by computing transitions according to their priority and also to investigate ways of using the Disabled and MaybeReady sets during state-space generation.

References

1. F. Bause. On the analysis of Petri nets with static priorities. *Acta Informatica*, 33:669–685, 1996. 10.1007/BF03036470.
2. F. Bause. Analysis of Petri Nets with a Dynamic Priority Method. In *Proc. of ATPN'97*, volume 1248 of *LNCS*, pages 215–234. Springer, 1997.
3. E. Best and M. Koutny. Petri net semantics of priority systems. *TCS*, 96(1):175–215, 1992.
4. CPN Tools webpage. Online: `cpntools.org`.
5. K.L. Espensen, M.K. Kjeldsen, and L.M. Kristensen. Modelling and Initial Validation of the DYMO Routing Protocol for Mobile Ad-Hoc Networks. In *ATPN'08*, volume 5062 of *LNCS*, pages 152–170. Springer, 2008.
6. T.B. Haagh and T.R. Hansen. Optimising a Coloured Petri Net Simulator. Master's thesis, Dept. of Computer Science, Aarhus University, 1994.
7. K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009.
8. H. Klaudel and F. Pommereau. A Concurrent and Compositional Petri Net Semantics of Preemption. In *Proc. of IFM'00*, volume 1945 of *LNCS*, pages 318–337. Springer, 2000.
9. M. Knoke, F. Kühling, A. Zimmermann, and G. Hommel. Towards Correct Distributed Simulation of High-Level Petri Nets with Fine-Grained Partitioning. In *Proc. of ISPA'04*, volume 3358 of *LNCS*, pages 64–74. Springer, 2004.
10. L.M. Kristensen and K. Jensen. Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad-hoc Networks. In *Integration of Software Specification Techniques for Application in Engineering*, volume 3147 of *LNCS*, pages 248–269. Springer, 2004.
11. K.H. Mortensen. Efficient Data-Structures and Algorithms for a Coloured Petri Nets Simulator. In *Proc. of 3rd CPN Workshop*, volume 554, pages 57–74. DAIMI PB, 2001.
12. M. Westergaard and F.M. Maggi. Modelling and Verification of a Protocol for Operational Support using Coloured Petri Nets. In *Proc. of ATPN'11*, LNCS. Springer, 2011.

Modelling Local and Global Behaviour: Petri Nets and Event Coordination

Ekkart Kindler

Informatics and Mathematical Modelling
Technical University of Denmark
eki@imm.dtu.dk

Abstract. Today, it is possible to generate major parts of a software system from models. Most of the generated code, however, concerns the structural parts of the software; the parts that concern the functionality or behaviour of a system are still programmed manually. In order to overcome this problem, we are developing the concept of *coordination diagrams* that define the global behaviour on top of structural software models. Basically, these diagrams define how the local behaviour of an element is coordinated with the behaviour of the elements it is connected to. The exact concepts of these coordination diagrams and their notation is still under development, but there exists a first prototype for experimenting and for fine-tuning its features. We call it the *Event Coordination Notation* (*ECNO*).
For experimenting with the ECNO, we implemented also a simple modelling notation for the local behaviour, which is based on Petri nets. In this paper, we briefly discuss the general idea of the ECNO and then present *ECNO nets* that define the local behaviour of elements. They are implemented as a Petri net type for the ePNK tool, together with a code generator that produces code that can be used in the ECNO framework and runtime environment. This way, all the behaviour of a system can be modelled – and code can be generated that easily integrates with the structural models and existing software.
Keywords: Model-based Software Engineering, Local and global behaviour modelling, Event coordination, Code generation, ECNO nets.

1 Introduction

Software models and the automatic generation of code from these models are becoming more and more popular in modern software development – as suggested by the success of one of the major approaches, the Model Driven Architecture (MDA) [1]. In many cases, however, code generation concerns the structural parts or the standard parts of the software only; as soon as the actual functionality and behaviour is concerned, major parts of the software are still programmed manually. As pointed out in previous work [2], the reason is not so much that there are no modelling notations for behaviour or that it would be difficult to generate code from these models. The actual problem is to integrate the behaviour models or the code generated from them with the code generated from the structural models and with pre-existing parts of the software. One of the main reasons for that problem is that, ultimately, the only mechanism for integrating different parts of the software is invocation – of functions, of methods, or of services.

Based on some earlier ideas [3, 4, 2], we set out to develop a concept that allows us to identify *events* in which different partners or parts of the software could or should participate; the notation allows us to define how the execution of these events should be *coordinated*. The overall behaviour of the system would then be a result of this coordination and synchronization of events combined with the local behaviour of the different parts. In a way, this is similar to aspect orientation with its join points and point cuts [5], but with a different focus. The development of this notation is still in progress; in order to gain some experience, though, we have implemented a first prototype, which consists of a modelling part and an execution runtime environment, which we call *Event Coordination Notation* (*ECNO*). From this prototype, we hope to learn more about which constructs do help to adequately model and coordinate behaviour and to collect efficiency and performance results for larger systems and for more complex coordinations. This way, we intend to fine-tune ECNO's constructs and notations and to strike a balance between obtaining an adequate notation on a high level of abstraction, with sufficient expressive power and universality on the one-hand side, and efficient and performant execution on the other side.

As stated above, the research on ECNO is still in progress. Its major concern is the integration of the *local behaviour* of elements by coordinating the execution of their events. In ECNO, the focus is on the coordination, i. e. on *global behaviour*. The local behaviour would still be programmed manually based on an API which is part of the ECNO framework (see [6] for more details). On the one hand side, the possibility of programming the local behaviour in a traditional way, makes it possible to integrate ECNO with classical software development approaches. We consider this possibility as a major feature of ECNO – in particular easing the gradual transition from programmed software to modelled software. One the other hand, programming is very tedious and not very much in the spirit of model-based software engineering (MBSE). Therefore, we started a side-line that is concerned with modelling the local behaviour and generating code from that. We use an extended version of Place/Transition nets (P/T-nets) [7, 8] for that purpose – mostly, because of our own background in Petri nets and because we have a generic tool, the ePNK [9], which allows us to easily define new net types of Petri nets and which is integrated with the eclipse platform that provides all the necessary infrastructure and technology for code generation from models and software development in general [10].

The main contribution of this paper is on the extended version of Petri nets (Sect. 3) for modelling local behaviour: ECNO nets. But, in order to understand the local behaviour models we need to see the context in which this local behaviour is coordinated. Therefore, we start with explaining the idea and concepts of ECNO (Sect. 2).

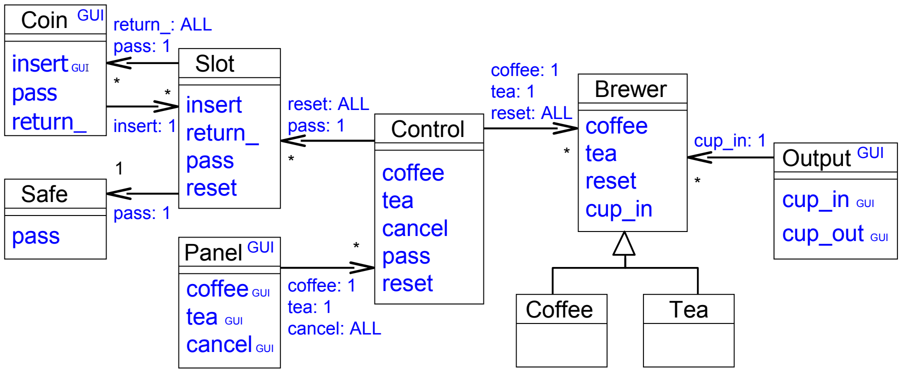


Fig. 1. A class and coordination diagram

2 The Event Coordination Notation

In this section, we discuss the main ideas and main concepts of ECNO. We start with an example in Sect. 2.1 and summarize the concepts in Sect. 2.2.

2.1 Example

The concepts of ECNO will be explained by using a simple example: the eternal coffee (and tea) vending machine. Figure 1 shows a UML class diagram[1] with some extensions concerning *events* and their *coordination*. Therefore, we call it *coordination diagram*.

Before explaining the extensions, let us have a brief look at it as a UML class diagram. The diagram shows the different possible *elements*[2] that are part of the system: A coin can be close to the slot, which is represented by the reference from Coin to Slot, or a coins can be in the slot, which is represented by the reference from Slot to Coin. There is a Safe to which the coin will be passed, once a coffee or tea is dispensed. There is a Panel for the user to interact with the vending machine. This panel is connected to controllers, which is represented by the reference from Panel to Control. The controllers are connected to the brewers, which can be either coffee or tea brewers. At last, there is an output device for the beverage, which is connected to the brewers. Note that Fig. 1 is a kind of class diagram. Therefore, there can be different configurations, which could be defined as an *instance* of this class diagram (in UML this would be an object diagram). In our example, we assume that there are (initially) three coins (not inserted yet to the slot), and that there are two coffee brewers and one tea brewer; for

[1] For the experts, this actually is an EMOF diagram [11] or an Ecore diagram [10]; but this is not relevant here.

[2] In order to point out that our objects are not objects in the traditional sense of object orientation, we call them *elements*.

all other classes, we assume that there is exactly one instance. Remember that, in these instances, the references of the class diagram are represented by *links* from one element to another (where the link's type is the reference).

Now, let us explain the extensions concerning the coordination of events between the different elements: First of all, there are some events mentioned in the operations section of the class diagram, like insert, pass and return_[3] for Coin. They define in which *events* the different elements could be involved or could participate. The actual definition of these events will be discussed later (see Fig. 2). More importantly, the references between the different elements are annotated with events and an additional quantifier, which can be 1 or ALL. These annotations define the coordination of events and which events and elements needed to be executed together. We call such a combination of elements and events an *interaction*. The meaning of these annotations is as follows: Let us assume that some element is involved in the execution of some event and that, in the coordination diagrams, the type of this element has a reference that is labelled with that event. Then, some elements at the other end of the respective link also need to participate: to be more precise, if the event is quantified by 1, exactly one of these elements must participate; if the event is quantified by ALL, all the elements at the other end of these links need to participate.

In our example, let us assume that there are two coins inserted to the slot. This would be represented by two links from the slot to a coin – one to each of the coins. If the slot does a return_ event, the annotation return_:ALL at the reference from Slot to Coin means, that both coins must participate in the execution of the event return_.

Note that, normally, this is required for all the references that are annotated with the event. In our example, there are two references from Slot that are annotated by pass – one to Coin and one to Safe. Therefore, when a slot participates in a pass event, exactly one coin and exactly one safe will need to participate in this interaction – this way, the coin will be passed from the slot to the safe, which will be discussed later in Sect. 3.1.

In some cases, we do not want all these references to be considered. To this end, the ECNO provides the concept of *coordination sets*, which allows us to define which references should be followed together. But, we do not discuss the details here since this is not relevant in this paper (see [6] for some more details).

Up to now, we have used events basically as names. In general, events can have parameters, which can be used to share information between the different partners that are participating in the event and the interaction. Figure 2 shows the declaration of all events of our vending machine along with the *event parameters*. On a first glance, this looks very much like method declarations. In contrast to methods, events do not have behaviour of their own, and they do not belong to a particular element (or are not owned by them). Therefore, event parameters can be contributed in many different ways, and by different elements. It is not defined in advance, who will provide and who will use the parameters.

[3] Since return is a key word of our target language Java, we use an additional underscore in the event name return_.

```
insert(Coin coin, Slot slot);    coffee();
pass(Coin coin, Slot slot);      tea();
return_(Slot slot);              cup_in();
reset();                         cup_out();
cancel();
```

Fig. 2. Event declarations

And it is not clear in which direction the values will be propagated. The execution engine of ECNO, however, guarantees that all elements participating in an interaction have the same parameters for the same event – if two partners contribute inconsistent values, the interaction would not be possible. Likewise, the interaction would not be possible if some partner needs[4] a parameter, but there is no partner that provides it.

A minor extension to class diagrams are the GUI annotations. These annotations indicate which elements and events are relevant to the user – and actually can be triggered by the user. As the name GUI indicates, this is mostly relevant for the GUI part of execution engine for generating buttons and user dialogs. In our case, there will be only buttons (see Fig. 7 in Sect. 4). Actually, it will not always be the GUI that triggers events; events can be triggered also programmatically; to this end, some elements would be registered with some specific controllers.

2.2 Summary of concepts

Altogether, ECNO extends class diagrams by the explicit definition of *events* and in which way different elements need to participate when an event is executed. This is defined by *coordination diagrams*, which are an extension of class diagrams. The basic mechanism for defining these coordination requirements is annotating references with an *event* and a *quantification.* Each of these references defines a bilateral coordination. In combination, however, they might require that many different elements participate in an *interaction*: First, there might be different references for the same event, which require different other elements to participate. Second, the other elements that are required to participate might have references with annotations, which require further elements to participate; in this way, establishing a chain of required elements. Third, an event annotation with quantification ALL requires that all the elements at the other end of the respective links participate. As we will see later, there is a fourth possibility: the local behaviour of an element can require synchronization of two different events. This way, cooperation diagrams define the *global behaviour* of a system based on the *local behaviour* of its elements[5].

[4] We will see later in the Petri nets defining the local behaviour what that means.

[5] Harel and Marelly called the global behaviour inter-object behaviour and the local behaviour intra-object behaviour [12] – in a different setting though.

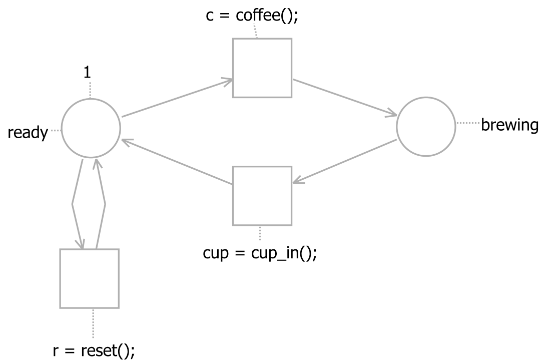


Fig. 3. Local behaviour of the coffee brewer

In the ECNO, there exist some more concepts (e. g. collective parameters of events) and we have some ideas of additional constructs, which are not discussed here (see [6] for a bit more information).

Figure 1 does not say anything about the local behaviour of the elements. The definition of the local behaviour will answer the following questions: when can an event be executed by an element, what is the local effect, and which (different) events need to be executed together (synchronized). ECNO provides an API for defining, or more precisely, for programming local behaviour for every element. But, in the context of this paper, we will use Petri nets for defining the local behaviour. These extended Petri nets will be discussed in the next section.

3 Modelling local behaviour with Petri nets

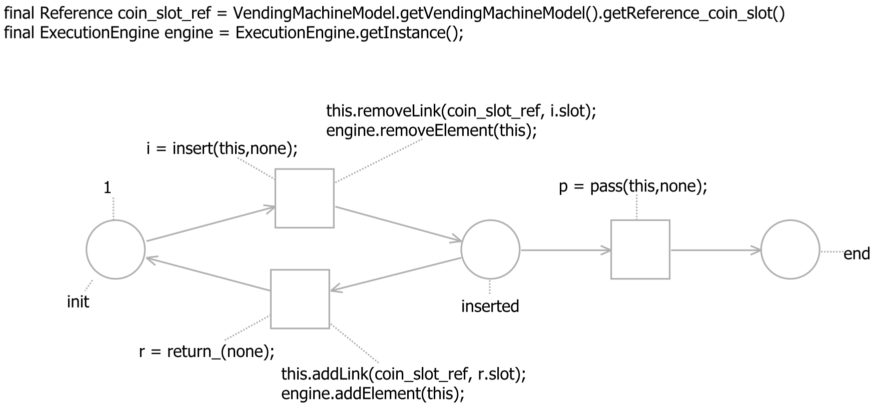
The *local behaviour* of an element defines when and under which (local) conditions an event or a combination of events can be executed locally, and it defines what happens locally when the event is executed as part of the global interaction.

3.1 Examples

Not surprisingly, this local behaviour can be defined by a slightly extended version of P/T-nets. We will discuss the main concepts by the help of some examples first. The concepts will then be clarified and explained in general in Sect. 3.2.

We start with a simple ECNO net that models the coffee brewer. It is shown in Fig. 3. Except for the annotations associated with the transitions, this is a conventional P/T-net. The transition annotations relate a transition of the Petri net to an[6] event; we call this annotation an *event binding.* After the event coffee, which represents the user pressing the coffee button, the coffee is brewed, which

[6] We will see later, that one transition can actually be bound to more than one event – this way enforcing the synchronization of different events.



```
final Reference coin_slot_ref = VendingMachineModel.getVendingMachineModel().getReference_coin_slot()
final ExecutionEngine engine = ExecutionEngine.getInstance();
```

this.removeLink(coin_slot_ref, i.slot);
engine.removeElement(this);
i = insert(this,none);
1
p = pass(this,none);
end
init
inserted
r = return_(none);
this.addLink(coin_slot_ref, r.slot);
engine.addElement(this);

Fig. 4. Local behaviour of the coin

will be dispensed, when a cup is inserted (event cup_in). The reset event is possible only when the coffee machine is in the initial state (no coffee is being made). In this example, the notation for event bindings might appear a bit verbose, and it is not clear why, the event is assigned to some "variable". This will become clear in the next example, when referring to the events' parameters in conditions or actions. In the coffee brewer, these assignments do not have any particular meaning.

The behaviour of the coin is more interesting. It is shown in Fig. 4. First of all, the event bindings and the involved events have parameters. Let us consider the one with the insert event: as we have seen earlier, insert has two parameters, the coin and the slot. The annotation refers to these two parameters. The first one, this, assigns the coin itself as the first parameter (coin) to this event. The second parameter is none, which is the keyword indicating that in this instance, the coin does not assign a parameter to the event insert (it could do that in other instances though). The other annotation of this transition is the *action*, which will be executed when all partners of an interaction are found. In this case, the coin does two things: First, it deletes the link to the slot (since it is inserted now) and it also removes itself from the engine so as not to be visible at the GUI anymore. The first line removes the link, which is "addressed" by the coin_slot_ref attribute (which is a constant which will be discussed later). The slot from which it is removed is denoted by i.slot, where i is the variable to which the insert event was assigned, and slot is one of its parameters, which is assigned to the event by the slot. This is why we needed to give a name to an event in the event binding.

Once the coin is inserted, the Petri net for the coin allows two things to happen: either the coin can be passed to the slot by the transition that is bound to event pass, or the coin is returned by the transition bound to event return_. In the case of pass, the coin assigns itself (this) as the coin parameter; and there

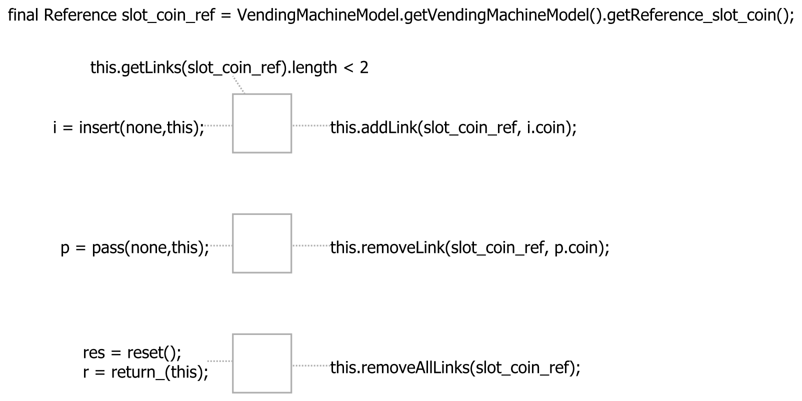

final Reference slot_coin_ref = VendingMachineModel.getVendingMachineModel().getReference_slot_coin();
this.getLinks(slot_coin_ref).length < 2
i = insert(none,this);
this.addLink(slot_coin_ref, i.coin);
p = pass(none,this);
this.removeLink(slot_coin_ref, p.coin);
res = reset();
r = return_(this);
this.removeAllLinks(slot_coin_ref);

Fig. 5. Local behaviour of the slot

is no action. In the case of a return_, no parameter is assigned to the event (none), but the action will add a link from the coin to the slot again, where the slot is coming from the parameter r.slot of the event return_. Moreover, the coin registers itself with the engine again, so as to be visible in the GUI again.

Another extension that we see in this example is the declaration of attributes at the top of Fig. 4. These declarations follow the syntax of the Java language. Therefore, the additional keyword final actually defines a constant. In the declarations of Fig. 4, the first line uses the ECNO mechanism to get access to the reference feature from Coin to Slot in the model – this is very similar to the mechanisms of the Eclipse Modeling Framework (EMF) [10] and we do not go into details here. This feature is assigned to the attribute coin_slot_ref, which is then used in the two actions to add, resp. remove a link of that kind. The second line gets access to the execution engine of the ECNO framework (for adding and removing the coin from and to the GUI).

Figure 5 shows the local behaviour of the slot. This is a rather degenerated net. As a P/T-net, all transitions would be enabled all the time since their presets are empty. Due to the event bindings, however, the local behaviour becomes a bit more interesting. We start with explaining the bottom transition: This transition, actually, has two events bound to it: reset and return_. This implies that reset and return_ must be executed together; this way, all coins will be returned during a reset. The slot assigns itself as a parameter to the return_ event. Moreover, the slot deletes all the links to the coins it contains (i. e. it returns the coins).

The transition bound to the pass event is even simpler. When it happens, the link to the coin that is passed (accessible via the parameter p.coin) is removed (since it is removed from the slot and passed to the safe).

At last, let us discuss the top transition of Fig. 5. It is bound to the insert event, where the slot assigns itself to the event's slot parameter. In the action, it sets a link to that coin. This transition uses another concept: the *condition* which

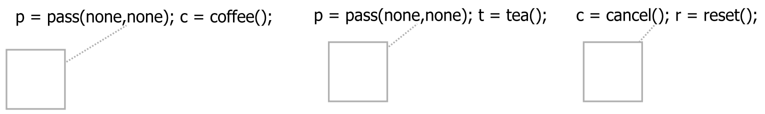

p = pass(none,none); c = coffee();
p = pass(none,none); t = tea();
c = cancel(); r = reset();

Fig. 6. Local behaviour of the control

is shown above the transition. This condition guarantees that an insert event can happen only when there are less than two coins in the slot. The condition refers to the attributes (the references to the coin) in this example, but it could in principle also refer to the parameters of the involved events.

The Petri net models for the other elements are similar. The last one that we discuss here is the one for the control. This net is shown in Fig. 6. The first transition guarantees that the event coffee (pressing the coffee button on the panel) must go together with an event pass (passing a coin from the slot to the safe). This way, it is indirectly guaranteed that there is a coin inserted. Therefore, the coffee button on the panel is enabled only when there is a coin inserted to the slot and when the coffee brewer is ready. The second transition does the same for event tea. The last event synchronizes the cancel event (which is triggered by pressing the cancel button on the panel) with the reset event: this way, it is indirectly guaranteed that all coins that are currently inserted to the slot are returned (see discussion of the local behaviour of the slot).

3.2 Concepts

As we have seen in Sect. 2.2, the main concepts of ECNO are the explicit definition of events and the coordination annotations of references that define how to coordinate the execution of events. At runtime, the combination of all the elements and events that meet these requirements will form an interaction.

The main concept for modelling the local behaviour of an element is to define when the element can participate in an event and what happens when the event is executed. In our Petri net notation, the event binding for transitions define when an event or a combination of some events is possible. To this end, the transitions of the Petri net refer to the respective events, and at the same time provide the parameter that this element would contribute to the event.

In our example, these parameters where expressions with values of the element (mostly this in our example). But, actually this can be more involved. For example, if we have some event event(Integer x, Integer y), it would be possible that an event binding looks as follows: e = event(none,e.x+1). The meaning of this is that the element takes the first parameter of the event increments it and assigns that value to the second parameter. If there were more events in the binding, we could also use a parameter of one event and assign it as a parameter to another event. Actually, there is no restriction in which way this could be done – if there are cyclic dependencies in these assignments in some interaction, the assignments will not be possible; the interactions will not be complete and, therefore, will not be executable. It could also be that two partners would provide a value to the same parameter of the same event. The result will also be an illegal interaction, if the provided parameter are not the same (equal in Java terms).

On the one hand, this mechanism of parameter passing provides much expressive power and flexibility. Within the same interaction, data can be passed in opposite directions. On the other hand, such power requires great care in using this mechanism in order not to preclude desired executions by unintended cyclic dependencies. This, however, is a question of methodology and analysis functions that can check that no cycles of that nature would occur (which however are yet to be developed). The execution engine of ECNO copes with these situations – though computing all the parameters in complex situations might be quite computation intensive. Making this more efficient is ongoing research.

In the extended Petri nets, transitions have two more extensions: conditions and actions. Conditions are expressions that may refer to local attributes of the element (and whatever the element can access from there); and they may refer to the parameters of the events in which it participates. Conceptually, the condition is evaluated when all necessary parameters of the events are available. If, in a given combination of events, the condition evaluates to false, the complete interaction is not executable. Only if the conditions of all participating elements evaluate to true, the interaction is executable; its execution amounts to executing all the actions of all participating elements (in an arbitrary order). An action may refer to and access and change the local attributes and the parameters. The parameters or rather the objects they refer to can be changed – a parameter object itself cannot not be replaced, which follows the principle of parameters in the Java language.

3.3 Discussion

Altogether, the ECNO together with ECNO nets for the local behaviour allow to fully model the behaviour of a system. The generated code together with the ECNO runtime environment implement that behaviour (see Sect. 4).

Since the ECNO is still under development, there are still some issues open that need to be adjusted in order to strike a balance between usability of the language (expressing behaviour on a high level of abstraction) and performance. This is ongoing research and requires some larger case studies.

Technically, the ECNO nets define the local behaviour of the elements or parts of a system only. To some Petri net people, this might look a bit strange, since Petri nets are typically used to describe the overall behaviour or rather the global behaviour of a system. In fact, this is more a question on how these nets are used. The approach used here could be used for coordinating behaviour in a more global way: to do that, we would have one or more elements that are global and coordinate other elements by the mechanisms proposed here. That is why our approach is technically local, but the Petri nets could still be used for coordinating behaviour globally.

4 ECNO engine and generated code

Though the ECNO is still under development, the current prototype fully supports the concepts discussed in our example, and there is a code generator that fully automatically generates the code for the elements from the ECNO nets. An experimental ECNO runtime environment as well as the code generator for ECNO nets are deployed as an extension to Eclipse (Galileo or Helios) via the ePNK update site; You will find the information on how to install this experimental ECNO release, the example discussed in this paper, and some explanations on how to generate the code from ECNO nets at `http://www2.imm.dtu.dk/~eki/projects/ECNO/`.

In this paper, we give a brief overview of the resulting software and of how the generated code could be used. The generated code and the ECNO runtime environment can be run as a stand-alone Java application (under JRE 1.6 or higher). When the "VendingMachineInstance" is started as a Java application, a GUI will start, which looks like the ones in Fig. 7. The left one shows the GUI immediately after startup; the middle one shows the GUI after pressing the "insert" buttons on two coins (the last coin cannot be inserted anymore); the right one shows the situation after pressing the "coffee" button (note that since two coins were inserted and there are two independent coffee brewers, you could order the next coffee right away).

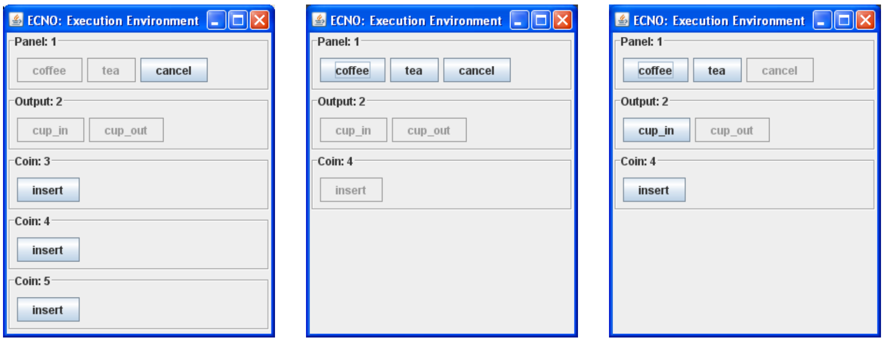


Fig. 7. Three screenshots: initial, after inserting two coins, after ordering a coffee

Note that a code generator for the coordination diagrams is not yet deployed. Therefore, the model from Fig. 1 is "programmed" in class "VendingMachineModel", in a style that is similar to the automatically generated Package and Factory classes for Ecore models of the Eclipse Modeling Framework (EMF) [10]. Based on that "programmed model", the class "VendingMachineInstance" sets up the concrete configuration of a coffee machine.

In this paper, we cannot discuss the details of the generated code since this would require a more detailed discussion of the ECNO runtime engine, its architecture, and the ECNO programming API (see [6] for some more details).

Basically, each element has a method that, for a given event, defines all the possible *choices*; in turn, a choice defines how to calculate the parameters for the events it participates in (which might depend on parameters of other events), a condition defining whether the choices is enabled, and a method that finally executes the action of the element. If you install the experimental prototype, there will be an example project which includes the generated code.

As motivated earlier, the generated code should integrate with other existing code, which could be legacy code, manually written code, or code generated from other models. This integration is possible along different lines: First, the generated code can be used by other parts of the software in the classical object oriented way[7] without taking its ECNO specific extensions into account – and without compromising the ECNO interactions. Note that, in order to be a bit more agile, the current prototype uses its own philosophy to generate the object oriented code from models. In principle however, ECNO could use any other technology for automatically generating code from class diagrams; once the prototype of ECNO stabilizes, we intend to use the EMF technology with its vast amount of related technologies and tools. Second, the code snippets in the ECNO nets (and in general any other implementations of the local behaviour) could use and invoke any other part of the software. Third, other parts of the software could use the API of the ECNO runtime to find out about enabled interactions and initiate their execution (see [6] for details). At last, existing software could be integrated with the ECNO by using adapter elements in the coordination diagrams, which delegate their behaviour to other parts of the software. Clearly, the first two "lines of integration" would be the most convenient ways; the last one is the most tedious one, since the adapters would require manual implementation – but it should not be too difficult.

5 Related work

The ideas for ECNO have developed over some years; they started out in the field of Business Process Modelling, where we used events and their synchronization for identifying and formalizing the basic concepts of business process models and their execution: AMFIBIA [13, 3]. It turned out that these ideas are much more general and do not only apply in the area of business processes. This generalisation resulted in MoDowA [4]. MoDowA, however, was tightly coupled to aspects, and event coordination was possible only for very specific types of relations. Therefore, the quest for distilling the basic coordination primitive was still on. In [2], we pointed out some first ideas for such an event coordination notation, which we call ECNO now.

Actually, none of the concepts of ECNO are particularly new or original. For example, Petri nets [7, 8] have been made exactly for the purpose of modelling

[7] This would require the ECNO engine to notify the controllers that are triggering possible interactions about changes in the possible interactions, which is not yet implemented.

behaviour. And many different mechanisms have been proposed for coordinating different Petri nets. For example, the box calculus and M-nets [14, 15] use synchronization of transitions, for coordinating different Petri nets. Also Renew [16, 17] uses executable Petri nets and has a concept of synchronous channels for synchronizing Petri nets.

The idea of events and the way they are synchronized dates back even further (in the earlier work, they would rather be called actions, which should however not be confused with our concept of action). ECNO's synchronization mechanism resembles process algebras like CSP [18], CSS [19], or the π-calculus [20]. But, we are a bit more explicit with whom to synchronize and with how many partners. Actually, there can be arbitrarily long chains or networks of required participants in our approach. One approach that allows to define such interactions (via connectors) is BIP [21]; but ECNO embeds a bit smoother with class diagrams and allows for and is tuned to dynamically changing structures. And it works together with classical programming and method invocation. Other parts of the software could call methods of our elements as they please; we just need to make sure that after such calls the possible interactions are updated. And, our actions can call methods of other parts of the software, and via an API other parts of the software can initiate and hook into interactions.

The ideas of the ECNO are an extension of our MoDowA approach (Modelling Domains with Aspects) [4], which has some similarities with the Theme approach [22]. In MoDowA, the interactions were restricted and implicitly defined in two special kinds of relations. Therefore, we introduced a separate concept and notation on top of references for making interactions explicit [6]; some of these ideas came up during the work on the master thesis [23]. Technically, ECNO is independent from aspect oriented modelling. Still, it was inspired by aspect orientation and is close in spirit to aspect oriented modelling (see [24, 25] for an overview) or subject orientation [26]. Moreover, ECNO could serve as an underlying technology for easily implementing aspect oriented models. In a way, events are join points and the interactions are point cuts as, for example, in AspectJ [5] in aspect oriented programming. There are two main differences though: in aspect oriented programming, the join points are defined in the final program; in that sense, the join points are artifacts and not concepts of the domain model. By contrast, our events are concepts of the domain! The other difference is the more symmetric interpretation and participation in an interaction. The participants in an interaction are not only invoked by another element; they can actively contribute parameters, and even prevent an interaction from happening (if no other partners are available). A more subtle difference is that interactions in our approach are attached to objects (actually, we called them elements) and not to lines in a program, and interactions can only be defined along existing links between the elements. This is a restriction – but a deliberate one: it provides more structure and avoids clutter and unexpected interactions between completely unrelated elements.

One of the main objectives of ECNO and coordination diagrams is that the coordination of events should relate to the structural models (a UML class diagram). The rationale is the smooth extension of used technologies and easing the integration of behaviour models with structural ones and with pre-existing software.

To sum up, all the individual concepts of ECNO existed before – we did not invent them. The main contribution is the combination of these concepts, and this way, making them more usable in practical software development in general – and in model-based software engineering in particular.

6 Conclusion

In this paper, we have briefly discussed an event coordination notation, which we call ECNO. This notation allows defining global behaviour of a system by coordinating local behaviour: this global coordination is defined on top of structural diagrams. The focus of this paper is on a specific modelling notation for the local behaviour which we called ECNO nets (for a discussion of the ECNO engine and its API, we refer to [6]). From these models, the code for the complete system including its behaviour can be generated. The example shows that the coordination mechanisms of ECNO for defining global behaviour together with the mechanisms for defining local behaviour are powerful enough to completely define a system and generate code for it.

The most interesting research on ECNO, however, is yet to come: Scalability, performance, and adequateness of the modelling notation still need investigation; the constructs need to be adjusted, so as to strike a balance between these different objectives. The prototype implementation discussed in this paper, lays the foundation for these investigations.

Acknowledgements Since ECNO has developed over several years, many colleagues, students, and anonymous reviewers have contributed to what ECNO is now. They are too many to name them here; but, there is a half-way complete list on the ECNO Home page: `http://www2.imm.dtu.dk/~eki/projects/ECNO/`.

References

1. OMG: MDA guide. `http://www.omg.org/cgi-bin/doc?omg/03-06-01` (2003)
2. Kindler, E.: Model-based software engineering: The challenges of modelling behaviour. In Aksit, M., Kindler, E., Roubtsova, E., McNeile, A., eds.: Proceedings of the Second Workshop on Behavioural Modelling - Foundations and Application (BM-FA 2010). (2010) 51–66 (Also published in the ACM electronic libraries).
3. Axenath, B., Kindler, E., Rubin, V.: AMFIBIA: A meta-model for the integration of business process modelling aspects. International Journal on Business Process Integration and Management **2**(2) (2007) 120–131
4. Kindler, E., Schmelter, D.: Aspect-oriented modelling from a different angle: Modelling domains with aspects. In: 12^{th} International Workshop on Aspect-Oriented Modeling. (2008)
5. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Proc. of *ECOOP 2001 – Object-Oriented Programming*, 15^{th} European Conference, Springer (2001) 327–353

6. Kindler, E.: Integrating behaviour in software models: An event coordination notation – concepts and prototype. In: 3rd Workshop on Third Workshop on Behavioural Modelling - Foundations and Application, Proceedings. (2011) Accepted.
7. Reisig, W.: Petri Nets. Volume 4 of EATCS Monographs on Theoretical Computer Science. Springer-Verlag (1985)
8. Murata, T.: Petri nets: Properties, analysis and applications. In: Proceedings of the IEEE. Volume 77. (1989) 541–580
9. Kindler, E.: ePNK: A generic PNML tool - users' and developers' guide : version 0.9.1. Tech. Rep. IMM-Tech. Rep.2011-03, DTU Informatics, Kgs. Lyngby, Denmark (2011)
10. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. 2nd edition edn. The Eclipse Series. Addison-Wesley (2006)
11. OMG: Meta Object Facility (MOF) specification, version 1.4.1. Tech. Rep. formal/05-05-05, The Object Management Group, Inc. (2005)
12. Harel, D., Marelly, R.: Come let's play: Scenario-based programming using LSCs and the Play-engine. Springer (2003)
13. Axenath, B., Kindler, E., Rubin, V.: An open and formalism independent metamodel for business processes. In Kindler, E., Nüttgens, M., eds.: Workshop on Business Process Reference Models 2005 (BPRM 2005), Satellite event of the third International Conference on Business Process Management. (2005) 45–59
14. Best, E., Devillers, R., Hall, F.: The box calculus: A new causal algebra with multi-label communication. In Rozenberg, G., ed.: Advances in Petri nets 1992. Volume 609 of LNCS. Springer-Verlag (1992) 21–69
15. Best, E., Fraczak, W., Hopkins, R.P., Klaudel, H., Pelz, E.: M-nets: An algebra of high-level Petri nets, with an application to the semantics of concurrent programming languages. Acta Inf. **35**(10) (1998) 813–857
16. Kummer, O.: A Petri net view on synchronous channels. Petri Net Newsletter **56** (1999) 7–11
17. Kummer, O., Wienberg, F., Duvigneau, M., Schumacher, J., Köhler, M., Moldt, D., Rölke, H., Valk, R.: An extensible editor and simulation engine for Petri nets: Renew. In Cortadella, J., Reisig, W., eds.: ICATPN. Volume 3099 of Lecture Notes in Computer Science., Springer (2004) 484–493
18. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
19. Milner, R.: Communication and Concurrency. International Series in Computer Science. Prentice Hall (1989)
20. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes (Parts I & II). Information and Computation **100**(1) (1992) 1–40 & 41–77
21. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Software Enigneering and Formal Methods, Forth IEEE International Conference, IEEE Computer Society (2006) 3–12
22. Clarke, S., Baniassad, E.: Aspect-oriented analysis and design: The Theme approach. Addison-Wesley (2005)
23. Nowak, L.: Aspect-oriented modelling of behaviour – imlementation of an execution engine based on MoDowA. Master's thesis, DTU Informatics (2009)
24. Brichau, J., Haupt, M.: Survey of aspect-oriented languages and execution models. Tech. Rep. AOSD-Europe-VUB-01, AOSD-Europe (2005)
25. Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Alarcon, M.P., Bakker, J., Tekinerdogan, B., Siobhán Clarke and, A.J.: Survey of aspect-oriented analysis and design approaches. Tech. Rep. AOSD-Europe-ULANC-9, AOSD-Europe (2005)
26. Harrison, W., Ossher, H.: Subject-oriented programming (a critique of pure objects). In: OOPSLA, ACM (1993) 411–428

Towards Verifying Parallel Algorithms and Programs using Coloured Petri Nets

Michael Westergaard

Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands
m.westergaard@tue.nl

Abstract. Coloured Petri nets have proved to be a useful formalism for modeling distributed algorithms, i.e., algorithms where nodes communicate via message passing. Here we describe an approach for modeling parallel algorithms and programs, i.e., algorithms and programs where processes communicate via shared memory. The model is verified for correctness, here to prove absence of mutual exclusion violations and to find dead- and live-locks. The approach can be used in a model-driven development approach, where code is generated from a model, in a model-extraction approach, where a model is extracted from running code, or using a combination of the two, supporting extracting a model from an abstract description and generation of correct implementation code. We illustrate our idea by applying the technique to a parallel implementation of explicit state-space exploration.

1 Introduction

Parallel and distributed computing address important problems of scalability in computer science, where some problems are too large or complex to be handled by just one computer. Until now, focus has mostly been on distributed algorithms, i.e., algorithms running on multiple computers communicating via a network, as access to parallel computers, i.e., computers capable of running multiple processes communicating via shared memory (RAM), has been limited. For this reason, there are many papers on modeling distributed algorithms, such as network protocols [3, 5, 11, 12]. With the advance of cheap multi-core processors and cheap multi-processor systems, access to multiple cores has become more common, and the development and analysis of algorithms for parallel processing becomes very interesting. As parallel computing allows much faster communication between processes, tasks that were not previously feasible or efficient to do concurrently becomes interesting. In this paper, we present our experiences developing parallel algorithms with synchronization mechanisms developed and verified by means of *coloured Petri nets* (CPNs) [10]. This work was motivated by the requirement for a parallel state-space exploration algorithm. In this paper, we provide an approach that allows us to extract a model for analysis from a program or abstractly described algorithm in a systematic way. We do this in a way that allows us to automatically generate a skeleton implementation of the algorithm subsequently. We use a simple state-space algorithm as example, but the approach has also been used for other parallel algorithms, such as parts of a protocol for operational support [16].

Classically, formal models can partake in a development in two different ways: by extracting an implementation from a model, which we call *model-driven software engineering*, or by extracting a model from an implementation, which we call *model-extraction*. Our focus in this paper is on model-extraction but in a way that allows us to also do code generation, thereby allowing a new combined approach. The model-driven engineering approach is shown in Fig. 1 (top) and shows that we start with model which is verified according to one or more properties. If it is ok, we can extract a program, and otherwise we refine the model. Examples of this approach are within hardware synthesis [9, 17], using a CPN simulator to drive a security system [14], or general code generation from a restricted class of CPNs [13]. The model-extraction approach is shown in Fig. 1 (bottom). Here, we do not start with a model, but rather with a program. From the program, we extract a model and verify it for correctness. If an error is found, the resulting error-trace is replayed on the original program to determine if it can be reproduced here. If not, the abstraction used to extract the model is refined and the cycle restarts. This approach is rarely used in the high-level Petri net world, but is employed by, e.g., FeaVer [6] to translate C code to PROMELA code usable in SPIN [8], Java PathFinder [7] to translate Java programs to PROMELA, SLAM [1] for automatically translating C device drivers to boolean programs, BLAST [2] for model-checking C programs, and many other tools. Both of these approaches may terminate the loop without providing a definite response.

The model-driven software engineering and model-extraction approaches have different strengths and weaknesses. The main strength of the model-driven soft-

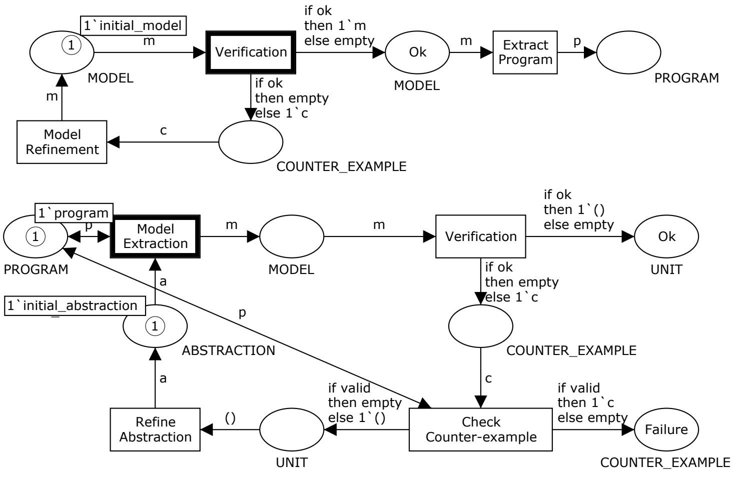


Fig. 1: Model-driven software engineering (top) and model-extraction (bottom).

ware engineering approach is that it is possible to verify an algorithm before implementation and we can even get a guaranteed-correct (template) implementation with little or no user-interaction. The disadvantage is that the approach is of little use for already existing software. The model-extraction approach precisely alleviates this by extracting a model from an existing implementation automatically, ensuring there is correspondence between the model and implementation. The main disadvantage is that we need an implementation of a, perhaps faulty, algorithm before analysis can start.

We would like to provide a translation supplying as many of the strengths of these approaches as possible. By supporting model-extraction in a way that allows subsequent code generation, we can support both approaches plus a new merged approach, shown in Fig. 2. Here we extract a model from a description in one – possibly abstract – language, manually or automatically refine the abstraction until we can prove the system to be correct (optionally modifying the original description if we find errors), and then extract an implementation. While the last step seems superfluous if we already have an implementation, it can be useful, for example, to use a program written in pseudo-code as input, automatically derive and verify a model, and from the model extract a runnable program in a desired implementation language. Alternatively, we can do round-trip engineering, where we take an implementation as input, verify and correct it on the level of a model, and update the original implementation to reflect the changes.

To do this, we use systematic extraction and abstraction methods that can be derived from a program or from an algorithm written in pseudo code. Our approach builds on *process-partitioned coloured Petri nets* (PP-CPNs) as described in [13]. The use of a (slightly restricted class of) CPNs allows us to refine data-structures as much as required and even using actual data-structures of the original algorithm or program. The restricted class of PP-CPNs allows us to automatically generate executable code from the model. Derivation of abstractions of the data-types used can be done by the user or automatically using counter-example guided abstraction refinement (CEGAR) [4] as implemented in SLAM [1] and BLAST [2]. CEGAR automatically improves abstractions by

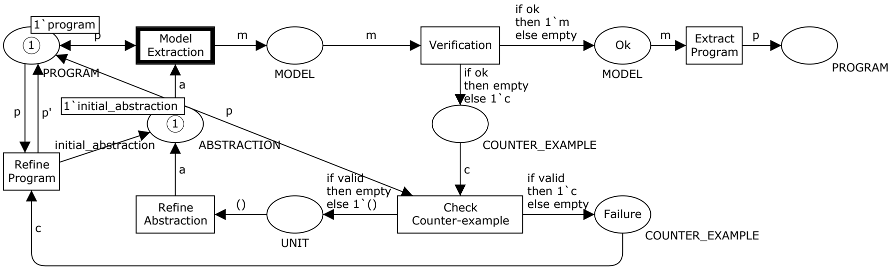


Fig. 2: Our approach combining model-driven software engineering and model-extraction.

replaying errors found in an abstract model on the original program and using about why a given error-trace cannot be replayed in the original program to refine the abstraction.

In this paper we focus on model-extraction. Our goal is to provide a proof-of-concept, so we do certain steps that can be automated by hand, such as the translation from code to a model using patterns. We do not address refinement after discovery of errors in this paper, but assume an external library using CEGAR or a user takes care of that. We have already treated the code generation aspect in [13].

The rest of this paper is structured as follows: in the next section, we introduce process-partitioned coloured Petri nets as defined in [13] and a simple algorithm for state-space generation which we use as running example to illustrate our idea. In Sect. 3, we introduce our approach to generating PP-CPN models from algorithms using a naive parallel version of the algorithm presented in Sect. 2. In Sect. 4, we use state-space generation to identify a problem in the original parallelization, fix the problem and show that the problem no longer is present in a modified version. Finally, in Sect. 5, we sum up our conclusions and provide directions for future work.

2 Background

In this section, we briefly introduce process-partitioned CPNs as defined in [13]. We also give a simple algorithm for explicit state-space generation which we use as example in the remainder of the paper.

Process-partitioned CPNs. Coloured Petri nets (CPNs) consist of *places*, *transitions*, and *arcs*. Places are typed and arcs have expressions that may contain variables. Places may contain tokens and we call the distribution of tokens on all places a *marking* of the net, and the marking before executing any transitions the *initial marking*. CPNs have a module concept, where *subpages* are represented by *substitution transitions*.

In [13] we introduce the notion of *process-partitioned CPNs* (PP-CPNs). These are CPNs, which are partitioned into separate kinds of processes. In this paper, we are only interested in models containing a single kind of process, so we just look at *process subnets* (Def. 2 in [13]). A single process subnet is a PP-CPN, but not necessarily the other way around, but in this paper, whenever we talk about PP-CPNs, we assume they consist of exactly one process subnet. A process subnet is a CPN with a distinguished *process colour set* serving as process identifier. The model in Fig. 6 (left) is an example of a PP-CPN (we provide a detailed description of the model in Sect. 3). The process colour set of this model is P. The places of a process subnet are partitioned into *process places*, *local places*, and *shared places* (in [13], we additionally introduce *buffer places* for asynchronous communication between processes, but these are not used here). These places correspond to the control flow, the local variables, and shared variables of normal programs. Process places must have the process colour set as type (in the example S and E and all unnamed places are process places), local places must have a product of the process colour set and any other type as type (in the example, s, b, and s'), and shared places can have any type (in the example, Waiting and Visited).

In the initial marking, exactly one of the process places must contain all tokens of the process colour set and the remaining process places must be empty (modeling that all processes must start in the same location in the program). Local places must initially contain exactly one token for each process so that if we project onto the component of the process colour set, we obtain exactly one copy of all values of the set (modeling that all local variables must be initialized). All shared places must contain exactly one value (modeling that shared variables must be initialized). All arc expression must ensure that tokens are preserved.

We have chosen to adopt the notion that we cannot create new processes or destroy processes from [13] even though nothing in our approach breaks if we allow dynamic instantiation and destruction of processes. This is mainly for simplicity as we did not need dynamic instantiation in our examples.

State-space Generation. State-space generation is a means of analysis of formal models, such as the ones specified by means of CPNs. A simple implementation is shown in Fig. 3. We start in the initial marking of the model and compute all enabled bindings. We then systematically execute each, note the marking we reach by executing bindings, store them in Waiting, and repeat the procedure for each of these newly discovered markings. To also terminate in case of loops, we store all markings for which we have already computed successors in Visited and avoid expanding them again. We often call a marking a *state* in the context of state-space analysis.

```
WAITING ← {MODEL.initial()}
VISITED ← {MODEL.initial()}
while WAITING $\neq \emptyset$ do
  Pick a $s \in$ WAITING
  WAITING ← WAITING $\setminus \{s\}$
  // Do any handling of $s$ here
  for all $b$ enabled in $s$ do
    Execute $b$ in $s$ to get $s'$
    if $s' \notin$ VISITED then
      WAITING ← WAITING$\cup\{s'\}$
      VISITED ← VISITED $\cup \{s'\}$
```

Fig. 3: Simple state-space exploration algorithm.

3 Approach

We introduce our approach to verifying parallel algorithms by a parallel version of the algorithm for generating state-spaces shown in Fig. 3. The basic idea is to use the loop of Fig. 3 for each process and share the use of Waiting and Visited, naturally with appropriate locking. From this algorithm, we illustrate our approach to extract a PP-CPN model. The last step, going from a PP-CPN model to implementation code, is handled in [13].

A simple way to parallelize Fig. 3 is shown in Fig. 4 (left). Here, we initialize as before (ll. 1–2). We have moved the main loop to a separate procedure, *computeStateSpace*. We perform mostly the same loop as before (ll. 16–24), but instead of testing for emptiness and picking an element of the queue in three steps, we do so using a procedure *pickAndRemoveElement* (ll. 17 and 24). The implementation of *pickAndRemoveElement* (ll. 4–9) does the same as we did before, except we return a bottom element $\perp$ if no elements are available and use that in the condition of the loop (l. 18). This forces us to perform the pick in two places: before the first invocation of the loop (l. 17) and at the end of the loop (l. 24). Handling of states (l. 19) and iteration over all enabled bindings (ll. 20–21) is the same as before. Now, instead of checking if a state is a member of VISITED and conditionally adding it to the set, we do both in a single step as shown in the procedure *addCheckExists* (ll. 22 and 11–14). We do this under the assumption that adding an element to the set does nothing if the element is already there. If the state was not already in VISITED, we add it to WAITING (l. 23). The reason for this re-organization is that we now assume that *pickAndRemoveElement*, *addCheckExists*, and the access to WAITING in line 23 are atomic, e.g., by creating a data-structure ensuring this or by requesting a lock for each data-structure before the start of an operation and releasing it afterward. This allows us to start two instances of *computeStateSpace* in parallel in lines 26–27. We will not argue for the correctness of neither Fig. 3 nor Fig. 4 (left), but note that it is easy to convince ourselves that if one is correct, so is the other with the assumption that *pickAndRemoveElement* and *addCheckExists* happen atomically.

Model Extraction. To go from Fig. 4 (left) to a PP-CPN model, we first extract the control-flow of the algorithm including generating representations of data, and then we refine the update of the data until we can prove the properties of the model we want.

Extracting the control-flow consists of creating the process places and transitions of the model. We do that using templates, very similar to the workflow-patterns [15] for low-level Petri nets. In Fig. 5 we show the patterns necessary to translate programs using our simple pseudo-code language to a PP-CPN model. From left to right the patterns match an atomic action (*Atomic*), a sequence of two subprograms ($S1; S2$), a conditional branch (**if** *condition* **then** $S1$ **else** $S2$), a while loop (**while** *condition* **do** S), and a critical section (**atomic** S). The type P is the process colour set and for each pattern, the place S is the start place and E the end place. All places created are process places except for the Mutex place, which is a shared place. We put all processes on the start place of the top level. We can add new templates or derive other constructs, such as a simplified conditional branch omitting the else path, a repeat/until loop, a for loop, and a for all loop.

In the initial abstraction, we translate a condition to an unbound boolean variable in the PP-CPN model. We add a local place for each local variable and a shared place for each global variable. These are available on all subpages where they are within the scope. In the initial abstraction, we approximate the type

```
 1: Waiting $\leftarrow$ {Model.$initial()$}
 2: Visited $\leftarrow$ {Model.$initial()$}
 3:
 4: proc $pickAndRemoveElement()$ is
 5:   if Waiting $= \emptyset$ then
 6:     return $\perp$
 7:   Pick a $s \in$ Waiting
 8:   Waiting $\leftarrow$ Waiting $\setminus \{s\}$
 9:   return $s$
10:
11: proc $addCheckExists(s')$ is
12:   $result \leftarrow s' \notin$ Visited
13:   Visited $\leftarrow$ Visited $\cup \{s'\}$
14:   return $result$
15:
16: proc $computeStateSpace()$ is
17:   $s \leftarrow pickAndRemoveElement()$
18:   while $s \neq \perp$ do
19:     // Handle $s$ here
20:     for all $b$ enabled in $s$ do
21:       Execute $b$ in $s$ to get $s'$
22:       if $addCheckExists(s')$ then
23:         Waiting $\leftarrow$ Waiting $\cup \{s'\}$
24:     $s \leftarrow pickAndRemoveElement()$
25:
26: $computeStateSpace()$
27:         || $computeStateSpace()$
```

```
 1: Waiting $\leftarrow$ {Model.$initial()$}
 2: Visited $\leftarrow$ {Model.$initial()$}
 3: MayAdd $\leftarrow 0$
 4:
 5: proc $pickWithCounter()$ is
 6:   $s \leftarrow pickAndRemoveElement()$
 7:   if $s \neq \perp$ then
 8:     MayAdd $\leftarrow$ MayAdd $+ 1$
 9:   return $s$
10:
11: proc $computeStateSpace()$ is
12:   repeat
13:     $s \leftarrow pickWithCounter()$
14:     while $s \neq \perp$ do
15:       // Do any handling of $s$ here
16:       for all $b$ enabled in $s$ do
17:         Execute $b$ in $s$ to get $s'$
18:         if $addCheckExists(s')$ then
19:           Waiting $\leftarrow$ Waiting $\cup$ $\{s'\}$
20:       MayAdd $\leftarrow$ MayAdd $- 1$
21:       $s \leftarrow pickWithCounter()$
22:   until MayAdd$=0$
23:
24: $computeStateSpace()$
25:         || $computeStateSpace()$
```

Fig. 4: Naive parallel state-space algorithm (left) and more involved algorithm (right).

of all variables with UNIT, and local and shared places are not connected to transitions.

Applying this extraction to the *computeStateSpace* procedure of Fig. 4 (left) and flattening it, we obtain Fig. 6 (left). Transitions are named after the line number they correspond to and conditions after the performed tests. Assign To Value stems from expanding the for all loop in line 18 to a while loop and line 19 has been omitted. We have instantiated the process twice (initial marking of S).

Abstraction Refinement. The initial abstraction allows execution of traces not allowed in the original program. In the model in Fig. 6 (left), it is possible to first terminate p(1) and have p(2) continue computation. This is not possible in Fig. 4 (left). The model does find all possible interleavings of the process, though, so if the state-space does not contain any erroneous states, neither will the program.

Abstraction refinement consists of using more elaborate types on local and shared places, of adding arcs for reading and updating local and shared places,

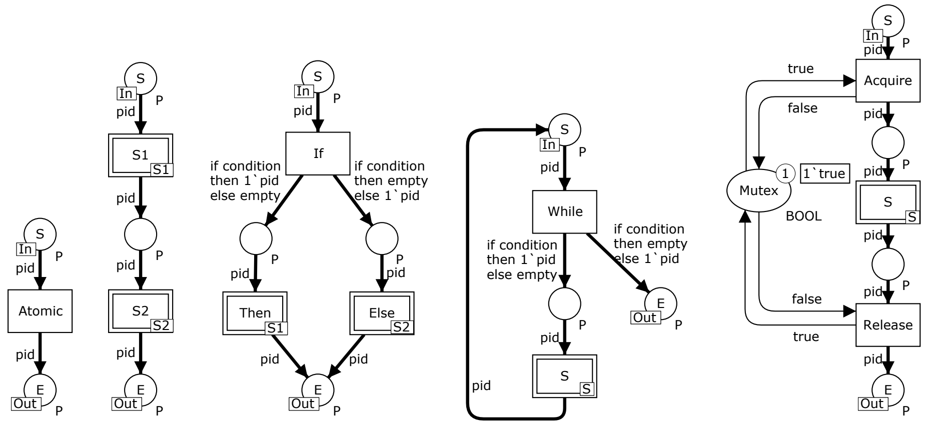


Fig. 5: Patterns for control structures.

and of limiting the values of condition variables, often using values from the local or shared places. Refinements must limit the behavior of previous models (which formally must be able to simulate any refinement if we ignore local and shared places), and must be true to the original program.

The basic idea is to refine the types of places modeling local and global variables and add tests accordingly. When we cannot determine an exact test or update, we solve it by non-deterministically choosing among the possible values. Here we only do a simple refinement to avoid the situation where one process can decide that Waiting is empty just to have the other immediately afterward decide it is not. For this, we refine the type of Waiting to a BOOL, indicating whether the Waiting set is empty or not, and refine the type of s to indicate whether $\perp$ was returned from *isEmpty*. The value of Waiting is initially false (as we add the initial state in line 1 of Fig. 4 (left)) and we do not care about the initial value of s, as it will be set before it is read. We make sure to read and update the values of Waiting and s faithfully. Adding an element to Waiting (l. 23) implies setting the value to true and for picking elements (ll. 17 and 24), we read the previous value of Waiting, use that to set the value of s, and non-deterministically set the value of the waiting set to true or false (though if the value already was true it remains so) as we do not have enough information to know which is the correct answer. We obtain the model in Fig. 6 (right). For readability, we have merged the transitions 17 and 24, but this is not necessary for analysis. We no longer can execute the erroneous trace on this refined model.

4 Analysis

The main reason we started this work in the first place is to analyze parallel algorithms. Our focus is on new problems arising when creating parallel algorithms, not on proving correctness of serial algorithms. We therefore assume that

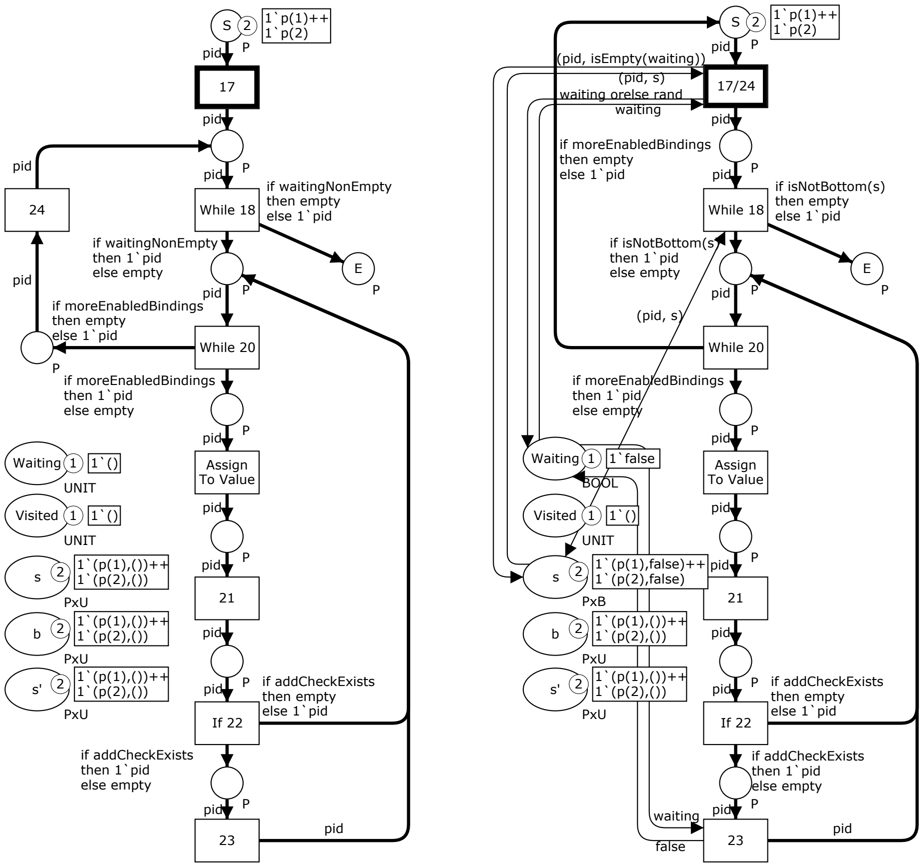


Fig. 6: Control-flow of (left) and a simple refinement of the model (right).

algorithms are correct under certain mutual-exclusion assumptions, and search for such violations. We are also interested in potential dead- and live-locks. Assuming a valid refinement, we can ensure that absence of safety violations in the model guarantees absence in the real program as we can simulate all executions of the algorithm. This includes proving absence of mutual-exclusion violations. We cannot use our approach to ensure the absence of dead-locks, as we deal with over-approximations of the possible interleavings and further restricting the behavior may introduce new dead-locks, but we can still find dead-locks and remove them from the implementation.

We can do state-space analysis of the derived models from Fig. 6, and obtain a state-space with 81 states for the abstract model (left) and a state-space with 130 states for the refined model. As the models do not have any critical regions, they of course have no mutual exclusion violations.

Dead-locks and Live-locks. As all processes have a distinguished start and end-state, we can recognize dead-locks and live-locks in the model. A *dead-lock* is a state without successors (a *dead state*) where not all processes reside on E. Neither of the models in Fig. 6 has dead-locks; the model to the left has exactly one dead state, where all process ids reside on E and the shared and local places retain their initial marking. The model in Fig. 6 (right) also has one dead state, where all process ids reside on E, Waiting is true, s is true for all processes, and all remaining local and shared places have their initial value.

Live-locks are a harder to recognize. We only consider live-locks in the absence of dead-locks. A model has a *strong live-lock* if the dead states of the model do not constitute a *home space*, i.e., if it is not always possible to reach one of the dead states. A strong live-lock in the model does not necessarily imply a live-lock in the original algorithm, but can be used to identify parts of the original program that should be further investigated. None of the models in Fig. 6 have strong live-locks.

A model may have a *weak live-lock* if its state-space has a loop. A loop may also just indicate that a loop may execute an unbounded number of times. Both models in Fig. 6 have loops, but analysis shows that the transition While 20 is *impartial*, i.e., that in any infinite execution it occurs an infinite number of times. This happens if we compute infinitely many successors (the state-space has infinitely many states), and makes sense in our algorithm.

A particular interesting kind of live-lock is a loop reachable from a state where E contains tokens. This means that even after one of the processes have terminated, the amount of work done by another process is unbounded. We have already seen that Fig. 6 (left) exhibits this due to too abstract modeling, i.e., that process p(1) may decide that Waiting is empty initially and terminate, just to have p(2) decide it is non-empty and continue computation. We have seen this is not possible in the original algorithm, which caused us to refine the model to Fig. 6 (right). We would therefore expect that no such live-lock was present in the refined model. Maybe surprisingly, one such does exist. This is seen by having p(1) check Waiting in 17/24, modify Waiting to be empty, and successfully continue. Then, p(2) checks Waiting, notices it is empty and terminates. Now, p(1) continues. This is also possible in Algorithm 4 (left), and even quite likely as the two processes will test Waiting initially, one of them will consume the only element it contains initially, and other processes terminate. This also occurred in reality in our first implementation of a parallel state-space exploration algorithm using Algorithm 4 (left).

To fix this, we notice that the reason p(2) terminates prematurely in the previous example is that it decided to terminate while p(1) can still add new states to Waiting. The idea of an improved algorithm is to ensure that no processes may terminate when others may produce new states. This prompts us to make Algorithm 4 (right). We reuse *calculateStateSpace*, *addCheckExists*, and *pickAndRemoveElement* from Algorithm 4 (left) and define a new procedure for picking, *pickWithCounter* (ll. 5–9) which is used in place of the original *pickAndRemoveElement* (ll. 13 and 21). We use MayAdd as a counter of the number of processes which may add new states to WAITING. We add an additional loop around the previous main loop ensuring we only quit when MAYADD is zero. We inline the call to *pickWithCounter* for the translation and use a mutex around the call to ensure atomicity.

We use the same approach to translate the model to a CPN model. We maintain the abstraction of WAITING and do no abstraction of MAYADD, i.e., we increment and decrement it according to the algorithm. We implement the mutex around the call to *pickWithCounter* as a place with type UNIT containing a single token acting as the mutex. We then obtain the model in Fig. 7. We have not completely flattened it for readability and to keep resemblance with the previous models in Fig. 6. At the left we have the loop in lines 13–21, which mostly corresponds to Fig. 6 (right). The only changes are that aside from renaming transitions to correspond with the line numbers of Algorithm 4 (right), we have added a transition for the new line 20 causing some rerouting of the flow, added a place representing MayAdd, and changed 13/21 (named 17/24 in Fig. 6) to a substitution transition. We have modeled the new outer loop as a separate page shown in Fig. 7 (top/right). We implement the semantics of a repeat/until loop looping over the page in Fig. 7 (left). We share access to MayAdd. The Test transition is explained later. The page corresponding to the substitution transition modeling *pickWithCounter* is shown in Fig. 7 (bottom/right). Again, we share access to a local place (s) and two shared places (MayAdd and Waiting). In all cases, the S and E places of a subpage is in port/socket relationship with the input and output place of the corresponding substitution transition.

We can use state-space analysis to verify that (for this configuration) we do not violate the mutex property in *pickWithCounter*. The state-space for this model contains 399 states and 934 bindings. We can check this explicitly by looking at the total number of tokens on the four unnamed places in Fig. 7 (bottom/right). This is either 0 or 1, showing that never do we have more than one process inside *pickWithCounter*. The Test transition in Fig. 7 is added to easily test whether it is ever possible for a process to execute lines 15–20 while another process has terminated, i.e., whether it is possible for a process to terminate while there is still work to do. The transition requires a token from E, i.e., a process that terminated and that the value of MayAdd is non-zero (this is handled by the *guard* [n<>0], which has not been explained but exactly ensures this). If Test is enabled in any state, it means that a process has terminated while another has more work to do. State-space analysis shows this is not the case. This is also a safety property, so absence of violations in the model implies absence of violations in the original algorithm. We can prove the property without adding Test by searching for a state where E has tokens and Waiting is false or one of the unnamed places below While 14 contains tokens.

We can convince ourselves that the mutex around *pickWithCounter* is necessary by removing it and repeating analysis. We then get a state where Test is enabled, and we can verify the same error is present in the original model (have one process enter *pickWithCounter* and consume the last element of Waiting,

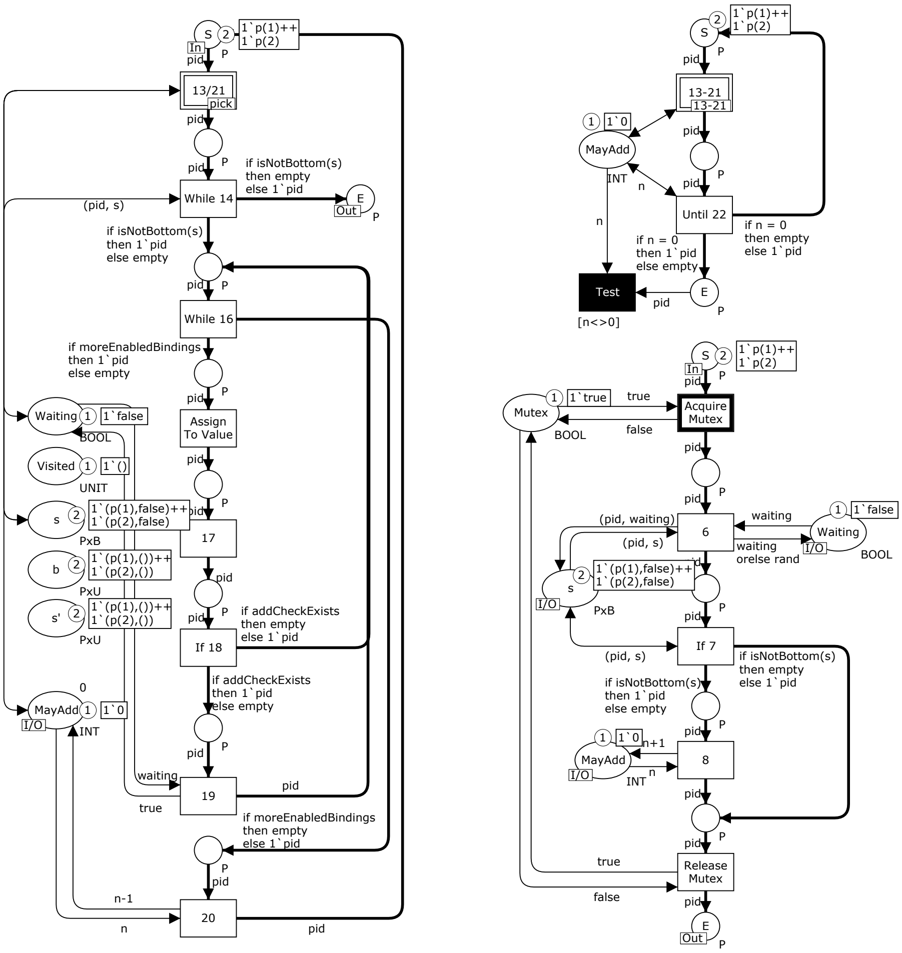


Fig. 7: CPN model of more elaborate state-space algorithm.

then have the other process enter *pickWithCounter*, notice there is no element and return ⊥, and terminate, just to have the first process continue alone).

In addition to the analysis of non-termination, we naturally also check for dead- and live-locks as before and find nothing unexpected. While Algorithm 4 (left) is quite simple and it is probably possible to convince oneself that it is correct without verification, the introduction of MayAdd makes Algorithm 4 (right) sufficiently complicated that correctness is not immediately apparent. We have verified the algorithm with up to 4 processes (yielding 55.709 states), which is the configuration we use in practice. Verification has given us confidence that the algorithm will work with any number of processes. We have also investigated an extended version additionally adding a checkpointing mechanism where all threads are paused while Waiting and Visited are written to disk in a consistent configuration.

We also used the method to verify the implementation of a slightly simplified version of the protocol for operational support developed in [16]. The protocol supports a client which sends a request to an operational support service, which mediates contact to a number of operational support providers. The protocol developed in [16] has support for running all participants on separate machines, i.e., using asynchronous communication, but we are satisfied with an implementation running the operational support server and providers on the same server. We therefore have to send fewer messages, but need to access shared data on the server. We devised a fine-grained locking mechanism using the method devised in this paper and proved that it enforced mutual exclusion and well as caused no dead-locks, increasing our confidence that the implementation works.

5 Conclusion and Future Work

We have sketched an approach for correct implementation of parallel algorithms. The approach allows users to extract a model from an algorithm written in an implementation or abstract language and verify correctness using state-space analysis. The approach also facilitates the generation of skeleton implementation code from the verified model using the approach from [13] as we rely on process-partitioned coloured Petri nets. Finally, we can also combine the two approaches, which facilitates writing an algorithm in an abstract language, extract a model for verification, and then extract a skeleton implementation.

Verification of software by means of models is not new. Code-generation from models have been used in numerous projects. The approach has been most successful for generating specification of hardware from low-level Petri nets and other formalisms to synthesize hardware such as computer chips [9, 17]. The approach has also been applied to high-level Petri nets to generate lower level controllers [14] and more general software [13]. Model extraction was pioneered by FeaVer [6], which made it possible to extract PROMELA models from C code using user-provided abstractions, and Java PathFinder [7] which did the same for Java programs. The approach has successfully been refined using counter-example guided abstraction refinement (CEGAR) [4] which was first implemented by Microsoft SLAM [1], which extracts and automatically refines abstractions from C code for Microsoft Windows device drivers, and refined by BLAST [2]. While the tools for model-extraction support a full development cycle by abstraction refinement and reuse for modified implementations, the idea of combining the two approaches is to the best of our knowledge new. The combination allows some interesting perspectives. The perspective we have focused on in this paper is the ability to write an algorithm in pseudo-code, extract a model from the code, and generate an implementation in a real language. Another perspective is supporting a full cycle as well, where we extract a model from a program, find and fix an error in the model, and emit code that is merged with the original code, supporting a cycle where we do not need to fix problems on the original code but can do so at the model level. The use of coloured Petri nets instead of a low-level formalism allows us to use the real data-types used in the program instead of abstractions, much like how FeaVer allows using C code as part of PROMELA models, but with the added bonus that the operations are a true part of the modeling language rather than an extension that requires some trickery to handle correctly.

The work presented here is only in the initial stages, but looks very promising. We have several ideas for future work. Currently, we have to manually extract the model from patterns. This is tedious and error-prone, and it would be nice to have automatic extraction. Such a translation should implement the patterns included in Fig. 5, but could also use explicit patterns for repeat/until loops and other constructs. Given an implementation of the translation to and from models, we could look at supporting a full development cycle allowing us to update existing code with changes to the model.

An implementation could also implement reduction rules like the one we used to merge lines 17 and 24 in the model in Fig. 6 (right). We can also add simplifications collapsing long traces of unconditional progress not modifying any data to reduce the state-space without removing behavior. For example, in Fig. 7 we can remove transitions Assign To Value and 17, merging the input place of Assign To Value and the output place of 17, to obtain a smaller state-space of 26.909 states for 4 processes, down from 55.709 states. We can also merge the acquisition of a lock with the first regular transition (merging the transitions Acquire Lock with 6 in Fig. 7 (bottom/right)) and merging releases with the last. This reduces the state-space to 14.841 states.

Currently, we provide abstractions manually. Like SLAM, we could easily replay found errors on the original code and provide assistance in the development of refinements, possibly even making them automatically. In our example, replaying the early termination error trace found in Fig. 6 (left) on Algorithm 4 (left) would show that it is not possible for *isEmpty* to return false initially and that it can only change from returning false to returning true if we execute line 23. Even though we might not be able to provide the abstraction refinement in Fig. 6 (right) fully automatically, providing such diagnostics can be very useful for the user for improving the refinement.

Our current method focuses on parallel algorithms with a fixed number of identical processes, but there is nothing in our approach preventing us from extending this to also handle distributed settings with asynchronous communication using buffer places and different kinds of processes; the code generation in [13] even supports that out of the box. While the fixed number of processes used in this paper works well for simple algorithms, more advanced algorithms may need to spawn processes. Nothing in our approach inherently forbids this, but the code generation in [13] does not support this out of the box. We believe that it should be quite easy to devise a construction for starting new processes and adapt the code generation to handle this.

One thing our approach does not support very well at the moment is intra-procedure calls. We can currently simulate this in simple cases by inlining procedure calls, but this is not possible when using recursion. One way to fix it is to view a recursive call as starting a new process for executing the child and waiting for the result. If we support different kinds of processes, communication between processes, and dynamic instantiation of processes, this should be easy to add.

References

1. T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proc. of POPL'02*, pages 1–3. ACM Press, 2002.
2. D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker BLAST: Applications to Software Engineering. *STTT*, 7(5):505–525, 2007.
3. J. Billington, M.C. Wilbur-Ham, and M.Y. Bearman. Automated protocol Verification. In *Proc. of IFIP WG 6.1 5th International Workshop on Protocol Specification, Testing, and Verification*, pages 59–70. Elsevier, 1985.
4. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. *J. ACM*, 50:752–794, 2003.
5. K.L. Espensen, M.K. Kjeldsen, and L.M. Kristensen. Modelling and Initial Validation of the DYMO Routing Protocol for Mobile Ad-Hoc Networks. In *Proc. of ATPN*, volume 5062 of *LNCS*, pages 152–170. Springer, 2008.
6. The FeaVer Feature Verification System webpage. Online: `cm.bell-labs.com/cm/cs/what/feaver/`.
7. K. Havelund and T. Presburger. Model Checking Java Programs Using Java PathFinder. *STTT*, 2(4):366–381, 2000.
8. G.J. Holzmann. *The SPIN Model Checker.* Addison-Wesley, 2003.
9. IEEE Standard System C Language Reference Manual. IEEE-1666.
10. K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems.* Springer, 2009.
11. L.M. Kristensen and K. Jensen. Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad-hoc Networks. In *Integration of Software Specification Techniques for Application in Engineering*, volume 3147 of *LNCS*, pages 248–269. Springer, 2004.
12. L.M. Kristensen, J.B. Jørgensen, and K. Jensen. Application of Coloured Petri Nets in System Development. In *Proc. of 4th Advanced Course on Petri Nets*, number 3098 in LNCS, pages 626–685. Springer, 2004.
13. L.M. Kristensen and M. Westergaard. Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept. In *Proc. of FMICS'10*, LNCS, pages 215–230. Springer, 2010.
14. J.L. Rasmussen and M. Singh. Designing a Security System by Means of Coloured Petri Nets. In *Proc. ATPN'96*, volume 1091 of *LNCS*, pages 400–419. Springer, 1996.
15. W.M.P. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems.* MIT Press, 2002.
16. M. Westergaard and F.M. Maggi. Modelling and Verification of a Protocol for Operational Support using Coloured Petri Nets. In *Proc. of ATPN*, LNCS. Springer, 2011.
17. A. Yakovlev, L. Gomes, and L. Lavagno. *Hardware Design and Petri Nets.* Kluwer Academic Publishers, 2000.

Bounded Model Checking Approaches for Verification of Distributed Time Petri Nets*

Artur Męski[1,2], Wojciech Penczek[2,3], Agata Półrola[1], Bożena Woźna-Szcześniak[4], and Andrzej Zbrzezny[4]

[1] University of Łódź, FMCS, Banacha 22, 90-238 Łódź, Poland
polrola@math.uni.lodz.pl
[2] Institute of Computer Science, PAS, Ordona 21, 01-237 Warsaw, Poland
{meski,penczek}@ipipan.waw.pl
[3] University of Natural Sciences and Humanities, Institute of Informatics,
3 Maja 54, 08-110 Siedlce, Poland
[4] Jan Długosz University, IMCS, Armii Krajowej 13/15, 42-200 Częstochowa, Poland
{b.wozna,a.zbrzezny}@ajd.czest.pl

Abstract. We consider two symbolic approaches to bounded model checking (BMC) of distributed time Petri nets (DTPNs). We focus on the properties expressed in Linear Temporal Logic without the neXt-time operator ($\mathrm{LTL}_{-\mathrm{X}}$) and the existential fragment of Computation Tree Logic without the neXt-time operator ($\mathrm{ECTL}_{-\mathrm{X}}$). We give a translation of BMC to SAT and describe a BDD-based BMC for both $\mathrm{LTL}_{-\mathrm{X}}$ and $\mathrm{ECTL}_{-\mathrm{X}}$. The two translations have been implemented, tested, and compared with each other on two standard benchmarks. Our experimental results reveal the advantages and disadvantages of both the approaches.

1 Introduction

Verification of time dependent systems is a very active field of research. Many efficient approaches have been put forward for the verification of timed automata [1] and time Petri nets [22] by means of model checking [12, 26]. However, the *state explosion* still remains the main problem to deal with while verifying a timed system by searching through its state space, which in most cases is very large due to infinity of the dense time domain. Furthermore, the size of the state space is likely to grow exponentially in the number of the concurrent system components. Symbolic model checking techniques [21] can be used to overcome the above problem. These exploit various kinds of binary decision diagrams to represent the model [24] or are based on a translation to a propositional satisfiability problem.

Bounded model checking (BMC) is an efficient verification method using a model truncated up to a specific depth only. In turn, SAT-based BMC verification consists in translating a model checking problem solvable on a fraction of a

* Partly supported by the Polish Ministry of Science and Higher Education under the grant No. N N206 258035.

model into a test of propositional satisfiability, which is then performed using a SAT-solver [28]. The method has been successfully applied to verification of both timed and untimed systems [3–5, 33]. Alternatively, one can use binary decision diagrams to represent a truncated model and to perform BDD-based verification [2, 13].

In this paper we investigate bounded model checking (BMC) approaches to verification of Distributed Time Petri Nets with discrete semantics, based on both SAT and BDD translations. There are several decisions taken that aim at making the verification of TPNs as efficient as possible. Below, we discuss them in detail to motivate clearly our approach. First of all, we believe that BMC is one of the main practical approaches, which can be used in case of dealing with huge or infinite state spaces. We motivate this point of view by comparing our experimental results with these of Tina, which operate on full state spaces. Clearly, BMC is restricted to verifying existential properties only, but it allows for tackling bounded models of large systems, whereas other approaches suffer from lack of memory.

Our second choice consists in dealing with distributed TPNs rather than with just 1-safe TPNs. The reason is that a representation of a global state contains only one clock for each process rather than one clock for each transition, which makes the encoding and the verification much more efficient. Another choice is related to the semantics. In this paper we investigate discrete semantics as we believe that in this case model checking is again more efficient. However, independently we are working on extending our approach to the dense semantics as we are aware that this is also a very interesting issue. Since there are several discrete semantics, we consider for each translation these which can be applied.

As to the temporal properties, we start with defining $\mathrm{CTL}^*_{-\mathrm{X}}$, but restrict ourselves two its two subsets $\mathrm{CTL}_{-\mathrm{X}}$ and $\mathrm{LTL}_{-\mathrm{X}}$, as these sublanguages allow for optimising the translations to SAT and BDDs. The languages do not contain the next step operator X as we are dealing with time systems, in which, for some discrete semantics, the next step may be not definable.

Next, we need to motivate our translations to SAT and BDDs. We are aware of the fact that there has been a tremendous speed-up due to applying the saturation technique [15] when performing decision diagram based verification. Moreover, the saturation combined with BMC was presented in [34], however only reachability checking was considered. Still, we believe, in most cases, BMC approach to BDD-based verification can be viewed as an alternative way of avoiding the BDD peak size when using BFS. In case of SAT we exploit the most efficient translations known for $\mathrm{ECTL}_{-\mathrm{X}}$ and $\mathrm{ELTL}_{-\mathrm{X}}$.

The above discussion motivates all the choices made in our paper and leads us to the main result, which is offering and comparing two symbolic BMC approaches for DTPNs. We show that for existential properties our BMC is much more efficient than Tina. We also compare efficiency of BMC depending on whether it is based on a translation to SAT or to BDD.

The main contribution of this paper is thus the combination of the three issues, as BMC has been studied, with both BDDs and, especially, SAT, but mostly for standard untimed models, while discrete time Petri nets have been studied with BDDs and extensions (e.g., [31]), but not for BMC.

To our best knowledge, no BMC method supporting $\mathrm{ELTL}_{-\mathrm{X}}$ and $\mathrm{ECTL}_{-\mathrm{X}}$ for time Petri nets has been defined so far, although some solutions for untimed Petri nets exist [27, 16]. Symbolic model checking has been investigated in many papers [2, 5]. Verification of CTL properties based on BDDs was introduced in [9]. In [27] SAT-based BMC for the existential fragment of CTL was described and implemented for elementary net systems. Verification methods using BDD-based BMC were studied in [10, 13] for simple invariant properties, in [23] for CTL over elementary nets systems, and in [19] for CTL extended with an epistemic component over multi-agent systems. On the other hand, verification of temporal properties for time Petri nets was a subject of intensive research of the teams of H. Boucheneb and O.H. Roux [6, 7, 20].

The rest of the paper is organised as follows. Section 2 presents logics $\mathrm{LTL}_{-\mathrm{X}}$ and $\mathrm{CTL}_{-\mathrm{X}}$. Section 3 introduces time Petri nets. SAT-based BMC for $\mathrm{ELTL}_{-\mathrm{X}}$ and $\mathrm{ECTL}_{-\mathrm{X}}$ is described in Section 4, whereas BDD-based BMC for these logics is in Section 5. Sections 6 and 7 contain respectively experimental results and concluding remarks.

2 Temporal Logics $\mathrm{LTL}_{-\mathrm{X}}$ and $\mathrm{CTL}_{-\mathrm{X}}$

We start with defining the logic $\mathrm{CTL}^*_{-\mathrm{X}}$, which can express both linear- and branching-time properties. Then, we introduce variants of linear time temporal logic ($\mathrm{LTL}_{-\mathrm{X}}$) as well as of branching time temporal logic ($\mathrm{CTL}_{-\mathrm{X}}$) as sublogics of $\mathrm{CTL}^*_{-\mathrm{X}}$. All the considered logics do not contain the next step operator X, which is reflected in their acronyms by $-$X.

Let PV be a set of propositional variables such that $\{true, false\} \subseteq PV$, and $\wp \in PV$. The language of $\mathrm{CTL}^*_{-\mathrm{X}}$ is given as the set of all the state formulae φ_s (interpreted at states of a model), defined using path formulae φ_p (interpreted along paths of a model), by the following grammar:

$$\varphi_s := \wp \mid \neg\varphi_s \mid \varphi_s \vee \varphi_s \mid \mathrm{A}\varphi_p$$
$$\varphi_p := \varphi_s \mid \neg\varphi_p \mid \varphi_p \vee \varphi_p \mid \varphi_p \mathrm{U} \varphi_p \mid \varphi_p \mathrm{R} \varphi_p$$

In the above A ('for All paths') is a path quantifier, whereas U ('Until'), and R ('Release') are state operators. Further, the following standard abbreviations are used in writing $\mathrm{CTL}^*_{-\mathrm{X}}$ formulae: $\varphi_s \wedge \varphi_s \stackrel{def}{=} \neg(\neg\varphi_s \vee \neg\varphi_s)$, $\varphi_p \wedge \varphi_p \stackrel{def}{=} \neg(\neg\varphi_p \vee \neg\varphi_p)$, $\mathrm{E}\varphi_p \stackrel{def}{=} \neg\mathrm{A}(\neg\varphi_p)$, $\mathrm{G}\varphi_p \stackrel{def}{=} false\mathrm{R}\varphi_p$, and $\mathrm{F}\varphi_p \stackrel{def}{=} true\mathrm{U}\varphi_p$.

Next, we define several sublogics of $\mathrm{CTL}^*_{-\mathrm{X}}$ including variants of $\mathrm{LTL}_{-\mathrm{X}}$ as well as of $\mathrm{CTL}_{-\mathrm{X}}$. Although a standard model for $\mathrm{LTL}_{-\mathrm{X}}$ is a path, for verification aims the logic is typically interpreted over all the paths of a Kripke model. So, two semantics are possible depending on whether a formula holds at all the paths or at some path of a model. Since we need to distinguish between these two semantics (in order to specify counterexamples), we find it useful to do it already at the level of the language by defining the universal ($\mathrm{ALTL}_{-\mathrm{X}}$)

and the existential ($\mathrm{ELTL}_{-\mathrm{X}}$) versions of the logic. In the literature on the verification of linear time properties, if this distinction is not necessary, then $\mathrm{ALTL}_{-\mathrm{X}}$ is typically called $\mathrm{LTL}_{-\mathrm{X}}$.

$\mathrm{ALTL}_{-\mathrm{X}}$ ($\mathrm{ELTL}_{-\mathrm{X}}$) is the fragment of $\mathrm{CTL}^*_{-\mathrm{X}}$ in which only the formulae of the form $\mathrm{A}\varphi_p$ ($\mathrm{E}\varphi_p$, respectively) are allowed, where φ_p is a path formula which does not contain the path quantifiers A, E.

$\mathrm{CTL}_{-\mathrm{X}}$ is the fragment of $\mathrm{CTL}^*_{-\mathrm{X}}$ in which the syntax of path formulae is restricted such that each state operators must be preceded by a path quantifier (i.e., the modalities A, E, U, R can only appear paired in the combinations AU, EU, AR, ER).

$\mathrm{ACTL}_{-\mathrm{X}}$ ($\mathrm{ECTL}_{-\mathrm{X}}$) is the fragment of $\mathrm{CTL}_{-\mathrm{X}}$ such that the formulae are restricted to the positive boolean combinations of $\mathrm{A}(\varphi \mathrm{U} \psi)$ and $\mathrm{A}(\varphi \mathrm{R} \psi)$ ($\mathrm{E}(\varphi \mathrm{U} \psi)$ and $\mathrm{E}(\varphi \mathrm{R} \psi)$, respectively). Negation can be applied to propositions only.

A *model* for $\mathrm{CTL}^*_{-\mathrm{X}}$ is a Kripke structure $M = (L, S, s^0, \rightarrow, V)$, where L is a set of labels, S is a set of states, $s^0 \in S$ is the initial state, $\rightarrow \ \subseteq S \times L \times S$ is a total successor relation (i.e., $(\forall s \in S)(\exists s' \in S)(s \rightarrow s')$), and $V : S \longrightarrow 2^{PV}$ is a valuation function.

In our paper we assume the standard semantics of $\mathrm{CTL}^*_{-\mathrm{X}}$which can be found in several papers, among others in [11, 12], so we do not deliver it here. Moreover, we assume that a $\mathrm{CTL}^*_{-\mathrm{X}}$ formula φ is true in the model M (denoted by $M \models \varphi$) iff φ is true at the initial state of the model, i.e., $M, s^0 \models \varphi$.

3 Time Petri Nets

Let $\mathbb{N}$ denote the set of natural numbers. We start with a definition of time Petri nets:

Definition 1. *A time Petri net (TPN) is a tuple $\mathcal{N} = (P, T, F, m^0, Eft, Lft)$, where $P = \{p_1, \ldots, p_{n_P}\}$ is a finite set of* places, *$T = \{t_1, \ldots, t_{n_T}\}$ is a finite set of* transitions, *$F \subseteq (P \times T) \cup (T \times P)$ is the* flow relation, *$m^0 \subseteq P$ is the* initial marking *of $\mathcal{N}$, and $Eft : T \to \mathbb{N}$, $Lft : T \to \mathbb{N} \cup \{\infty\}$ are functions describing the* earliest *and the* latest firing time *of the transition; where for each $t \in T$ we have $Eft(t) \leq Lft(t)$.*

For a transition $t \in T$ we define its *preset* $\bullet t = \{p \in P \mid (p, t) \in F\}$ and *postset* $t\bullet = \{p \in P \mid (t, p) \in F\}$, and consider only the nets such that for each transition the preset and the postset are nonempty. We need also the following notations and definitions:

- a *marking* of $\mathcal{N}$ is any subset $m \subseteq P$;
- a transition $t \in T$ is called *enabled* at m ($m[t\rangle$ for short) if $\bullet t \subseteq m$ and $t \bullet \cap (m \setminus \bullet t) = \emptyset$; and *leads from m to m'*, if it is enabled at m, and $m' = (m \setminus \bullet t) \cup t\bullet$. The marking m' is denoted by $m[t\rangle$ as well, if this does not lead to misunderstanding;
- $en(m) = \{t \in T \mid m[t\rangle\}$ is the set of all the transitions enabled at the marking m;

- a marking $m \subseteq P$ is *reachable* if there exists a sequence of transitions $t_1, \ldots, t_l \in T$ and a sequence of markings $m_0, \ldots, m_l$ such that $m_0 = m^0$, $m_l = m$, and for each $i \in \{1, \ldots, l\}$ $t_i \in en(m_{i-1})$ and $m_i = m_{i-1}[t_i\rangle$;
- a marking m *concurrently enables* two transitions $t, t' \in T$ if $t \in en(m)$ and $t' \in en(m \setminus \bullet t)$;
- a net is *sequential* if no reachable marking of $\mathcal{N}$ concurrently enables two transitions.

It should be mentioned that the time Petri nets defined as above are often called *1-safe* in the literature. In this work we consider a subclass of TPNs – *distributed time Petri nets* (DTPNs) [26]:

Definition 2. *Let $\mathfrak{I} = \{\mathfrak{i}_1, \ldots, \mathfrak{i}_n\}$ be a finite ordered set of indices, and let $\mathfrak{N} = \{N_\mathfrak{i} = (P_\mathfrak{i}, T_\mathfrak{i}, F_\mathfrak{i}, m^0_\mathfrak{i}, Eft_\mathfrak{i}, Lft_\mathfrak{i}) \mid \mathfrak{i} \in \mathfrak{I}\}$ be a family of 1-safe, sequential time Petri nets (called* processes*), indexed with $\mathfrak{I}$, with the pairwise disjoint sets $P_\mathfrak{i}$ of places, and satisfying the condition $(\forall \mathfrak{i}_1, \mathfrak{i}_2 \in \mathfrak{I})(\forall t \in T_{\mathfrak{i}_1} \cap T_{\mathfrak{i}_2})$ $(Eft_{\mathfrak{i}_1}(t) = Eft_{\mathfrak{i}_2}(t) \ \wedge \ Lft_{\mathfrak{i}_1}(t) = Lft_{\mathfrak{i}_2}(t))$. A* distributed time Petri net *$\mathcal{N} = (P, T, F, m^0, Eft, Lft)$ is the union of the processes $N_\mathfrak{i}$, i.e., $P = \bigcup_{\mathfrak{i}\in\mathfrak{I}} P_\mathfrak{i}$, $T = \bigcup_{\mathfrak{i}\in\mathfrak{I}} T_\mathfrak{i}$, $F = \bigcup_{\mathfrak{i}\in\mathfrak{I}} F_\mathfrak{i}$, $m^0 = \bigcup_{\mathfrak{i}\in\mathfrak{I}} m^0_\mathfrak{i}$, $Eft = \bigcup_{\mathfrak{i}\in\mathfrak{I}} Eft_\mathfrak{i}$, and $Lft = \bigcup_{\mathfrak{i}\in\mathfrak{I}} Lft_\mathfrak{i}$.*

Notice that the function $Eft_{\mathfrak{i}_1}$ ($Lft_{\mathfrak{i}_1}$) coincides with $Eft_{\mathfrak{i}_2}$ ($Lft_{\mathfrak{i}_2}$, resp.) for the joint transitions of each two processes $\mathfrak{i}_1$ and $\mathfrak{i}_2$. The interpretation of such a system is a collection of sequential, nondeterministic processes with communication capabilities (via joint transitions).

In what follows, we consider DTPNs only, assume that their initial markings contain exactly one place of each of the processes of the net, and that all their processes are *state machines* (i.e., for each $\mathfrak{i} \in \mathfrak{I}$ and each $t \in T_\mathfrak{i}$, it holds $|\bullet t| = |t \bullet| = 1$). This implies that in any marking of $\mathcal{N}$ there is exactly one place of each process. It is important to mention that a large class of distributed nets can be decomposed to satisfy the above requirement [18]. Moreover, for $t \in T$ we define $\mathcal{IV}(t) = \{\mathfrak{i} \in \mathfrak{I} \mid \bullet t \cap P_\mathfrak{i} \neq \emptyset\}$, and say that a process $N_\mathfrak{i}$ is *involved in a transition t* iff $\mathfrak{i} \in \mathcal{IV}(t)$.

3.1 Concrete State Spaces and Models

The current state of the net is given by its marking and the time passed since each of the enabled transitions became enabled (which influences the future behaviour of the net). In our work we assume a discrete-time semantics of DTPNs, i.e., consider integer time passings only (cf. [26]). Thus, a *concrete state* σ of a distributed TPN $\mathcal{N}$ can be defined as an ordered pair $(m, clock)$, where m is a marking, and $clock : \mathfrak{I} \to \mathbb{N}$ is a function which for each index $\mathfrak{i}$ of a process of $\mathcal{N}$ gives the time elapsed since the marked place of this process became marked most recently [29]. The set of all the concrete states is denoted by Σ. The initial state of $\mathcal{N}$ is $\sigma^0 = (m^0, clock^0)$, where m^0 is the initial marking, and $clock^0(\mathfrak{i}) = 0$ for each $\mathfrak{i} \in \mathfrak{I}$.

For $\delta \in \mathbb{N}$, let $clock + \delta$ denote the function given by $(clock + \delta)(\mathfrak{i}) = clock(\mathfrak{i}) + \delta$, and let $(m, clock) + \delta$ denote $(m, clock + \delta)$. The states of $\mathcal{N}$ can change when the time passes or a transition fires. In consequence, we introduce a labelled timed consecution relation $\rightarrow_c \subseteq \Sigma \times (T \cup \mathbb{N}) \times \Sigma$ given as follows:

- In a state $\sigma = (m, clock)$ a time $\delta \in \mathbb{N}$ can pass leading to a new state $\sigma' = (m, clock + \delta)$ (denoted $\sigma \xrightarrow{\delta}_c \sigma'$) iff for each $t \in en(m)$ there exists $\mathfrak{i} \in \mathcal{IV}(t)$ such that $clock(\mathfrak{i}) + \delta \leq Lft(t)$ (*time-successor relation*);
- In a state $\sigma = (m, clock)$ a transition $t \in T$ can fire leading to a new state $\sigma' = (m', clock')$ (denoted $\sigma \xrightarrow{t}_c \sigma'$) if $t \in en(m)$, for each $\mathfrak{i} \in \mathcal{IV}(t)$ we have $clock(\mathfrak{i}) \geq Eft(t)$, and there is $\mathfrak{i} \in \mathcal{IV}(t)$ such that $clock(\mathfrak{i}) \leq Lft(t)$. Then, $m' = m[t\rangle$,and for all $\mathfrak{i} \in \mathfrak{I}$ we have $clock'(\mathfrak{i}) = 0$ if $\mathfrak{i} \in \mathcal{IV}(t)$, and $clock'(\mathfrak{i}) = clock(\mathfrak{i})$ otherwise (*action-successor relation*).

Intuitively, the time-successor relation does not change the marking of the net, but increases the clocks of all the processes, provided that no enabled transition becomes disabled by passage of time (i.e., for each $t \in en(m)$ the clock of at least one process involved in the transition does not exceed $Lft(t)$). Firing of a transition t takes no time – the action-successor relation does not increase the clocks, but only sets to zero the clocks of the involved processes (note that each of these processes contains exactly one input and one output place of t, as the processes are state machines); and is allowed provided that t is enabled, the clocks of all the involved processes are greater than $Eft(t)$, and there is at least one such process whose clock does not exceed $Lft(t)$.

We define a *timed run* of $\mathcal{N}$ starting at a state $\sigma_0 \in \Sigma$ (*σ_0-run*) as a maximal sequence of concrete states, transitions, and time passings $\rho = \sigma_0 \xrightarrow{a_0}_c \sigma_1 \xrightarrow{a_1}_c \sigma_2 \xrightarrow{a_2}_c \ldots$, where $\sigma_i \in \Sigma$ and $a_i \in T \cup \mathbb{N}$ for all $i \in \mathbb{N}$. An *alternating run* is a timed run in which $a_i \in \mathbb{N}$ when i is even, and $a_i \in T$ when i is odd. A *non-alternating run* is a timed run with $a_i \in T \cup (\mathbb{N} \setminus \{0\})$ for all i. Given a set of propositional variables PV, we introduce a *valuation function* $V_c : \Sigma \rightarrow 2^{PV}$ which assigns the same propositions to the states with the same markings. We assume the set PV to be such that each $q \in PV$ corresponds to exactly one place $p \in P$, and use the same names for the propositions and the places. The function V_c is defined by $p \in V_c(\sigma) \Leftrightarrow p \in m$ for each $\sigma = (m, \cdot)$. The structure $M_c(\mathcal{N}) = (T \cup \mathbb{N}, \Sigma, \sigma^0, \rightarrow_c, V_c)$ is called a *concrete* (*discrete-timed*) *model of* $\mathcal{N}$.

3.2 A Model for $\mathbf{CTL}^*_{-\mathbf{X}}$Verification of DTPNs

The concrete model $M_c(\mathcal{N}) = (T \cup \mathbb{N}, \Sigma, \sigma^0, \rightarrow_c, V_c)$ for a DTPN $\mathcal{N}$ defined in Section 3 involves timed steps of arbitrary length. However, it can be proven that without loss of generality one can consider a model with a restricted set of timed labels, and of restricted values of the *clock* function. Let $c_{max}(\mathcal{N})$ denote the greatest finite value of Eft and Lft of the net $\mathcal{N}$, $\mathbf{c_{m1}}$ denote the value $c_{max}(\mathcal{N}) + 1$, and $\mathbb{C}_{\mathcal{N}}$ be the set of natural numbers from the interval $[0, \mathbf{c_{m1}}]$. Next, for a function $f : T \rightarrow \mathbb{N}$ and $a \in \mathbb{N}$, let $f|_a$ denote the function given by $f|_a(t) = f(t)$ if $f(t) \leq a$, and $f|_a(t) = a$ otherwise. Let $clock_s : \mathfrak{I} \rightarrow \mathbb{C}_{\mathcal{N}}$ denote the function which for each index $\mathfrak{i}$ of a process of $\mathcal{N}$ gives the time either elapsed since the marked place of this process became marked most recently, or "frozen" on the value $\mathbf{c_{m1}}$ if the time elapsed since the marked place becomes marked exceeded $c_{max}(\mathcal{N})$. Let $\sigma|_{\mathbf{c_{m1}}}$, for $\sigma = (m, clock) \in \Sigma$, be the state $(m, clock_s)$ with $clock_s = clock|_{\mathbf{c_{m1}}}$. Moreover, for $\delta \in \mathbb{N}$, let $clock_s \oplus \delta$ denote the function given by $(clock_s \oplus \delta)(\mathfrak{i}) = clock_s(\mathfrak{i}) + \delta$ if $clock_s(\mathfrak{i}) + \delta \leq \mathbf{c_{m1}}$, and $(clock_s \oplus \delta)(\mathfrak{i}) = \mathbf{c_{m1}}$ otherwise. The *reduced (discrete-timed) model* for DTPN $\mathcal{N}$ is defined as follows: $\widehat{M_c}(\mathcal{N}) = (T \cup \mathbb{C}_{\mathcal{N}}, \Sigma_s, \sigma^0, \rightarrow_s, V_s)$, where $\Sigma_s = \{\sigma|_{\mathbf{c_{m1}}} \mid \sigma \in \Sigma\}$, V_s is given by $V_s(\sigma|_{\mathbf{c_{m1}}}) = V_c(\sigma)$, and the relation $\rightarrow_s \subseteq \Sigma_s \times (T \cup \mathbb{C}_{\mathcal{N}}) \times \Sigma_s$ is defined by

- in a state $\sigma_s = (m, clock_s)$ a time $\delta \in \mathbb{C}_{\mathcal{N}}$ can pass leading to a new state $\sigma'_s = (m, clock_s \oplus \delta)$ (denoted $\sigma_s \xrightarrow{\delta}_s \sigma'_s$) iff for each $t \in en(m)$ there exists $\mathfrak{i} \in I\mathcal{V}(t)$ such that $clock_s(\mathfrak{i}) \oplus \delta \leq Lft(t)$,
- a transition $t \in T$ can fire in a state $\sigma_s = \sigma|_{\mathbf{c_{m1}}}$ leading to a state σ'_s (denoted $\sigma_s \xrightarrow{t}_s \sigma'_s$) iff $\sigma \xrightarrow{t}_c \sigma'$ for some $\sigma' \in \Sigma$ s.t. $\sigma'_s = \sigma'|_{\mathbf{c_{m1}}}$.

In order to show that $\widehat{M_c}(\mathcal{N})$ can replace $M_c(\mathcal{N})$ in $\mathrm{CTL}^*_{-\mathrm{X}}$ verification we shall prove the following lemma:

Lemma 1. *For a given DTPN $\mathcal{N}$ the models $M_c(\mathcal{N}) = (T \cup \mathbb{N}, \Sigma, \sigma^0, \rightarrow_c, V_c)$ and $\widehat{M_c}(\mathcal{N}) = (T \cup \mathbb{C}_{\mathcal{N}}, \Sigma_s, \sigma^0, \rightarrow_s, V_s)$ are bisimulation equivalent.*

The proof can be found in the appendix. In the proof we use an "intermediate" model $\widetilde{M_c}(\mathcal{N}) = (T \cup \mathbb{C}_{\mathcal{N}}, \Sigma, \sigma^0, \rightarrow_r, V_c)$ with $\rightarrow_r \subseteq \Sigma \times (T \cup \mathbb{C}_{\mathcal{N}}) \times \Sigma$ given by

- in a state $\sigma = (m, clock)$ a time $\delta \in \mathbb{C}_{\mathcal{N}}$ can pass leading to a new state $\sigma' = (m, clock + \delta)$ (denoted $\sigma \xrightarrow{\delta}_r \sigma'$) iff for each $t \in en(m)$ there exists $\mathfrak{i} \in I\mathcal{V}(t)$ such that $clock(\mathfrak{i}) + \delta \leq Lft(t)$,
- a transition $t \in T$ can fire in a state σ leading to a state σ' ($\sigma \xrightarrow{t}_r \sigma'$) iff $\sigma \xrightarrow{t}_c \sigma'$,

(i.e., the model which differs from $\widehat{M_c}(\mathcal{N})$ in such a way that the values of the *clock* function are not restricted to $\mathbf{c_{m1}}$) which is bisimulation equivalent to $M_c(\mathcal{N})$. Further, we define the following equivalence relation, which is used in the next section to define a SAT-based BMC method.

Definition 3. *Let $\sigma = (m, clock)$ and $\sigma' = (m', clock')$ be two states of a DTPN $\mathcal{N}$ ($\sigma, \sigma' \in \Sigma$). The states σ, σ' are* $\star$-equivalent *(denoted $\sigma \simeq_\star \sigma'$) iff $m = m'$ and $\forall_{t \in en(m)}[(\min_{\mathfrak{i} \in I\mathcal{V}(t)} clock(i) = \min_{i \in I\mathcal{V}(t)} clock'(i) \wedge \min_{i \in I\mathcal{V}(t)} clock(\mathfrak{i}) \leq c_{max}(\mathcal{N})) \vee (\min_{\mathfrak{i} \in I\mathcal{V}(t)} clock(i) > c_{max}(\mathcal{N}) \wedge \min_{\mathfrak{i} \in I\mathcal{V}(t)} clock'(i) > c_{max}(\mathcal{N}))]$.*

The following lemma shows that the equivalence preserves the behaviours of the net. Its proof can be found in the appendix.

Lemma 2. *Let $\sigma, \sigma' \in \Sigma$ be $\star$-equivalent. Thus, for any $l \in T \cup \mathbb{N}$ we have:*

- *if $\sigma \xrightarrow{l}_c \sigma_1$ for some $\sigma_1 \in \Sigma$ then there is $\sigma'_1 \in \Sigma$ s.t. $\sigma' \xrightarrow{l}_c \sigma'_1$ and $\sigma_1 \simeq_\star \sigma'_1$,*
- *if $\sigma' \xrightarrow{l}_c \sigma'_1$ for some $\sigma'_1 \in \Sigma$ then there is $\sigma_1 \in \Sigma$ s.t. $\sigma \xrightarrow{l}_c \sigma_1$ and $\sigma'_1 \simeq_\star \sigma_1$.*

4 SAT-Based BMC for $\mathrm{ELTL}_{-\mathrm{X}}$ and $\mathrm{ECTL}_{-\mathrm{X}}$

BMC is a verification technique whose main idea consists in considering a model truncated up to a specific depth. Thus, BMC is mostly used to find counterexamples for the properties expressed in "universal" logics (in our case $\mathrm{ACTL}_{-\mathrm{X}}$ and

$\mathrm{ALTL_{-X}}$), or to prove that properties expressed in "existential" logics ($\mathrm{ECTL_{-X}}$, $\mathrm{ELTL_{-X}}$) hold.

The BMC method used in our paper is based on the BMC method for the existential fragment of $\mathrm{CTL}^*_{-\mathrm{X}}$ ($\mathrm{ECTL}^*_{-\mathrm{X}}$) [32], and on an improved BMC translation for the $\mathrm{ECTL_{-X}}$ fragment [35]. In particular, in the paper we adapt the BMC techniques mentioned above to the DTPN setting. Let $\widetilde{M}_c(\mathcal{N}) = (T \cup \mathbb{C}_{\mathcal{N}}, \Sigma, \sigma^0, \to_r, V_c)$ be a model for a given DTPN $\mathcal{N} = (P, T, F, m^0, Eft, Lft)$, and φ an $\mathrm{ECTL_{-X}}$ or $\mathrm{ELTL_{-X}}$ formula describing an undesired property. To show that φ is true in $\widetilde{M}_c(\mathcal{N})$, it is enough to prove that φ holds in a fragment (submodel) M' of $\widetilde{M}$. Thus, we start by taking a submodel M' of the model $\widetilde{M}_c(\mathcal{N})$ that consists of the finite prefixes of paths from $\widetilde{M}_c(\mathcal{N})$ restricted by a bound $k \in \mathbb{N}$ – traditionally called k-paths. The number of k-paths in M' depends on the checked formula φ, and it is specified by a value of a function $f_k : \mathcal{F}_{\mathrm{ECTL}^*_{-\mathrm{X}}} \to \mathbb{N}$ defined by: for $\wp \in PV$, $f_k(\wp) = f_k(\neg\wp) = 0$, $f_k(\varphi \wedge \psi) = f_k(\varphi) + f_k(\psi)$, $f_k(\varphi \vee \psi) = \max\{f_k(\varphi), f_k(\psi)\}$, $f_k(E\varphi) = f_k(\varphi) + 1$, $f_k(\varphi \mathrm{U} \psi) = k \cdot f_k(\varphi) + f_k(\psi)$, $f_k(\varphi \mathrm{R} \psi) = (k+1) \cdot f_k(\psi) + f_k(\varphi)$. Next, we translate the problem of checking whether the M' is a model for φ to the problem of checking the satisfiability of the following propositional formula:

$$[\widetilde{M}_c(\mathcal{N}), \varphi]_k := [\widetilde{M}_c(\mathcal{N})^{\varphi,\sigma^0}]_k \wedge [\varphi]_{M'} \tag{1}$$

The first conjunct of Formula 1 represents all the submodels M' of $\widetilde{M}_c(\mathcal{N})$ that consists of $f_k(\varphi)$ k-paths, and the second a number of constraints that must be satisfied on these submodels for φ to be satisfied. Once this translation is defined, satisfiability of an $\mathrm{ECTL_{-X}}$ or $\mathrm{ELTL_{-X}}$ formula can be tested with a SAT-solver.

How to define the formula $[\widetilde{M}_c(\mathcal{N})^{\varphi,\sigma^0}]_k$ in the DTPN settings we show in the next subsection. Note however that for a given DTPN $\mathcal{N}$, the formula $[\widetilde{M}_c(\mathcal{N})^{\varphi,\sigma^0}]_k$ can be implemented either using the model $\widetilde{M}_c(\mathcal{N})$ or using $\widehat{M}_c(\mathcal{N})$. We have chosen $\widetilde{M}_c(\mathcal{N})$ in order to simplify the implementation. It should be explained that although in $\widetilde{M}_c(\mathcal{N})$ there is no upper bound on the values of clocks, restricting the lengths of the time steps allows to bound the values of clocks on k-paths by a value depending on k and $c_{max}(\mathcal{N})$. The definition of the formula $[\varphi]_{M'}$ depends on whether φ is in $\mathrm{ECTL_{-X}}$ or in $\mathrm{ELTL_{-X}}$, and whether considered k-paths are, or are not *loops*; a k-path $\pi_k = (\sigma_0, \sigma_1, \ldots, \sigma_k)$ is called a (k,l)-*loop*, if

- $\sigma_k \simeq_\star \sigma_l$ for some $0 \leq l < k$ (the non-alternating semantics).
- $\sigma_k \simeq_\star \sigma_l$ for some $0 \leq l < k$, and either both k and l are odd or they are both even (the alternating semantics).

The difference in the above definitions follows from the fact that in the alternating semantics the looping runs need to preserve the alternating structure when "unfolded", while in the non-alternating semantics their stucture is preserved without any additional conditions. Using $\simeq_\star$ instead of the standard equality of states follows from the fact that for the further possible behaviours of the net at a given state only the minimal values of the clocks of the processes involved in the enabled transitions are important.

A k-path π_k is called a *loop*, if it is (k, l)-loop for some $l \in \{0, \dots, k-1\}$. In this paper we assume the definitions of $[\varphi]_{M'}$ that can be found, respectively, in [35] (ECTL$_{-\mathrm{X}}$), and in [32, 5] (ELTL$_{-\mathrm{X}}$). However, to apply them to the DTPN setting, we have changed the definition of the loop to the one presented above.

Definition of formula $[\widetilde{M}_c(\mathcal{N})^{\varphi,\sigma^0}]_k$. Let $\widetilde{M}_c(\mathcal{N}) = (T \cup \mathbb{C}_{\mathcal{N}}, \Sigma, \sigma^0, \rightarrow_r, V_c)$ be a model of a given DTPN $\mathcal{N} = (P, T, F, m^0, Eft, Lft)$, φ an ECTL$_{-\mathrm{X}}$ or ELTL$_{-\mathrm{X}}$ formula, and $k \in \mathbb{N}$ a bound. In order to define the formula $[\widetilde{M_c(\mathcal{N})}^{\varphi,\sigma^0}]_k$ that constrains the $f_k(\varphi)$ symbolic k-paths to be valid k-paths of $\widetilde{M}_c(\mathcal{N})$, we proceed as follows. We assume that each state $\sigma \in \Sigma$ is given in a unique binary form, i.e., every state σ can be encoded as a bit vector $(\sigma[1], \dots, \sigma[\mathfrak{l}_b])$ of length $\mathfrak{l}_b$ depending on the number of places P of $\mathcal{N}$, the bound k , the value $c_{max}(\mathcal{N})$ (i.e., the greatest finite value of Eft and Lft), and the value $f_k(\varphi)$. Thus, each state σ can be represented by a valuation of a vector $w = (w[1], \dots, w[\mathfrak{l}_b])$ (called a *global state variable*), where $w[i]$, for $i = 1, \dots, \mathfrak{l}_b$ is a propositional variable (called *state variable*)[5]. A finite sequence $(w_0, \dots, w_k)$ of global state variables is called a *symbolic k-path*. Since in the ECTL$_{-\mathrm{X}}$ case we shall need to consider not just one but a number of symbolic k-paths, we use the notation of j-th symbolic k-path $(w_{0,j}, \dots, w_{k,j})$, where $w_{i,j}$ are global state variables for $0 \leq j < f_k(\varphi)$ and $0 \leq i \leq k$; the number of symbolic k-paths depends on the formula φ under investigation, and it is returned as the value $f_k(\varphi)$ of the function f_k; note that if φ is an ELTL$_{-\mathrm{X}}$ formulae then $f_k(\varphi) = 1$.

Let w, w' be two global state variables. We define the following auxiliary propositional formulae:

- $I_\sigma(w)$ is a formula that encodes the state σ of the model $\widetilde{M}_c(\mathcal{N})$, i.e., $\sigma[i] = 1$ is encoded by $w[i]$, and $\sigma[i] = 0$ is encoded by $\neg w[i]$.
- $\mathcal{TS}(w, w')$ ($\mathcal{TS}'(w, w')$) is a formula over w and w' which is true for two valuations σ_w of w and $\sigma_{w'}$ of w' iff $\sigma_w \xrightarrow{\delta}_r \sigma_{w'}$, for $\delta \in \mathbb{C}_{\mathcal{N}}$ ($\delta \in \mathbb{C}_{\mathcal{N}} \setminus \{0\}$, respectively). It encodes the time-successor relation of $\widetilde{M}_c(\mathcal{N})$.
- $\mathcal{AS}(w, w')$ is a formula over w and w' which is true for two valuations σ_w of w and $\sigma_{w'}$ of w' iff $\sigma_w \xrightarrow{t}_r \sigma_{w'}$, for $t \in T$. It encodes the action-successor relation of $\widetilde{M}_c(\mathcal{N})$.

The propositional formula $[\widetilde{M}_c^{\varphi,\sigma^0}]_k$ is defined as follows:

$$[\widetilde{M}_c^{\varphi,\sigma^0}]_k := I_{\sigma^0}(w_{0,0}) \wedge \bigwedge_{j=0}^{f_k(\varphi)-1} \bigwedge_{i=0}^{k-1} \mathcal{R}(w_{i,j}, w_{i+1,j})$$

where $w_{i,j}$ for $0 \leq i \leq k$ and $0 \leq j < f_k(\varphi)$ are global state variables, and

(a) $\mathcal{R}(w_{i,j}, w_{i+1,j}) := \mathcal{TS}(w_{i,j}, w_{i+1,j})$ when i is even, and $\mathcal{R}(w_{i,j}, w_{i+1,j}) := \mathcal{AS}(w_{i,j}, w_{i+1,j})$ when i is odd (the alternating semantics), or

(b) $\mathcal{R}(w_{i,j}, w_{i+1,j}) := \mathcal{TS}'(w_{i,j}, w_{i+1,j}) \vee \mathcal{AS}(w_{i,j}, w_{i+1,j})$ (the non-alternating semantics).

[5] Notice that we distinguish between states of Σ encoded as sequences of 0's and 1's (we refer to these as valuations of w), and their representations in terms of propositional variables $w[i]$.

Note that if φ is an $\mathrm{ELTL}_{-\mathrm{X}}$ formula, then $f_k(\varphi) = 1$, and the above definition is equivalent to the following one: $[\widetilde{M}_c^{\varphi,\sigma^0}]_k := I_{\sigma^0}(w_{0,0}) \wedge \bigwedge_{i=0}^{k-1} \mathcal{R}(w_{i,0}, w_{i+1,0})$.

5 BDD-based BMC for $\mathrm{ELTL}_{-\mathrm{X}}$ and $\mathrm{ECTL}_{-\mathrm{X}}$

Binary decision diagrams (BDDs) [8, 17] are an efficient data structure widely used for storing and manipulating boolean functions. In the paper we use Reduced Ordered Binary Decision Diagrams (ROBDDs) instead of the "pure" BDD structures. The advantage of ROBDDs is that they are canonical for a particular function and variable order.

To introduce a BDD-based bounded model checking method, we start with describing $\mathrm{ECTL}_{-\mathrm{X}}$ in terms of sets of reachable states at which the given formula holds [17]. For this purpose we need the notion of a fixed point.

Let S be a finite set and $\tau : 2^S \longrightarrow 2^S$ a *monotone* function, i.e., $X \subseteq Y$ implies $\tau(X) \subseteq \tau(Y)$ for all $X, Y \subseteq S$. Let $\tau^i(X)$ be defined by $\tau^0(X) = X$ and $\tau^{i+1}(X) = \tau(\tau^i(X))$. We say that $X' \subseteq S$ is a *fixed point* of τ if $\tau(X') = X'$. It can be proven that if τ is monotone, S is a finite set and $|S|$ is a number of its elements, then there exist $m, n \leq |S|$ such that $\tau^m(\emptyset)$ is the least fixed point of τ (denoted by $\mu X.\tau(X)$) and $\tau^n(S)$ is the greatest fixed point of τ (denoted by $\nu X.\tau(X)$).

Let $M = (L, S, s^0, \rightarrow, V)$ be a model, and $S_R \subseteq S$ a set of all the reachable states of the model M. For $X \subseteq S_R$, let $pre_\exists(X) = \{s \in S_R \mid (\exists s' \in X)(\exists l \in L)\ s \xrightarrow{l} s'\}$ be a set of all the reachable states from which there is a transition to some state in X. Further, we denote the set of all the reachable states of the model M at which φ holds by $[\![M, \varphi]\!]$ or by $[\![\varphi]\!]$, if M is implicitly understood. For $\mathrm{ECTL}_{-\mathrm{X}}$ formulae φ and ψ we define the following sets: $[\![true]\!] \overset{def}{=} S_R$, $[\![\wp]\!] \overset{def}{=} \{s \in S_R \mid \wp \in V(s)\}$, $[\![\neg\varphi]\!] \overset{def}{=} S_R \setminus [\![\varphi]\!]$, $[\![\varphi \wedge \psi]\!] \overset{def}{=} [\![\varphi]\!] \cap [\![\psi]\!]$, $[\![\varphi \vee \psi]\!] \overset{def}{=} [\![\varphi]\!] \cup [\![\psi]\!]$. The remaining operators can be defined as fixed points in the following way: $[\![\mathrm{EG}\varphi]\!] \overset{def}{=} \nu X.[\![\varphi]\!] \cap pre_\exists(X)$, $[\![\mathrm{E}[\varphi \mathrm{U} \psi]]\!] \overset{def}{=} \mu X.[\![\psi]\!] \cup ([\![\varphi]\!] \cap pre_\exists(X))$.

To define the sets corresponding to $\mathrm{ELTL}_{-\mathrm{X}}$ formulae we proceed as follows. Let $M = (L, S, s^0, \rightarrow, V)$ be a model, and φ an $\mathrm{ELTL}_{-\mathrm{X}}$ formula. We begin with constructing the tableau for φ, as described in [11], that is then combined with the model M to obtain their product, which contains these paths of M where the formula φ potentially holds. The product is then verified in terms of CTL model checking of EG*true* formula under fairness constraints. The fairness constraints, corresponding to sets of states, allow to choose only these paths of the model, along which at least one state in each set representing fairness constraints appears infinitely often. In the case of $\mathrm{ELTL}_{-\mathrm{X}}$ model checking, fairness is applied to guarantee that $\mathrm{E}(\varphi \mathrm{U} \psi)$ really holds, i.e., to eliminate paths where φ holds continuously, but ψ never holds. Finally, we choose only these reachable states of the product that belong to some particular set of states computed for the formula. The corresponding states of the verified system that are in this set, comprise the set $[\![M, \varphi]\!]$, i.e., the set of the reachable states where the verified formula holds. As we are unable to include a more detailed description of the method (due to the page limit), we refer the reader to [11] for more details.

Before describing the BDD-based bounded model checking method, we first define a submodel. Namely, for the model $M = (L, S, s^0, \rightarrow, V)$ and $U \subseteq S$ such that $s^0 \in U$, we define a *submodel* $M|_U = (L', U, s^0, \rightarrow', V')$, where: $L' = \{l \in L \mid (\exists s, s' \in U)\ s \xrightarrow{l} s'\}$, $\rightarrow' = \{s \xrightarrow{l} s' \mid s, s' \in U\}$, $V' : U \longrightarrow 2^{PV}$ is defined by $V'(s) = V(s)$ for all $s \in U$. As the method can be applied to BMC of both ECTL$_{-\mathrm{X}}$ and ELTL$_{-\mathrm{X}}$, we do not distinguish between ECTL$_{-\mathrm{X}}$ and ELTL$_{-\mathrm{X}}$ formulae, and in what follows, by φ we understand either an ECTL$_{-\mathrm{X}}$ formula or an ELTL$_{-\mathrm{X}}$ formula.

Let $M = (L, S, s^0, \rightarrow, V)$ be a model. For any set $X \subseteq S$ we define the set of successors of all the states in X by $X_{\leadsto} \overset{def}{=} \{s' \in S \mid (\exists s \in X)(\exists l \in L)\ s \xrightarrow{l} s'\}$. The complete set of the reachable states is obtained by computing the least fixed point $\mu Reach.\{s^0\} \cup Reach \cup Reach_{\leadsto}$. With each iteration, when the set *Reach* is extended with new states, i.e., with the set $Reach_{\leadsto}$, the verified formula is checked in the submodel $M|_{Reach}$. The loop terminates and the algorithm returns *true*, if the initial state s^0 is in the set of states of the obtained submodel at which φ holds. The search continues until no new states can be discovered from the states in *Reach*, i.e., the fixed point is reached. When we obtain the complete set of reachable states, and a path from the initial state on which φ holds could not be found in any of the obtained submodels, the algorithm terminates with $false$.

BDD-based Verification of DTPNs In order to verify a DTPN using BDDs first we need to translate its underlying reduced model into boolean formulae that are encoded with BDDs. Let $\widehat{M_c}(\mathcal{N}) = (T \cup \mathbb{C}_{\mathcal{N}}, \Sigma_s, \sigma^0, \rightarrow_s, V_s)$ be a model of a given DTPN $\mathcal{N} = (P, T, F, m^0, Eft, Lft)$. We assume that every state $\sigma \in \Sigma_s$ can be encoded as a bit vector $(\sigma[1], \ldots, \sigma[\mathfrak{l}_b])$ of length $\mathfrak{l}_b$ depending on the number of places P of $\mathcal{N}$, and the value $c_{max}(\mathcal{N})$. Thus, each state σ can be represented by a valuation of a vector $w = (w[1], \ldots, w[\mathfrak{l}_b])$ (called a *global state variable*), where $w[i]$, for $i = 1, \ldots, \mathfrak{l}_b$ is a propositional variable (called *state variable*).

Let w, w' be two global state variables. We define the following boolean formulae that are used in the encoding:

- $\mathcal{I}_\sigma(w)$ is a formula that encodes the state σ of the model $\widehat{M}(\mathcal{N})$, i.e., $\sigma[i] = 1$ is encoded by $w[i]$, and $\sigma[i] = 0$ is encoded by $\neg w[i]$.
- $TS(w, w')$ is a formula over w and w' which is true for two valuations σ_w of w and $\sigma_{w'}$ of w' iff $\sigma_w \xrightarrow{\delta}_s \sigma_{w'}$, for $\delta \in \mathbb{C}_{\mathcal{N}} \setminus \{0\}$. It encodes the time-successor relation of $\widehat{M_c}(\mathcal{N})$.
- $AS_t(w, w')$, where $t \in T$, is a formula over w and w' which is true for two valuations σ_w of w and $\sigma_{w'}$ of w' iff $\sigma_w \xrightarrow{t}_s \sigma_{w'}$. It encodes the action-successor relation of $\widehat{M_c}(\mathcal{N})$ for the transition $t \in T$.

- $T(w, w') = \left(\bigvee_{t \in T} AS_t(w, w')\right) \vee TS(w, w')$ is a formula over w and w' which is true for two valuations σ_w of w and $\sigma_{w'}$ of w' iff $\sigma_w \rightarrow_s \sigma_{w'}$. It encodes the transition relation of $\widehat{M_c}(\mathcal{N})$.

Notice that due to the fact that an implementation of the alternating semantics would be inefficient in the case of the BDD-based method, we apply only the non-alternating semantics.

In our implementation we use the order of variables suggested in [17] where the variables encoding the states and their successors are interleaved. The explanation of how we can compute the BDDs for the sets $X_{\leadsto}$ and $pre_{\exists}(X)$ (where $X \in \Sigma_s$) that are needed by the described fixed point methods can be found also in [17]. Moreover, we encode each disjunct of the formula encoding the transition relation, with separate BDDs.

6 Experimental Results

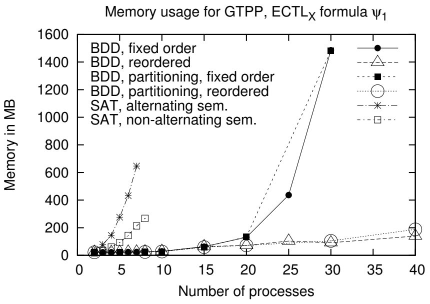
In this section we consider two scalable DTPNs which we use to evaluate the performance of our SAT- and BDD-based BMC algorithms, as well as of the tool Tina, for the verification of several properties expressed in $\mathrm{ECTL}_{-\mathrm{X}}$ and $\mathrm{ELTL}_{-\mathrm{X}}$. The evaluation is given by means of the running time and the consumed memory. Graphs representing the benchmarks described below can be found at the webpage of VerICS – http://verics.ipipan.waw.pl/.

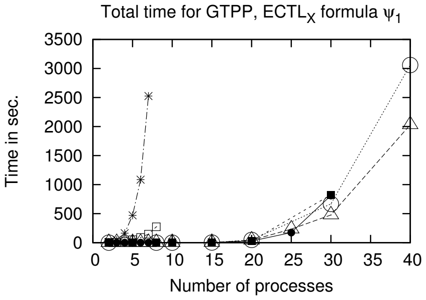
The first benchmark we consider is the Generic Timed Pipeline Paradigm (GTPP) Petri net model [25], which consists of Producer producing data (*ProdReady*) or being inactive, Consumer receiving data (*ConsReady*) or being inactive, and a chain of n intermediate Nodes which can be ready for receiving data ($Node_iReady$), processing data ($Node_iProc$), or sending data ($Node_iSend$). The example can be scaled in the number of intermediate nodes. The intervals are used to adjust the time properties of Producer, Consumer, and of the intermediate Nodes.

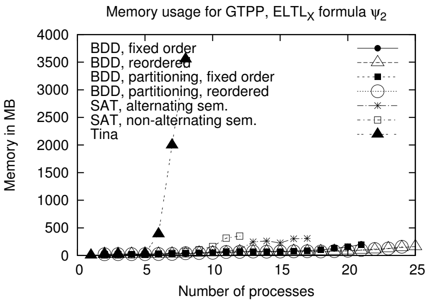
The second benchmark of our interest is the DTPN model for *Fischer's mutual exclusion protocol* (Mutex). The model consists of n time Petri nets, each one modelling a process, plus one additional net used to coordinate the access of processes to their critical sections *Mutual exclusion* means that no two processes are in their critical sections at the same time. The preservation of this property depends on the relative values of the time-delay constants δ and Δ. In particular, Fischer's protocol ensures mutual exclusion iff $\Delta < \delta$. This DTPN can be scaled in the number of processes.

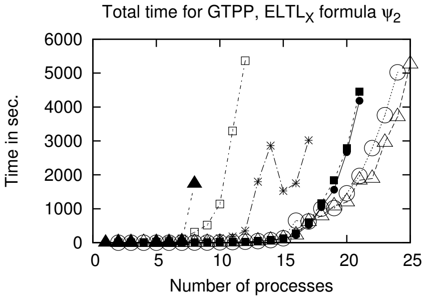
The GTPP Petri net model, where all the intervals are set to $[0, 2]$, was tested with the $\mathrm{ECTL}_{-\mathrm{X}}$ formula $\psi_1 = \mathrm{EG}(\mathrm{EF}(\neg ConsReady))$, and the $\mathrm{ELTL}_{-\mathrm{X}}$ formula $\psi_2 = \mathrm{EGF}(\neg ConsReady)$. The Mutex protocol, with $\Delta = 1$ and $\delta = 2$, was tested with the $\mathrm{ECTL}_{-\mathrm{X}}$ formulae: $\psi_1 = \mathrm{EGEF}(critical1 \vee \ldots \vee criticalN)$, $\psi_2 = \mathrm{EF}(trying_1 \wedge \ldots \wedge trying_N \wedge \mathrm{EG}(\neg critical_2 \wedge \ldots \wedge \neg critical_N))$, and the $\mathrm{ELTL}_{-\mathrm{X}}$ formulae: $\psi_3 = \mathrm{EGF}(critical_1 \vee \ldots \vee critical_N)$, $\psi_4 = \mathrm{EF}(trying_1 \wedge \ldots \wedge trying_N \wedge \mathrm{G}(\neg critical_2 \wedge \ldots \wedge \neg critical_N))$.

The above systems have been carefully selected in order to reveal the advantages and disadvantages of both SAT- and BDD-based BMC approaches.









For the SAT-based BMC method two semantics are implemented: the alternating and the non-alternating one. The results obtained for the non-alternating semantics are superior to those for the alternating one in the following two cases: (1) the length of the witness and the number of k-paths depends on the number of components of the considered system; (2) the number of k-paths is constant and their lengths are at least twice as long in the alternating semantics as in the non-alternating one. On the other hand, the non-alternating semantics is inferior to the alternating one in the case when the length of k-paths is independent of the number of components of the considered system and their number is independent of their lengths. Further, the assumed time limit (1800s) prefers the non-alternating semantics, i.e., if a larger time limit than 1800s is set, then for the alternating semantics much more components of a given system can be verified than for the non-alternating one (see Mutex and the formula ψ_1). The reason is that the SAT-based BMC method for systems with a large number of components (for the non-alternating semantics) generates the propositional formulae that are more complicated than in case of the alternating semantics. This results in the fact that the memory consumed by the SAT-solver is significantly larger for the set of clauses generated in case of the non-alternating semantics, therefore only smaller systems can be model-checked.

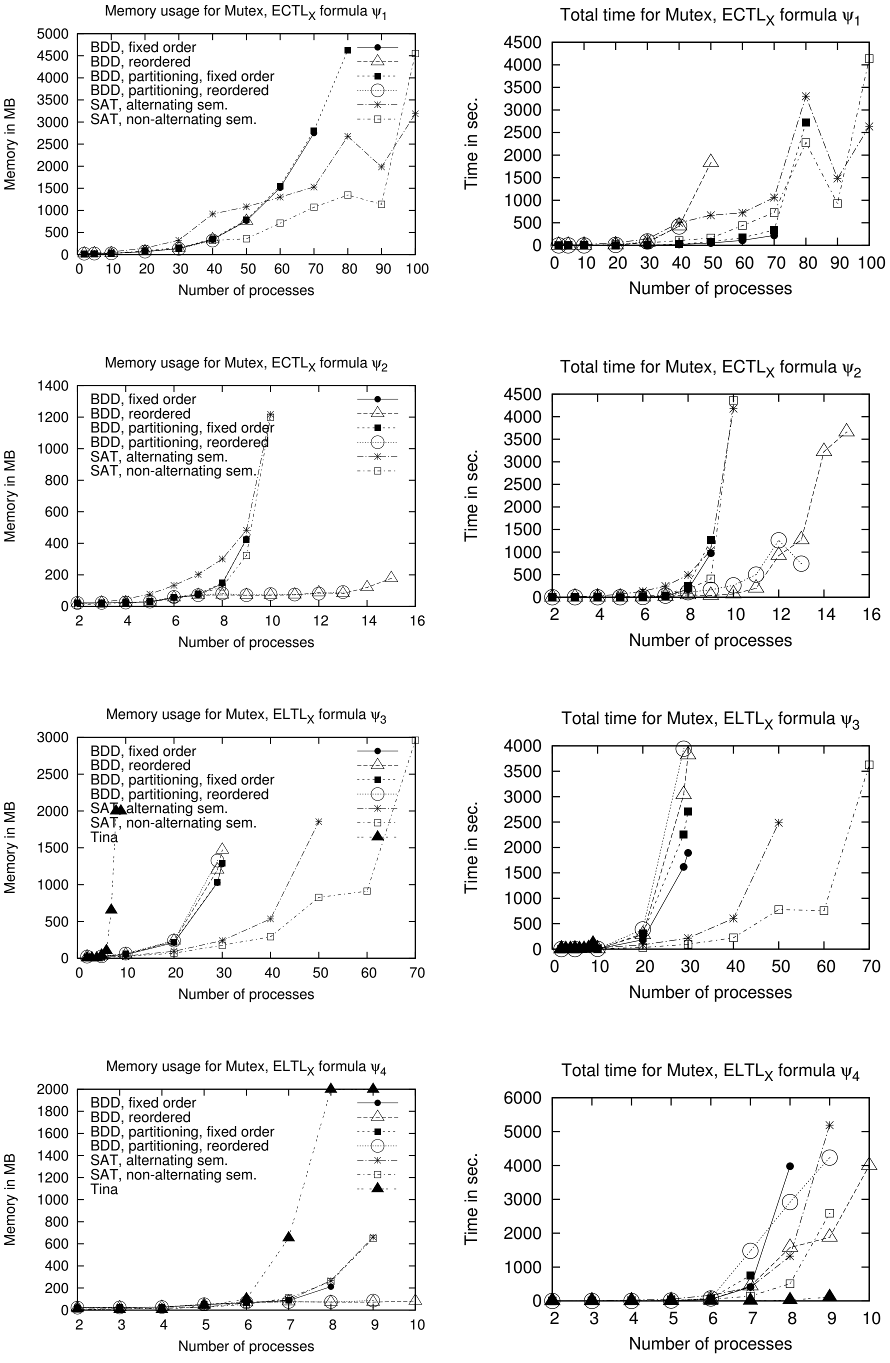
The method based on BDDs is implemented with reordering, and with the fixed interleaving order of the BDD variables. The reordering is performed by the Rudell's sifting algorithm available in the implementation of CUDD library. Moreover, we also use partitioned transition relations. In the case of GTPP, the BDD-based method is remarkably superior to the SAT-based method in terms of the verification times and the consumed memory for the tested formulae. The reason is the substantial number of k-paths in SAT-BMC, which causes a larger memory consumption and longer running times in comparison with the BDD-based method. Where the length of the witness is constant regardless of the number of the processes (i.e., in Mutex for ψ_1 and the corresponding formula ψ_3), the SAT-based method is more efficient than the BDD-based one. Our partitioning of the transition relation does not reduce noticeably the memory usage, although in most of the considered cases the method benefits from the reordering of the BDD variables. The BDD-based method deals better with the increasing length of the witness when scaling in the number of processes or nodes. In the case of Mutex, our experiments revealed that the method based on BDDs is more efficient for small and medium models, but it consumes more memory. The above observations seem to be consistent with other existing comparisons of SAT versus BDD [2].

We compare also our results with those of Tina, however, as Tina does not support a verification of $\mathrm{ECTL}_{-\mathrm{X}}$ formulae, the results only for $\mathrm{ELTL}_{-\mathrm{X}}$ are taken into account. Unsurprisingly, as Tina is a non-bounded model checker, the results are inferior to the results of our BMC methods. Although Tina seems to perform well in the case of ψ_4 for Mutex, it suffers from a significant increase of the memory usage for 8 processes and is unable to verify more than 9 processes.

All the benchmarks can be found at the webpage of VerICS, together with an instruction how to reproduce our results. For the tests we have used a computer running Linux 2.6.38 with two Intel Xeon 2.00GHz processors and 4 GB of memory. Both the algorithms have been implemented in C++. The BDD-based method uses CUDD [30], which is a general purpose BDD library, while the SAT-based technique uses MiniSat2 [14] for testing satisfiability of the generated propositional formulae.

7 Conclusions

In this paper we have presented two different approaches for bounded model checking of DTPNs: via a reduction to SAT and via BDDs. The two methods have been tested and compared to each other on two standard benchmarks. The specifications were given in the $\mathrm{ECTL}_{-\mathrm{X}}$ and $\mathrm{ELTL}_{-\mathrm{X}}$ languages. Additionally we have compared our results with those obtained using the tool Tina. The experimental results revealed that SAT-based BMC and BDD-based BMC are complementary solutions to the BMC problem, as their performance depends


Memory usage for Mutex, ECTL$_X$ formula ψ_1
Memory in MB
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
0
10
20
30
40
50
60
70
80
90
100
Number of processes
BDD, fixed order
BDD, reordered
BDD, partitioning, fixed order
BDD, partitioning, reordered
SAT, alternating sem.
SAT, non-alternating sem.
Total time for Mutex, ECTL$_X$ formula ψ_1
Time in sec.
0
500
1000
1500
2000
2500
3000
3500
4000
4500
0
10
20
30
40
50
60
70
80
90
100
Number of processes
Memory usage for Mutex, ECTL$_X$ formula ψ_2
Memory in MB
0
200
400
600
800
1000
1200
1400
2
4
6
8
10
12
14
16
Number of processes
BDD, fixed order
BDD, reordered
BDD, partitioning, fixed order
BDD, partitioning, reordered
SAT, alternating sem.
SAT, non-alternating sem.
Total time for Mutex, ECTL$_X$ formula ψ_2
Time in sec.
0
500
1000
1500
2000
2500
3000
3500
4000
4500
2
4
6
8
10
12
14
16
Number of processes
Memory usage for Mutex, ELTL$_X$ formula ψ_3
Memory in MB
0
500
1000
1500
2000
2500
3000
0
10
20
30
40
50
60
70
Number of processes
BDD, fixed order
BDD, reordered
BDD, partitioning, fixed order
BDD, partitioning, reordered
SAT, alternating sem.
SAT, non-alternating sem.
Tina
Total time for Mutex, ELTL$_X$ formula ψ_3
Time in sec.
0
500
1000
1500
2000
2500
3000
3500
4000
0
10
20
30
40
50
60
70
Number of processes
Memory usage for Mutex, ELTL$_X$ formula ψ_4
Memory in MB
0
200
400
600
800
1000
1200
1400
1600
1800
2000
2
3
4
5
6
7
8
9
10
Number of processes
BDD, fixed order
BDD, reordered
BDD, partitioning, fixed order
BDD, partitioning, reordered
SAT, alternating sem.
SAT, non-alternating sem.
Tina
Total time for Mutex, ELTL$_X$ formula ψ_4
Time in sec.
0
1000
2000
3000
4000
5000
6000
2
3
4
5
6
7
8
9
10
Number of processes

	SAT-BMC		**BDD-BMC**
	alternating sem.	**non-alternating sem.**	**non-alternating sem.**
Formula	$(\mathbf{k}, \mathbf{f_k}(\psi))$	$(\mathbf{k}, \mathbf{f_k}(\psi))$	**the number of iterations**
GPP, ψ_1	$(4 \cdot n + 6, 4 \cdot n + 8)$	$(2 \cdot n + 3, 2 \cdot n + 5)$	$2 \cdot n + 2$
GPP, ψ_2	$(4 \cdot n + 6, 1)$	$(2 \cdot n + 3, 1)$	$2 \cdot n + 2$
Mutex, ψ_1	$(8, 10)$	$(4, 6)$	4
Mutex, ψ_2	$(2 \cdot n + 8, 2)$	$(n + 2, 2)$	$2 \cdot n + 1$
Mutex, ψ_3	$(14, 1)$	$(6, 1)$	5
Mutex, ψ_4	$(4 \cdot n + 8, 1)$	$(2 \cdot n + 2, 1)$	$2 \cdot n + 1$

Table 1. The sizes of the witnesses. The number of nodes/processes is denoted by n.

on the system and the property that are verified. The approach based on BDDs scales better than the SAT-based one, when witnesses are found at small and constant depths with respect to the scaling parameter. From two of the considered semantics for SAT-BMC, the non-alternating one is more efficient.

The paper is the first one to present bounded model checking methods for verifying $\mathrm{ECTL}_{-\mathrm{X}}$ and $\mathrm{ELTL}_{-\mathrm{X}}$ properties of time Petri nets. The encodings that are used in the SAT-based method, are applied in the context of BMC and DTPNs for the first time. Similarly, the verification methods for $\mathrm{ECTL}_{-\mathrm{X}}$ and $\mathrm{ELTL}_{-\mathrm{X}}$ used in BDD-BMC have not been considered before in the bounded model checking of time Petri nets. The dependence on the length of the witnesses, and the performance of the two BMC methods for DTPNs has not been observed before as well.

As this is our early attempt at BDD-based bounded model checking, it suffers from some weaknesses. In particular, the encoding of the transition relation could be improved, and some more recent developments in BDD-based symbolic model checking could be applied.

In our future work we are going to consider dense semantics and more general time Petri nets.

References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. N. Amla, R. Kurshan, K. McMillan, and R. Medel. Experimental analysis of different techniques for bounded model checking. In *Proc. of TACAS'03*, volume 2619 of *LNCS*, pp. 34–48. Springer-Verlag, 2003.
3. G. Audemard, A. Cimatti, A. Kornilowicz, and R. Sebastiani. Bounded model checking for timed systems. In *Proc. of FORTE'02*, volume 2529 of *LNCS*, pp. 243–259. Springer-Verlag, 2002.
4. M. Benedetti and A. Cimatti. Bounded model checking for Past LTL. In *Proc. of TACAS'03*, volume 2619 of *LNCS*, pp. 18–33. Springer-Verlag, 2003.
5. A. Biere, A. Cimatti, E. Clarke, M.Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. of DAC'99*, pp. 317–320, 1999.

6. H. Boucheneb, G. Gardey, and O. H. Roux. TCTL model checking of time Petri nets. *Journal of Logic and Computation*, 19(6):1509–1540, 2009.
7. H. Boucheneb and R. Hadjidj. CTL* model checking for time Petri nets. *Theoretical Computer Science*, 353(1):208–227, 2006.
8. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, 35(8):677–691, 1986.
9. J. R. Burch, E. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1990.
10. G. Cabodi, P. Camurati, and S. Quer. Can BDD compete with SAT solvers on bounded model checking? In *Proc. of DAC'02*, pp. 117–122, 2002.
11. E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In *Proc. of CAV'94*, volume 818 of *LNCS*, pp. 415–427. Springer-Verlag, 1994.
12. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
13. F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Vardi. Benefits of bounded model checking at an industrial setting. In *Proc. of CAV'01*, volume 2102 of *LNCS*, pp. 436–453. Springer-Verlag, 2001.
14. N. Eén and N. Sörensson. MiniSat - A SAT Solver with Conflict-Clause Minimization. In *Proc. of SAT'05*, LNCS. Springer-Verlag, 2005.
15. G. Luettgen G. Ciardo and A. S. Miner. Exploiting interleaving semantics in symbolic state-space generation. *Formal Methods in System Design*, 31:63–100, 2007.
16. K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. In *Proc. of LPNMR'01*, volume 2173 of *LNCS*, pp. 200–212. Springer-Verlag, 2001.
17. M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.
18. R. Janicki. Nets, sequential components and concurrency relations. *Theoretical Computer Science*, 29:87–121, 1984.
19. A. Jones and A. Lomuscio. A BDD-based BMC approach for the verification of multi-agent systems. In *Proc. of CS&P'09*, volume 1, pp. 253–264. Warsaw University, 2009.
20. D. Lime and O. H. Roux. Model checking of time Petri nets using the state class timed automaton. *Discrete Event Dynamic Systems*, 16(2):179–205, 2006.
21. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
22. P. Merlin and D. J. Farber. Recoverability of communication protocols – implication of a theoretical study. *IEEE Trans. on Communications*, 24(9):1036–1043, 1976.
23. A. Męski, W. Penczek, and A. Półrola. BDD-based bounded model checking for elementary net systems. In *Proc. of CS&P'10*, volume 237(1) of *Informatik-Berichte*, pp. 219–230. Humboldt University, 2010.
24. A. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In *Proc. of ICATPN'99*, volume 1639 of *LNCS*, pp. 6–25. Springer-Verlag, 1999.
25. D. Peled. All from one, one for all: On model checking using representatives. In *Proc. of CAV'93*, volume 697 of *LNCS*, pp. 409–423. Springer-Verlag, 1993.
26. W. Penczek and A. Półrola. *Advances in Verification of Time Petri Nets and Timed Automata: A Temporal Logic Approach*, volume 20 of *Studies in Computational Intelligence*. Springer-Verlag, 2006.
27. W. Penczek, B. Woźna, and A. Zbrzezny. Bounded model checking for the universal fragment of CTL. *Fundamenta Informaticae*, 51(1-2):135–156, 2002.

28. Knot Pipatsrisawat and Adnan Darwiche. Rsat 2.0: Sat solver description. Technical Report D–153, Automated Reasoning Group, Computer Science Department, UCLA, 2007.
29. A. Półrola and W. Penczek. Minimization algorithms for time Petri nets. *Fundamenta Informaticae*, 60(1-4):307–331, 2004.
30. F. Somenzi. CUDD: CU decision diagram package - release 2.3.1. http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html.
31. M. Wan and G. Ciardo. Symbolic reachability analysis of integer timed petri nets. In *Proc. of SOFSEM'2009*, pp. 595–608, 2009.
32. B. Woźna. ACTL* properties and bounded model checking. *Fundamenta Informaticae*, 63(1):65–87, 2004.
33. B. Woźna, A. Zbrzezny, and W. Penczek. Checking reachability properties for timed automata via SAT. *Fundamenta Informaticae*, 55(2):223–241, 2003.
34. A. J. Yu, G. Ciardo, and G. Luettgen. Decision-diagram-based techniques for bounded reachability checking of asynchronous systems. *Software Tools for Technology Transfer*, 11(2):117–131, 2009.
35. A. Zbrzezny. Improving the translation from ECTL to SAT. *Fundamenta Informaticae*, 85(1-4):513–531, 2008.

A Appendix: Models for DTPNs - Proofs

In order to show that $\widehat{M_c}(\mathcal{N})$ can replace $M_c(\mathcal{N})$ in CTL$^*_{-\mathrm{X}}$verification (i.e., to prove Lemma 1) we shall prove the following lemma:

Lemma 3. *For a given distributed time Petri net $\mathcal{N}$ the models $M_c(\mathcal{N}) = (T \cup \mathbb{N}, \Sigma, \sigma^0, \rightarrow_c), V_c$ and $\widetilde{M_c}(\mathcal{N}) = (T \cup \mathbb{C}_{\mathcal{N}}, \Sigma, \sigma^0, \rightarrow_r, V_c)$ are bisimulation equivalent.*

Proof. We shall show that the relation $\mathcal{R} = \{((m, clock), (m', clock')) \mid m = m' \wedge \forall(\mathfrak{i} \in \mathfrak{I} \text{ s.t. } clock(\mathfrak{i}) \leq c_{max}(\mathcal{N}))\ clock(\mathfrak{i}) = clock'(\mathfrak{i}) \wedge \forall(\mathfrak{i} \in \mathfrak{I} \text{ s.t. } clock(\mathfrak{i}) > c_{max}(\mathcal{N}))\ clock'(\mathfrak{i}) > c_{max}(\mathcal{N})\}$ is a bisimulation. It is easy to see that $\sigma^0 \mathcal{R} \sigma^0$, and the valuations of the related states are equal (due to equality of their markings). Consider $\sigma = (m, clock) \in \Sigma$ and $\sigma' = (m, clock') \in \Sigma$ such that $\sigma \mathcal{R} \sigma'$.

- if $\sigma \overset{\delta}{\rightarrow}_c \sigma_1$, where $\delta \in \mathbb{N}$, then for each $t \in en(m)$ there exists $\mathfrak{i} \in I\mathcal{V}(t)$ s.t. $clock(\mathfrak{i}) + \delta \leq Lft(t)$. Consider the following cases:
  - if $en(m)$ contains at least one transition t with $Lft(t) < \infty$, then this implies that $\delta \leq c_{max}(\mathcal{N})$. In this case consider $\delta' = \delta$; it is easy to see from the definition of $\mathcal{R}$ that for any $t \in en(m)$ s.t $Lft(t) < \infty$ if in σ for some $\mathfrak{i} \in \mathfrak{I}$ we have $clock(\mathfrak{i}) + \delta \leq Lft(t)$, then in σ' $clock'(\mathfrak{i}) + \delta' \leq Lft(t)$ holds as well, and therefore the time δ' can pass at σ', leading to the state $\sigma' + \delta'$, which satisfies $(\sigma + \delta)\mathcal{R}(\sigma' + \delta')$ in an obvious way.
  - if $en(m)$ contains no transition t with $Lft(t) < \infty$, then we can have either $\delta < c_{\mathbf{m1}}$ or $\delta \geq c_{\mathbf{m1}}$, where by $c_{\mathbf{m1}}$ we mean the value $c_{max}(\mathcal{N}) + 1$. In the first case consider $\delta' = \delta$; it is obvious that such a passage of time at σ' disables no transition and is allowed therefore; it is also easy to see

that $(\sigma + \delta)\mathcal{R}(\sigma' + \delta')$. In the case $\delta \geq c_{\mathbf{m1}}$ assume $\delta' = c_{\mathbf{m1}}$. Again, it is obvious that such a passage of time at σ' disables no transition and due to this is allowed, and that in both the states $\sigma + \delta$ and $\sigma' + \delta'$ we have $clock(\mathtt{i}) > c_{max}(\mathcal{N})$ for all $i \in \mathfrak{I}$, and therefore $(\sigma + \delta)\mathcal{R}(\sigma' + \delta')$.
- the three remaining cases are straightforward.

Next, we can prove Lemma 1:

Proof. To prove the lemma, it is sufficient to show that $\widetilde{M}_c(\mathcal{N})$ and $\widehat{M}_c(\mathcal{N})$ are bisimulation equivalent. So, we shall show that the relation $\mathcal{R} \subseteq \Sigma \times \Sigma_s$ given by $\mathcal{R} = \{((m, clock), (m', clock_s)) \mid m = m' \wedge clock_s = clock|_{c_{\mathbf{m1}}}\}$ is a bisimulation. It is easy to see that $\sigma^0 \mathcal{R} \sigma^0$, and that the valuations of the related states are equal (due to equality of their markings). Consider $\sigma = (m, clock) \in \Sigma$ and $\sigma' = (m, clock_s) \in \Sigma_s$ with $clock_s = clock|_{c_{\mathbf{m1}}}$.

- if $\sigma \overset{\delta}{\to}_r \sigma_1$, where $\delta \in \mathbb{C}_{\mathcal{N}}$, then for each $t \in en(m)$ there exists $\mathtt{i} \in I\mathcal{V}(t)$ s.t. $clock(\mathtt{i}) + \delta \leq Lft(t)$. Due to the fact that for each $\mathtt{i} \in \mathfrak{I}$ it holds $clock|_{c_{\mathbf{m1}}}(\mathtt{i}) \leq clock(\mathtt{i})$, the time δ can pass at σ' as well, leading to the state $(m, clock|_{c_{\mathbf{m1}}} \oplus \delta)$. Consider the states $(m, clock + \delta)$ and $(m, clock|_{c_{\mathbf{m1}}} \oplus \delta)$; we should show that $(clock + \delta)|_{c_{\mathbf{m1}}} = clock|_{c_{\mathbf{m1}}} \oplus \delta$. We have the following cases: if $clock(\mathtt{i}) = clock|_{c_{\mathbf{m1}}}(\mathtt{i})$ and $clock(\mathtt{i}) + \delta < c_{\mathbf{m1}}$, then $clock(\mathtt{i}) + \delta = clock|_{c_{\mathbf{m1}}}(\mathtt{i}) + \delta = clock|_{c_{\mathbf{m1}}}(\mathtt{i}) \oplus \delta$. If $clock(\mathtt{i}) = clock|_{c_{\mathbf{m1}}}(\mathtt{i})$ and $clock(\mathtt{i}) + \delta \geq c_{\mathbf{m1}}$ then $clock|_{c_{\mathbf{m1}}}(\mathtt{i})$
$\oplus\, \delta = c_{\mathbf{m1}}$, and therefore $(clock + \delta)|_{c_{\mathbf{m1}}}(\mathtt{i}) = clock|_{c_{\mathbf{m1}}}(\mathtt{i}) \oplus \delta$. If $clock(\mathtt{i}) \geq c_{\mathbf{m1}}$ and $clock|_{c_{\mathbf{m1}}}(\mathtt{i}) = c_{\mathbf{m1}}$ then $clock(\mathtt{i}) + \delta \geq c_{\mathbf{m1}}$ and $clock|_{c_{\mathbf{m1}}}(\mathtt{i}) \oplus \delta = c_{\mathbf{m1}} = (clock + \delta)|_{c_{\mathbf{m1}}}(\mathtt{i})$, which ends this part of the proof.
- if $\sigma' \overset{\delta}{\to}_s \sigma'_1$, where $\delta \in \mathbb{C}_{\mathcal{N}}$ then for each $t \in en(m)$ there exists $\mathtt{i} \in I\mathcal{V}(t)$ s.t. $clock|_{c_{\mathbf{m1}}}(\mathtt{i}) \oplus \delta \leq Lft(t)$. If $Lft(t) < \infty$, then this implies $clock|_{c_{\mathbf{m1}}}(\mathtt{i}) \oplus \delta \leq c_{max}(\mathcal{N})$, which in turn gives that $clock|_{c_{\mathbf{m1}}}(\mathtt{i}) \leq c_{max}(\mathcal{N})$, and therefore $clock(\mathtt{i}) = clock|_{c_{\mathbf{m1}}}(\mathtt{i})$, $clock(\mathtt{i}) + \delta \leq c_{max}(\mathcal{N})$ and finally $clock(\mathtt{i}) + \delta \leq Lft(t)$, while if $Lft(t) = \infty$ then $clock(\mathtt{i}) + \delta \leq Lft(t)$ in an obvious way. Thus, the time δ can pass in σ as well. Consider the states $(m, clock + \delta)$ and $(m, clock|_{c_{\mathbf{m1}}} \oplus \delta)$; we should show that $(clock + \delta)|_{c_{\mathbf{m1}}} = clock|_{c_{\mathbf{m1}}} \oplus \delta$, which can be done analogously as in the previous part of the proof.
- The remaining two cases are straightforward.

Finally, we prove that the relation $\simeq_\star$ preserves the behaviours of the net (Lemma 2):

Proof. Consider the states $\sigma = (m, clock)$ and $\sigma' = (m, clock')$ $(\sigma, \sigma' \in \Sigma)$ s.t. $\sigma \simeq_\star \sigma'$.

- Consider $l = \delta \in \mathbb{N}$. The time δ can pass in σ iff for each $t \in en(m)$ there is $\mathtt{i} \in I\mathcal{V}(t)$ s.t. $clock(\mathtt{i}) + \delta \leq Lft(t)$. If $Lft(t) < \infty$, then we have that $\min_{\mathtt{i} \in I\mathcal{V}(t)} clock(\mathtt{i}) + \delta \leq Lft(t) \leq c_{max}(\mathcal{N})$, which implies that the states σ, σ' satisfy $\min_{\mathtt{i} \in I\mathcal{V}(t)} clock(i) = \min_{\mathtt{i} \in I\mathcal{V}(t)} clock'(i)$, and in turn $\min_{\mathtt{i} \in I\mathcal{V}(t)} clock'(\mathtt{i}) + \delta = \min_{\mathtt{i} \in I\mathcal{V}(t)} clock(\mathtt{i}) + \delta \leq Lft(t)$. If $Lft(t) = \infty$ we can have two cases: if $\min_{\mathtt{i} \in I\mathcal{V}(t)} clock(i) = \min_{\mathtt{i} \in I\mathcal{V}(t)} clock'(i) \leq c_{max}(\mathcal{N})$ then $\min_{\mathtt{i} \in I\mathcal{V}(t)} clock(i) + \delta = \min_{\mathtt{i} \in I\mathcal{V}(t)} clock'(i) + \delta$ which is not greater than $Lft(t)$ in an obvious way, while if $\min_{\mathtt{i} \in IV(t)} clock(i) > c_{max}(\mathcal{N})$ and $\min_{\mathtt{i} \in IV(t)} clock'(i) > c_{max}(\mathcal{N})$ then both $\min_{\mathtt{i} \in IV(t)} clock(i) + \delta$ and $\min_{\mathtt{i} \in IV(t)} clock'(i) + \delta$ are greater than $c_{max}(\mathcal{N})$ and do not exceed $Lft(t)$. Thus, the same time can pass at σ and at σ', and the obtained states are $\star$-equivalent.

- Consider $l = t \in T$ such that $t \in en(m)$. The transition t can fire at σ leading to a state $\sigma_1 = (m_1, clock_1)$ iff for each $\mathtt{i} \in IV(t)$ we have $clock(\mathtt{i}) \geq Eft(t)$ and there is $\mathtt{i} \in IV(t)$ such that $clock(\mathtt{i}) \leq Lft(t)$.
  - If $Lft(t) < \infty$ then from $\sigma \simeq_\star \sigma'$ we have that $\min_{\mathtt{i} \in IV(t)} clock(i) = \min_{\mathtt{i} \in IV(t)} clock'(i)$, which implies that for each $\mathtt{i} \in IV(t)$ $clock'(\mathtt{i}) \geq Eft(t)$ and there is $\mathtt{i} \in IV(t)$ such that $clock'(\mathtt{i}) \leq Lft(t)$, which means that t can fire at σ' as well, leading to a state $\sigma'_1 = (m'_1, clock'_1)$. In the obtained states we have $m_1 = m'_1$, $clock_1(\mathtt{i}) = 0 = clock'_1(\mathtt{i})$ for each $\mathtt{i} \in IV(t)$, and $clock_1(\mathtt{i}) = clock(\mathtt{i})$, $clock'_1(\mathtt{i}) = clock'(\mathtt{i})$ otherwise. Consider a transition $t' \in en(m')$. If $IV(t) \cap IV(t') \neq \emptyset$ then $\min_{\mathtt{i} \in IV(t')} clock_1(i) = \min_{\mathtt{i} \in IV(t')} clock'_1(i) = 0$, while if $IV(t) \cap IV(t') = \emptyset$ then for each $i \in IV(t')$ the relation between $clock_1(\mathtt{i})$ and $clock'_1(\mathtt{i})$ is the same as between $clock(\mathtt{i})$ and $clock'(\mathtt{i})$, which implies that either $\min_{\mathtt{i} \in IV(t')} clock_1(i) = \min_{\mathtt{i} \in IV(t')} clock'_1(i) \leq c_{max}(\mathcal{N})$ or $\min_{\mathtt{i} \in IV(t')} clock_1(i) > c_{max}(\mathcal{N}) \wedge \min_{\mathtt{i} \in IV(t')} clock'_1(i) > c_{max}(\mathcal{N})$. Thus, we have $\sigma_1 \simeq_\star \sigma'_1$.
  - If $Lft(t) = \infty$ then from the definition of $c_{max}(\mathcal{N})$ we have that $Eft(t) \geq c_{max}(\mathcal{N})$, and therefore from the definition of $\simeq_\star$ for each $\mathtt{i} \in IV(t)$ it holds $clock'(\mathtt{i}) \geq Eft(t)$, while for all $\mathtt{i} \in IV(t)$ $clock'(t) < Lft(t)$ in an obvious way. Thus, the transition can fire at σ' as well, leading to a state $\sigma'_1 = (m_1, clock'_1)$. The proof that $\sigma_1 \simeq_\star \sigma'_1$ is analogous to the case $Lft(t) < \infty$.
- The rest of the proof is straightforward.

Extending PNML Scope: the Prioritised Petri Nets Experience

Lom-Messan Hillah[1], Fabrice Kordon[2], Charles Lakos[3], and Laure Petrucci[4]

[1] LIP6, CNRS UMR 7606
and Université Paris Ouest Nanterre La Défense
200, avenue de la République, F-92001 Nanterre Cedex, France
Lom-Messan.Hillah@lip6.fr

[2] Université P. & M. Curie LIP6 - CNRS UMR 7606
4 Place Jussieu, F-75252 Paris cedex 05, France
Fabrice.Kordon@lip6.fr

[3] University of Adelaide, Adelaide, SA 5005, Australia
Charles.Lakos@adelaide.edu.au

[4] LIPN, CNRS UMR 7030, Université Paris XIII
99, avenue Jean-Baptiste Clément, F-93430 Villetaneuse, France
Laure.Petrucci@lipn.univ-paris13.fr

Abstract. The Petri net standard ISO/IEC 15909 comprises 3 parts. The first one defines the most used net types, the second an interchange format for these — both are published. The third part deals with Petri net extensions, in particular structuring mechanisms and the introduction of additional, more elaborate net types within the standard.

This paper focuses on the latter issue: how should a new net type be added, while guaranteeing the compatibility with the current standard. The extension of Petri nets with static or dynamic priorities is studied, showing design choices to ensure the desired compatibility. The result is integrated within the standard companion tool, PNML Framework. Then, the approach is generalised so as to be used at a later stage for other Petri nets extensions.

Keywords: Standardisation, PNML, Prioritised Petri Nets

1 Introduction

The International Standard on Petri nets, ISO/IEC 15909, comprises three parts. The first one (ISO/IEC 15909-1) deals with basic definitions of several Petri net types: Place/Transition, Symmetric, and High-level nets. It was published in December 2004 [5].

The second part, ISO/IEC 15909-2, defines the interchange format for Petri net models: the Petri Net Markup Language [7] (PNML, an XML-based representation). This part of the standard was published on February 2011 [6]. It can now be used by tool developers in the Petri Nets community with, for example, the companion tool to the standard, PNML Framework [4].

The standardisation effort is now focussed on the third part. ISO/IEC 15909-3 aims at defining enrichments and extensions on the whole family of Petri nets.

Extensions are, for instance, the support of modularity, time or probabilities. Enrichments consider less significant semantic changes such as inhibitor arcs, capacity places, etc. This raises flexibility and compatibility issues in the standard.

One of the interesting features in Petri nets is priorities. There are a number possibilities: static priorities and dynamic priorities, as summarised in [9]. Since such characteristics are of interest for several classes of Petri nets (from P/T up to high-level), it is highly desirable to investigate their definition in an orthogonal way that can be associated with any of the existing Petri net types.

This paper focuses on this objective: enrich existing Petri net types with both static and dynamic priorities. To do so, we explore a modular and generic enrichment mechanism that benefits from the current metamodels architecture of the standard. Thus, we can preserve consistency between existing Petri net types and those obtained with the proposed enrichments.

The paper is structured as follows. Section 2 summarises the specific aims of part 3 of the standard. Section 3 defines prioritised Petri nets, before describing, in Section 4, the introduction of Prioritised Petri nets types in the standard metamodeling framework. The metamodels obtained are then integrated within PNML Framework, and experimental results are reported in Section 4.3. Section 5 discusses the expertise drawn from the case of Prioritised Petri nets, so as to give general guidelines for the integration of a new Petri net type within the standard.

2 Aims of ISO/IEC 15909-3

While parts 1 and 2 of the ISO/IEC 15909 standard address simple and common Petri nets types, part 3 is concerned with extensions. These can take several forms, as described in Section 2.1. The work on these issues started with a one-year study group drawing conclusions w.r.t. the scope to be addressed. According to the study group conclusions, the standardisation project was launched in November 2010, for delivery within 5 years. The choices to be made must of course ensure compatibility with the previous parts of the standard, as discussed in Section 2.2.

2.1 Petri nets extensions

The Petri net extensions considered can be of different kinds: nodes or arcs extensions, structuring mechanisms, new Petri net types. One can even consider the possibility of tools exchanging Petri net properties through the net files. In this section, we present the main ideas underlying these possibilities.

Enrichments are concerned with the addition of a new type of node or arc to an already existing Petri net type. Typical examples of such extensions are inhibitor arcs or capacity places.

Enrichments[5] are rather simple extensions since, although they modify the net semantics, they do not not require the manipulation of data which is not already described in the Petri net type. Indeed, let us illustrate this with the capacity place example. If a net is extended with capacity places, the capacity only indicates a maximum marking the place can hold. The marking being already an element of the Petri net type description, adding capacities is straightforward. The mechanism for doing so is independent of the Petri net type: it is defined as a maximal marking for the place in the same manner as its initial marking is defined.

We have already tested enrichment mechanisms in practice within PNML Framework, the companion tool to the standard [11, 4], by introducing special arcs such as inhibitor, test, and reset arcs. As expected, this experiment proved successful. An extension of the PNML (Petri Net Markup Language) grammar for these special arcs is available online at [7].

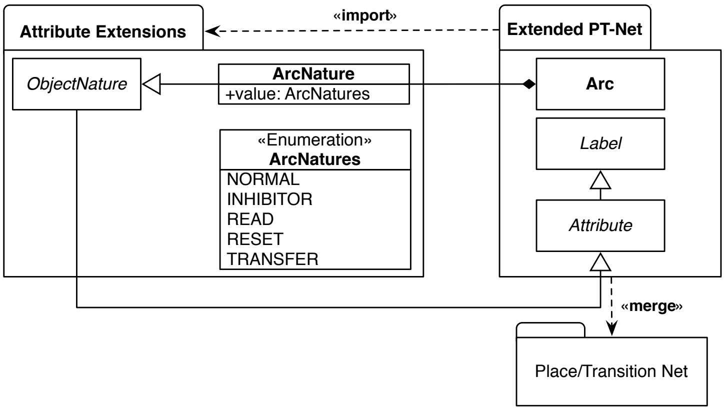


Fig. 1. Extending PT-Net with special arcs.

The experiment consisted in extending the metamodel of PT-Net, first by defining the special arcs nature as attribute extensions, whose metamodel is depicted in Fig. 1. Then, the extended PT-net metamodel is built by merging the current PT-Net metamodel and importing the attribute extensions one. This modular definition approach is put into practice in Section 4 and the use of the `import` and `merge` relationships explained.

Modularity and structuring mechanisms are essential for modelling and analysis purposes. Such mechanisms are independent of the Petri net type and should thus be general enough. Preliminary theoretical work in that direction has been presented in [8]. The main features are the following:

[5] The name chosen is consistent with the notion of enrichment for abstract data types [3].

- each module is composed of an interface and an implementation,
- the implementation is a Petri net,
- interfaces import and export Petri net elements (according to the implementation Petri net type): places, transitions, data types and operations,
- modules can be instantiated and connected so as to constitute an actual complex system.

Even though the work on structuring mechanisms has progressed well, there are still numerous issues to be considered, as detailed in [8]. For example, the semantics for connecting modules can vary, and node fusion policies could be defined. This leads to a more elaborate extension of Petri net types. Moreover, practical experiments still need to be conducted.

New Petri net types build on existing types — at least the Core Petri Net model — enhancing them with specific features which could be a new attribute or a new element. However, they are distinguished from “enrichments” in that they necessitate the elaboration of specific additional constructs. As an example, dynamically prioritised Petri nets associate with each Petri net transition a priority function with a marking as input and some value, e.g. a real number, as output. In this case, markings are already part of the defined Petri net elements, but real numbers are not. Therefore, in contrast to the capacity place example discussed above, dynamic priority is not an enrichment.

The core of this paper concerns how to introduce new Petri net types, and will thus be detailed by first studying the case of prioritised Petri nets in Section 4, and then generalising the approach in Section 5 to give guidelines for introducing new Petri net types in the future.

Properties such as safety and liveness properties, are a much longer term issue for standardisation, and will certainly not be achieved in the first release of part 3 of the standard, but might be in a future revision. The idea is to define storage mechanisms for properties so as to include properties within Petri net files. Thus, properties could be computed by one tool and later be exploited by another one.

2.2 Compatibility issues

The new features brought by part 3 of the standard must of course ensure compatibility with the previous stages, i.e. parts 1 and 2. This is essential since the already defined Petri net types constitute the building blocks.

Hence, an enrichment, such as the aforementioned capacity places, basically involves the addition of a new attribute to an already existing class of objects. In the case of capacity places, this attribute is a maximum marking. Note that this kind of extension easily applies to any type of net: a maximum marking is a marking as defined for the net type, be it a P/T net or a high-level net.

Similarly, the modular or structuring constructs must apply to all kinds of nets. They define the different parts of a module, that is the interface and the implementation, the composition policy, etc. The implementation is then an already known net type and the interface adds attributes telling which objects are imported or exported. The composition policy has so far been quite simple: place fusion and transition fusion, essentially. Nonetheless, work in this area must be pursued further.

Finally, adding new Petri net types will, as discussed in Section 2.1, build on at least the Petri net core model, thus preserving the essential characteristics common to all Petri net types. However, one can easily imagine that Prioritised P/T nets build on the P/T nets model while Prioritised high-level nets build on high-level nets. Further, several kinds of extensions could be applied so as to obtain a more elaborate net type, e.g. Prioritised Modular High-Level nets. Therefore, extensions must be carefully designed, allowing for a high degree of compatibility.

An important point must be investigated while some characteristics of Petri Nets can be considered as "orthogonal" (i.e. without influence on each other). As an example, priorities and colours can be considered separately and combined together because they do not affect the same attributes[6]. Such an orthogonality is important in the design of new Petri net types in the standard. This is discussed further in section 5.

3 Prioritised Petri Nets

This section introduces the definition of prioritised Petri nets, starting with static priorities.

Definition 1 (Statically Prioritised Petri net).
A Statically Prioritised Petri net *is a tuple SPPN* $= (P, T, W, M_0, \rho)$*, where:*

- (P, T, W, M_0) *is a Petri net.*
- ρ *is the* static priority function *mapping a transition into* $\mathbb{R}^+$*.*

We can also consider the case where the priority of transitions is *dynamic*, i.e. it depends on the current marking [1]. This definition was introduced in [9]. Note that the only difference with statically prioritised Petri nets concerns the priority function ρ.

Definition 2 (Prioritised Petri net).
A Prioritised Petri net *is a tuple PPN* $= (P, T, W, M_0, \rho)$*, where:*

- (P, T, W, M_0) *is a Petri net.*
- ρ *is the* priority function *mapping a marking and a transition into* $\mathbb{R}^+$*.*

The behaviour of a prioritised Petri net is now detailed, markings being those of the associated Petri net. Note that the firing rule is the same as for non-prioritised Petri nets, the priority scheme influencing only the enabling condition.

[6] for shared attributes like marking, we so far duplicate them. As an example, there are PTMarking for P/T nets and HLMarking for high-level nets.

Definition 3 (Prioritised enabling rule).

– *A transition $t \in T$ is* priority enabled *in marking M, denoted by $M[t\rangle^\rho$, iff:*
  - *it is enabled, i.e. $M[t\rangle$, and*
  - *no transition of higher priority is enabled, i.e. $\forall t' : M[t'\rangle \Rightarrow \rho(M, t) \geq \rho(M, t')$.*

– *The definition of the priority function ρ is extended to sets and sequences of transitions (and even markings M):*
  - $\forall X \subseteq T : \rho(M, X) = \max\{\rho(M, t) \mid t \in X \wedge M[t\rangle\}$
  - $\forall \sigma \in T^* : \rho(M, \sigma) = \min\{\rho(M', t') \mid M'[t'\rangle^\rho$ *occurs in* $M[\sigma\rangle^\rho\}$.

In the definition of $\rho(M, X)$, the set X will often be the set T of all transitions, in which case the T could be omitted and we could view this as a priority of the marking, i.e. $\rho(M)$. The definition of $\rho(M, X)$ means that we can write the condition under which transition t is priority enabled in marking M as $M[t\rangle^\rho$, or in the expanded form $M[t\rangle \wedge \rho(M, t) = \rho(M, T)$. We prefer the latter form if the range of transitions is ambiguous.

If the priority function is constantly zero over all markings and all transitions, then the behaviour of a Prioritised Petri Net is isomorphic to that of the underlying Petri Net.

Note that we choose to define priority as a positive real-valued function over markings and transitions — the higher the value, the greater the priority. We could equally define priority in terms of a *rank function* which maps markings and transitions to positive real values, but where the smaller value has the higher priority. This would be appropriate, for example, if the rank were an indication of earliest firing time. Note that the dependence of the priority function on the markings (as well as the transitions) means that *the priority is dynamic.*

4 Adding Prioritised Petri Nets to the Standard

This section introduces the Petri nets metamodels modular definition approach defined in Part 2 of the standard and how we put it into practice to design prioritised Petri nets.

4.1 Current Metamodels Architecture

Figure 2 shows an overview of the metamodels architecture currently defined in Part 2 of the standard. This architecture features three main Petri net types: Place/Transition, Symmetric and High-level Petri nets. They rely on the common foundation offered by the PNML Core Model. The PNML Core Model provides the structural definition of all Petri nets, which consists of nodes and arcs and an abstract definition of their labels. There is no restriction on labels since the PNML Core Model is not a concrete Petri net type.

Such a modular architecture favours reuse between net types. Reuse takes two forms in the architectural pattern of the standard: package `import` and `merge` relationships, as defined in the UML standard [12].

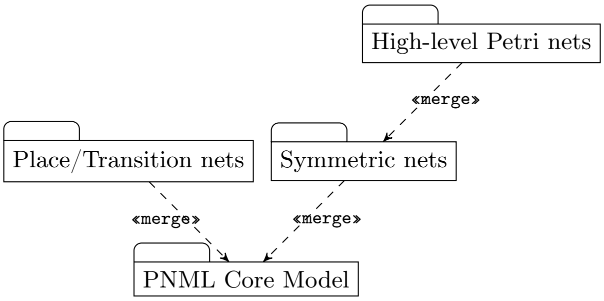


Fig. 2. Metamodels architecture currently defined in ISO/IEC-15909 Part 2.

`Import` is meant to use an element from another namespace (package) without the need to fully qualify it. For example, when package `A` includes: `import B.b`, then in `A` we can directly refer to `b` without saying `B.b`. But `b` still belongs to the namespace `B`. In the ISO/IEC 15909-2 standard, Symmetric nets import sorts packages such as `Finite Enumerations, Cyclic Enumerations, Booleans`, etc.

`Merge` is meant to combine similar elements from the merged namespace to the merging one. For example, let us assume that `A.a`, `B.a` and `B.b` are defined. If `B` is merged into `A` (`B` being the target of the relationship), it will result in a new package name `A'`:

- all elements of `B` now explicitly belong to `A'` (e.g., `A'.b`);
- `A.a` and `B.a` are merged into a single `A'.a` which combines the characteristics of both;
- actually, since `A` is the merging package (or the receiving package), `A` becomes `A'` (in the model, it is still named `A`).

`Merge` is useful for incremental definitions (extensions) of the same concept for different purposes.

In the standard, this form of reuse is implemented for instance by defining Place/Transition nets upon the Core Model and High-level nets upon Symmetric nets, as depicted in Figure 2. That is why Symmetric nets elements and annotations are also valid in High-Level Petri nets (but not considered as Symmetric nets namespace elements anymore).

This extensible architecture is compatible with further new net types definitions, as well as with orthogonal extensions shared by different net types. These two extension schemes will be put into practice for defining prioritised Petri nets, as discussed in the next section.

4.2 Metamodels for PT-Nets with Priorities

A prioritised Petri net basically associates a priority description with an existing standardised Petri net, thus building a new Petri net type. The metamodel in

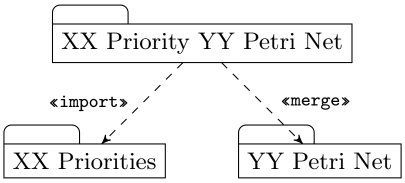


Fig. 3. Modular construction of prioritised Petri Nets metamodels.

Figure 3 illustrates this modular definition approach. It shows a blueprint for instantiating a concrete prioritised Petri net type, by merging a concrete Petri net type and importing a concrete priority package. The `XX Priority` package is the virtual representation of a concrete priority package and the `YY Petri net` is the virtual representation of a concrete Petri net type.

For example, Figure 4 shows a prioritised PT-Net using static priorities only. It is built upon a standardised PT-Net which it merges, and a `Priority Core` package, which it imports. The `Priority Core` package provides the building blocks to define `Static Priorities`, as depicted by Figure 5.

The purpose of the `Priority Core` package is to provide :

- the root metaclass for priorities, represented by the `Priority` metaclass;
- a priority level, which is an evaluated value represented by `PrioLevel`, associated with each instance of `Priority`;
- the ordering policy among the priority values of the prioritised Petri net. This ordering policy is represented by the `PrioOrderingPolicy` metaclass.

The purpose of priority levels is to provide an ordered scalar enumeration of values such that either the higher the value, the higher the priority, or the lower the value, the higher the priority. With the `Priority Core` package, and thanks to the `PrioLevel` metaclass, static priorities can thus be attached to transitions, as in the `Static Priority PT-Net` shown in Figure 4.

Using the same approach, Figure 6 shows a prioritised PT-Net which uses dynamic priorities. Dynamic priorities are built upon `Priority Core`.

This modular construction follows the extension schemes adopted so far in the PNML standard, that were explained earlier in this section. For instance,

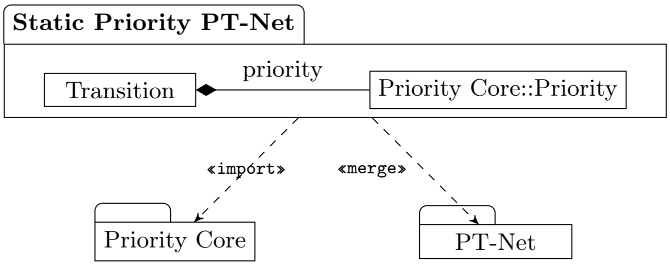


Fig. 4. Prioritised PT-Net metamodel showing how the priority description is attached.

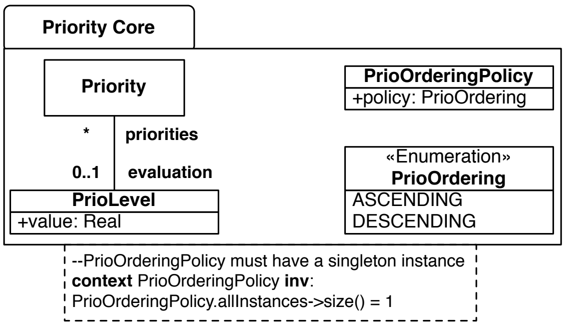


Fig. 5. Core package of priorities.

High-Level Petri nets build upon Symmetric nets that they merge, and new specific sorts (such as List, String and arbitrary user-defined sorts) that they import. The use of the `merge` and `import` relationships is therefore consistent.

This approach is consistent with the idea that a new Petri net type subsumes the underlying one it builds upon, but the algebraic expressions it reuses are generally orthogonal to net types. Next, we introduce the metamodel for priorities.

Priority Metamodel Prioritised Petri nets augment other net models (e.g. PT or Symmetric nets) by associating a priority description with the transitions. Such priority schemes are of two kinds:

- *static priorities*, where the priorities are given by constant values which are solely determined by the associated transition[7];
- *dynamic priorities*, where the priorities are functions depending both on the transition and the current net marking.

Figure 7 shows the modular architecture of priorities metamodels. The `Priority Core` package (detailed in Figure 5) provides the building blocks to define

[7] For high-level nets such as Coloured nets, the priorities are given by constant values which are solely determined by the associated binding element.

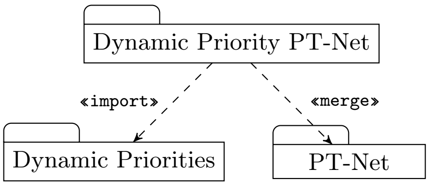


Fig. 6. Prioritised PT-Net metamodel using dynamic priority

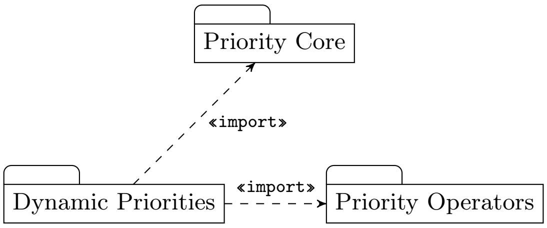


Fig. 7. Metamodel for priorities.

both `Static Priorities` and `Dynamic Priorities`. However, dynamic priorities are further defined using `Priority Operators`. Dynamic priorities can encompass static ones by using a constant function (for the sake of consistency in the use of priority operators).

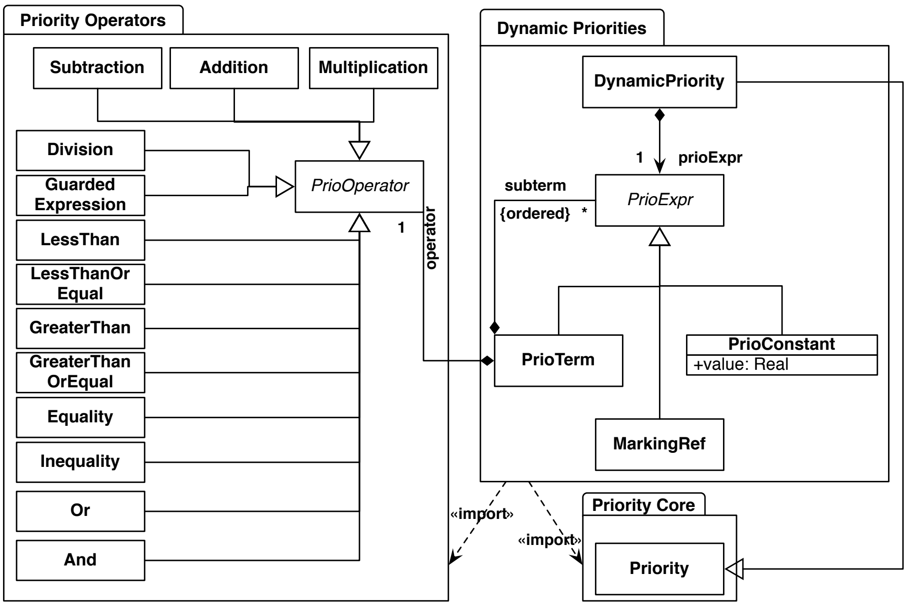


Fig. 8. Dynamic priorities and priorities operators packages.

Figure 8 shows how the `Dynamic Priorities` metamodel is built. A `DynamicPriority` *is a* `Priority Core::Priority`. It contains a priority expression (`PrioExpr`). A concrete priority expression is either a `PrioTerm` which represents a term, a `PrioConstant` which holds a constant value or `MarkingRef` which will hold a reference to the marking of a place.

Note that the actual reference to the metaclass representing markings is missing. It must be added as an attribute (named `ref`) to `MarkingRef` once the concrete prioritised Petri net type is created. Its type will then be a reference to the actual underlying Petri net type marking metaclass. For instance, in the case of prioritised PT-Net, this `ref` attribute will refer to the `PTMarking` metaclass.

A `PrioTerm` is composed of an operator (`PrioOperator`) and ordered subterms. This definition enables priority expressions to be encoded in abstract syntax trees (AST). For example, the conditional priority expression: `if` $M(P2) > 3$ `then` $3 * M(P2)$ `else` $2 * M(P1)$, is encoded by the AST of Figure 9, assuming that:

- P1 and P2 are places;
- M(P1) and M(P2) are respectively markings of P1 and P2;
- T1 is a transition the dynamic priority expression is attached to.

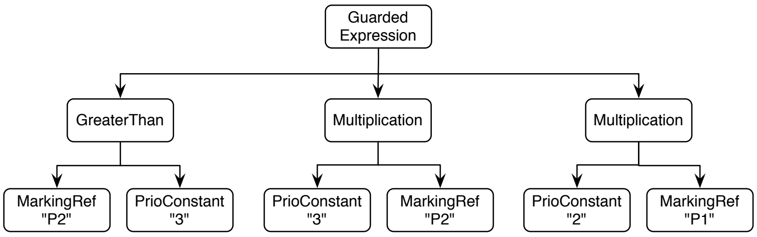


Fig. 9. AST of the conditional expression: `if` $M(P2) > 3$ `then` $3 * M(P2)$ `else` $2 * M(P1)$.

The priority operators are gathered within the `Priority Operators` package to allow for more flexibility in extending this priority framework. New operators can thus be added easily to this package.

Note that all these operators can also be found in ISO/IEC 15909-2, but are scattered among different sorts packages, thus directly tied to the sort they are most relevant for. We suggest for the next revision of the standard that they be gathered in separate and dedicated packages (e.g. arithmetic operators, relational operators, etc.). This refactoring will allow for more reusability across different Petri net type algebras definitions.

4.3 Towards experimentation with PNML Framework

PNML Framework is one of the standard's companion tools, which provides an intuitive and easy way to use Java Application Programing Interface (API) to handle standardised Petri nets models. The design and development of PNML Framework follows model-driven engineering (MDE) principles and relies on implementing mature technology such as Eclipse Modeling Framework (EMF).

Thanks to MDE, PNML Framework already uses the same modular definition approach as in the standard. Dealing with new Petri net types as proposed in this extension framework will thus be implemented in the same way as it was for the first standardised net types [4]. PNML-specific information (i.e., XML tags and their relationships) is embedded in the metamodels as annotation. The metamodels (in EMF) are thus self-contained w.r.t to PNML. This PNML-specific information can thus be captured during code generation of the appropriate reader and writer methods.

Using such an approach, an API to handle the new net type models can be generated in any output language, not only Java. This is possible because EMF format is the standardised eXtended Metadata Interchange (XMI), which is XML-based. It is thus open to any technology which can handle XML. In PNML Framework, code generation templates are designed to automatically capture these annotations.

The standard uses UML `merge` relationship to implement the reuse between metamodels. Therefore, UML `merge` constraints (preconditions) and transformations (postconditions) should also apply in our framework as well, at the design level.

When we started the development of PNML Framework as a means for early assessment of the standard design choices, EMF did provide powerful code generation capabilities (including code merging). However it did not provide models merging in the sense defined by UML, which was a disappointment. It was up to the modeler to come up with a way to implement this. UML plugin did provide models merging, but this needs a round-trip transformation from EMF to UML and vice versa, which practically turned out to be messy.

Recently, EMF Compare plugin now provides a workspace editor and a programmaging interface to compare and merge models, in a version control fashion. This environment could be used to perform a basic merger in the following main steps:

1. (a) If the source package of the merge relationship does not have any specific elements different from the merge relationship target package, duplicate the package of target Petri net type in the merge relationship and rename it to the source of the merge relationship.
   (b) If there are more than one merge relationship, perform this iteratively by pair of packages, the source being incrementally augmented.
   (c) If the source package does have specific elements, then design it first and apply the merge with one target. If there many targets, apply previous step.
2. Select the packages to import and import them.
3. Add relationships and attributes which need the merge operation to complete first.

IBM's Rational Software Architect also performs UML models comparison and merger [10] but it is not free or open software. Up to now, we let the modeler choose the means to perform the merger. Models merging is an important topic which is addressed, not only in the UML standard, but also by several studies [13, 14].

In the next section, we propose a generalisation of the metamodel modular definition approach we presented in this section, as an extension framework for the definition of new Petri net types.

5 Generalisation

We now generalise the approach used to create prioritised Petri nets metamodels to set up an extension framework as a proposal to be considered for ISO/IEC 15909-3. The purpose is to compose extensions on an existing Petri net type to build a new net type. It is illustrated by Figure 10, where `XX Extension` and `ZZ Extension` are extensions metamodels (e.g. priority and time) which are composed over an existing `YY Petri net` type to build the `(XX ∘ ZZ) YY Petri net` type metamodel.

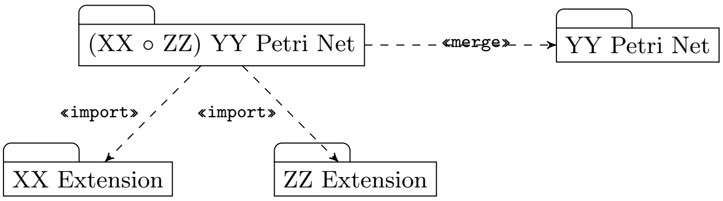


Fig. 10. Modular construction of a new net type by composing selected extensions.

This generalised modular definition approach involves two important characteristics to build the new net type, that are the *orthogonality* and *compatibility* of the combined features (extensions).

Orthogonality must guarantee upward compatibility: current net types definitions must have the ability to be extracted from new definitions that build upon them. For example, a Core model can currently be extracted from a Symmetric net model; a partial Symmetric net model can be extracted from a High-Level net model, after having pruned annotations and inscriptions that are not recognised in Symmetric nets.

Let us consider a Petri net type made of n extensions that are orthogonal in the sense defined in section 2.2. Compatibility must guarantee that the firing rule of the new net is sound. This is the case when the set of firable transitions can be expressed as follows:

$$T_f = \bigcap_{i=1}^{n} \mathit{firing}_i(T)$$

where T_f is the set of firable transitions and firing_i are partial firing functions using the dedicated attributes associated with a Petri net type extension.

These characteristics yield semantic issues that we are currently investigating with a group of Petri nets experts to assess how they should be tackled.

Underlying issues regarding semantics of Petri nets in terms of: (*i*) abbreviation (e.g. P/T vs. Colored nets), (*ii*) extension of the modeling power (e.g. inhibitor arcs) and (*iii*) change of semantic domain (e.g. time vs. stochastic) must be properly addressed [2]. We will continuously submit the outcome of this investigative work to the ISO/IEC JTC1/SC7 WG19 working group, responsible for the standardisation of Petri nets.

Experts must therefore pay careful attention to the compatibility issue between features since it is not considered at the syntactic level which concerns their metamodels definition.

6 Conclusion

The standardisation effort of the International Standard ISO/IEC 15909 is currently focused on the third part, where enrichments and extensions to Petri nets are being defined. This paper presents an extension scheme based on the standard approach, in order to define prioritised Petri nets. The presented work is structured in two steps: first the formal definition of prioritised Petri nets and their enabling rule, and then their metamodels definition.

Prioritised nets metamodels are built in a modular way. First, the priority core metamodel defines the necessary concepts for static priorities. Then the dynamic priorities package reuses the core package, while adding operators from the priority operators package to its definition. Using these building blocks, a static priority PT-Net can thus be defined upon the classic PT-Net package from the standard, using the priority core. A dynamic priority PT-Net can also be defined in the same way, this time using the dynamic priorities package.

Integrating new Petri net types defined using such an approach into the companion tool, PNML Framework, is no more different than the initial work which enables the support of the current standardized types (PT-Net, Symmetric Net and High-Level Petri Net). PNML Framework being based on mature model-driven engineering tools such as Eclipse Modeling Framework, enabling support of new Petri net types follows three simple steps: (*i*) create the metamodels of the extensions and the new type, (*ii*) annotate the metamodels with PNML-specific information (XML tags and attributes), and finally (*iii*) click on a button to generate the Java API to handle the new type. The annotation step follows simple conventions that are embedded in the current metamodels, and which are easy to reproduce. Code generation templates have been designed to capture these annotations.

The purpose of this investigative work is to propose an extension framework which enables experts to easily define new Petri net types, in a consistent way with the current standard approach. Orthogonality and compatibility of combined extensions to define new Petri net types are paramount for the semantic aspect of the new net types, in particular regarding firing rules. Syntax will usually not be an issue, as the metamodel definition approach presented does not consider semantic rules.

Perspectives to this work include presenting this approach as a contribution to the next plenary session of the ISO/SC 7/WG 19 working group (responsible for the standardisation of Petri nets in the ISO/IEC 15909 series), and experiment new Petri net types definitions involving different features combination such as time and priorities.

References

1. F. Bause. Analysis of Petri nets with a dynamic priority method. In Azéma, P. and Balbo, G., editors, *Proc. 18th International Conference on Application and Theory of Petri Nets, Toulouse, France, June 1997*, volume 1248 of *LNCS*, pages 215–234, Berlin, Germany, June 1997. Springer-Verlag.
2. M. Diaz, editor. *Petri Nets, Fundamental Models, Verification and Applications.* Wiley-ISTE, 2009.
3. J. A. Gougen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In *Current Trends in Programming Methodology*, pages 80–149. Prentice Hall, 1978.
4. L. Hillah, F. Kordon, L. Petrucci, and N. Trèves. PNML Framework: an extendable reference implementation of the Petri Net Markup Language. In *Proc. 31st Int. Conf. Application and Theory of Petri Nets and Other Models of Concurrency (PetriNets'2010), Braga, Portugal, June 2010*, volume 6128 of *Lecture Notes in Computer Science*, pages 318–327. Springer, June 2010.
5. ISO/IEC. Software and Systems Engineering - High-level Petri Nets, Part 1: Concepts, Definitions and Graphical Notation, International Standard ISO/IEC 15909, December 2004.
6. ISO/IEC. Software and Systems Engineering - High-level Petri Nets, Part 2: Transfer Format, International Standard ISO/IEC 15909, February 2011.
7. ISO/IEC/JTC1/SC7/WG19. *The Petri Net Markup Language home page.* `http://www.pnml.org`, 2010.
8. E. Kindler and L. Petrucci. Towards a standard for modular Petri nets: A formalisation. In *Proc. 30th Int. Conf. Application and Theory of Petri Nets and Other Models of Concurrency (PetriNets'2009), Paris, France, June 2009*, volume 5606 of *Lecture Notes in Computer Science*, pages 43–62. Springer, June 2009.
9. C. Lakos and L. Petrucci. Modular state spaces for prioritised Petri nets. In *Proc. Monterey Workshop, Redmond, WA, USA*, volume 6662 of *Lecture Notes in Computer Science*, pages 136–156. Springer, Apr. 2010.
10. K. Letkeman. *Comparing and merging UML models in IBM Rational Software Architect: Part 3 -A deeper understanding of model merging.* IBM, `http://www.ibm.com/developerworks/rational/library/05/802_comp3/`, 2005.
11. LIP6. *The PNML Framework home page.* `http://pnml.lip6.fr/`, 2011.
12. OMG. *Unified Modeling Language: Superstructure - Version 2.4 - ptc/2010-11-14*, Jan. 2011.
13. P. Sriplakich, X. Blanc, and M.-P. Gervais. Supporting transparent model update in distributed CASE tool integration. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1759–1766, New York, NY, USA, 2006. ACM.
14. B. Westfechtel. A formal approach to three-way merging of emf models. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, IWMCP '10, pages 31–41, New York, NY, USA, 2010. ACM.

Part III

Short Presentations

Specialisation and Generalisation of Processes

Christine Choppy[1], Jörg Desel[2]*, and Laure Petrucci[1]

[1] LIPN, CNRS UMR 7030, Université Paris 13, 93430 Villetaneuse, France
[2] FernUniversität in Hagen, 58084 Hagen, Germany

Abstract. In data modelling, one of the most important abstraction concepts is specialisation, with generalisation being the converse. Although there are already some approaches to define generalisation for process modelling as well, there is no generally accepted notion of generalisation for processes.

In this paper, we introduce a general definition of process specialisation and generalisation. Instead of concentrating on a specific process description language, we refer to labelled partial orders. For most process description languages, behaviour (if defined at all) can be expressed by means of this formalism. We distinguish generalisation from aggregation, and specialisation from instantiation. For Petri nets, we provide examples and suggest associated notations.

Our generalisation notion captures various previous approaches to generalisation, for example ignoring tasks, allowing alternative tasks and deferring choices between alternative tasks. A general guideline is that a more general process contains less features and/or less information than a more specific one.

In the conclusion, we also consider the question of common generalisation of a set of processes. Finally, we suggest a generalisation concept that goes beyond labelled partial orders, including additional behavioural constraints and process data generalisation.

Keywords: Process generalisation; process specialisation

1 Introduction

Specialisation, and its counterpart generalisation, is an important concept of data modelling which has been known for many years in database research [9]. The core idea of generalisation is to combine object types which share common attributes to a more general supertype which only has the common attributes whereas each more specific type inherits these attributes from the supertype and has its own, private attributes. We can also adopt a top-down view instead of a bottom-up one: starting with an object type, we might identify subtypes such that the objects in each of the subtypes share common additional attributes which might be meaningless for other objects. We can specialise the initial type to these subtypes and distinguish common attributes and additional attributes

* This work was achieved while the second author was visiting University Paris 13.

of the subtypes. Such a specialisation can cover the supertype (every object belongs to at least one subtype) or not, and it can divide the supertype into disjoint subtypes (no object belongs to more than one subtype) or not.

Instead of attributes of objects, generalisation and specialisation also apply to classes and their methods in object-oriented modelling, and in an even broader scope, to arbitrary features that are inherited from the more general to the more specific component. Generalisation and specialisation are very important abstraction concepts in models that clarify mutual dependencies. They are the prerequisite for reuse of system components and they allow to avoid redundancy. For the maintenance of systems and of models, only these concepts support that common features of different system or model components can be handled at a single place, instead of considering various copies.

Whereas generalisation and specialisation are abstraction techniques that have their own graphical representation in data modeling, there are only few suggestions how to apply comparable concepts to process models. On the other hand, the same arguments as above would apply to a generalisation and specialisation concept in behavioural modelling as well. At several places, this demand was explicitly expressed, see e.g. [3]. Actually, Ulrich Frank pointed us to this question and provided more (unpublished) examples and papers that identify the problem of process generalisation and specialisation.

We also considered other papers on specialisation for particular process models, such as Petri nets. In [14] a particular extensions of Petri nets is suggested, based on [7]. Unfortunately, it remains unclear how this approach can be transferred to other modelling languages. Another relevant approach is given in [11], however, this paper also restricts to a particular language. Moreover, it emphasises inheritance and change instead of specialisation and generalisation. In particular, the inheritance is based on *blocking* and *hiding* of transitions whereas in our notion blocking a transition will turn out to be a specialisation and hiding a transition will turn out to be a generalisation.

Another important difference to previous papers is that we consider partial order behaviour of process models instead of strings and sequential automata. This choice is justified by the observation, that many process languages emphasise concurrency between activities which is most appropriately represented by means of partial orders.

Our approach should be applicable to as many process modelling languages as possible. Therefore, we do not start with any particular syntactical description but rather concentrate on the behaviour of models. The behavioural notion used in this paper is given by partially ordered sets of activities representing runs, labelled by respective tasks that are expected to appear in a process model. Although this formalism is quite easy to understand, it needs some more involved technical notations that will be carefully introduced in Section 2. In our examples, we use Petri nets as a modelling language, but this does not restrict our approach to this particular language. We use only elementary Petri nets and expect the reader to understand them without any formal definition.

Our core criteria for a specialisation definition are that specialisation means to add something (features, tasks, information) to a process model and that everything valid for a more general process model should also hold for its specialisation. For example, adding a task to a process model results in the addition of respective activities in the runs, where the remaining structure of the runs is not changed. If, according to a process model, two tasks can be executed independently (in any order or concurrently), then, in a specialisation an order between these tasks can be specified. This results in runs that are also runs of the more general system. If two tasks can be executed alternatively, then a specialisation might add the information to decide which of the tasks occurs; in this case, some of the runs of the more general model are ruled out in the specialisation. All these relations can be formulated by means of the respective sets of runs, and will be the subject of Section 3.

We also identify more subtle specialisation relations that refer to the branching behaviour of process models; a more special model could have "earlier" information about the decision of a choice than a more general model, although their respective sets of runs are identical. For a motivating example and our solution to this phenomenon, see Section 4.

Section 5 comes back to the previous work on specialisation of Petri nets mentioned above. We show that these concepts can be viewed as a special case of our approach.

2 Basic Setting

In this paper, no formal definition of a process model is given since we do not stick to any particular process modelling language. Instead, some characteristics of a process model are expected to be defined: its set of tasks; its runs containing task executions; and a precedence relation between task executions in runs. This precedence relation should be a partial order, i.e. a transitive and irreflexive relation (in accordance with the WorkFlow Management Coalition [13], we use the term activity for a single execution of a task in a process run). Our definition resembles the definition of Partial Languages as defined by Grabowski [4] and Pomsets as defined by Pratt [8].

Given a process model P with set of tasks T, its behaviour is defined by its set of possible runs.

Definition 1 (Process run). *A* process run *(or just* run*)* π *of a process model* P *with set of tasks* T *is given by*

- *a finite set of* activities A_π*,*
- *partially ordered by a* precedence relation $\rightsquigarrow_\pi$*, and*
- *a* mapping $\lambda_\pi\colon A \to T$ *mapping each activity* to a task*.*

Remark 1. We have decided to consider finite runs only because infinite runs of process models are only of theoretical interest. However, each infinite run can be approximated by an infinite sequence of finite runs where each run is a proper prefix (defined below) of its successor.

If an activity x is mapped to a task $\lambda(x)$ then it represents an occurrence of $\lambda(x)$. Activities have to be distinguished from tasks since there can be more than one occurrence of a task in one run, with different precedence relations to other activities.

Definition 2 (Immediate precedence).
Given a process run π, we denote by $\rightarrow_\pi := \rightsquigarrow_\pi \setminus (\rightsquigarrow_\pi \circ \rightsquigarrow_\pi)$ the immediate precedence relation.

Remark 2. Since the set of activities A_π is finite, $\rightsquigarrow_\pi$ is the transitive closure of $\rightarrow_\pi$.

In graphical representations, we depict the relation $\rightarrow_\pi$ by means of directed arcs, i.e., we give the Hasse-diagram of partial orders. Two activities x and y satisfy $x \rightsquigarrow_\pi y$ if and only if there is a nonempty path leading from x to y.

Example 1.
Figure 1 depicts a process run π such that :
$A_\pi = \{1, 2, 3, 4, 5\}$, $\rightarrow_\pi = \{(1, 2), (2, 3), (3, 4), (1, 5)\}$, and $\lambda_\pi(1) = a$, $\lambda_\pi(2) = b$, $\lambda_\pi(3) = c$, $\lambda_\pi(4) = b$, $\lambda_\pi(5) = b$. Here, activities are denoted by numbers and tasks by lowercase letters.

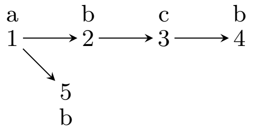


Fig. 1. A process run π

Note that a process run can feature several branches, and that a given task can occur at different places in the process run. In the above example, task b can occur after another occurrence of task b, and both occur concurrently to a third one. However, they are associated with different activities, respectively 4 and 2, which are ordered by the precedence relation, and 5.

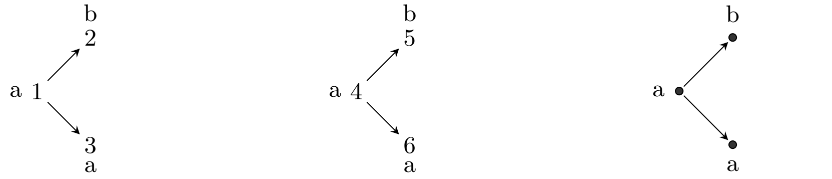
In case of sequential semantics of processes, where each run is represented by a sequence of occurring tasks, we can easily distinguish the first, the second, etc. occurrence of a single task. Each sequence can be viewed as a mapping from the set $\{1, 2, 3, \ldots, \mathit{length}(\mathit{run})\}$ to the set of tasks. For partial-order semantics, there is no such unique representation of a run. Hence the "same behaviour" can be represented by runs which only differ w.r.t. the activities. Such runs are said to be isomorphic.

Definition 3 (Isomorphic runs). *Let π and π' be two process runs of the same process model P with set of tasks T. Then, π' is* isomorphic *to π iff there is a bijection $\beta\colon A_\pi \rightarrow A_{\pi'}$ such that*

1. $\forall x, y \in A_\pi$: $x \rightarrow_\pi y \Longleftrightarrow \beta(x) \rightarrow_{\pi'} \beta(y)$ *and*
2. $\forall x \in A_\pi$: $\lambda_\pi(x) = \lambda_{\pi'}(\beta(x))$

Clearly, process run isomorphism is an *equivalence relation.* If isomorphic process runs are not distinguished, any representative of the equivalence class is used.

Example 2. Figure 2 schematises the isomorphism between two process runs. Process run π (Figure 2(a)) and process run π' (Figure 2(b)) both yield the same graph of tasks when abstracting away the activities.



(a) Process run π (b) Process run π' (c) Task occurrences

Fig. 2. A process run π' isomorphic to a process run π

In general, activities are formalised differently even though they have the same meaning in terms of tasks. Sometimes it is still necessary to distinguish activities within a process run in order to be able to refer to precise occurrences of a task. When this distinction is irrelevant in the pictures, only a • will be used instead of the actual activity (Figure 2(c)).

3 Linear Time Specialisation

The meaning of $a \leadsto_\pi b$ is that $\lambda(a)$ is executed before $\lambda(b)$ in run π. If activities a and b are not ordered by means of $\leadsto_\pi$, they occur in any order (this order is not captured by our notion of run) or concurrently. Each *linearisation* of a run π, obtained by adding elements to the order relation $\leadsto_\pi$, yields another run, which we consider more *specific* than π. In turn, each run *generalises* its set of linearisations.

The following definition not only compares the precedence relations between activities but also the respective sets of activities of two runs. It formalises the observation that a run of a more specific process definition might contain more details than a run of a less specific process definition. Hence the runs in the following definition can belong to distinct processes.

Definition 4 (Process run specialisation). *Let P and P' be two process models with sets of tasks T and T', respectively. A process run π' of P'* specialises *a process run π of P (denoted by $\pi' \geq \pi$) if there is an injective mapping μ: $A_\pi \rightarrow A_{\pi'}$ such that*

1. $\forall x, y \in A_\pi,\ x \rightarrow_\pi y \Rightarrow \mu(x) \rightsquigarrow_{\pi'} \mu(y)$ *and*
2. $\forall x \in A_\pi,\ \lambda_\pi(x) = \lambda_{\pi'}(\mu(x))$.

Remark 3. As a consequence of Definition 4, if $x \rightsquigarrow_\pi y$ then $\mu(x) \rightsquigarrow_{\pi'} \mu(y)$.

Each activity in the process run π has a corresponding activity in the specialised process (by the mapping μ), mapped to the corresponding task (2). Moreover, for each precedence relation in π, there is a corresponding sequence of precedences in π' (1).

Example 3. Figure 3 shows a process run π' specialising a process run π together with the mapping μ.

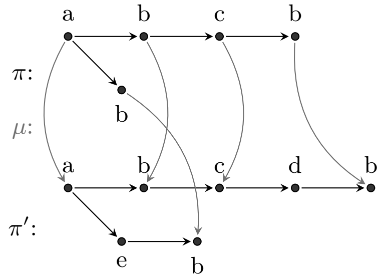


Fig. 3. Process run π' specialises process run π

Up to an isomorphism, a process run π' of P' specialises a process run π of P if and only if $A_\pi \subseteq A_{\pi'}$ and $\rightsquigarrow_\pi \subseteq \rightsquigarrow_{\pi'}$. This justifies the "$\geq$"-notion for specialisations. It is formalised by the following lemma.

Lemma 1. *Let P and P' be two process models with sets of tasks T and T', and runs π and π', respectively. $\pi' \geq \pi$ if and only if there exists a process run $\overline{\pi}$ such that*

1. *$\overline{\pi}$ is isomorphic to π*
2. $A_{\overline{\pi}} \subseteq A_{\pi'}$
3. $\rightarrow_{\overline{\pi}} \subseteq \rightsquigarrow_{\pi'}$
4. $\forall x \in A_{\overline{\pi}} : \lambda_{\overline{\pi}}(x) = \lambda_{\pi'}(x)$.

Proof. ($\Rightarrow$) Assume that π' specialises π by means of mapping μ. Process run $\overline{\pi}$ is constructed as follows:

$A_{\overline{\pi}} = \{x' \in A_{\pi'} \mid \exists x \in A_\pi : x' = \mu(x)\}$
$\rightsquigarrow_{\overline{\pi}} = \{(x', y') \in A_{\overline{\pi}}^2 \mid \exists (x, y) \in \rightsquigarrow_\pi : \mu(x) = x' \wedge \mu(y) = y'\}$
$\forall x \in A_{\overline{\pi}} : \lambda_{\overline{\pi}}(x) = \lambda_{\pi'}(x)$

($\Leftarrow$) Assume that $\overline{\mu} : A_{\overline{\pi}} \rightarrow A_{\pi'}$ is an isomorphism. Mapping $\mu : A_\pi \rightarrow A_{\pi'}$ is constructed by: $\forall x \in A_\pi, \mu(x) = \overline{\mu}(x)$.

A process run π' specialises π if π boils down to the same tasks graph as in process $\overline{\pi}$ (4), $\overline{\pi}$ is isomorphic to π (1), but with activities in π' (2). The precedence relation in $\overline{\pi}$ respects the order relation in π' (3).

Definition 5 (Linear time specialisation). *A process model P' is a* linear time specialisation *of a process model P if, for each run π' of P' there exists a run π of P such that $\pi' \geq \pi$.*

The process model P is then said to be a a linear time generalisation *of P'.*

As mentioned in the introduction, every valid statement about the more general process should hold for the specialised process as well. In a more formal setting, which is beyond the scope of this paper, any formula of an appropriate logic that evaluates to true for the general process, should be evaluated to true for the specialized process as well. Since in this section we concentrate on a relation based on the runs only, this logic would be Linear Time and based on partial orders, such as the one of [1].

The idea is that a less general process contains more features/information than a more general one but inherits all properties from the more general one.

Representative examples of specialisation/generalisation: Table 1 depicts some representative examples of generalisation and specialisation. The process models are given as Petri nets, and their runs are pictured as well. For each example, both a specialised and a general version are given. The example is named after the specialisation characteristic. Hence, the first one *forces an activity* since it prevents activity associated with task b from occurring. The second example *adds an activity* associated with task b. Finally, the last example imposes a sequential *ordering between activities* that are concurrent in the general model, namely those associated with tasks b and c.

4 Branching Time Specialisation

Let us first consider an example motivating further enhancement of the specialisation notion.

Example 4. Consider two processes P and P' modelled by the Petri nets in Table 2. These nets are actually labelled Petri nets. The two transitions of the net on the left-hand side labelled by a represent the same task a. As shown in the pictures, they have identical sets of runs. Both processes start with an occurrence of a and then either continue with an occurrence of b or with an occurrence of c. Hence they cannot be distinguished just by inspecting their runs. However, in process P, after the occurrence of a, there is a choice between continuing with b or with c, whereas in process P' we have to choose immediately between the run containing occurrences of a and b and the one containing occurrences of a and c. In this case we might consider process P' as a specialisation of P since additional (more precisely, earlier) information about the choice between b and c is necessary.

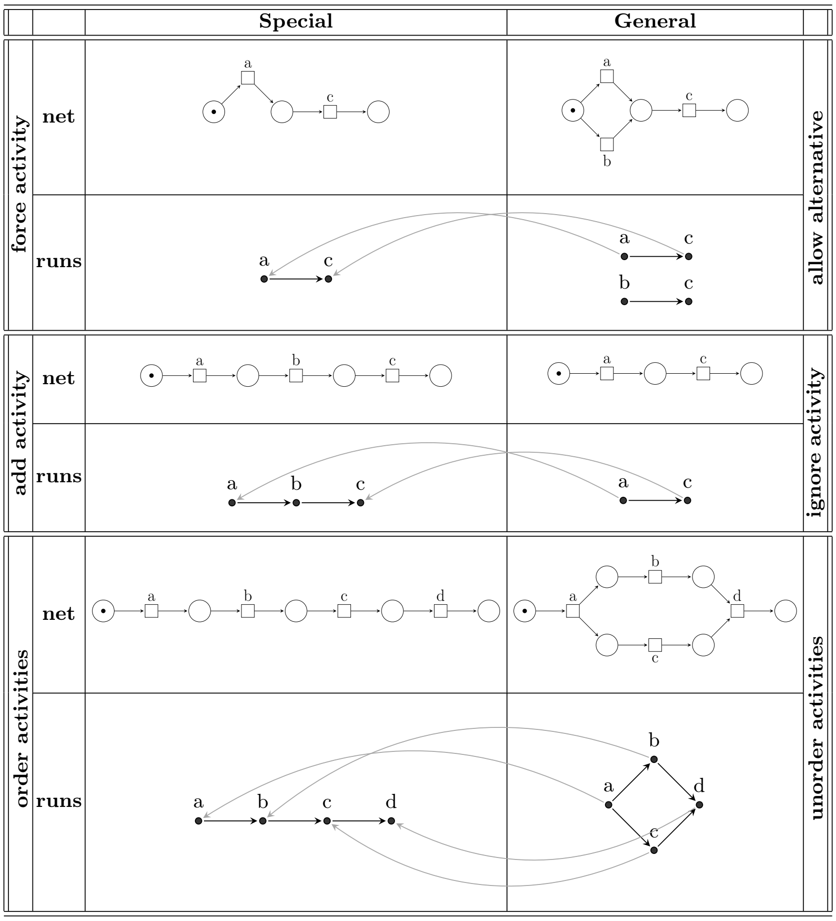


Table 1. Representative examples of specialisation/generalisation

We cannot distinguish processes P and P' by means of their runs as in the previous section. Therefore, it is necessary to carefully examine all possible behaviours. To do so, we first define the prefix of a run.

Definition 6 (Prefix of a run). *A run π is a* prefix of a run *π' if there is an injective mapping $\mu: A_\pi \to A_{\pi'}$ such that*

1. $\forall x, y \in A_\pi,\ x \to_\pi y \iff \mu(x) \to_{\pi'} \mu(y)$
2. $\forall x \in A_\pi,\ \lambda_\pi(x) = \lambda_{\pi'}(\mu(x))$.
3. $\forall x', y' \in A_{\pi'} : x' \to_{\pi'} y' \Rightarrow [(\exists y \in A_\pi, y' = \mu(y)) \Rightarrow (\exists x \in A_\pi, x' = \mu(x))]$.

Explanation: Each precedence relation in the prefix has a correspondence in the complete run π' (1) and corresponding activities are mapped to the same task (2). Moreover, each precedence relation of the complete run π' leading to an activity

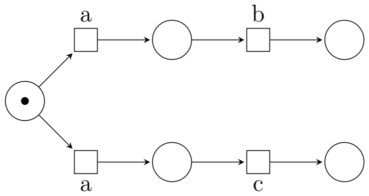
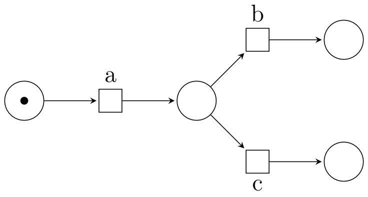
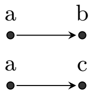
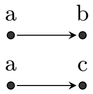
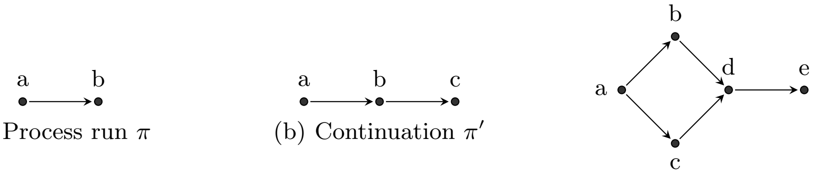
	P'	**P**
net		
runs		

Table 2. Runs do not always distinguish processes

that corresponds to one of the prefix also has its starting point corresponding to an activity of the prefix (3).

Example 5. The process run of Figure 4(a) is a prefix of the process runs of Figures 4(b) and 4(c).



(a) Process run π

(b) Continuation π'

(c) Continuation π''

Fig. 4. A process run π with different extended runs π' and π''

Roughly speaking, a prefix of a run is constituted by a set of activities together with all their predecessors, up to an isomophism. This observation is formalised in the following lemma:

Lemma 2. *A run π is a prefix of a run π' if and only if there exists a process run $\overline{\pi}$ such that*

1. *$\overline{\pi}$ is isomorphic to π*
2. *$A_{\overline{\pi}} \subseteq A_{\pi'}$*
3. *$\rightarrow_{\overline{\pi}} = \rightarrow_{\pi'} \cap A_{\overline{\pi}} \times A_{\overline{\pi}}$*

4. $\forall x \in A_{\overline{\pi}}$, $\lambda_{\overline{\pi}}(x) = \lambda_{\pi'}(x)$
5. $\forall y \in A_{\overline{\pi}}$, $x \rightarrow_{\pi'} y \Rightarrow x \in A_{\overline{\pi}}$.

Proof. ($\Rightarrow$) Let μ be the mapping mentioned in the definition of a prefix. Set $A_{\overline{\pi}} = \{x' \in A_{\pi'} \mid \exists x \in A_{\pi} \colon x' = \mu(x)\}$. The other defining components of $\overline{\pi}$ are items 3 and 4 of the lemma.
($\Leftarrow$) Choose $\mu(x) = \mu_i(x)$ for every $x \in A_\mu$ where μ_i is the isomorphism from π to $\overline{\pi}$.

Explanation: This lemma is very similar to lemma 1. Run $\overline{\pi}$ is exactly the restriction of run π' to the activities in $\overline{\pi}$.

Now, for a run that is a prefix of another run, we define the set of its continuations.

Definition 7 (Extended run, continuations). *An* extended run *is a run π together with a set of runs $\mathcal{C}(P)$, called its* continuations*, such that π is a prefix of each run in $\mathcal{C}(P)$.*

Note that in general $\mathcal{C}(P)$ does not contain all runs with prefix π. Two extended runs are different if their continuations are different, even if their respective runs are isomorphic.

Example 6. Figure 4 pictures a process run π (Figure 4(a)), and two different continuations: π' in Figure 4(b) and π'' in Figure 4(c). Then $(\pi, \{\pi'\})$ and $(\pi, \{\pi''\})$ are two different extended runs of π.

Isomorphic runs can lead to different states. Definition 7 avoids considering states of processes. For applying our concept to concrete process definition languages, we might consider the states reached by runs instead, determining the possible continuations.

Remark 4. For unlabelled Petri nets the distinction does not occur because isomorphic runs lead to identical markings. For labelled Petri nets the problem can occur.

Definition 8 (Branching Time Specialisation). *A process model P' is a* branching time specialisation *of a process model P if for each extended run π' of P' with continuation $\mathcal{C}(P')$ there exists an extended run π of P with continuation $\mathcal{C}(P)$ such that*

- $\pi' \geq \pi$
- *for each run $\tilde{\pi}' \in \mathcal{C}(P')$ there exists a run $\tilde{\pi} \in \mathcal{C}(P)$ such that $\tilde{\pi}' \geq \tilde{\pi}$.*

Example 7. The nets in Table 2 can be distinguished using specialisation of Definition 8, while they could not be with the linear time specialisation of Definition 5.

In Table 3, the activities are numbered after transitions names. The process model P' has a run π', consisting only of activity 1 labelled by a. Its set of continuations $\mathcal{C}(P')$ contains this run itself and also the run $1 \to 2$, as given in the figure. For P, we also have the run $\pi = 1$, labelled by a. Its set of continuations $\mathcal{C}(P)$ contains also the run $1 \to 4$, as given in the figure. So, in this example, for every continuation in $\mathcal{C}(P')$ we find an isomorphic continuation in $\mathcal{C}(P)$. Conversely, consider the extended run π consisting only of activity 1 labelled by a in process model P, which is a prefix of both depicted runs. Its continuations is the set of both runs depicted in the table. So it is possible to continue with a b-activity and it is also possible to continue with a c-activity. Now activity 1 (i.e., the run consisting only of this activity) can not be an associated run of P' because it misses a continuation with an activity labelled with c. Similarly, activity 3 can not be an associated run of P' because this one misses a continuation with an activity labelled with b. Arguing about all possible runs and continuations of P' as above shows that P' is a specialisation of P.

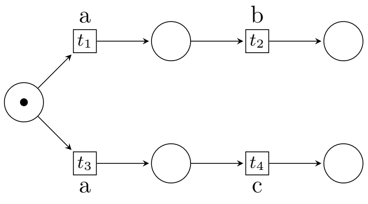
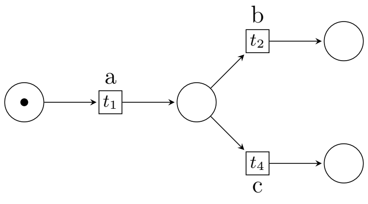
	P'	**P**
net		
runs	$\overset{a}{1} \longrightarrow \overset{b}{2}$ $\overset{a}{3} \longrightarrow \overset{c}{4}$	$\overset{a}{1} \longrightarrow \overset{b}{2}$ $\overset{a}{1} \longrightarrow \overset{c}{4}$

Table 3. Runs display different activities

We call this concept of specialisation Branching Time Specialisation because, again, we have that the valid statements (now expressable in Branching Time Temporal Logic) of the more general process should also hold for the specialised one.

5 Related Works

Abstraction and refinement have been the subject of numerous researches, and we just mention some of them here. The goals include to keep the modelling and reasoning as close as possible to the essence of the system to be developed, while still taking into account that a concrete representation (also called concrete implementation) of data should be provided at some later point together with the way it is related with the abstract data.

Tony Hoare [6] proposed a method to prove program correctness in the context of stepwise refinement where a representation of abstract data is chosen. Then John Guttag et al. [5] showed how the use of algebraic axiomatizations can simplify the process of proving the correctness of an implementation of an abstract data type. A number of works followed in the algebraic specification field. Now, an explicit data type refinement construct is introduced in programming languages like Scala [10].

As mentioned in the introduction, abstraction was studied for database modelling by John and Diane Smith [9], with the concepts of generalisation and aggregation. These concepts are also part of the object-oriented approaches (e.g. UML based approaches), or languages.

The same concepts should also be adapted to the modelling of the behaviour of a system, described by a process, a state-based diagram, etc.

Lee and Wyner [7] define a specialisation concept for dataflow diagrams. They distinguish minimal execution set semantics in which adding an activity is a specialisation (as in Table 1), and the maximal execution set semantics which is the option they choose. So obviously their definition of specialisation differs from ours.

In [14], they work on specialisation for a variant of Petri net (Workflow Process Definition). They analyse the approach of van der Aalst and Basten [11] who identify four types of inheritance of workflows, and propose an extension.

Wang et al. [12] stress that design issues are important, and, in the context of component-based model-driven development, they present two refinement relations, a trace-based refinement and a state-based (data) refinement, that provide different granularity of abstractions. So they combine the data refinement and the behaviour refinement.

6 Conclusions

Summary

We have presented a very general notion of specialisation and generalisation of processes which does not stick to a specific process modelling language but can be applied to all process languages where behaviour can be expressed in terms of partially ordered activities. Processes with sequential runs are a special case where all activities in runs are mutually ordered. Specialisation is interpreted as addition of features, where features can be additional tasks, additional ordering information, additional information on choices and earlier information on choices. All these features except the last one can be expressed by a specialisation relation on runs whereas the last one concerns branching points of process models that do not appear in runs and hence require a more involved definition based on runs and their possible continuations.

Specialisation versus Refinement/Generalisation versus Aggregation

One could argue that refinement of a process element is a form of specialisation because the more detailed view adds information. Conversely, what distinguishes generalisation and aggregation? According to our definition, specialisation *adds* something to a process, whereas refinement *replaces* something by something else, which should be more detailed. For example, if a task is added to the process, then the process is more special and associated activities show up in its runs. If a task is refined to two subsequent tasks, then the process is more detailed and the respective activities are refined accordingly in its runs.

Specialisation versus Instantiation

In one of our previous examples we have shown that information about the decision of choices can be viewed as a particular specialisation. The specialised process contains less alternatives. If all choices are decided, then the process is deterministic and contains no alternative at all. In other words, it only has a single run (remember that our notion of a run captures possible concurrency so that no additional runs caused by interleaving appear). Depending of the representation of the process and of its single run, both might look very similar. However, the run represents an instance of the process (and was an instance of the original process, too) whereas the process is on a lower meta-level. Whereas repeated specialisation of processes is possible and always yields new processes, instantiation of processes decreases the meta-level by one and can only occur once.

Extensions

It is obvious that there are features, that can be added, which cannot be handled by our concept so far. One example considers concurrency. Our notion of partially ordered runs is interpreted in such a way that activities that are not ordered can occur concurrently or in any order. However, it could be possible to specialise such a specification by demanding that activities have to occur concurrently whereas any order would be illegal. This cannot be expressed by our notion unless we add an additional “concurrent” relation. Other, similar relations are “not later than” or “at most two of three activities occur concurrently”.

Another extension refers to data. Usually tasks are performed for some or several data objects. This data does not appear in runs of our processes. For a combined view of processes and data, and for a combined notion of specialisation, the process view and the data view have to be integrated. There is an obvious way to do this integration by carrying over the data attributes from tasks to activities. Then the mapping between activities constituting our specialisation relation has to be extended to these data objects; the more general process handles the more general data.

Representation

In data modelling, generalisation and other abstraction techniques such as aggregation are represented graphically in the model. While this is nicely possible for aggregation in process models (see, for example, [2]), we do not see an elegant solution for generalisation in process models. One obvious approach is to depict additional tasks and additional relations between tasks by means of different symbols (colours or lines, respectively). However, if a single specialisation considers changes in different parts of a process then it is not obvious to express that these changes can only occur together.

The Partial Order of Specialisation

Our notion of generalisation and specialisation is not primarily given as a problem (is x a specialisation of y ?) but as a specification means. However, it is an interesting question whether the above question is decidable and, in the positive case, what is the complexity of a decision algorithm or of the problem in general (i.e., the complexity of the most efficient algorithm).

It is not difficult to see that the specialisation relation between single process runs is decidable, although any algorithm heavily depends on the data structure used to represent the respective process runs. Notice that by definition process runs are finite, i.e., have a finite set of activities. We cannot deal with isomorphism classes of process runs in algorithms but instead consider arbitrary representative process runs.

Proposition 1. *Given two process runs π and π' of two process models P and P', it is decidable whether π' specialises π.*

Proof. π' can only be a specialisation of π if, for each task t, π' has at least as many activities labelled by t as π has. This condition is easy to check. Assume that it holds true.

There are finitely many injective label-preserving mappings from the activities of π to the activities of π', and it is not difficult to construct them in a systematic way. For each pair of activities of π which are in the immediate precedence relation we check whether the respective target activities are ordered in π' (they do not necessarily have to be immediate successors!). If this is true for at least one mapping, then π' is a specialisation of π.

Things become more difficult when process models are compared, because each process model can have infinitely many runs. Since we do not consider any particular process model, nothing can be said about decidability in general. Clearly, there is only a chance for decidability of specialisation if a process model cannot generate infinitely many arbitrary runs.

It is not difficult to prove that "being a specialisation" is a partial order on process models. Its minimal element is the empty process model which is a (useless, though) generalisation of all process models. Using this order one might ask, for a given set of process models, whether there is a "largest" generalisation, whether this is unique, and whether it can be constructed algorithmically. Similarly, it would be interesting to find the "smallest" specialisation of a set of process models. There are no obvious answers to these questions, and hence the study of the partial order of specialisation is subject to future work.

References

1. Girish Bhat and Doron Peled. Adding partial orders to linear temporal logic. *Fundamenta Informaticae*, 36(1):1–21, 1998.
2. Jörg Desel and Agathe Merceron. Vicinity respecting homomorphisms for abstracting system requirements. *Transactions on Petri Nets and Other Models of Concurrency*, 4:1–20, 2010.
3. Ulrich Frank and Bodo Van Laak. A Method for the Multi-Perspective Design of Versatile E-Business Systems. In *Proceedings of the 8th Americas Conference on Information Systems*, pages 621–627, 2002.
4. Jan Grabowski. On partial languages. *Fundamenta Informaticae*, 4(2):428–498, 1981.
5. John V. Guttag, Ellis Horowitz, and David R. Musser. Abstract data types and software validation. *Commun. ACM*, 21(12):1048–1064, 1978.
6. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
7. Jintae Lee and George M. Wyner. Defining specialization for dataflow diagrams. *Information Systems*, 28(6):651–671, 2003.
8. Vaughan Pratt. Modelling concurrency with partial orders. *Int. Journal of Parallel Programming*, 15:33–71, 1986.
9. John Miles Smith and Diane C. P. Smith. Database abstractions: aggregation and generalization. *ACM Trans. Database Systems*, 2(2):105–133, 1977.
10. The Scala Programming Language. `http://www.scala-lang.org/`.
11. Will M. P. van der Aalst and Twan Basten. Inheritance of workflows: an approach to tackling problems related to change. *Theoretical Computer Science*, 270(1-2):125–203, 2002.
12. Zizhen Wang, Hanpin Wang, and Naijun Zhan. Refinement of Models of Software Components. In *Proceedings of SAC'10*, pages 2311–2318, 2010.
13. Workflow Management Coalition. `http://www.wfmc.org/`.
14. George M. Wyner and Jintae Lee. Applying Specialization to Petri Nets: Implications for Workflow Design. In Christoph Bussler and Armin Haller, editors, *Business Process Management Workshops*, volume 3812, pages 432–443, 2005.

Integrating Verification into the Paose Approach

Marcin Hewelt, Thomas Wagner, and Lawrence Cabac

University of Hamburg, Department of Informatics
http://www.informatik.uni-hamburg.de/TGI/

Abstract. The Paose approach to software engineering combines Petri nets with the paradigm of agent-orientation and utilises the agent metaphor to structure large software systems and their development. Up until now the Paose approach only exhibited aspects of verification in a rudimentary way. This is due to the complexity of the systems to be verified on the one hand, and the expressiveness of the employed Petri net formalism of reference nets on the other hand. This contribution deals with enhancing the tool support for Paose in this regard.
We present how we technically integrate the functionality of LoLA, a sophisticated verification tool, into Renew, the development and runtime environment that backs Paose. Furthermore we sketch how structural aspects of multi-agent systems developed with the agent framework of Mulan can be exploited for verification. The results of the integration are applied in the context of distributed network security for the Herold research project.

Keywords: Development approach, Petri nets, Petri net tools, verification

1 Introduction

In order to successfully tackle the challenges of creating large, complex software systems, the adoption of a development approach is almost mandatory. Paose (**P**etri Net-Based **A**gent-**O**riented **S**oftware **E**ngineering, [3]) is a sophisticated development approach that utilises Petri nets and the multi-agent system (MAS) metaphor to structure software systems and their development. It has been used in a variety of projects and has proven to be suitable for developing large and complex systems with large groups of developers.

One aspect that has, up until now, been largely left out of the Paose approach is the verification of the resulting software systems. The main challenge in this regard is posed by the highly expressive nature of the Petri net formalism of reference nets [12], utilised in Paose. Generally speaking, (low-level) Petri nets offer a formal and clear way of specifying complex systems, while at the same time retaining an intuitive and compact graphical representation. This allows for the use of analysis techniques, which can verify certain properties of a Petri net specification. Reachability of certain markings, boundedness of places and liveness of transitions are just some examples of properties that can be verified for some Petri net classes. Reference nets, however follow the nets-within-nets paradigm. This allows tokens to be nets themselves and consequently leads to a nested system of nets. In addition tokens can also refer to Java objects which can be manipulated using (Java) inscriptions on the transitions. Finally reference nets employ so-called *test arcs*, which allow concurrent reading access to a resource. These features enable us, on the one hand, to use reference nets as a full-fledged programming language to develop software systems. But on the other hand classical verification algorithms can no longer be directly applied to reference net systems. Adaptions and abstractions have to be undertaken in order to make verification available in the context of Paose. Our current work constitutes the first steps in this endeavour.

Verification of a software system is especially important, when certain properties of the system need to be guaranteed after deployment. Aspects, like deadlock-freeness or boundedness, are often not just simple inconveniences, but can endanger critical components. This is especially true in the case of the Herold project [17], which researches the application of agent-orientation in the domain of distributed network security. In this project we employ the Paose approach to develop a MAS for the management of network security components (e.g. firewalls), because a MAS is capable of capturing the distributed nature of the domain. If the deployed system contained deadlocks or interactions between agents failed to terminate the result could endanger the entire network under management by leaving it open to attacks. Because this has to be prevented under all circumstances, the need for verification support is clear and the integration of such functionality into Paose becomes a priority.

The goal of the particular research presented in this contribution is to enhance the tool support for Paose with regards to verification. The designated tool for Paose is Renew (**Re**ference **ne**t **w**orkshop, [13]), which was especially developed to simulate reference nets, although it is not restricted to this formalism. It has served as the build- and run-time environment for many software systems using reference nets [4, 23].

We have chosen to integrate an existing, external tool for verification, instead of developing this functionality from scratch. This tool is LoLA (Low Level Net Analyser, [21]). It provides the desired verification functionality in an efficient and accessible way. This contribution presents the technical and conceptual enhancements to Paose and Renew, especially the integration of LoLA.

Concerning related work, we researched other verification tools prior to choosing LoLA for the integration. These tools included Netlab (see [8, 18]) and Maria (see [15]). Netlab offers functionality to edit, simulate and analyse place/transition nets. Analysis functions include computation of S- and T-invariants and the reachability/coverability graphs of a given net. Maria (The Modular Reachability Analyzer) employs algebraic system nets. For verification functionality it features reachability analysis and LTL model checking. Both Netlab and Maria are sophisticated verification tools, which provide extensive techniques. In the end, we chose LoLA since it provides comprehensive verification functionality, is quite efficient and its integration is straight-forward.

The paper is structured in the following way. Section 2 discusses the technical aspects of the integration of LoLA into Renew. Section 3 then discusses, how this integration can conceptually be used within Paose. Finally Section 4 summarises and concludes the paper.

2 Technical Integration

Renew, the **Re**ference **Ne**t **W**orkshop, is a high-level Petri net editor and simulator created at the University of Hamburg. It is described in [13] and was developed alongside the reference net formalism [12]. It supports multiple formalisms, including the aforementioned reference nets, workflow nets [7, 22] and place/transition nets. With regards to verification it provides a type and syntax check when editing nets and some effort has been made to support external verification tools (e.g. [16]).

LoLA, the **Lo**w **L**evel Petri Net **A**nalyser, is a verification tool for place/transition and coloured nets described, for example, in [21]. It is based on state space exploration exploiting state-of-the-art reduction techniques and has a high standing in industry and academia, see e.g. [6, 14, 10] for recent applications. Functionality includes the creation of useful additional information about nets, such as reachability and coverability graphs, and the verification of deadlock-freeness, reachability of markings, non-deadness of transitions, state predicates, CTL formulas and other properties.

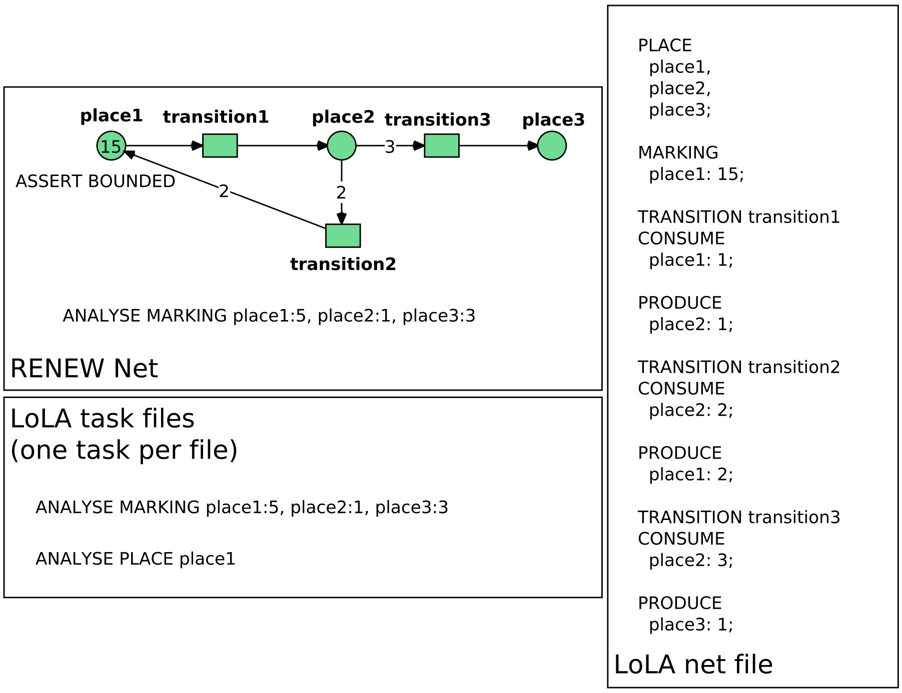
In order to use LoLA within the Renew tool, we implemented an export from Renew nets to the textual net representation that LoLA requires. Figure 1 gives an example of a simple Renew net and the corresponding textual LoLA net file, as generated by the presented Renew plugin.

Renew nets consist of an unsorted set of figure objects, each of which represents one net element[1]. The figures of a net can be accessed by means of a Java enumeration, which is processed to generate the representation LoLA understands. In order to do so, we have to identify the places, the transitions (together with their pre- and post-set) and the initial marking of the net[2]. These are extracted from the Renew net and written into a new file that complies with the LoLA syntax.

The export of Renew nets is restricted to those concepts that can be expressed within LoLA. Therefore we employ the following *projection*, which strips some semantic information from the reference net. We first need to ignore all inscriptions on places, transitions and arcs (type declarations, guards, synchronous channels, Java code and variables), except for names of places and transitions.

[1] Net elements in this context are not only transitions, places and tokens, but *everything* in the net drawing, including inscriptions, names and comments.

[2] We can restrict ourselves to only these three, because, for now, we are only using the P/T-net functionality of LoLA. However, we continue to work on mapping Renew's type declarations to LoLA's specification format for high-level nets, which allows the declaration of sorts and operations. These can later on be referred in the verification requests.



```
ANALYSE MARKING place1:5, place2:1, place3:3
```

```
ANALYSE MARKING place1:5, place2:1, place3:3

ANALYSE PLACE place1
```

```
PLACE
  place1,
  place2,
  place3;

MARKING
  place1: 15;

TRANSITION transition1
CONSUME
  place1: 1;

PRODUCE
  place2: 1;

TRANSITION transition2
CONSUME
  place2: 2;

PRODUCE
  place1: 2;

TRANSITION transition3
CONSUME
  place2: 3;

PRODUCE
  place3: 1;
```

Fig. 1. A RENEW net and the corresponding LoLA files.

Then we need to treat object and net tokens (i.e. tokens that represent Java objects or itself nets) as black tokens, wherever they appear. Finally we need to replace test arcs (which are bidirectional) with normal arcs leading to and from the elements involved.

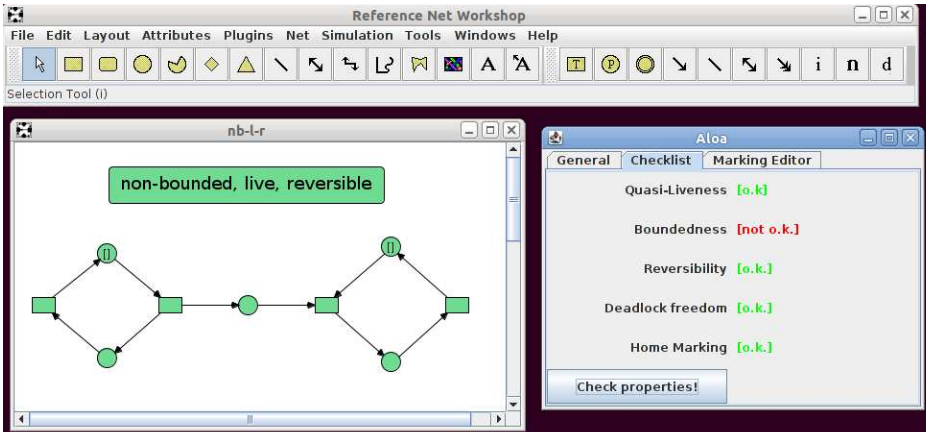
Additionally we have implemented the possibility to annotate verification tasks inside a RENEW net. The corresponding text figures are written into separate files, since LoLA requires them to be independent from the textual net representation[3].

Verification tasks for LoLA can be defined in two ways in the RENEW tool. They can either be added directly as unconnected text figures, using the LoLA Syntax (e.g. the "ANALYSE MARKING . . . " text in Figure 1, which corresponds to the task of checking if the specified marking is reachable) or, more comfortably, as a special inscription for the concerning element. For example, the inscription of the place *place1* in Figure 1 stating "ASSERT BOUNDED" is automatically translated into a verification task to check the boundedness of the place. The latter is more convenient for the user, since it does not require him to keep track of the names of the net elements for verification.

[3] It is also possible to add some verification tasks directly to the net file. Another viable alternative is to provide the tasks to LoLA as streams.

Providing LoLA functionality with the exported nets from within Renew is handled by calling LoLA externally and feeding the results back into Renew. There are other options how to integrate LoLA, however missing insight into the internal workings of LoLA we were not able to implement them yet. The work on this is still ongoing.

Now that we have outlined *how* LoLA was technically integrated into Renew we can proceed to discuss in *which ways* the functionality is used. Apart from simply being able to specify nets and manually exporting and verifying them, we have decided to provide a more convenient tool set for the Renew user to access the functionality. Therefore we implemented a plugin for Renew that offers the described export capabilities and aggregates some of the functionality offered by LoLA into a graphical view, which makes it comfortably accessible in the Renew context. The work on this plugin is still ongoing. Some features have already been implemented while others are considered work-in-progress.



Quasi-Liveness [o.k]

Boundedness [not o.k.]

Reversibility [o.k.]

Deadlock freedom [o.k.]

Home Marking [o.k.]

Fig. 2. Screenshot of checklist

On-the-fly check for transitions and places:
The plugin provides the means to check the transitions and places of a net currently being drawn. It will check if transitions are dead and places are bounded. Dead transitions and unbounded places will be marked in the net drawing, so the user can revise the relevant parts of the net.

Net "checklist":
The plugin offers a kind of checklist for a net. In this checklist the properties of quasi-liveness, reversibility, deadlock freedom, existence of home markings and boundedness are displayed. For now, for each of these properties an indicator is given (either positive or negative), but the checklist can be extended to supply additional information, such as concrete home markings

or witness paths. In Figure 2 we included a screenshot of the net checklist that shows the result we obtained for the net on the left.

Reachability/coverability graphs: *Work-in-progress*
The plugin will provide a window in which the reachability and coverability graphs of the net currently being drawn are displayed. This creation of the graphs is initiated manually by the user.

Marking editor: *Work-in-progress*
The plugin will provide an editor for entering a specific marking for the current net. The fields of the editor are updated to display all current places. The user can enter the tokens of each place and then check the marking for example for its reachability (from the initial marking or a second marking input into the plugin window), coverability or status as a home state. Feedback is immediately given to the user within the plugin window.

CTL formula editor: *Work-in-progress*
Similar to the marking editor, the plugin will provide the means to input a CTL formula for the current net, which can then be checked directly from within Renew.

3 Conceptual Integration into Paose approach

Applying Petri nets for software engineering (SE) is a common practice (see e.g. [19]). Nonetheless the Paose approach developed at the theoretical foundations group (TGI) in Hamburg is quite unique, in that it combines Petri nets with the paradigm of agent-orientation to build software systems. We first give a short introduction to the Paose approach and then discuss the role of verification. Finally we show how the results of the technical integration presented in this paper can be applied to the verification of multi-agent systems (MAS).

3.1 Petri net agents

Paose employs concepts from agent-orientation to structure software systems on four different levels, which together form the Mulan architecture (**Mul**ti-**a**gent **n**ets, see [20]). Agents are active, goal-oriented and autonomous software components that only communicate by means of asynchronous messages. They are situated in a logical environment, called platform, that allows for agent migration, facilitates message transport and provides a service registry. The behaviour of agents is autonomous, so unlike function calls, agents choose how to answer to a received message (if at all). For the concrete handling of a message the agent executes a protocol[4] from his knowledge base, which constitutes a repository of known behaviours and data.

Each of the four levels is realised with reference nets, with higher levels embedding the lower ones as net tokens. In this way a platform contains references to agent nets, which, in turn, hold references to protocol nets. Due to the

[4] The term does not refer to a communication protocol that encompasses several agents, but rather stands for a specification of a specific behaviour of an agent.

operational semantics of reference nets, a modelled multi-agent system can be simulated directly.

Being an exhaustive SE-approach PAOSE also encompasses techniques for the specification of interactions between agents, for the definition of ontologies and for the definition of agent roles and knowledge bases, all of which are fully supported by internally and externally developed tools. A model for the development process, as well as an agile method and facilities for the automatic generation of nets and documentation are available. However, we will not go into detail here, but refer the reader to [2] for a complete, detailed presentation of the PAOSE approach.

3.2 Role of Verification

Up until the research presented here, the PAOSE approach did not include formal verification, but instead relied on the sophisticated type and syntax checking offered by RENEW. Because the ontology of a multi-agent application is transformed into a Java type hierarchy, the RENEW tool can provide the developer with type checking for ontology objects. The same is true for the meta-ontology describing platforms, agents, services and messages among other MULAN-relevant objects.

An additional feature at least partially fills the role of verification. During the development process the multi-agent application can be simulated and inspected thoroughly, and can even be modified on the fly during a simulation run [5]. All these measures reduce the cost of development, as they make the process less error-prone.

This contribution addresses the afore-mentioned shortcoming of PAOSE, by providing "classical" Petri net verification. However, we tailor the functionality to the specific context of PAOSE. Due to the Turing completeness of reference nets [11] the properties of liveness and boundedness, among others, are undecidable and thus the problem of general verification has not been tackled before. We have come to realise that although no general verification can be achieved, interesting properties can be validated to some degree. To make reference nets suitable for verification with the LoLA plugin, we employ the projection described in Section 2, which replaces object and net tokens with black tokens and strips away certain inscriptions e.g. Java calls.

It needs to be examined what assertions can be made about the original net. One can observe that if a transition is activated in the original net, it is also activated in the projection[5]. On the other hand, if a transition in the projected net is not activated, it will not be activated under any circumstances in the original net. So we can conclude that if a deadlock occurs in the projected net, it also occurs in the original one. The opposite does not hold true however.

[5] There is a minor exception to this. In the case where several transitions are connected to the same place with a test arc, they can fire concurrently in the original net. Because test arcs are replaced by usual arcs, this no longer can happen in the projected net.

We will now detail how the LoLA plugin can play an important role in supporting the development of multi-agent systems by utilising the structure of Mulan nets for verification.

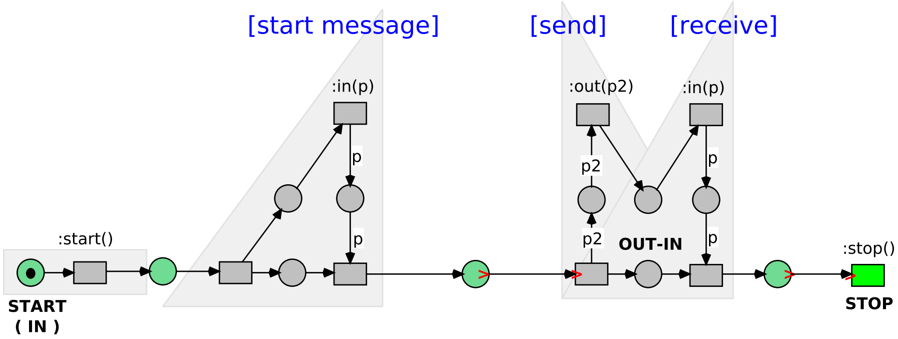


Fig. 3. A Mulan protocol net

3.3 Towards MAS verification

We claim that the reference nets used in the Paose context exhibit some structural features that make them well suited for verification. For this we have to take a more detailed look at the utilised net systems. Agent and platform nets are fixed, this means all agents and platforms share a common structure, while application specific functionality is added through protocol nets and the knowledge base (and application specific infrastructure). While agent and platform nets need to be live, it is essential for protocol nets to terminate, i.e. to reach a terminal state. Some groundwork on this topic has been presented in [9].

Figure 3 illustrates the typical form of a protocol net used in Mulan. Structurally speaking, protocol nets start and end with a transition, which synchronises with the embedding agent net. The initial synchronisation causes the newly instantiated protocol net to be put on a special place in the agent net, dedicated for ongoing conversations. Messages an agent receives can be directly passed to an ongoing conversation, if their type matches the type expected in the protocol net. The final synchronisation removes the instance of the protocol net from the conversation place, once it is finished (meaning it reached its terminal state). In this way, it is essentially a workflow net [22]. Using this perspective the closure of the workflow net would be provided by the agent, which is capable of re-instantiating the protocol/workflow.

While a conversation is ongoing, agents send and receive messages in their corresponding protocol nets that together form the conversation. The platform net is also involved in the conversation, because it routes the message to its destination. Deadlock-freeness and thus termination can only be verified under the assumption that the communication partners and the message transport service behave well. On the other hand, stripping the inscriptions from the protocol net, one can verify deadlock-freeness of the projected protocol net independently of its environment. Using the LoLA tool in combination with the existing type check we can enhance verification in cases where alternative behaviours are chosen according to message content (which is the standard case).

Some other verification tasks for multi-agent systems spring to mind and will be the focus of future work:

– *Do agents possess matching protocols so that they can communicate?*
  Given two, or more, agents this task can determine if they possess matching protocols. This means that an outgoing message in one agent's protocol has to correspond to an incoming message in another agent's protocol. In terms of practical implementation this boils down to matching the outgoing message as a type to the incoming message as a type, possibly by merging the protocol nets.
– *Does an agent have required protocols to be embedded on a platform?*
  While similar to the previous task, this task is a bit more general. Instead of explicitly matching a number of agents for communication, this task determines if an agent possesses the required functionality (i.e. protocols) needed to be active on a specific platform. This task is more comprehensive and complex, since it addresses further aspects concerning functionality, not just communication. In practical terms, more nets have to be matched and certain assumptions about the content of protocols (for example through type matching) have to be verified.
– *Is the overall communication structure deadlock-free?*
  Another related task is to check the overall interaction and communication structure of the multi-agent application for deadlocks. Taking the agent interactions into account we could generate a simplified net version of the overall communication, assuming that the internal workings of the agents are correct and do not deadlock. The check would then again be confined to the messages and their types, but instead of having a particular set of agents and protocols in the focus, it would examine the overall multi-agent application.

These last points dealing with the orchestration of protocol nets illustrates the intimate relation between protocol nets in the Mulan architecture and open nets of [24]. Open nets are used in the context of service synthesis, composition and orchestration and are supported by an exhaustive tool suite[6] that includes LoLA. Therefore we started our investigation into verification tools with LoLA and plan to extend it to other tools from the suite.

4 Conclusion

We have presented a way to make use of an existing tool as a plugin for Renew and how the Paose approach profits from this integration. The technical real-

[6] Available online as open source from http://service-technology.org/.

isation was easily achieved due to the flexible plugin mechanisms provided by Renew. So far we have used the tool to check for very basic properties, but as was advocated in Section 3, we are working on applying it to multi-agent systems. Conceptually we gain enhanced tool support for Paose, making it easier to develop functioning-as-designed multi-agent applications.

Concerning future work, the efforts of enhancing Renew and Paose with the help of LoLA are still ongoing. We are looking at further ways of extending the functionality of the LoLA plugin for Renew and are researching additional areas of our multi-agent systems for verification, as was outlined in Section 3.3. As for the technical aspects, a promising avenue of work is to also incorporate the support for sorts and operations offered by LoLA into the plugin. This will improve the modelling effort for verification and, in turn, make using LoLA within Renew more comfortable.

From a practical point of view, we will use the results we have obtained within the Herold research project[7]. Herold deals with distributed network security and the management of network security components (see [1] for more information). The complex processes within the project, coupled with the critical nature of the application domain, require the use of verification techniques in order to ensure correct execution of the produced multi-agent system. The LoLA plugin will play an invaluable role for the verification aspects of the project.

References

1. Simon Adameit, Tobias Betz, Lawrence Cabac, Florian Hars, Marcin Hewelt, Michael Köhler-Bußmeier, Daniel Moldt, Dimitri Popov, Jose Quenum, Axel Theilmann, Thomas Wagner, Timo Warns, and Lars Wüstenberg. Herold - agent-oriented, policy-based network security management. In *Future Security, 5th Security Research Conference, Berlin*, 2010.
2. Lawrence Cabac. *Modeling Petri Net-Based Multi-Agent Applications*, volume 5 of *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2010.
3. Lawrence Cabac, Till Dörges, Michael Duvigneau, Christine Reese, and Matthias Wester-Ebbinghaus. Application development with Mulan. In Daniel Moldt, Fabrice Kordon, Kees van Hee, José-Manuel Colom, and Rémi Bastide, editors, *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'07)*, pages 145–159, Siedlce, Poland, June 2007. Akademia Podlaska.
4. Lawrence Cabac, Michael Duvigneau, Michael Köhler, Kolja Lehmann, Daniel Moldt, Sven Offermann, Jan Ortmann, Christine Reese, Heiko Rölke, and Volker Tell. PAOSE Settler demo. In *First Workshop on High-Level Petri Nets and Distributed Systems (PNDS) 2005*, Vogt-Kölln Str. 30, D-22527 Hamburg, March 2005. University of Hamburg, Department of Computer Science.
5. Lawrence Cabac, Daniel Moldt, and Jan Schlüter. Adding runtime net manipulation features to MulanViewer. In *15. Workshop Algorithmen und Werkzeuge für Petrinetze, AWPN'08*, volume 380 of *CEUR Workshop Proceedings*, pages 87–92. Universität Rostock, September 2008.

[7] The Herold project is funded by the German Federal Government, through its Ministry of Education and Research (Grant No. 01BS0901).

6. Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming BPEL to Petri nets. In Wil M. P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Proceedings of the Third International Conference on Business Process Management (BPM 2005)*, volume 3649 of *Lecture Notes in Computer Science*, pages 220–235, Nancy, France, September 2005. Springer-Verlag.
7. Thomas Jacob. Implementierung einer sicheren und rollenbasierten Workflowmanagement-Komponente für ein Petrinetzwerkzeug. Diploma thesis, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg, 2002.
8. Stephan Kleuker. *Formale Modelle der Softwareentwicklung. Model-Checking, Verifikation, Analyse und Simulation.* Vieweg+Teubner, Wiesbaden, 2009.
9. Michael Köhler, Daniel Moldt, and Heiko Rölke. Liveness preserving composition of behaviour protocols for Petri net agents. Report of the research program: Act in Social Contexts FBI-HH-M-316/02, University of Hamburg, Department of Computer Science, June 2002.
10. Milos Krstic, Eckhard Grass, and Christian Stahl. Request-driven GALS technique for wireless communication system. In *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 76–85, Washington, DC, USA, 2005. IEEE Computer Society.
11. Olaf Kummer. Undecidability in object-oriented Petri nets. *Petri Net Newsletter*, 59:18–23, 2000.
12. Olaf Kummer. *Referenznetze.* Logos Verlag, Berlin, 2002.
13. Olaf Kummer, Frank Wienberg, Michael Duvigneau, Michael Köhler, Daniel Moldt, and Heiko Rölke. Renew – the Reference Net Workshop. In Eric Veerbeek, editor, *Tool Demonstrations. 24th International Conference on Application and Theory of Petri Nets (ATPN 2003). International Conference on Business Process Management (BPM 2003)*, pages 99–102. Department of Technology Management, Technische Universiteit Eindhoven, Beta Research School for Operations Management and Logistics, June 2003.
14. Niels Lohmann, Oliver Kopp, Frank Leymann, and Wolfgang Reisig. Analyzing BPEL4Chor: Verification and participant synthesis. In Marlon Dumas and Reiko Heckel, editors, *Web Services and Formal Methods*, volume 4937 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2008.
15. Marko Mäkelä. Maria: Modular reachability analyser for algebraic system nets. In *Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets*, ICATPN '02, pages 434–444. Springer-Verlag, 2002.
16. Marco Mascheroni, Thomas Wagner, and Lars Wüstenberg. Verifying reference nets by means of hypernets: A plugin for Renew. In Michael Duvigneau and Daniel Moldt, editors, *Proceedings of the International Workshop on Petri Nets and Software Engineering, PNSE'10, Braga, Portugal*, number FBI-HH-B-294/10 in Bericht, pages 39–54, Vogt-Kölln Str. 30, D-22527 Hamburg, June 2010. University of Hamburg, Department of Informatics.
17. Daniel Moldt, Michael Köhler-Bußmeier, Axel Theilmann, Simon Adameit, Tobias Betz, Lawrence Cabac, Florian Hars, Marcin Hewelt, Dimitri Popov, José Quenum, Thomas Wagner, Timo Warns, and Lars Wüstenberg. Modelling distributed network security in a Petri net and agent-based approach. In Jürgen Dix and Witteveen Cees, editors, *Multiagent System Technologies. 8th German Conference, MATES 2010,Leipzig, Germany, September 27-28, 2010. Proceedings*, volume 6251 of *Lecture Notes in Artificial Intelligence*, pages 209–220, Berlin Heidelberg New York, September 2010. Springer-Verlag.

18. Philipp Orth and Dirk Abe. Rapid Control Prototyping petrinetzbasierter Steuerungen mit dem Tool NETLAB. *at-Automatisierungstechnik*, 54, Issue: 5:222–227, 2006.
19. Wolfgang Reisig. Petri nets in software engineering. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *Lecture Notes in Computer Science*, pages 62–96. Springer, 1987.
20. Heiko Rölke. *Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen*, volume 2 of *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2004.
21. Karsten Schmidt. LoLA: A Low Level Analyser. In Mogens Nielsen and Dan Simpson, editors, *Application and Theory of Petri Nets 2000: 21st International Conference, ICATPN 2000, Aarhus, Denmark, Proceedings*, volume 1825 of *Lecture Notes in Computer Science*, pages 465–474. Springer-Verlag, June 2000.
22. Wil M. P. van der Aalst. Verification of workflow nets. In Pierre Azéma and Gianfranco Balbo, editors, *ICATPN*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426. Springer, 1997.
23. Thomas Wagner. Prototypische Realisierung einer Integration von Agenten und Workflows. Diploma thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2009.
24. Karsten Wolf. Does my service have partners? *Transactions on Petri Nets and Other Models of Concurrency*, 2:152–171, 2009.

Transitions as Transactions [⋆]

Shengyuan Wang, Weiyi Wu, Yao Zhang, and Yuan Dong

Department of Computer Science and Technology
Tsinghua University, Beijing, 100084, China
{wwssyy}@tsinghua.edu.cn
http://soft.cs.tsinghua.edu.cn/~wang

Abstract. As newly developed transactional memory technology has received significant attention as a way to dramatically simplify shared-memory concurrent programming, user-level transactional concurrent programming models have become a very interesting topic in the programming community. However, the fact is that, in existing transactional concurrent programming models, user-level mechanisms have not been well developed. The dilemma is how to make a balance between the performance and correctness of a program. Explicit concurrency among cooperative transactions can undoubtedly decrease the rate of conflicts and improve the performance, but it is harmful to the correctness. In this paper, a transactional concurrent programming approach, based on Petri nets, is presented, which can easily specify concurrency among transactions and do not aggravate programmers remarkably in writing correct transactional concurrent programs. In this approach, a special Petri net system with transition markings is developed. Although such a Petri net system is not defined conventionally, it is shown that its behavior can be simulated through a conventional net, so existing analysis and verification approaches for usual Petri nets can be applied indirectly.

Key words: Concurrent Programming; Transactional Memory; Petri Nets

1 Introduction

Transactional memory mechanism has recently received significant attention as a way to dramatically simplify memory-sharing concurrent programming, in which mutual exclusion and synchronization can be constructed without using any locks [1]. For convenience,a concurrent programming model based on transactional memory mechanism is called a *transactional concurrent programming model* in this paper.

In existing user-level transactional concurrent programming models, there are two major solutions. One of them is to use directly some API's for transactional memory mechanism, which may be implemented by hardware, software or hybrid. For example, programmers can write transactional concurrent programs

[⋆] Supported by the National Natural Science Foundation of China under grant No. 90818019

in Java together with the library DSTM2 [2], or in C together with the library TL2-x86 [3]. The advantage of this solution is that one can use existing common languages without changing their compilers, but programmers have to use non-structural library functions carefully.

Another solution is to extend conventional programming languages with some transactional features, such as atomic statement-blocks, as in some new languages Fortress [4], X10 [5], Chapel [6], etc. In this solution, it is easier for programmers to write correct transactional concurrent programs, however, an appropriate compiler must be provided.

To develop a user-level transactional concurrent programming mechanisms, one dilemma is how to make a balance between the performance and correctness of a program. On the one hand, to write an efficient program in the transactional programming paradigm, it still needs programmers' wisdom to build the explicit parallelism among cooperative transactions, in order to decrease the rate of conflicts. On the other hand, however, explicit parallelism is harmful to the correctness of a program, while one of the initial intents of the transactional memory mechanism is to alleviate the burden for a programmer to write concurrent programs.

There have been some contributions in the literature to introduce transactions into existing concurrent programming model. For example, a CCR-based transactional concurrent programming model was proposed by T. Harris and K. Fraser [7], and Baek et al extend the API's of OpenMP [8] to OpenTM [9]. Unfortunately, these approaches still have the usual drawbacks of concurrent programs, that is, not easy to write and not easy to verify.

As well known, Petri nets [10] are useful tools in the specification and verification of concurrent applications. With true concurrency dynamical semantics, a Petri net system has a good opportunity to become a realistic part of a concurrent program for multi-core or multi-thread architecture. In this paper, we present informally a transactional concurrent programming mechanism based on a Petri net, in order to specify concurrency among transactions explicitly while not to aggravate programmers remarkably in writing correct concurrent programs.

Fig.1 shows a simple example described in a typical transactional concurrent program structure, where an atomic statement-block declares a transactional region, and $fork1$, $fork2$, $fork3$ and $cake\_c$ are shared objects among cooperative transactions.

In the Petri net system shown in Fig.2, to be explained in more details, each of the transitions declares a transactional region, and $fork1$, $fork2$, $fork3$ and $cake\_c$ are shared objects among three cooperative transactions. Since the accesses of $fork1$, $fork2$, and $fork3$ will never conflict, the rate of access conflicts among transactions is decreased, compared to the program in Fig.1.

Extremely, we can protect all shared objects by the Petri net system, corresponding to the so-called conservative concurrency control. However, if the number of shared objects increases dramatically, the net system may get too big in size. Fortunately, we can leave some shared objects to be protected by the

```
int fork1 = 0, fork2 = 0, fork3 = 0;
int cake_c = 12;


thread ph1:
  while ( true ) {
    atomic {
      read fork1 ;
      read fork2 ;
      write fork1+1  to fork1 ;
      write fork2+1  to fork2 ;
    }
  }
```

```
thread ph2:
  while ( true ) {
    atomic {
      read fork2 ;
      read fork3 ;
      write fork2+1  to fork2 ;
      write fork3+1  to fork3 ;
      if ( fork2 mod 10 == 0 ) {
        read cake_c ;
        write cake_c-1  to cake_c ;
      }
    }
  }
```

```
thread ph3:
  while ( true ) {
    atomic {
      read fork3 ;
      read fork1 ;
      write fork3+1  to fork3 ;
      write fork1+1  to fork1 ;
      if ( fork1 mod 20 == 0 ) {
        read cake_c ;
        write cake_c-1  to cake_c ;
      }
    }
  }
```

Fig. 1. A simple example of typical transactional concurrent program structure

transactional memory mechanism, if the probability of access conflicts for those shared objects is not that big. For example, we have not made the accesses to *cake_ c* protected by the net system in Fig.2.

We call a shared object to be *critical* or *non-critical* according to its probability of access conflicts. So in the transactional concurrent programming approach suggested in this paper, programmers are encouraged to implement the protection of critical shared objects through Petri net systems, and to leave the non-critical shared objects to be protected automatically by the transactional memory system. In the example shown in Fig.1 and Fig.2, the shared object *cake_ c* is less frequently accessed than *fork*i's, hence, it is assumed that *fork*i's be critical shared objects among *ph*i's, and *cake_ c* be the non-critical shared object among them.

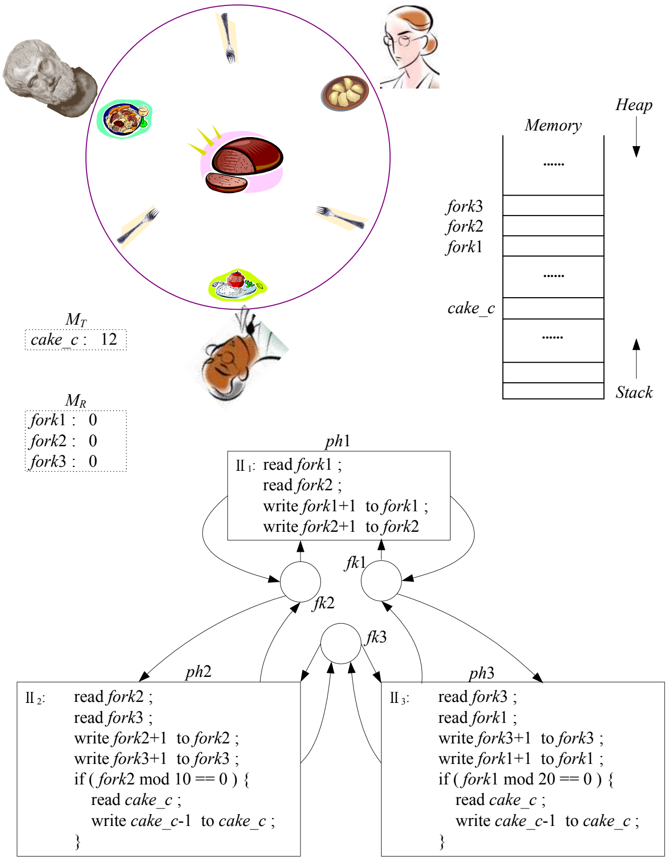
The Petri net model we use is a special colored Petri net model [11], called *resource nets*, which guarantee the access consistency for shared objects. The semantics for a transactional memory mechanism is inspired by the implementation of DSTM2 [2].

The rest of the paper is organized as follows. In Section 2, we make some informal interpretation to the Petri net model, *resource nets*. Further in Section 3, the behavior simulation of a *resource net system* is discussed. Then in Section 4, the program model is briefly presented. Section 5 shows a sample user-level transactional concurrent programming tool, where the concept *resource nets* is applied. Finally, Section 6 gives some remarks and the future work.

2 The Net Model

As stated above, a transactional concurrent program can access two classes of shared objects, *critical* or *non-critical* ones. We use *resource variables* to access critical shared objects, and *global variables* to access non-critical shared objects. In the following, the set of *resource variables* is denoted by V_R, and the set of *global variables* is denoted by V_T.

A *resource net system* is a special colored Petri net system $\mathcal{N} = (\mathbb{P}, \mathbb{T}, \mathbb{A}, W, m_0, M_F)$, where



```
II 1: read fork1 ;
      read fork2 ;
      write fork1+1  to fork1 ;
      write fork2+1  to fork2
```

```
II 2:     read fork2 ;
          read fork3 ;
          write fork2+1  to fork2 ;
          write fork3+1  to fork3 ;
          if ( fork2 mod 10 == 0 ) {
             read cake_c ;
             write cake_c-1  to cake_c ;
          }
```

```
II 3:     read fork3 ;
          read fork1 ;
          write fork3+1  to fork3 ;
          write fork1+1  to fork1 ;
          if ( fork1 mod 20 == 0 ) {
             read cake_c ;
             write cake_c-1  to cake_c ;
          }
```

Fig. 2. A resource net system for Dining Philosophers

- $\mathbb{P} \subseteq \{\rho_k \mid k \in \mathbb{N}\}$, and $\mathbb{T} \subseteq \{(\tau_k, \mathbb{I}_k) \mid k \in \mathbb{N}\}$ are the set of places and the set of transitions respectively.
- $\mathbb{A} = (\mathbb{P} \times \mathbb{T}) \cup (\mathbb{T} \times \mathbb{P})$ is the set of arcs.
- $W : \mathbb{A} \to \{\mathbb{S} \mid \mathbb{S} \subseteq V_R\}$ is the inscription function.
- $m_0 \in \mathit{Marking}$ is the initial marking, where $\mathit{Marking} = \{m \mid m : (\mathbb{P} \cup \mathbb{T}) \to \{\mathbb{S} \mid \mathbb{S} \subseteq V_R\}\}$.
- $M_F \subseteq \mathit{Marking}$ is the set of final markings.

A resource net system $\mathcal{N} = (\mathbb{P}, \mathbb{T}, \mathbb{A}, W, m_0, M_F)$ has the following features:

- For each transition $(\tau_k, \mathbb{I}_k) \in \mathbb{T}$, a command sequence $\mathbb{I}_k$ is attached. When a transition $(\tau_k, \mathbb{I}_k)$ is fired, it starts a transaction for the command sequence $\mathbb{I}_k$. A command in $\mathbb{I}_k$ can access shared variables in $V_R \cup V_T$, and the variables local to τ_k.
- A transition can hold a token while its transaction is executing, and the token does not return to the net system until the computation is committed or aborted. So we extend the definition of marking with transition markings.

- It is possible that M_F is empty, which is the usual case in conventional Petri net systems..

It is worth to noting that variables in V_R should be disjoined in locations with each other, which are usually implemented by the compiler.

2.1 An Example

***Example* 1** Fig.2 shows a transactional concurrent program with $V_R = \{\ \mathit{fork}1, \mathit{fork}2, \mathit{fork}3\}$ and $V_T = \{\ \mathit{cake\_\ c}\ \}$. The *resource net system* $\mathcal{N} = (\mathbb{P}, \mathbb{T}, \mathbb{A}, W, m_0, M_F)$, where

- $\mathbb{P} = \{fk1, fk2, fk3\}$.
- $\mathbb{T} = \{tr1, tr2, tr3\}$, where $tr1 = (ph1, \mathbb{I}_1)$, $tr2 = (ph2, \mathbb{I}_2)$, and $tr3 = (ph3, \mathbb{I}_3)$, where $\mathbb{I}_1$, $\mathbb{I}_2$ and $\mathbb{I}_3$ are command sequences attached to transitions ph1, ph2, and ph3 respectively, as is illustrated in Fig.2.
- $\mathbb{A} = \{(tr1,fk1), (tr1,fk2), (tr2,fk2), (tr2,fk3), (tr3,fk3), (tr3,fk1), (fk1,tr1), (fk1, tr3), (fk2, tr2), (fk2, tr1), (fk3, tr3), (fk3, tr2)\}$.

- W is defined by:
  $W(fk1, tr1) = W(tr1, fk1) = W(fk1, tr3) = W(tr3, fk1) = \{\mathit{fork}1\}$, $W(fk2, tr2) = W(tr2, fk2) = W(fk2, tr1) = W(tr1, fk2) = \{\mathit{fork}2\}$, $W(fk3, tr2) = W(tr2, fk3) = W(fk3, tr3) = W(tr3, fk3) = \{\mathit{fork}3\}$.
- $m_0 \in \mathit{Marking}$ is defined by:
  $m_0(fk1) = \{\mathit{fork}1\}$, $m_0(fk2) = \{\mathit{fork}2\}$, $m_0(fk3) = \{\mathit{fork}3\}$, and $m_0(\tau) = \emptyset$ for $\tau = tr1, tr2, tr3$.
- $M_F = \emptyset$.

2.2 Well-Formed Resource Net Systems

A resource net system $\mathcal{N} = (\mathbb{P}, \mathbb{T}, \mathbb{A}, W, m_0, M_F)$ is *well-formed*, if

- $\mathbb{P} \cap \mathbb{T} = \emptyset$.
- $\forall \rho \in \mathbb{P}.\forall v_1, v_2 \in m_0(\rho).(v_1 \neq v_2)$, that is, at the initial marking, all tokens owned by a place are corresponding to different resource variables.
- $\forall \rho_1, \rho_2 \in \mathbb{P}.\forall v_1 \forall v_2.(\rho_1 \neq \rho_2 \wedge v_1 \in m_0(\rho_1) \wedge v_2 \in m_0(\rho_2) \rightarrow v_1 \neq v_2)$, that is, at the initial marking, tokens owned by different places have disjoint resource variables.
- $\forall \tau \in \mathbb{T}.(m_0(\tau) = \emptyset)$, that is, at the initial marking, every transition contains no tokens.
- $\forall m \in M_F.\forall \tau \in \mathbb{T}.(m(\tau) = \emptyset)$, that is, at each of final markings, every transition will not contain any tokens.
- $\forall \tau \in \mathbb{T}.\forall \rho_1, \rho_2 \in \bullet\tau.\forall v_1 \forall v_2.(\rho_1 \neq \rho_2 \wedge v_1 \in W(\tau, \rho_1) \wedge v_2 \in W(\tau, \rho_2) \rightarrow v_1 \neq v_2)$, that is, all sets of resource variables on the incoming arcs of the same transition are disjoined with each other.

- $\forall \tau \in \mathbb{T}.\forall \rho_1, \rho_2 \in \tau \bullet .\forall v_1 \forall v_2.(\rho_1 \neq \rho_2 \wedge v_1 \in W(\rho_1, \tau) \wedge v_2 \in W(\rho_2, \tau) \rightarrow v_1 \neq v_2)$, that is, all sets of memory blocks on the outgoing arcs of the same transition are disjoined with each other.
- $\forall \tau \in \mathbb{T}.\forall \rho \in \tau \bullet .(W(\tau, \rho) \subseteq \bigcup_{\rho' \in \bullet\tau}(W((\rho', \tau)))$, that is, no extra shared objects be produced within a transaction associated to each of transitions. For simplification,in this paper, we don't consider the dynamic memory allocation for a shared object within a transaction.

In the above,the pre-set and post-set of a transition $\tau \in \mathbb{T}$ or a place $\rho \in \mathbb{P}$ are used, which are defined as usual: $\bullet\tau = \{\rho \mid (\rho, \tau) \in \mathbb{A}\}$, $\tau\bullet = \{\rho \mid (\tau, \rho) \in \mathbb{A}\}$, $\bullet\rho = \{\tau \mid (\tau, \rho) \in \mathbb{A}\}$, and $\rho\bullet = \{\tau \mid (\rho, \tau) \in \mathbb{A}\}$.

***Example* 2** It is easy to show that the resource net system in Example 1 is well-formed.

2.3 Execution Semantics

To show the execution semantics of a resource net system, we define *TranState* = $\{$*blocked*, *active*, *aborted*, *committed* $\}$, consisting of 4 transaction states of a transition. At the initial marking, every transition has the state *blocked*.

Entering a Transition Whenever a transition τ in the resource net system is in state *blocked*, and the firing condition for τ is satisfied under the current marking m, that is, $\forall \rho \in \bullet\tau.(m(\rho) \supseteq W(\rho, \tau))$, and in the same time, the current marking is not a final state, that is, $m \notin M_F$, the system can enter the transition such that a transaction is started and the transition gets to hold tokens. When it occurs, the state of the τ will be *active*.

Execution of a Command Sequence Whenever a transition τ in the resource net system is in state *active*, and the next command in its remained command sequence is c, the transition can make a progress by executing the command c. We need several separate rules respectively for several cases:

(1) If c reads or writes to a global variable which has been written most recently by some other transition but τ, a *read/write confliction* occurs, and τ will be in the state *aborted*.

(2) If the execution of c has no *read/write confliction* and c has no *write* operation to any global variables, the transition will keep in state *active*.

(3) If the execution of c has no *read/write confliction* and c has a *write* operation to some global variable x, the transition will keep in state *active*, while the system will record τ to be the transition that most recently written to x.

Ready to Commit a Transaction Whenever a transition τ in the resource net system is in state *active*, and there is no next command in its remained command sequence, the system can make a progress to change the state of τ from *active* to *committed*, meaning that the transaction associated to τ is ready to commit.

Committing a Transaction Whenever a transition τ in the resource net system is in state *committed*, the system can make a progress to commit the transaction associated to τ, changing the state of τ from *committed* to *blocked*.

Aborting a Transaction Whenever a transition τ in the resource net system is in state *aborted*, the system can make a progress to return tokens to places in the pre-set of τ, changing the state of τ from *aborted* to *blocked.*

Let $\mathcal{N} = (\mathbb{P}, \mathbb{T}, \mathbb{A}, W, m_0, M_F)$ be a well-formed resource net system, it is easy to show that for any reachable marking m from the initial marking m_0, the following two properties are satisfied

- $\forall x \in \mathbb{P} \cup \mathbb{T}.\forall v_1, v_2 \in m(x).(v_1 \neq v_2)$,that is, at the marking m, all tokens owned by a place or a transition are corresponding to different resource variables.
- $\forall x_1, x_2 \in \mathbb{P} \cup \mathbb{T}.\forall v_1 \forall v_2.(x_1 \neq x_2 \wedge v_1 \in m(x_1) \wedge v_2 \in m(x_2) \rightarrow v_1 \neq v_2)$, that is, at the marking m, all tokens owned by different places or transitions have disjoint resource variables.

***Example* 3** Since the resource net system $\mathcal{N} = (\mathbb{P}, \mathbb{T}, \mathbb{A}, W, m_0, M_F)$ in Example 1 is well-formed. So the above two properties will keep in a well-formed program state during its execution.

3 Behavior Simulation of a Resource Net System

In this section, it will be shown that a well-formed resource net system $\mathcal{N}$ can be reduced to an usual colored Petri net system *desugar*($\mathcal{N}$) so that the behavior of $\mathcal{N}$ can be simulated by *desugar*($\mathcal{N}$), which can be analyzed and verified by using existing approaches in the Petri net community.

The transformation from $\mathcal{N}$ to *desugar*($\mathcal{N}$) is called *desugaring*. Before and after desugaring, the change of net structure can be illustrated by Fig.3.

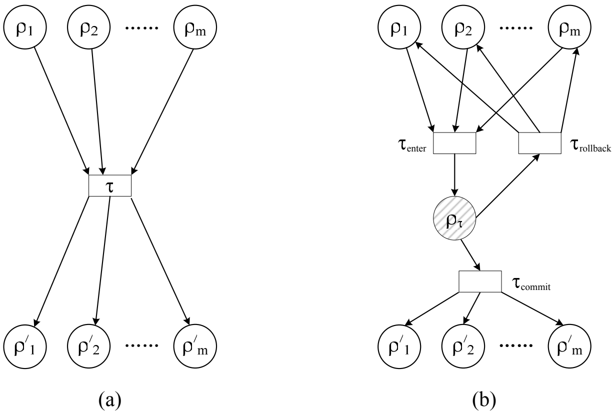


Fig. 3. Net structure before and after desugaring

***Example* 4** Consider the well-formed resource net system $\mathcal{N} = (\mathbb{P}, \mathbb{T}, \mathbb{A}, W, m_0, M_F)$ in Example 1. *desugar*($\mathcal{N}$) $= (\mathbb{P}', \mathbb{T}', \mathbb{A}', W', m_0', M_F')$ can be depicted in Fig.4, where

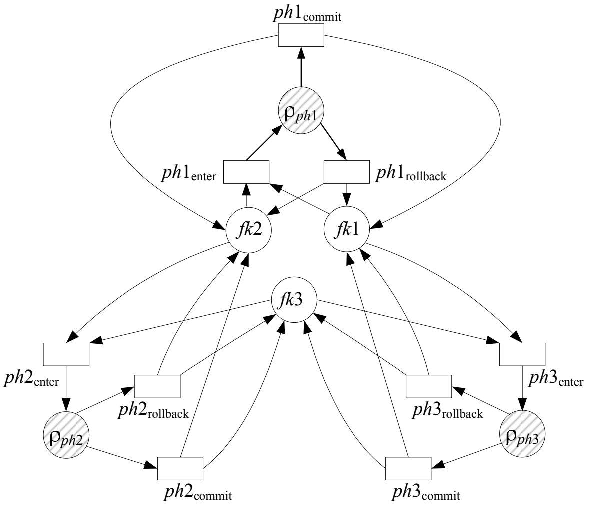


Fig. 4. Example of a desugaring net system

- $\mathbb{P}' = \{fk1, fk2, fk3, \rho_{ph1}, \rho_{ph2}, \rho_{ph3}\}$.
- $\mathbb{T}' = \{ph1_{enter}, ph2_{enter}, ph3_{enter}, ..., ph3_{commit}\}$
- $\mathbb{A}' = \{(ph1_{enter}, \rho_{ph1}), (ph2_{enter}, \rho_{ph2}), (ph3_{enter}, \rho_{ph3}),$
  $...,$
  $(fk1, ph1_{enter}), (fk1, ph3_{enter}), ...,$
  $(ph1_{rollback}, fk1), (ph3_{rollback}, fk1), ...,$
  $(ph1_{commit}, fk1), (ph3_{commit}, fk1), ...,$
  $..., (ph3_{commit}, fk3)\}$.
- The definition of W' is illustrated in Table 1 (partly).
- m'_0 is defined by $m'_0(\rho_{ph1}) = m'_0(\rho_{ph2}) = m'_0(\rho_{ph3}) = \emptyset$, $m'_0(fk1) = \{\{fork1\}\}$, $m'_0(fk2) = \{\{fork2\}\}$, and $m'_0(fk3) = \{\{fork3\}\}$.
- $M'_F = \emptyset$.

It is not difficult to establish a behavior simulation relation between $\mathcal{N}$ and *desugar*($\mathcal{N}$), and show that many behavior properties, including *deadlock-freeness*, for $\mathcal{N}$ can be verified indirectly by verifying those for *desugar*($\mathcal{N}$). For example, it is easy to verify that the usual Petri net system *desugar*($\mathcal{N}$) in Example 4 is deadlock-free, so we can conclude that the resource net system $\mathcal{N}$ is also deadlock-free.

It is worth to noting that the execution semantics can guarantee behaviour consistence between $\mathcal{N}$ and *desugar*($\mathcal{N}$). For every transition τ in $\mathcal{N}$ and ρ_τ in *desugar*($\mathcal{N}$) as illustrated in Fig.3, we have

- If τ is in state *blocked*, there no token in ρ_τ, and τ_{enter} is enabled. If τ_{enter} fires, τ will be in state *active* at the same time.
- If τ is in state *active*, nether τ_{commit} or $\tau_{rollback}$ will fire though there exist tokens in ρ_τ.
- If τ is in state *committed*, τ_{commit} is enabled.
- If τ is in state *aborted*, $\tau_{rollback}$ is enabled.
- τ is in state *active*, *committed*, or *aborted*, iff there no token in ρ_τ.

Table 1. $W' : \mathbb{A}' \to \{\mathbb{L} \mid \mathbb{L} \subseteq 2^{Label}\}$

if $a =$	then $W'(a) =$
$(ph1_{enter}, \rho_{ph1})$	$\{\{fork1\}, \{fork2\}\}$
$(ph2_{enter}, \rho_{ph2})$	$\{\{fork2\}, \{fork3\}\}$
$(ph3_{enter}, \rho_{ph3})$	$\{\{fork3\}, \{fork1\}\}$
...	...
$(fk1, ph1_{enter})$	$\{\{fork1\}\}$
$(fk1, ph3_{enter})$	$\{\{fork1\}\}$
...	...
$(ph1_{rollback}, fk1)$	$\{\{fork1\}\}$
$(ph3_{rollback}, fk1)$	$\{\{fork1\}\}$
...	...
$(ph1_{commit}, fk1)$	$\{\{fork1\}\}$
$(ph3_{commit}, fk1)$	$\{\{fork1\}\}$
...	...
$(ph3_{commit}, fk3)$	$\{\{fork3\}\}$

For the sake of limited space, in this paper, we have not formally defined the desugaring and the behavior simulation relation.

4 The Program Model

In the transactional concurrent programming approach of this paper, a program consists of a set of Petri net systems, which are protected parts in the system, and a set of unprotected threads which contains an initial thread identified *root* and other unprotected threads. Resource variables can only be accessed within protected parts. At the beginning, the thread *root* is initialized to execute at the level which we call *top level.*

A set of Petri net systems can spawn outside a Petri net system, initialized with new allocated resource variables or their references. When a transition in a Petri net system is fired, it becomes a (transactional) transition thread, which will eventually commit, or rollback due to conflicts to access the shared memory.

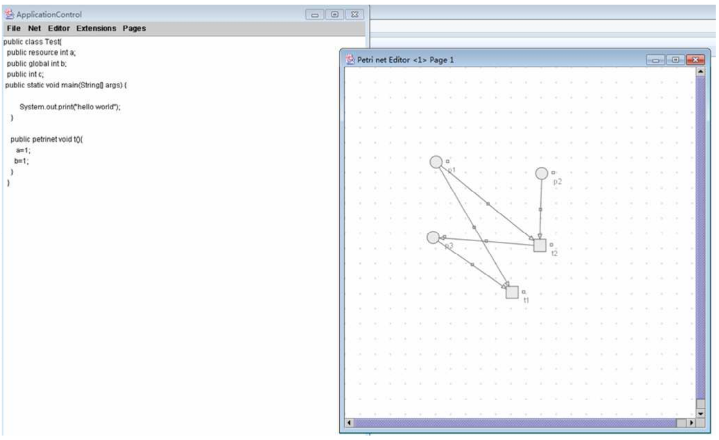
An unprotected thread except for the thread *root* can be spawned outside a Petri net system.

The program ends if all the Petri net systems achieve one of their final states, and in the same time all the unprotected threads execute to the end.

5 A sample user-level transactional concurrent programming tool

A sample user-level transactional concurrent programming tool has been developing in our lab, based on available software sources, DSTM2 [2], PNK [13] and GJC [14]. In the programming model of this sample programming tool, a program consists of a set of Petri net systems, corresponding to resource net systems in this paper, and other part written in Java Language.

A simple visual IDE for this programming model has been developing.



```
public class Test{
 public resource int a;
 public global int b;
 public int c;
 public static void main(String[] args) {

      System.out.print("hello world");
  }

  public petrinet void t0{
    a=1;
   b=1;
  }
 }
```

Fig. 5. A basic editing view

Editors In the IDE, each of the elements of a program can be visually edited. Fig.5 shows a basic editing view. A editor for a Petri net system is similar to that provided in PNK source, but some modifications to add code editing area, to make the code editing to be the main input area, and to integrate with associate compilation operations.

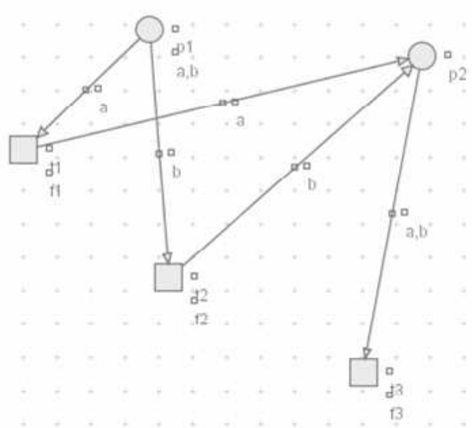
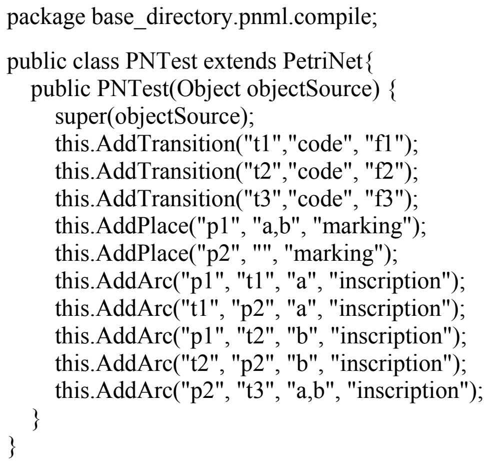


Fig. 6. A visual Petri net system

A visually edited Petri net system will be automatically translated to some code to be fed to the compiler. It actually completes a class inherited from a class PetriNet for the programmer, where PetriNet is the class encapsulated for a special Petri net virtual machine. For example, for the Petri net system in Fig.6, the result class will be the one in Fig.7.



```
package base_directory.pnml.compile;

public class PNTest extends PetriNet{
   public PNTest(Object objectSource) {
      super(objectSource);
      this.AddTransition("t1","code", "f1");
      this.AddTransition("t2","code", "f2");
      this.AddTransition("t3","code", "f3");
      this.AddPlace("p1", "a,b", "marking");
      this.AddPlace("p2", "", "marking");
      this.AddArc("p1", "t1", "a", "inscription");
      this.AddArc("t1", "p2", "a", "inscription");
      this.AddArc("p1", "t2", "b", "inscription");
      this.AddArc("t2", "p2", "b", "inscription");
      this.AddArc("p2", "t3", "a,b", "inscription");
   }
}
```

Fig. 7. Class for a Petri net system

Associate Compilation The *associate compilation* makes the static check of a Petri net system, then associates its code with all other parts of opened codes and compiles them together with each other. At the early time of the compilation, the Petri net system is translated into its internal form as the Petri net virtual machine instructions, which then is added to its specific class.

Variables corresponding to transaction memory blocks and resource memory blocks are declared with the modifiers *global* and *resource* respectively. Besides, the code for each transition of a Petri net system is defined by a specific member function with the modifier *petrinet*. In the associate compilation, the lexical, syntactical, and semantical analysis associate to the modifiers *global*, *resource* and *petrinet* has been processed carefully.

The compiler has been implemented based on GJC [14], a open Java compiler released by Sun, and kept the original logic of GJC unchanged.

The statical semantic check for a Petri net system is corresponding to the definition of a *well-formed resource net system* in Section 2.2.

STM Integration The transactional memory support is based on DSTM2 [2], with each piece of transition code automatically trisected by invoking provided STM APIs. Hence the variable with modifier Global can be protected by the transactional memory system.

In order to integrate DSTM2's API, we need to make some modification to GJC. We need to change the type of every variable with modifier Global to a wrapper class with a factory. Also we need to change every reference of those variables and every left-value consisted of those variables to corresponding DSTM2's APIs.

Fig.8 illustrates how to transact a global variable in our implementation. Currently we only support basic types or simple “copyable” types.

```
                      @atomic interface _T {
                        T getValue ();
                        void setValue (T value);
                      }
                      Factory<_T> factory_T =
                        Thread.makeFactory(_T.class);
global T a;           _T _a = factory_T.create();
a = x;                _a.setValue(x);
x = a;                x = _a.getValue();
```

Fig. 8. Transactional global variables

Petri Net Virtual Machine As stated above, a Petri net system is finally translated to some class inherited from a class PetriNet, which encapsulates interfaces for a special Petri net virtual machine. The Petri net virtual machine is now simply designed with the following instructions:

- AddTransition (name,code)
- AddPlace (name,resource)
- AddArc (Source, Target, Inscription)
- Start ()
- Join ()

where AddTransition, AddPlace, and AddArc are used to construct a Petri net system, and Start and Join used for scheduling the execution of a Petri net system.

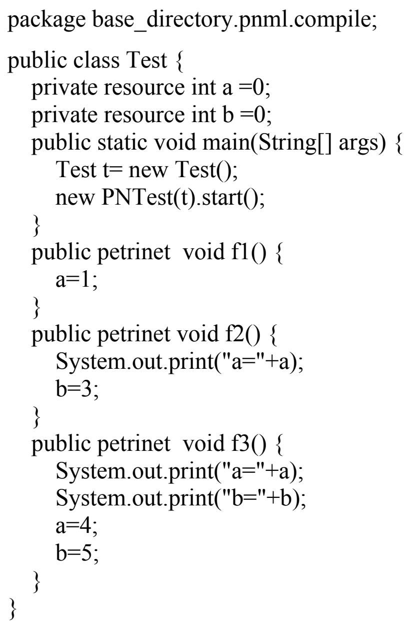
Fig.9 shows an example to start the Petri net system defined in Fig.6 or Fig.7. One possible execution result will be "a=1 a=1 b=3", and another possible result is "a=0 a=1 b=3", as is illustrated in Fig.10.

Fig.11 and Fig.12 illustrate the integration of a simple Petri net transition simulator and DSTM routine. The left routine in Fig.11 simulate a Petri net transition to do something, and the right part is the DSTM routine to do the same task. Fig. 12 integrates the functions of two routines in Fig.11, getting a so-called PNTM routine.

The Petri net virtual machine can be implemented on any architecture you like, especially, it will be helpful if the target architecture can efficiently support concurrent programming, such as a CMP system. Until now, we have just implemented the Petri net virtual machine based on JVM.

The latest stable version of source code, in which the invoking of STM API’s in DSTM2 has not been packaged, can be downloaded at the URL:
`http://soft.cs.tsinghua.edu.cn/~wang/projects/NSFC90818019/software/pntm.rar`

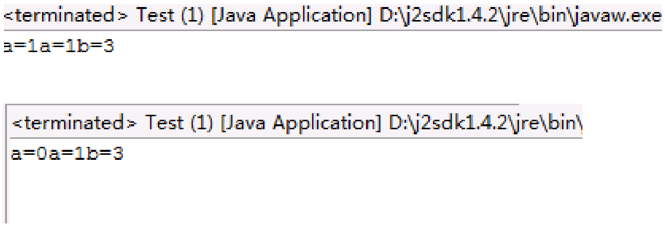
We are making a research plan to extend the virtual machine and its implementation based on some reconfigurable simulator for multi-core architectures such that some basic performance analysis could be made.



```
package base_directory.pnml.compile;

public class Test {
    private resource int a =0;
    private resource int b =0;
    public static void main(String[] args) {
        Test t= new Test();
        new PNTest(t).start();
    }
    public petrinet  void f1() {
        a=1;
    }
    public petrinet void f2() {
        System.out.print("a="+a);
        b=3;
    }
    public petrinet  void f3() {
        System.out.print("a="+a);
        System.out.print("b="+b);
        a=4;
        b=5;
    }
}
```

Fig. 9. Example for starting a Petri net system



```
<terminated> Test (1) [Java Application] D:\j2sdk1.4.2\jre\bin\javaw.exe
a=1a=1b=3
```

```
<terminated> Test (1) [Java Application] D:\j2sdk1.4.2\jre\bin\
a=0a=1b=3
```

Fig. 10. Two different executions on the simple virtual machine

6 Remarks and Future Work

The paper presents an approach to integrate a Petri net system with a transitional memory mechanism, which has currently been applied to the implementation of a user-level transactional concurrent programming tool in our lab. There is few formalism to play such a role as we have known so far. There exist researches based on Petri nets to model atomic or transactional threads, however the net system is not a part of the program. For example, an approach to check causal atomicity is proposed by modeling programs using Petri Nets [12]. At some extent, concurrent programming models based on Petri nets, such as OPN [15], CLOWN [16], COO [17], CO-OPN/2 [18], and Elementary Object Nets [19] may be extended to support various transaction semantics with the conservative concurrency control.

We observed that the integration of Petri nets with transactional memory can bring benefits to both side, which is the motivation of the paper. On the one hand, with

```
TestAndConsumeToken();
DoThings();
ProduceToken();
```

PN-Transition routine

```
Thread.onCommit(new Runnable() {
  public void run () {
    Commit();
  }
});
Thread.onAbort(new Runnable() {
  public void run () {
    Abort();
  }
});
Thread.doIt(new Callable<Void>() {
  public Void call ()
      throws Exception {
    DoThings();
  }
});
```

DSTM routine

Fig. 11. Petri net transition and DSTM routines

```
Thread.onCommit(new Runnable() {
  public void run () {
    ProduceToken();
  }
});
Thread.onAbort(new Runnable() {
  public void run () {
    ReturnToken();
  }
});
TestAndConsumeToken();
Thread.doIt(new Callable<Void>() {
  public Void call ()
      throws Exception {
    DoThings();
  }
});
```

Fig. 12. PNTM routine

transactional memory, a finer granularity of concurrency can be achieved in a Petri net system, and the scale of the net model can be controlled flexibly. On the other hand, with a Petri net system, the concurrency among cooperative transactions can be built explicitly, which can undoubtedly decrease the rate of conflicts and improve the performance, while the analysis and verification capability of a Petri net model can be inherited.

The main idea in a resource net system, the net system presented in the paper, is to classify shared resources in two classes: (1) resources such that the access policy is driven by the net structure, so that mutual exclusion is guaranteed; (2) resources whose access policy is driven by a transactional memory model, with possible conflicts, resolved by a commit-rollback protocol.

That is, our approach advocates the methodology that critical objects shared among concurrent transactions will be protected through a resource net system, while non-critical shared objects be left protected automatically by the transactional memory system. Thus the net system can be designed flexibly to keep a moderate size and in a finer granularity than usual net system.

It is shown that the behavior of a well-formed resource net system can be simulated by its desugar net system, which can be analyzed and verified by using existing approaches in the Petri net community. Behavior properties for a well-formed resource net system, such as deadlock-freeness, can be verified indirectly by verifying those for its desugaring net system. For example, INA tool [20] can be directly integrated into our programming tool being developed, as has been done in PNK.

A practical user-level transactional concurrent programming tool, based on DSTM2 [2], PNK [13] and GJC [14], has been developing in our lab. The version so far is not suitable to make a performance analysis because the target virtual machine on which a program with Petri net structures runs is implemented simply based on JVM. We are making a plan to extend the virtual machine and its implementation such that some basic performance analysis could be made.

Certainly, the approach could be extended to other formalisms as well. Furthermore, how to decide critical or non-critical shared objects, we believe, would become an interesting area in software design methodology.

References

1. Tim Harris, et al., Transactional Memory: An Overview, IEEE Micro, vol. 27, no. 3, Pages 8-29, 2007.
2. Maurice Herlihy, Victor Luchangco, Mark Moir, A Flexible Framework for Implementing Software Transactional Memory, In Preceedings of OOPSLA'06, Pages 253-262, 2006.
3. TL2-x86, Stanford Transactional Applications for Multi-Processing, http://stamp.stanford.edu/.
4. E. Allen et al., The Fortress Language Specification, Sun Microsystems, 2005.
5. P. Charles et al, X10: An Object-Oriented Approach to Non-Uniform Cluster Computing, Proc. 20th Ann. ACM SIGPLAN Conf Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA05), ACM Press, pp. 519-538, 2005.
6. Chapel Specification 0.4, Cray Inc., 2005; http://chapel.cs.washington.edu/Specification.pdf.
7. Tim Harris, Keir Fraser, Language Support for Lightweight Transactions, OOPSLA'03, October 26-30, 2003.
8. The OpenMP API specification for parallel programming. URL: http://openmp.org.
9. Baek W., Minh C. C., Trautmann M., Kozyrakis C., and Olukotun K., The OpenTM Transactional Application Programming Interface, In PACT'07: Proceedings of the 16th international conference on Parallel architectures and compilation techniques, Washington, DC, USA: IEEE Computer Society, pp. 376-387, 2007.
10. W. Reisig, Petri Nets, EATCS Monagraphs on Theoretical Computer Science, Springer Verlag, 1985.

11. K.Jensen. Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Volume 1, EATCS Monographs in Computer Science, Springer verlag, 1992.
12. Azadeh Farzan, P. Madhusudan,Causal Atomicity. In Proceedings of Computer Aided Verification (CAV) 2006, Lecture Notes in Computer Science, volume 4144: 315-328, 2006.
13. Ekkart Kindler, Michael Weber, The Petri Net Kernel: An infrastructure for building Petri net tools, International Journal on Software Tools for Technology Transfer (STTT), Vol 3, No.: 486-497, 2001. PNK available at : http://www2.informatik.hu-berlin.de/top/pnk/.
14. GJC available at : http://www.sun.com/software/communitysource/j2se /java2/download.xml
15. C.A. Lakos, Object-Oriented Modelling with Object Petri Nets, Concurrent Object-Oriented Programming and Petri Nets, G. Agha, F.D. Cindio, and G. Rozenberg (eds.), Lecture Notes in Computer Science 2001, Springer-Verlag, pages 1-37, 2001.
16. E.Batiston, A.Chizzoni, Fiorella De Cindo, CLOWN as a Testbed for Concurrent Object-Oriented Concepts, Concurrent Object-Oriented Programming and Petri Nets, G. Agha, F.D. Cindio, and G. Rozenberg (eds.), Lecture Notes in Computer Science 2001, Springer-Verlag, pages 131-163, 2001.
17. C.Sibertin-Blanc, CoOperative Objects: Principles, Use and Implementation, Concurrent Object-Oriented Programming and Petri Nets, G. Agha, F.D. Cindio, and G. Rozenberg (eds.), Lecture Notes in Computer Science 2001, Springer-Verlag, pages 216-246, 2001.
18. O. Biberstein, D. Buchs, N. Guelfi, Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 Formalism, Concurrent Object-Oriented Programming and Petri Nets, G. Agha, F.D. Cindio, and G. Rozenberg (eds.), Lecture Notes in Computer Science 2001, Springer-Verlag, pages 73-130, 2001.
19. R.Valk. Petri Nets as Token Objects An Introduction to Elementary Object Nets. Proceedings of 19th International Conference on the Application and Theory of Petri Nets, LNCS 1420, Springer-Verlag, 1998.
20. INA:Integrated Net Analyzer, at http://www.informatik.hu-berlin.de/lehrstuehle/automaten/ina.

A Component Framework where Port Compatibility Implies Weak Termination

Debjyoti Bera, Kees M. van Hee, Michiel van Osch, and
Jan Martijn van der Werf

Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands.
{d.bera, k.m.v.hee, m.p.w.j.van.osch, j.m.e.m.v.d.werf}@tue.nl

Abstract. The design and verification of an asynchronous communicating system can be very complex. In this paper we focus on weak termination: in each reachable state, the system has the option to eventually terminate. We present a component framework and construction method that guarantees weak termination. In the framework, communication between components is modeled by portnets, a special class of workflow nets. A basic component defines the orchestration of the portnets. For weak termination, the orchestration should accord to each of the portnets. A composite component is built from basic components that offer some service via a portnet. We provide sufficient conditions to guarantee weak termination for composite components. Furthermore, we present a refinement-based construction procedure to derive a weakly terminating composite from an architectural diagram of the system.

1 Introduction

The class of asynchronous communicating systems encompasses a wide range of software systems that include information systems, embedded systems, grid computing systems, etc. The distributed nature and growing complexity of these systems warrant the need for a component based development (CBD) approach with support for formal analysis techniques. The central idea in the design of such systems involves the construction of complex systems by assembling components while guaranteeing certain properties.

Over the past years, different formal models supporting component based development have been proposed, like Cadena [8] and SaveCCM [5]. Many of these techniques provide a model to specify the components and their composition while relying on state space based explorations to verify the correctness of the design. State space based explorations are generally time consuming and do not scale well to the complexities of real world models. For this reason we construct a framework to guarantee correctness properties by construction. We focus on one property: weak termination.

The weak termination property states that in each reachable state of the system, the system always has the possibility to reach a final state. Generalized soundness [10] is a generalization of weak termination for workflow nets.

A class of generalized sound workflow nets is the class of ST-nets [10] which are constructed by successive refinements of state machines and acyclic marked graphs [6].

Components are loosely coupled. As a consequence, their composition introduces a high degree of concurrency, and thus a state space explosion. In [2] a sufficient condition is presented to pairwise verify weak termination for tree structured compositions. For a subclass of compositions of pairs of components, called ATIS-nets, this condition is implied by their structure [11]. ATIS-nets are constructed from pairs of acyclic marked graphs and isomorphic state machines, and the simultaneous refinement of pairs of places [15].

In this paper, we present a component framework to construct a network of asynchronously communicating components that guarantees weak termination. The framework supports a best practice in communication protocol design: communication between two components is first modeled as a state machine. Then, each transition is assigned to one of the components, such that if any two transitions are in conflict, these transitions are designated to the same component. Then, the state machine is duplicated for each of the components. If a transition is assigned to that component, it sends a message; the corresponding transition of the other component receives this message. Such a net is called a *portnet*. In this way, a component consists of a set of portnets defining its behavior with the environment. A component needs to orchestrate all its portnets, such that for each component it communicates with, it acts as the corresponding portnet. This requirement is similar to the condition imposed by choreography standards like WS-CDL [12]. A Component may be either basic or composite. A basic component provides a service via a portnet. In order to do so, it consumes from other components. In a composition, we allow more than one component to consume a service from another component. Such a composition is a directed graph with edges representing dependency relationships between basic components. If the composition is acyclic, it is a *composite component*.

The orchestration of a component may nest portnets. To resemble this in the architecture, we introduce a simple architectural diagram. Furthermore, we study the behavior of an arbitrary composition of components and give sufficient conditions to guarantee weak termination. We also present a construction procedure based on the rules of [14] and [9] to derive a weakly terminating composition of components.

2 Preliminaries

Let S be a set. We denote the powerset by $\mathcal{P}(S)$. A *bag* over some set S is a function $m : \mathbb{N} \to S$, where $\mathbb{N} = \{0, 1, 2, \ldots\}$ denotes the set of natural numbers. For $s \in S$, $m(s)$ denotes the number of occurrences of s in m. We enumerate bags with square brackets, e.g. the bag $m = [a^2, b^3]$ has an element a occurring twice and element b occurring thrice and all other elements have multiplicity zero. The set of all bags over S is denoted by $B(S)$. We write $[]$ for an empty bag and we use $+$ and $-$ for the sum of two bags and $=, <, >, \leq, \geq$ to element wise compare bags, which are defined in the standard way. A set can be seen as a multiset in which each element of the set occurs exactly once.

A Petri net is a tuple $N = (P, T, F)$, where P is the set of *places*; T is the set of *transitions* such that $P \cap T = \emptyset$ and F is the *flow relation* $F \subseteq (P \times T) \cup (T \times P)$. We refer to elements from $P \cup T$ as *nodes* and elements from F as *arcs*. We denote the places of net N by P_N, transitions as T_N and similarly for other elements of the tuple. If the context is clear, we omit N in the subscript. We define the preset of a node n as ${}^{\bullet}_{N}n = \{m | (m, n) \in F\}$ and the postset as $m^{\bullet}N = \{n | (m, n) \in F\}$. We lift the notion of a preset and postset to sets: ${}^{\bullet}_{N}S = \cup_{s \in S} {}^{\bullet}_{N}s$ and $S^{\bullet}_{N} = \cup_{s \in S} s^{\bullet}_{N}$ for some set $S \subseteq (P \cup T)$. If the context is clear, the subscript is omitted. A *path* ν in a Petri net N of length $n \in \mathbb{N}$ is a function $\nu : 1, \ldots, n \to (P \cup T)$ such that $(\nu(i), \nu(i+1)) \in F$ for all $1 \leq i < n$. We denote a path of length n by $\nu = \langle x_1, \ldots, x_n \rangle$ where $x_i = \nu(i)$ for all $q \leq i \leq n$. The set of all paths of a Petri net N is called the *path space* and denoted by $PS(N)$. Two Petri nets N and M are disjoint if $(P_N \cup T_N) \cap (P_M \cup T_M) = \emptyset$. They are *isomorphic*, denoted by $N \cong_{\psi} M$ if and only if a bijective function $\psi : P_N \cup T_N \to P_M \cup T_M$ exists such that $P_M = \psi(P_N)$, $T_M = \psi(T_N)$ and $\forall (x, y) \in F_N \Leftrightarrow (\psi(x), \psi(y)) \in F_M$. We write $N \cong M$ if a bijective function ψ exists such that $N \cong_{\psi} M$. The state of a Petri net $N = (P, T, F)$ is determined by its marking which represents the distribution of tokens over places of the net. A marking m of a Petri net N is a bag over its places P, i.e., $m \in B(P)$. A transition $t \in T$ is enabled in m if and only if ${}^{\bullet}t \leq m$. An enabled transition may fire which results in a new marking $m' = m - {}^{\bullet}t + t^{\bullet}$, denoted by $m \xrightarrow{t} m'$. We define the set of reachable markings of a Petri net N with marking m inductively by $\mathcal{R}(m) = \{m\} \cup \bigcup_{m \xrightarrow{t} m'} \mathcal{R}(m')$. We define the *net system* of a Petri net N as a 3-tuple $M = (N, m_0, m_f)$, where $m_0 \in B(P_N)$ is the initial marking and $m_f \in B(P_N)$ is the final marking. The *weak termination* property for a *net system* M states that $\forall m \in \mathcal{R}(N, m_0) : m_f \in \mathcal{R}(N, m)$, i.e. for all reachable markings from the initial marking the final marking is reachable. If a marking does not enable any transition in the net, it is called a *dead marking.* A place is called safe in a net system (N, m_0, m_f) if $\forall m \in \mathcal{R}(N, m_0), m(p) \leq 1$. Let $N = (P, T, F)$ be a Petri net. Net N is a *workflow net* (WFN) if there exists exactly one place $i \in P$ with ${}^{\bullet}i = \emptyset$, called the initial place, one place $f \in P$ with $f^{\bullet} = \emptyset$, called the final place, and all nodes $n \in P \cup T$ are on a path from i to f. The *closure* of a workflow net N is a net $closure(N) = (P, T \cup \{\bar{t}\}, F \cup \{(\bar{t}, i), (f, \bar{t})\})$ such that $\bar{t} \notin T$ and ${}^{\bullet}i = f^{\bullet} = \{\bar{t}\}$. A WFN N weakly terminates if its net system $(N, [i], [f])$ weakly terminates. Note that in [10] this property is called *1-Soundness.* For an overview of soundness, see [1]. Net N is a *state machine* (*S-net*) [6] if and only if $\forall t \in T : |{}^{\bullet}t| = |t^{\bullet}| = 1$. In a state machine, a place p is called a *split* if $p^{\bullet} > 1$. Likewise, it is a join if ${}^{\bullet}p > 1$. Net N is a *marked graph* (*T-net*) [6] if and only if $\forall p \in P : |{}^{\bullet}p| = |p^{\bullet}| = 1$. A workflow net that is also a state machine is called an *S-WFN.* If it is both a workflow net and a marked graph, it is called a *T-WFN.* The class of *ST-nets* were introduced in [10]. These nets allow both concurrency and choice. Note that the class of *T-nets* used in [10] has transitions as the initial and final nodes of the net. We extend such a net to our definition of a *T-WFN* by adding one initial and one final place. The class of *ST-nets* that we will use in this paper includes the class of *S-nets*, *T-nets* and nets obtained after arbitrary successive refinement of places [10] within an *ST-net* by either an *S-WFN* or a *T-WFN*.

3 Component Framework

In this section, we introduce a compositional framework to describe component based systems built of components that are cyclic in their execution and react to inputs from their environment. The main concept of this framework is the notion of a *component*. A component may be *basic* or *composite*. A basic component provides a service and may in turn may use the services offered by other basic components. The interfaces of a basic component are modeled as a *portnet*. A portnet describes a communication protocol. Such a protocol describes all possible sequences of messages that may be exchanged during a service negotiation. A basic component is a closed ST-net providing some service by means of a *sell side portnet* and consuming services using *buy side portnets* from components that have a compatible *sell side portnet*. The sell side portnet of a basic component encapsulates all of its buy side portnets. Furthermore, we allow buy side portnets to be nested. A *composite component* is the composition of a set of pairwise composable basic components such that their dependency graph is *acyclic*.

A *component* is modeled as a Petri net. An *activity* within such a component is modeled by a transition. We distinguish between two types of places, namely *internal places* and *interface places*. An *interface place* is either an *input place* for one component or an *output place* for another component. An *input place* has an empty preset and an *output place* has an empty postset. All other places of a component are referred to as *internal places*. Tokens residing at interface places represent messages, otherwise they are simply *state markers*. Transitions are either *internal transitions* or *interface transitions*. An *internal transition* has no interface places in its preset and postset, whereas an *interface transition* has some interface places in its preset (then it is called a *receive transition*) or its postset (then it is called a *send transition*), but never in both.

3.1 Formalization

Our component framework is based on *open Petri nets* (OPN) which are a subclass of classical Petri nets. OPN are ideal to model communicating systems. This is because they have a distinguished set of *interface places* that represent the interfaces of the net. A direct consequence of the interaction of an OPN with its environment results in tokens being exchanged between these places. Furthermore, we add structural constraints to derive subclasses of an OPN. A *subworkflow net* is a OWN that is a subnet of an OPN.

Definition 1 (Open Petri net, subworkflow net). *An open Petri net (OPN) is defined as $N = (P, I, O, T, F, i, f)$, where* (1) *$P$ is the set of* internal places*;*

(2) *I is the set of* input places *with* $^\bullet I = \emptyset$*;* (3) *O is the set of* output places *with* $O^\bullet = \emptyset$*;* (4) *T is the set of* transitions*;* (5) *the sets P, I, O and T are pairwise disjoint;* (6) *$i \subseteq P$ is the set of* initial places*;* (7) *$f \subseteq P$ is the set of* final places*;* (8) $\forall t \in T : {}^\bullet t \cap I \neq \emptyset \Rightarrow t^\bullet \cap O = \emptyset \wedge t^\bullet \cap O \neq \emptyset \Rightarrow {}^\bullet t \cap I = \emptyset$*; and* $((P \cup I \cup O, T, F), i,\ f)$ *is the* net system*. We refer to the set $I \cup O$ as the* interface places *of the net. The* skeleton *of N is a Petri net defined as* $skeleton(N) = (P, T, F')$*, where* $F' = F \cap ((P \times T) \cup (T \times P))$*. The* skeleton system *is defined as* (P, T, F', i, f)*.*

If $skeleton(N)$ is a WFN *then N is called a* open workflow net (OWN)*. If $skeleton(N)$ is a* S-WFN *then N is called a* state machine open workflow net (S-OWN)*. If $skeleton(N)$ is a* T-WFN *then N is called a* marked graph open workflow net (T-OWN)*. If $skeleton(N)$ is a* ST-net *then N is called a* ST open workflow net (ST-OWN)*.*

Let N be an OPN and M be a OWN. We say that M is a subworkflow net of N denoted by $M \sqsubseteq N$ if and only if $P_M \subseteq P_N$, $T_M \subseteq T_N$, $F_M \subseteq F_N$, $I_M \subseteq I_N, O_M \subseteq O_N$, ${}^\bullet_N(T_M \cup O_M \cup P_M \setminus \{i_M\}) \cup (T_M \cup I_M \cup P_M \setminus \{f_M\})^\bullet_N \subseteq (T_M \cup P_M \cup I_M \cup O_M)$*.*

The transitions of an open Petri net are distinguished into three categories depending on the direction of communication, namely send, receive and internal. A *send transition* contains an output place in its postset. A *receive transition* has an input place in its preset. A transition that does not send or receive is called an *internal transition.*

Definition 2 (Direction of communication). *The* direction of communication *of a transition with respect to a place in an open Petri net N is a function* $\lambda : T \to \{send, receive, \tau\}$ *defined as* $\lambda(t) = send \Leftrightarrow t^\bullet \cap O \neq \emptyset \wedge {}^\bullet t \cap I = \emptyset$*;* $\lambda(t) = receive \Leftrightarrow {}^\bullet t \cap I \neq \emptyset \wedge t^\bullet \cap O = \emptyset$ *and* $\lambda(t) = \tau$*, otherwise, for all $t \in T$. We call a transition $t \in T$ a* communicating transition *if and only if* $\lambda(t) \neq \tau$*.*

The refinement of safe places in a Petri net is a well known refinement step and has been described in various contexts [10]. We present here the refinement of a safe place within an OPN by an ST-OWN.

Definition 3 (Place refinement and net reduction). *Given an OPN N and an OWN M such that N and M are disjoint, a safe place* $p \in P_N \setminus \{n \mid i_N(n) = f_N(n) = 0\}$ *can be refined by M, resulting in an OPN* $N' = N \odot_p M = (P, I, O, T, F, i, f)$ *with* $P = (P_N \setminus \{p\}) \cup P_M$, $I = I_N \cup I_M$, $O = O_N \cup O_M$, $T = T_N \cup T_M$, $F = (F_N \setminus (({}^\bullet p \times \{p\}) \cup (\{p\} \times p^\bullet))) \cup F_M \cup ({}^\bullet p \times \{i_M\}) \cup (\{f_M\} \times p^\bullet)$, $i = i_N$, $f = f_N$*. We define the reduction of net N' by the subworkflow net M by* $reduce(N', M) = N$ *if and only if* $N' = N \odot_p M$*.*

An OPN N is said to be *reducible* to another open Petri net N' if and only if successive applications of the reduce operation on net N results in the net N'. Note that we restrict the definition to reductions only by the class of ST-net, since this is the inherent structure of all nets in this component framework. Note that this relation is a preorder.

Definition 4 (Reducible nets). *Consider two OPN's N and N'. We say N is reducible to N' denoted by $N \rightsquigarrow N'$ if and only if $N = N' \vee \exists M : M$ is a ST-OWN $\wedge$ $(M \neq N) \wedge (M \sqsubseteq N) \wedge reduce(N, M) \rightsquigarrow N'$.*

Unlike in an OPN, interfaces in our component framework are more than just a set of interface places acting as message buffers. An interface is determined by a Petri net with a distinguished set of interface places, called the *portnet.* A portnet defines the communication protocol which specifies all acceptable sequences of messages that are permitted to be exchanged over the portnet.

A portnet is an *S-OWN* with structural constraints on the relation between transitions and interface places and paths through it. In a portnet, each interface place is connected to exactly one transition, and each transition is connected to exactly one interface place. Secondly, a portnet must satisfy the *leg property.* A path in a portnet is called a *leg* if it is a path from a split to a join. We also consider the initial place as a split and the final place as a join. The *leg property* requires every leg in a portnet to have at least two transitions with different directions of communication. Lastly, a portnet must satisfy the *choice property*, which requires all transitions belonging to the postset of a place to have the same direction of communication.

Definition 5 (Portnet). *A portnet C is a* S-OWN *such that*

- $\forall t \in T : |({}^\bullet t \cup t^\bullet) \cap (I \cup O)| = 1$;
- $\forall x \in I \cup O : |{}^\bullet x \cup x^\bullet| = 1$;
- (Leg property) $\forall \beta = \langle p_1, t_1 ... t_{n-1}, p_n \rangle \in PS(C) : (|p_1{}^\bullet| > 1 \vee p_1 = i_N) \wedge (|{}^\bullet p_n| > 1 \vee p_n = f_N) : \exists t, t' \in \beta : \lambda(t) \neq \lambda(t')$.
- (Choice property) $\forall t_1, t_2 \in T : {}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset \Rightarrow \lambda(t_1) = \lambda(t_2)$.

We distinguish between two types of portnets: A *sell side portnet* advertises a service and needs a startup message and terminates after sending a result message. A *buy side portnet* consumes a service by sending a startup message and terminates after receiving the result message.

Definition 6 (Portnet types). *Consider a portnet C. We call Portnet C a* sell side portnet *denoted by $sell(C)$ if and only if $\forall t \in i_C{}^\bullet : \lambda(t) = receive \wedge \forall t \in {}^\bullet f_C : \lambda(t) = send$ and we call Portnet C a* buy side portnet *denoted by $buy(C)$ if and only if $\forall t \in i_C{}^\bullet : \lambda(t) = send \wedge \forall t \in {}^\bullet f_C : \lambda(t) = receive$.*

Note that $\neg sell(C) \Leftrightarrow buy(C)$. A *component* is an OPN with a set of portnets. Every communicating transition in a component belongs to a portnet. Furthermore, every portnet of a component is either already a subworkflow net or there exists a subworkflow net that can be reduced to the corresponding portnet.

Definition 7 (Component). *A* component *is a pair (N, Γ) where N is an OPN and Γ is a set of portnets, such that:*

- $\forall t \in T_N : \lambda(t) \neq \tau \Rightarrow \exists C \in \Gamma : t \in T_C$
- $\forall C \in \Gamma : \exists N' : N'$ *is an OWN* $: N' \sqsubseteq N \wedge N' \rightsquigarrow C$

The set of all sell side portnets of a component is defined as: $sellside((N, \Gamma)) = \{C \in \Gamma \mid sell(C)\}$ *and the set of all buy side portnets of a component is defined as:* $buyside((N, \Gamma)) = \{C \in \Gamma \mid buy(C)\}$.

Lemma 8 (Preservation of weak termination). *Consider two OPN's N and M such that $N \rightsquigarrow M$. Then N is weakly terminating if and only if M is weakly terminating.*

Portnets of a component may be nested in each other.

Definition 9 (Nested portnets). *Consider a component (N, Γ) and two portnets $C_1, C_2 \in \Gamma$. We say portnet C_2 is nested in portnet C_1 denoted by $C_2 \lhd_N C_1$ if and only if $\exists M_1, M_2 : M_2 \sqsubseteq M_1 \sqsubseteq N \wedge M_1 \rightsquigarrow C_1 \wedge M_2 \rightsquigarrow C_2$.*

Two portnets are said to be *compatible* if their skeletons are isomorphic. Furthermore, the set of input places of one portnet must match the set of output places of the other portnet while preserving the relation with their associated transitions. Note that a portnet is not compatible with itself.

Definition 10 (Compatible portnets). *Portnets C_1 and C_2 are compatible with respect to some bijective function $\phi : (P_{C_1} \cup T_{C_1} \cup I_{C_1} \cup O_{C_1}) \rightarrow (P_{C_2} \cup T_{C_2} \cup I_{C_2} \cup O_{C_2})$, denoted by $C_1 \triangleq_\phi C_2$ if and only if:*

- $skeleton(C_1) \cong_\phi skeleton(C_2)$,
- $O_{C_2} = \phi(I_{C_1})$, $I_{C_2} = \phi(O_{C_1})$,
- $\forall x \in I_{C_1}, t \in T_{C_1} : (x, t) \in F_{C_1} \Leftrightarrow (\phi(t), \phi(x)) \in F_{C_2}$,
- $\forall x \in O_{C_1}, t \in T_{C_1} : (t, x) \in F_{C_1} \Leftrightarrow (\phi(x), \phi(t)) \in F_{C_2}$

We write $C_1 \triangleq C_2$ if a bijective function ϕ exists such that $C_1 \triangleq_\phi C_2$.

Basic components are the building blocks of this component framework. The Petri net structure of a basic component is modeled as an *ST-OWN* with a closure transition. A basic component has one sell side portnet by means of which it provides a service. The sell side portnet may have zero or more nested buy side portnets. Furthermore, each interface place belongs to a unique portnet.

Definition 11 (Basic component). *A* component $B = (N, \Gamma)$ *is a* basic component *if and only if $|i_N| = |f_N| = 1$, N is the closure of an ST-OWN and the following conditions are met:*

- $\forall x \in I_N \cup O_N, \exists! C \in \Gamma : x \in I_C \cup O_C$;
- $\exists C \in \Gamma : i_C = i_N \wedge f_C = f_N \wedge sellside(B) = \{C\}$.

Corollary 12. *Consider a basic component $B = (N, \Gamma)$ and a portnet $C \in \Gamma : sell(C)$. Then $N \rightsquigarrow closure(C)$.*

Note that the closure transition allows the basic component to handle more than one service request. Fig. 1 gives an example of a basic component $M = (N, \Gamma)$, where $\Gamma = \{S1, B1, B2\}$ and $S1$ is a sell side portnet. The sell side portnet has two nested buy side portnets: $B1 \lhd_N S1$ and $B_2 \lhd_N S1$. Net N contains a

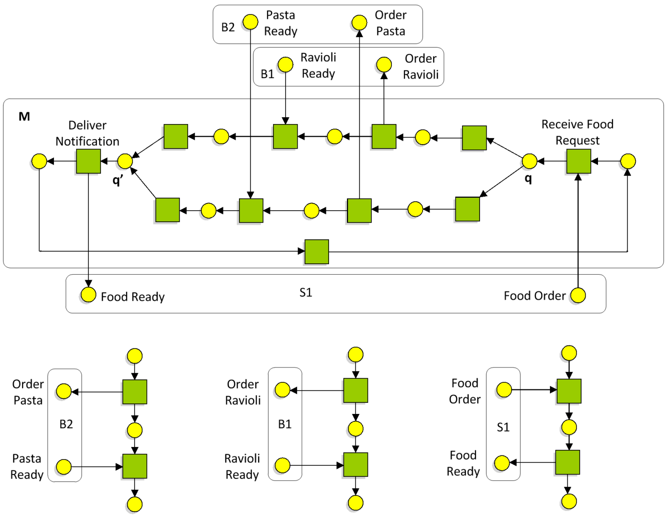


Fig. 1. A basic component

subworkflow net with initial place q and final place q'. This subworkflow net can be reduced by nets $B1$ and $B2$. We refer to the resulting net as an *orchestration net*. Such a net provides the logic behind the order of invocation of the different buy side portnets within a basic component.

Two components are said to be *composable* if and only if the only set of nodes they share are interface places and if this set is not empty, then either they have compatible portnets or they have identical buy side portnets. Note that we require unique sell side portnets.

Definition 13 (Composable components). *Two components $X = (N, \Gamma_N)$ and $Y = (M, \Gamma_M)$ are* composable *denoted by $composable(X, Y)$ if and only if*

- $(P_N \cup I_N \cup O_N \cup T_N) \cap (P_M \cup I_M \cup O_M \cup T_M) = (O_N \cup I_N) \cap (O_M \cup I_M)$;
- $\forall C_1 \in \Gamma_N, C_2 \in \Gamma_M : ((O_{C_2} \cap I_{C_1}) \cup (O_{C_1} \cap I_{C_2}) \neq \emptyset \Rightarrow C_1 \triangleq C_2) \wedge$
  $((I_{C_1} \cap I_{C_2}) \cup (O_{C_1} \cap O_{C_2}) \neq \emptyset \Rightarrow C_1 \cong C_2 \wedge buy(C_1))$.

A composition of a set of pairwise composable components is almost a pairwise union of the tuples of this set, except that the interface places belonging to pairs of compatible portnets, now become the internal places of this composition. Furthermore, the set of portnets of this composition is the set of all incompatible portnets. We extend the composition operation to portnets by treating portnets as components. This is possible in the following way: Consider a portnet C, then this portnet is also a component $(C, \{C\})$.

Definition 14 (Composition of components). *The composition of a set S of pairwise composable components is denoted by $comp(S) = (N, \Gamma)$, where $N = (P_N, I_N, O_N, T_N, F_N, i_N, f_N)$ such that*

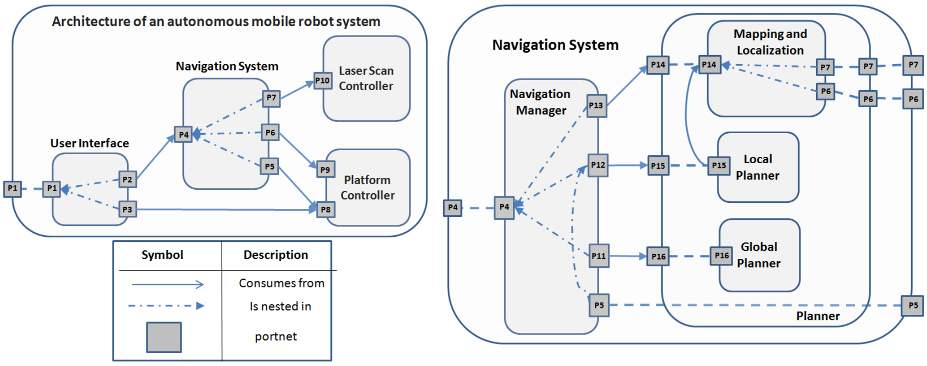


Fig. 2. Architecture diagram

- $P_N = (\bigcup_{(X,\Gamma')\in S} P_X) \cup (\bigcup_{(X,\Gamma')\in S} I_X \cap \bigcup_{(X,\Gamma')\in S} O_X)$,
- $I_N = \bigcup_{(X,\Gamma')\in S} I_X \setminus \bigcup_{(X,\Gamma')\in S} O_X$, $O_N = \bigcup_{(X,\Gamma')\in S} O_X \setminus \bigcup_{(X,\Gamma')\in S} I_X$,
- $T_N = \bigcup_{(X,\Gamma')\in S} T_X$, $F_N = \bigcup_{(X,\Gamma')\in S} F_X$,
- $i_N = \bigcup_{(X,\Gamma')\in S} i_X$, $f_N = \bigcup_{(X,\Gamma')\in S} f_X$, *and*
- $\Gamma = \{C \in \bigcup_{(X,\Gamma')\in S} \Gamma' | \forall C' \in \bigcup_{(X,\Gamma')\in S} \Gamma' : \neg(C \triangleq C')\}$.

Corollary 15. *The composition of a set of pairwise composable components is again a component.*

Note that the composition of one basic component is in fact the basic component itself. Furthermore, $comp(S_1 \cup S_2) \neq comp(S_1 \cup \{comp(S_2)\})$, where S_1 and S_2 are sets of pairwise composable basic components.

3.2 Architectural Diagram

We now present a graphical notation to represent a *composition of components* as an architectural diagram of the system. The diagram abstracts away from the underlying control flow and focuses on the relationships between components and the relationships between the portnets of a component. Components are depicted by a *rounded rectangle*. The portnets of a basic component are represented by a *square*. All entities are labeled. The *dependency relation* between a pair of portnets belonging to different components is represented by a directed arrow indicating the direction of communication initiation, i.e. from a buy side portnet to a sell side portnet. A buy side portnet may have at most one outgoing directed edge while a sell side portnet may have zero or more incoming directed edges. The sell side portnet with zero incoming directed edges becomes the portnet of the composition. The portnets of the composition are represented by extending the portnet with a dotted line to the boundary of the composition. By the structure of a component, we know that all the buy side portnets of a basic component are nested within the sell side portnet. Furthermore, a buy side portnet may nest one or more buy side portnets. We represent the *nesting of portnets* by a dotted directed edge leading from the child to its parent. Note that the Service Component Architecture assembly diagram [3] notation is similar but does not consider nested portnets. We present an architecture diagram of a navigation system in Fig. 2.

4 Behavior

In this section, we study the behavior of a composition of components. In particular, we are interested in weak termination of components, which we define on the skeleton system of the component.

Definition 16 (Weak termination of a component). *A component N is weakly terminating if its skeleton system weakly terminates.*

To prove weak termination for an arbitrary composition of components we first show that the composition of a sell side portnet with a set of compatible buy side portnets is weakly terminating. The crux of the proof relies on both the leg property and the choice property. These properties ensure that every choice and loop is properly communicated to the other compatible portnet, and once the choice to provide a service to a buy side portnet has been made no other buy side portnet can influence the service negotiation. The proof of the following theorem can be found in [4].

Theorem 17. *Let $A, B_1, ..., B_k$ be portnets such that $sell(A)$ and $B_i \triangleq A$ for all $1 \leq i \leq k$, then $comp(\{closure(A), B_1, ..., B_k\})$ weakly terminates.*

Weak termination for an arbitrary composition of portnets is not sufficient to guarantee weak termination for an arbitrary composition of components. To guarantee weak termination for a composition of components, we require the graph of the composition to be acyclic. This is because a cycle indicates a deadlock in the composition. In our framework, we call an acyclic composition of pairwise composable basic components a *composite component.* Note that we will use the shorthand D instead of $D = (N, \Gamma)$, D' instead of $D' = (N', \Gamma')$ and so on, to denote a component without explicitly labeling the tuples. We first introduce the notion of a partner for a buy side portnet in a component, which is the component that provides the compatible sell side portnet.

Definition 18 (Partner). *For a non-empty set S of composable basic components, and the set B consisting of all buy-side portnets of the components of S, we define the function partner : $S \times B \nrightarrow S$ by $\forall D, D' \in S, \forall C \in B : D' = partner(D, C) \Leftrightarrow \exists C' \in \Gamma' : sell(C') \wedge C \triangleq C'$.*

Definition 19 (Acyclic composition, composite component). *Consider a set of S of pairwise composable basic components. Let $R \subseteq S \times S$ be the relation such that $\forall D, D' \in S : (D, D') \in R \Rightarrow \exists C \in \Gamma : D' = partner(D, C)$. The composition is a* composite component *if and only if the transitive closure R^* is irreflexive.*

The result on weakly terminating composition of portnets in conjunction with Lemma 8 allows us to prove that an arbitrary composite component weakly terminates. Both the proof and an example of a deadlock in a cyclic composition can be found in [4].

Theorem 20. *A composite component weakly terminates.*

5 Construction Method

This section presents a construction method that derives a composition of basic components from an architectural diagram and ensures that the derived composition is weakly terminating. The construction method is based on place refinement and composition as defined in the previous sections.

The construction method starts with an architecture diagram of a composite component, like the one depicted in Fig. 2. To construct a basic component we require the three ingredients, namely a sell side portnet, a set of buy side portnets and a set of orchestration nets (ST-WFN). An orchestration net is used to elaborate the activities of a basic component by being able to introduce internal activities, concurrency and choice in a structured way. Furthermore, the places introduced by an orchestration net, may be refined with buy side portnets during construction, thereby allowing us to model both the choice of service invocations and concurrency in service invocations.

First, for each basic component in the diagram, design the sell side portnet. Next, for each basic component in the diagram, derive all its buy side portnets from existing compatible sell side portnets. Note that a buy side portnet may be derived from a sell side portnet by changing the direction of communication associated with each transition in the corresponding sell side portnet. Lastly, for each basic component in the diagram, design the necessary orchestration nets that will be required during the construction.

We may now convert all the sell side portnets into a basic component by introducing the closure transition. For each basic component, the architecture diagram gives the order of nesting of its portnets. Using this information, we may now start designing the control flow of a basic component by successive refinements of an existing internal place with either an orchestration net or a buy side portnet, until all the buy side portnets of the basic component have been added in the right order of nesting and the desired basic component has been constructed.

Construction method

1. Design an architecture diagram for an acyclic composition of basic components using the techniques of Sec. 3.
2. Design all the portnets and orchestration nets that we will need for this composition.
3. For each node in this architecture diagram select the corresponding sell side portnet and apply the closure operation.

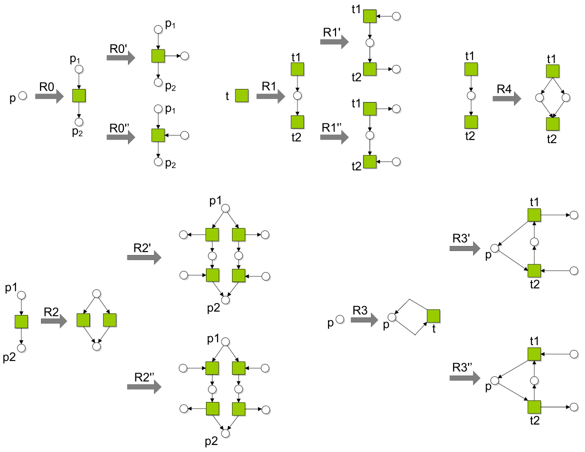


Fig. 3. Refinement rules to generate portnets and orchestration nets

4. For each basic component, repeat the following steps until in each basic component all the buy side portnets have been added in the right order of nesting, and the desired orchestration has been constructed:
   (a) If an orchestration needs to be added first, then choose an internal place and refine with the right orchestration net;
   (b) Otherwise, choose an internal place and refine with a buy side portnet in the order defined by the architectural diagram;
5. Compose the set of basic component using the composition operation.

Theorem 21. *The construction method always results in a composite component that weakly terminates.*

5.1 Construction of Orchestration Nets and Portnets

For the construction of portnets, we extend the Jackson refinement rules $R0$, $R1$, $R2$, and $R3$ with interface places as depicted in Fig. 3. Note that rule $R0$ is a special case of the refinement rule of Def. 3. Rule $R0'$ and $R0''$ extend rule $R0$ such that the refined transition can have either the communication direction send or receive. The extensions of rule $R3$ maintains the leg property by only adding loops with different directions of communication. Similarly, Rule $R2$, which adds a choice to the net is extended such that the choice property is maintained. Rule $R1$ is extended to allow two way communication.

The construction of an orchestration net starts with a single place. By applying the refinement rules of [9], we obtain larger nets that are guaranteed to be weakly terminating. We limit ourselves by applying the Jackson refinement rules $R0, R1, R2, R3, R4$ such that the result remains an ST-net. The construction of a portnet starts with the choice of the *sell side portnet* or *buy side portnet*. A sell side portnet is obtained by the sequence of refinements: $R0; R1; R1'$. A buy side portnet is obtained by the sequence of refinements: $R0; R1; R1''$. We may now further elaborate these portnets by arbitrary applications of the refinement rules $R0; R0'$, $R0; R0''$, $R1; R1'$, $R1; R1''$, $R2; R2'$, $R2; R2''$, $R3; R3'$, $R3; R3''$, while ensuring that the structure of the portnet remains a S-OWN by not allowing for place duplication (rule $R4$). Note that the choice of the place to apply the refinement sequence $R3; R3'$ or $R3; R3''$ must be such that the newly introduced legs do not violate the leg property.

Theorem 22. *The refinement rules for portnets preserve a portnet.*

6 The Control Flow of an Autonomous Mobile Robot

We will model the control flow of the navigation system on a mobile robot. The software system comprises of four main components: The user interface, the navigation system, the platform controller and the laser scan controller. The robot perceives its environment by means of a planar laser scanner. The laser scan controller provides the latest scan as a service. The platform controller is a composite component and provides two services (a) to set a desired velocity (b) queries on the latest odometry. The navigation system is capable of creating a map of its environment and localizing itself on this map using the current laser scan and odometry services. Furthermore, the navigation system can accept a waypoint and generate a sequence of velocity commands that drive the platform to this waypoint while avoiding obstacles. The user interface at the remote location allows an operator to visualize this map and give waypoint to the navigation system. While a waypoint is in progress an operator receives feedback on the progress of this goal. Once the robot has reached its waypoint, the operator is notified. An architecture of the system is presented in Fig. 2.

The navigation system is a composite component comprising of the navigation manager and planning. The latter is again a composite component and comprises of the global planner, the local planner and the mapping and localization system. The mapping and localization system is capable of generating a map and localizing itself using the services offered by the laser scan controller and platform controller. The global planner accepts a map and a waypoint goal and generates a global plan (trajectory) from the robot's current location to the waypoint goal. The local planner accepts a map and a global plan and generates a collision free sequence of velocity commands that drive the robot to the desired waypoint goal. The local planner generates these sequence of velocity commands in a loop until the destination is reached or a valid plan could not be found. In each cycle, the local planner makes use of the mapping and localization system to check its current location and generates feedback on the progress of this goal. If the destination has arrived then this is notified and the planner terminates. If at any moment, a valid velocity command could not be found then this situation is notified and the planner terminates. The navigation manager provides the waypoint navigation as a service to the user interface by orchestrating the components of the navigation system in the right order.

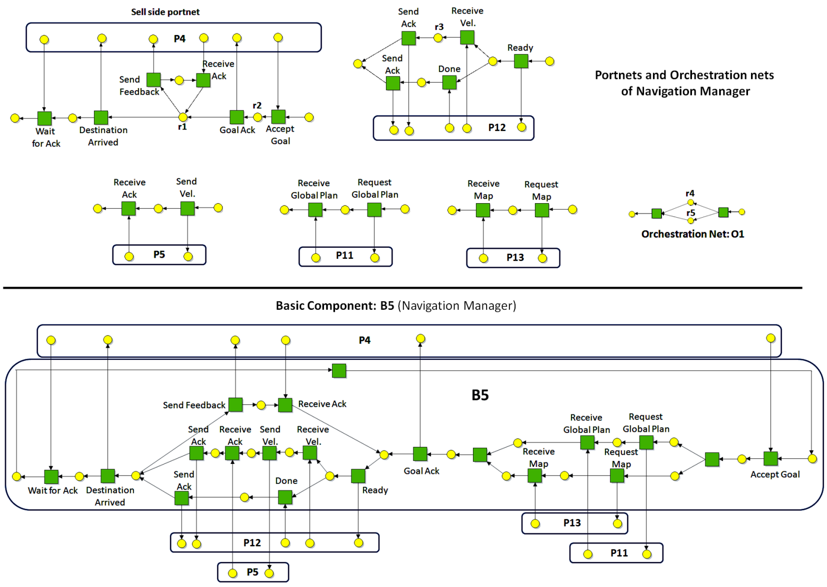


Fig. 4. Basic component: Navigation Manager

The Fig. 4 presents five portents and one orchestration net. From the architecture diagram in Fig. 2, we know portnet $P5$ is nested in $P12$ and all other buy side portnets are nested in the sell side portnet $P4$. We may now apply the construction method to derive the navigation manager in the following way: $(((closure(P4) \odot_{r2} O1) \odot_{r4} P11) \odot_{r5} P13) \odot_{r1} (P12 \odot_{r3} P5)$.

7 Conclusions

In this paper, we introduced a compositional component framework and a construction method to design the control flow of a network of components, while guaranteeing weak termination. The two main concepts of this framework are portnet and basic component. A portnet models the interface of a basic component as a state machine which describes the communication protocol underlying a service negotiation. A basic component provides a service by orchestrating its portnets in the right way. The weak termination property was then investigated by first considering compositions of portnets. It turns out that any pair of compatible portnets that satisfy the *leg property* and the *choice property* always weakly terminate. Furthermore, we prove that an acyclic composition of basic components also known as a composite component weakly terminates.

In [7, 13], the authors focus on constructing deadlock free systems using labeled transition systems, i.e., each component is a state machine, which after composition guarantee deadlock freedom. On the other hand, Petri nets offer a natural way to make formal models of the control flow of a software system. The

Petri net based construction method provides a structured way to design these control flows and guarantee weak termination by construction. In this way they can focus more on the design of each component without having to worry about deadlocks that could be introduced by a composition of components. The designers of software systems can use the guiding principles defined by the construction method during system design .

References

1. W.M.P. van der Aalst, K.M. van Hee, A.H.M. ter Hofstede, et al. Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Computing*, 23(3):333–363, 2011.
2. W.M.P. van der Aalst, K.M. van Hee, P. Massuthe, N. Sidorova, and J.M.E.M. van der Werf. Compositional Service Trees. In *ICATPN 2009*, volume 5606 of *LNCS*, pages 283–302. Springer, 2009.
3. M. Beisiegel et al. Service Component Architecture - Assembly Model Specification, SCA Version 1.00, 2007.
4. D. Bera, K.M. van Hee, M.P.W.J. van Osch, and J.M.E.M van der Werf. A Component Framework where Port Compatibility Implies Weak Termination. Technical Report CSR 11-08, Technische Universiteit Eindhoven, 2011.
5. J. Carlson, J. Hakansson, and P. Pettersson. SaveCCM: An Analysable Component Model for Real-Time Systems. *Electronic Notes in Theoretical Computer Science*, 160(1):127 – 140, 2006.
6. J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1995.
7. G. Gossler and J. Sifakis. Component-based construction of deadlock-free systems. In *FSTTCS 2003*, volume 2914 of *LNCS*, pages 420–433. Springer, 2003.
8. J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad. Cadena: An Integrated Development, Analysis, and Verification Environment for Componentbased Systems. In *ICSE 2003*, page 160. IEEE Press, 2003.
9. K.M. van Hee, A.J.H. Hidders, G.J.P.M. Houben, J. Paredaens, and P.A.P. Thiran. On the relationship between workflow models and document types. *Information Systems*, 34(1):178–208, 2009.
10. K.M. van Hee, N. Sidorova, and M. Voorhoeve. Soundness and Separability of Workflow Nets in the Stepwise Refinement Approach. In *ICATPN 2003*, volume 2679 of *LNCS*, pages 337–356. Springer, 2003.
11. K.M. van Hee, N. Sidorova, and J.M.E.M. van der Werf. Construction of asynchronous communicating systems: Weak termination guaranteed! In *Software Composition*, volume 6144 of *LNCS*, pages 106–121. Springer, 2010.
12. N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto. Web Services Choreography Description Language Version 1.0. `http://www.w3.org/TR/ws-cdl-10/`, November 2005.
13. K. Klai, S. Tata, and J. Desel. Symbolic Abstraction and Deadlock-Freeness Verification of Inter-enterprise Processes. In *Business Process Management*, volume 5701 of *LNCS*, pages 294–309. Springer, 2009.
14. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
15. J.M.E.M. van der Werf. *Compositional design and verification of component-based information systems.* PhD thesis, Technische Universiteit Eindhoven, 2011.

Improving the Development Tool Chain in the Context of Petri Net-Based Software Development

Tobias Betz, Lawrence Cabac, and Matthias Güttler

University of Hamburg
Faculty of Mathematics, Informatics and Natural Sciences
Department of Informatics
{2betz,cabac,3guettle}@informatik.uni-hamburg.de
`http://www.informatik.uni-hamburg.de/TGI`

Abstract. Modern, collaborative software development projects are highly supported by a variety of tools. Aside from the pure code development that is nowadays well supported by integrated development environments (IDEs) such as Eclipse, also other activities receive increasing attention in the matter of tool support. Recent improvements in tool support for source code management (SCM), process management and documentation management are driven by the changing demands of increasing distribution and agility of development projects. Moreover, the integration of tool support systems into integrated project management environments (IPMEs) increase the usefulness of each emerging tool support, especially for agile approaches. The Petri net-based agent-oriented software engineering (Paose) is highly influenced by agile methods and combines the agile concepts with aspects from model driven development. We have achieved increased efficiency by the adoption of IPMEs into the Paose development approach. However, the introduced tool set integration lacks in support for the model driven part of Paose, i.e. the tool integration for graphical (model-based) source code is missing. In this work we present an approach for an appropriate tool support of agile methods and model driven development. As a prototypical implementation of the concepts we present a Web service-based framework and plugin-based extension for Redmine as representative of open source IPMEs, which is currently in use in the context of the Paose approach.

Keywords: Renew, integrated project management, Paose, model-driven development, Redmine

1 Introduction

The history of software development shows a continuous increase of complexity in several aspects of the development process [1, p.10]. Furthermore, the increasing relevance of software in general and the higher demand in quality and speed contribute to the complexity of procedure. In comparison to conventional software development, today's modern development requires more flexible approaches to manage the given circumstances. In fact, project managers have to deal with distributed teams and software systems, concurrently working developers and fast and sporadic changes of demands.

Therefore we need even more adequate tools in order to keep the development process under control and to handle the different requirements in a satisfyingly and efficient manner. In the usual software development process, there are a lot of well-established tools in use, such as integrated development environments (IDEs) to support technical aspects of programming and project management tools, assisting developers and mainly superintendents in organizational matters. The centralized or distributed storage of source code is also well supported by various source code management (SCM) tools such as Subversion[1] or Git[2].

In contrast to the classical software development, the agent-oriented software development faces developers and project managers with some new aspects of tasks and therefore with some new fields to support, possibly even with an extended tool chain. One main reason for this is the self-organized and autonomous character of agents, which we also adopted into the organization of the development team [4]. This adoption goes well with the agile methods, which highly influenced the evolution of the Paose approach. Another reason is the model driven development based on Petri nets. In the Paose approach, we do not only operate with plain text files as source code, we also work with a graphical representation of Petri nets as source code to accomplish our targets of development. In contrast to the conventional model-driven development, in which the models are used to generate the executable source code, the Paose approach directly uses the models (Petri nets) as source code.

With this work we present an approach to improve the developers tool chain in context of the Petri net-based agent-oriented software development. For that reason, we advance the development process through adjusted and new tools, which fill the gap of partially unsupported tasks. Additionally, our focus lies on the adequate integration of these tools in our existent tool chain to advance the integrated project management environment (IPME). In Section 2 we investigate the developers' tasks, especially in context of the agent-oriented and agile procedure of development. Furthermore, we point out the deficits of the existent tool chain and additionally work out some possible supporting measures. We decided to primarily improve the topic of source code management within the Petri net-based development through an extended tool set. Section 3 describes these tools with regard to the architecture, realization and integration. We also present our prototypical implementation, which allows to include visual representations of any diagram type / net, available in the Renew tool set (see `http://www.renew.de`), in an IPME. This includes the simple displaying of diagrams / nets as well as the presentation of visual differences between versions available in the SCM.

[1] Apache Subversion, Enterprise-class centralized version control for the masses (http://subversion.apache.org)

[2] Git, distributed version control system (http://git-scm.com)

In the final conclusion we summarize our results of investigation and discuss the achieved benefits through our enhancement in the development tool chain.

2 Developer Tasks in Context of Paose

The process of software development consists of many different activities and phases we traverse during the specification, implementation, verification and deployment of software artifacts [13]. Figure 1 shows the various tasks a developer has to deal with. Besides the pure writing of source code, there are more tasks to focus on. Primarily, the definition of development tasks and their timing is essential concerning the planning and definition of pending work. These activities are supported by ticketing systems or bug trackers, which are widely applied in common tool chains of software development projects. Not only the current implementation phase is supported by tickets, it is also part of the software artifact specification within the meaning of agile methods.

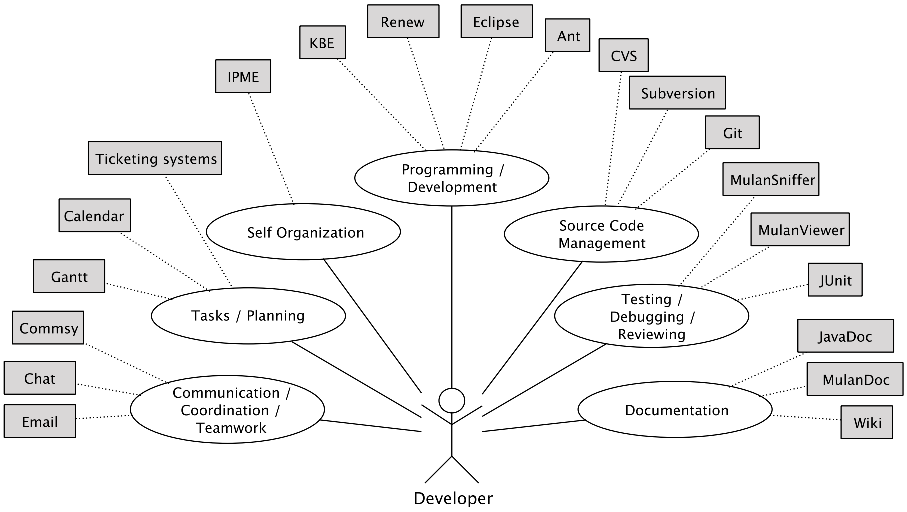


Figure 1. The developer's tasks and the supporting tools.

While communication is a prerequisite for the proper planning and coordination inside the development team[3], this activity gets more relevant within distributed teams and concurrently working developers. The terms self-organization and responsibility are outstanding in agent-oriented development [4,6], as well as in applied agile approaches [3]. Regular teamwork is already well supported

[3] In this context, we refer to development teams as individuals, who take part in the process of development and subsumes programmers, managers and contributors to the project.

by various communication platforms, but often less efficient than desired, especially in the context of agent-oriented software development[4]. Proper change notification in distributed planning is a complex task, regarding what Petrie et al. [11] investigated in this matter, and still not satisfactorily solved for modern, collaborative software development environments. In the context of agent oriented software development, especially the autonomous and self-organized aspect is predominant and consequently communication and coordination within the team becomes more important.

Furthermore, the source code management plays a significant role sharing resources within the development team. Regarding textually represented source code, a wide variety of tools exists to support the developer [9]. We already mentioned SCM tools like Subversion and Git. For graphically represented source code or models, on which the Petri net-based agent-oriented software development is based on, however, there is a lack of supporting tools to manage and visualize source code or changes made to it [9]. Finally one of the most useful features of modern SCM tools is the possibility to reveal differences between arbitrary revisions (versions) of the source code. This insufficiency within the Petri net-based development is a problem in order to accomplish the required information exchange and also the change notification between the developers of a (distributed) team, as mentioned before. Regarding Schipper et al. [12], a major concern is the depiction of changes in graphical models in a graphical way. We will present an appropriate solution for this missing feature in the following section.

Documentation is another task a developer is faced with. In agile approaches this task is less expensive, because running software has a higher value as comprehensive documentation if we follow the agile manifest [2]. However, apart from this, documentation is always a task to manage. Some IPMEs already support integrated documentation in the form of wikis for example. But also the documentation inside the source code is a part of the entire documentation and should be supported in some form by the integrated documentation system or the IPMEs as well. An IPME with integrated SCM and possibility to write and permanently save additional documentation fulfills this task sufficiently.

Finally, developers make use of a lot of tools. We aim at the integration of all used tools in the existent and usual tool chain and further connect these tools together. This means that developers are able to link the various contents and results of the tools' usages. More precisely, we target the linking of source code with task definitions, documentation and discussed topics relevant to the concrete realization. Additionally, the exchange of information between the tools should be realized as an automatic process embedded into the usual workflow. The result of this is an integrated project management environment, being able to handle many tasks belonging to different phases and areas of the complete development process and finally to make this information available through a single interface.

[4] Note that in Paose the agent-oriented paradigm is also applied to the development team and to the development process.

The free available open source project management tools Redmine[5] and Trac[6] support many of these needs out of the box and also provide a notification system via e-mail transparently embedded in the usual workflow. One missing feature, the possibility to view and compare graphical source code in a visual manner inside the IPME, is realized through our extension of the tool chain and will be presented in the following section.

3 Web Service-Based Tool Integration

In the previous section, we pointed out the requirement of a tool to visualize and compare graphical source code, particularly in context of Petri net-based development. This section deals with the architecture, implementation and integration of the outlined draft for such a tool, that fits perfectly into the distributed nature of agent-oriented software development, our existing tool chain and development workflow.

3.1 Architecture of the Tool Set

Due to the plugin-based design of the most project management tools, in our case Redmine, it is possible to improve the functionality of the IPME with respect to the integration of Petri nets and also other models such as sequence diagrams, class diagrams, etc. For this purpose, we provide an extension, which enables the IPME to (1) render net files in a graphical manner instead of showing the corresponding text representation and (2) compare two revisions of the same net file by highlighting their differences. The implementation of this extension is divided into two functional components. The first is designed as a Web service, which offers the capability to compute images of submitted nets that depict the requested functionality of the extension (detailed description in Section 3.3). The second component is located in the IPME as a plugin called Redmine Renew Plugin and integrates the functionality offered by the Web service into the web-based user interface (as described in Section 3.5). This is realized by extending the existing code view mechanisms so that net files supported by Renew (*.rnw, *.aip, *.draw, *.pnml, etc.) will be forwarded and handled by the corresponding Web service.

Each component of the tool chain can be assembled and hosted in a completely distributed environment, as depicted in Figure 2, or centralized on one host. The IPME is hosted on a web server that offers developers, managers and designers an easy access using a simple Web browser. Source code repositories are located on file servers and will be integrated into the environment with the help of the build-in source code management (SCM) connectors. The major workload of the net integration is done by the export server, which hosts the two Web services (Image Net Export and Image Net Diff).

[5] Redmine, A flexible project management web application (http://www.redmine.org)

[6] Trac, Integrated SCM & Project Management (http://trac.edgewall.org)

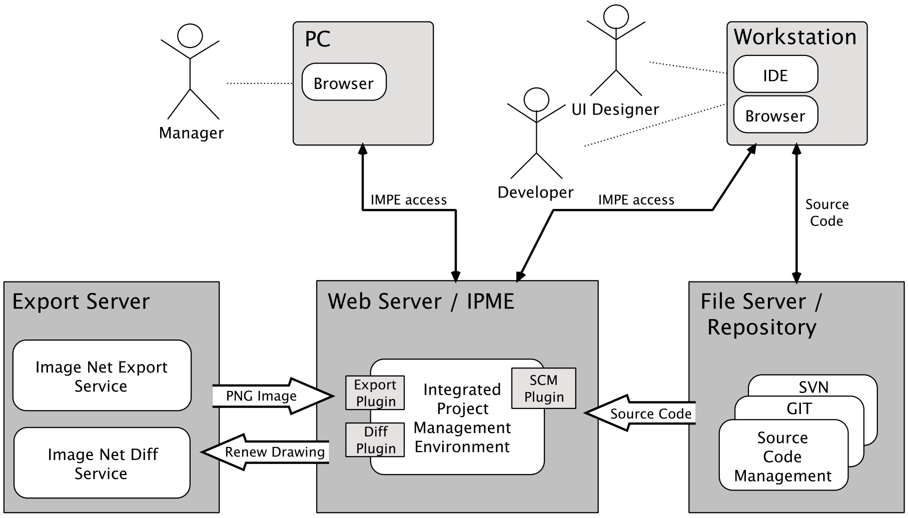


Figure 2. Web Service-based extension of the distributed tool chain environment.

3.2 Export and ImageNetDiff

Conversion of diagram / net representation into image representations is provided by two plugins available for Renew. The Export plugin is part of the distribution of Renew and makes use of the powerful Freehep libraries (see `http://www.freehep.org/`). It was first included in version 2.1 of Renew. The Export agent makes use of the PNG (Portable Network Graphics) export format, which is displayed directly in a Web browser.

The ImageNetDiff plugin for Renew is available as on optional plugin and allows producing a visual diff of two diagrams or nets. As the name suggests, the visual diff is represented as an image (compare with [7]) and is in fact produced through a comparison of two images, which are the result of an export operation. The functionality is available in Renew from the GUI or as command line argument, i.e. comparing two opened nets (or diagrams) as well as using the functionality in scripts is possible. Additionally, the comparison of a checked-out copy of a net against the revision base in a Subversion repository is directly supported. With the integration of the ImageNetDiff functionality in the IPME the comparison of arbitrary version in a repository is also easily possible. Instead of executing a diff on the textual representation and mapping the results back to the Petri net or diagram we make heavy use of the graphical (analogue) representation of the diagram. Thus the diff procedure in our implementation is not a semantic (or syntactic) diff but a purely visual one. On the one hand, this approach holds some disadvantage. Direct editing of nets on the basis of the diff representation is not possible and also our tool set does not support automatic merges. On the other hand, we can also derive some advantages from this approach. The implementation is very simple. In many relevant cases this approach shows the desired results. The results integrate directly into the Web-based applications. Note also, that the diff images are only transitional objects during the development of models / source code. For the means of comparison, they are only useful at the moment of creation. Once the desired information is found, e.g. the error is located, the diff object is deleted.

3.3 Web Service Implementation

Both services (Image Net Export Service and Image Net Diff Service) used in the Redmine Renew Plugin are realized as RESTful Web services [10]. This approach offers a high interoperability and simplifies the integration of services in distributed environments. The current implementation of the Export Server (see Figure 2) is based on Mulan/Capa [8]: a Petri net-based multi-agent framework. Thus, the functionality of both services is provided by an agent (called Export agent), which is able to offer its services as Web services. This is possible with the help of the Mulan WebGateway agent, which is acting between the export agent and the invoking Web client, in our case the Redmine Renew Plugin. As depicted in Figure 3 the WebGateway agent has two communication interfaces and is able to communicate on the right hand side with agents over FIPA-ACL[7] messages. On the left hand side the interface is implemented by a Jetty[8] Web server and is able to handle HTTP and WebSocket[9] connections. Besides the capability to register and manage agent Web services, the major purpose of the WebGateway is to translate incoming Web client messages to FIPA-ACL messages and vice versa. During agent service registration at the WebGateway, each service can submit a service representation in terms of ordinary JSP[10] Web sites. These sites serve as web-based service invocation applications and are available by opening the associated service URL in a Web browser.

To fulfill the desired behavior of the export agent services (Image Net Export Service and Image Net Diff Service), the implementation makes use of the Renew Export and ImageNetDiff plugins (mentioned in Section 3.2), which are available due to the fact that the Mulan multi-agent framework uses Renew as a runtime engine.

3.4 Redmine Renew Plugin Implementation

The current prototype of the Redmine Renew Plugin implements two different features, export and diff, which both work with source code files from the connected repository, as described in Section 3.1.

The visualization of a single net file is a relatively simple task and thus fairly simple to implement. The plugin has to fetch the net file from the repository and post the content of this file to the Web service to get a visual representation

[7] FIPA ACL: http://fipa.org/specs/fipa00061/SC00061G.html

[8] Jetty Web server: http://www.eclipse.org/jetty/

[9] W3C WebSocket API: http://dev.w3.org/html5/websockets/

[10] Java Server Page (JSP): http://www.oracle.com/technetwork/java/javaee/jsp/

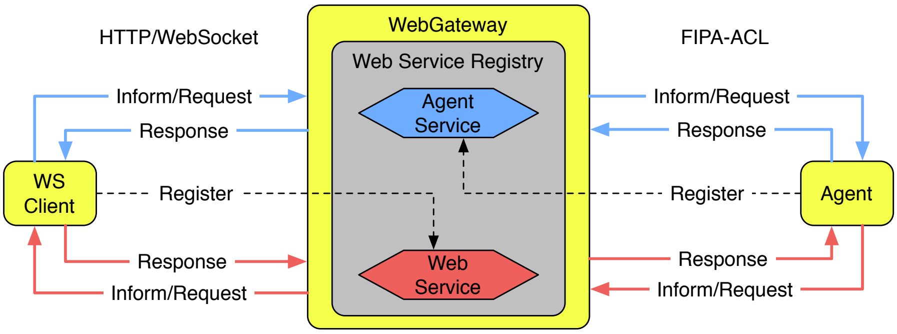


Figure 3. WebGateway Architecture.

of the net as result. This result, the received PNG image, will be shown to the user, embedded inside the repository view of Redmine (compare with Figure 4, in which a diff image is shown, which is explained in Section 3.5). The comparison of two revisions of the same net file from the repository does not make a big effort, too. In this case, the plugin has to fetch the two different revisions of the net file and post its contents to the Web service to receive the image with the highlighted differences.

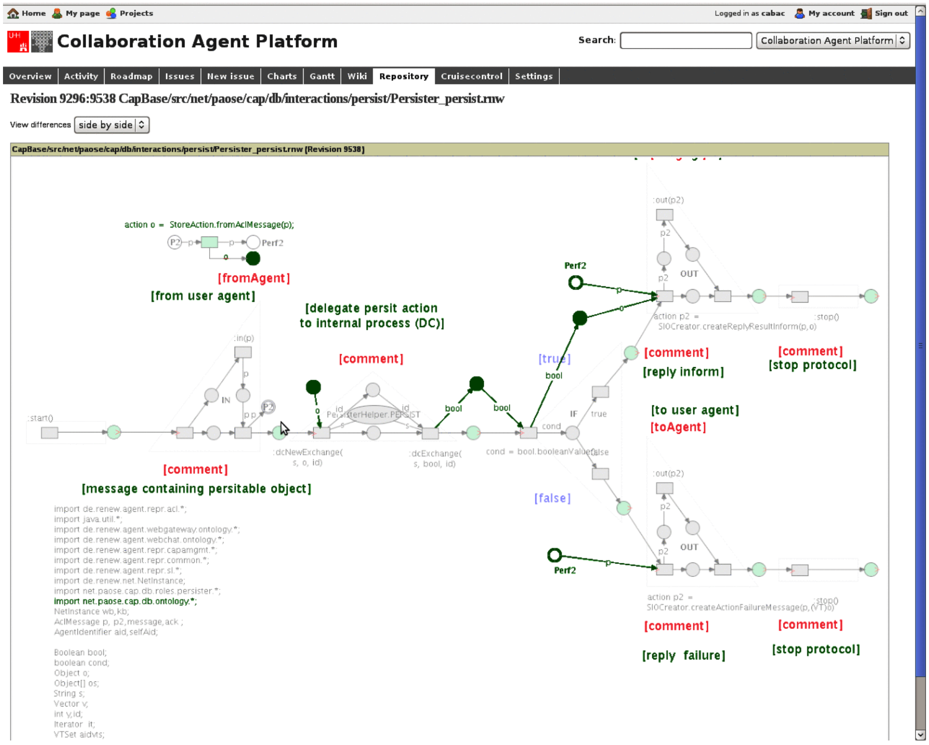
However, the case of comparing two complete changesets[11] is a complex task with regard to the technical aspect. This is firstly, because a changeset might contain graphical and textual source code changes, therefore the plugin has to decide for every file separately, if it will use the image net diff algorithm or the build-in textual one[12]. Secondly, the type of change can be one of the following for every single file in the changeset: addition, modification, deletion, duplication, moving or renaming. As a consequence, there are a lot of possible cases to take into account e.g. if we add a net file in one changeset, remove this file in the next one and after that we add a file with the same name but different contents again. Then, the complete history of changes made to this file name has to receive attention in order to decide which revisions should be compared. Another complex sequence of operations arises, when a file becomes renamed or moved. In this case the plugin has to fetch the previously named file and the current one in order to compare the changes, what results in a supplementary request to the repository.

[11] A changeset contains all changes, which were made in an atomic commit and contains multiple files in the majority of cases.

[12] The repository itself (i.e. Subversion) does not make any difference between file types and returns a single unified diff file for all changes made in a changeset. Therefore the Redmine Renew Plugin has to split this unified diff into separate parts, whereby every part contains the changes made to a single file.

3.5 Integration into the Existing Tool Chain

In the PAOSE approach we use project management tools that are already well-established in agile development approaches. By their usage, we directly apply the advantages of these tools to the agent-oriented development and benefit from the tight linking of source code with developer tasks and documentation, as requested within Section 2. Beyond that, we extend these tools with plugins according to the needs of the Petri net-based agent-oriented software development, which result from the model-driven nature of the Petri net-based development, as described in Section 3.1.



Revision 9296:9538 CapBase/src/net/paose/cap/db/interactions/persist/Persister_persist.rnw

View differences side by side

CapBase/src/net/paose/cap/db/interactions/persist/Persister_persist.rnw [Revision 9538]

```
import de.renew.agent.repr.acl.*;
import java.util.*;
import de.renew.agent.webgateway.ontology.*;
import de.renew.agent.webchat.ontology.*;
import de.renew.agent.repr.capamgmt.*;
import de.renew.agent.repr.common.*;
import de.renew.agent.repr.sl.*;
import de.renew.net.NetInstance;
import net.paose.cap.db.roles.persister.*;
import net.paose.cap.db.ontology.*;
NetInstance wb,kb;
AclMessage p, p2,message,ack ;
AgentIdentifier aid,selfAid;

Boolean bool;
boolean cond;
Object o;
Object[] os;
String s;
Vector v;
int y,id;
Iterator  it;
VTSet aidvts;
```

Figure 4. Redmine ImageNetDiff plugin in use.

Figure 4 shows a screenshot of the web-based IPME Redmine. The image shows the repository view extended by the Redmine Renew Plugin in diff mode. Inside the content area in the center, it shows the differences of two revisions (i.e. revisions 9296 and 9538) of a Petri net from the connected Subversion repository. The colored areas describe changed parts of the Petri net, whereby green[13] signifies added parts and red deleted parts, while the light-gray areas signify unchanged parts. The plugin integrates into the IMPE in a transparent way, without the need of changing anything in the general workflow. Additionally, the presentation of both, textual and graphical source code, is embedded into our IMPE homogeneously. The just described screenshot shows only one possible use of the plugin. There is also the ability to compare other graphical resources, such as use case diagrams or agent interaction protocols, which both were introduced to the Paose approach [5].

A demonstration of the new tool integrated into the IMPE Redmine is accessible under `https://paose.informatik.uni-hamburg.de/redmine/`. The connected Subversion repository contains some examples, which were developed in the context of the P*AOSE student project at the University of Hamburg. The presented IPME allows public users to access the source code, to visualize and to compare different versions.[14]

In Section 2 we described the developer tasks and discussed the need for tool support for the comparison of graphical source code (i.e. in our case Petri nets) and other diagrams in a visual way. The possibilities offered by our presented extensions especially support the developers in communication aspects and the exchange of information. Furthermore, reviewing and refactoring of code, for instance in order to discover and eliminate some bugs, are simplified, because the simple access to a visualization of the graphical source code is integrated into the applied IPME. Besides that, project leaders and other participants can inform themselves straightforwardly about the progress of development without the need of an IDE and thus also derive benefits from the integrated tool set.

A typical use case for the net diff inside the IPME is a session of finding the differences during a debugging session. However, one very useful simple use case is examining, whether changes were made in a model. For instance marginal changes in the layout (e.g. changing the z-order of elements) can result in repository check-ins. Also if changes from for example version A to version B are reverted in version C the control of absence of differences between version A and C and the easy access to this information is of great advantage.

4 Conclusion

In this work we investigated the development process in Petri net-based agent-oriented software development concerning the improvements of tool support. After outlining developer tasks and gaps in tool support, we introduced an extension to the tool set to accommodate the source code management of graphically represented source code, embedded into the widely used open source project management tool Redmine.

[13] A black and white print shows only light-gray parts and black net elements and inscriptions. With the exception of the comments (in square brackets) only additions have been made in the presented diff.

[14] Additionally the Web service-based functionality of diagram export and diagram image diff can be accessed manually via an included Web interface.

With the introduction of our extended tool chain in the context of the P*AOSE projects at the University of Hamburg, which follow the Paose approach, we improved various aspects of the developers' tasks. First of all, of course we supported the possibility to observe graphical source code and compare it concerning its different versions. Supplementary, we improved the speed of development through this feature, in respect of the possibility to visually compare model-based source code (i.e. in our case Petri nets) and other diagrams directly and effortlessly in an integrated environment for the project management and planning (IPME), which is closely linked to the SCM. Furthermore the presented tool integration supports the communication of the developers. Information exchange through source code changes between the developers gets easier than before and performs thereby the requirements delineated in Section 2. Finally, we come to the conclusion that our presented extended tool chain is one substantial step to improve the development of Petri net-based agent-oriented software. The gap between text-based and model-based source code – as well as other design models – is thus further closed in regard to the handling of these documents during development, reviewing, documentation and refactoring. The presented approach could easily be adapted to support other development approaches that make use of graphical representations (e.g. UML diagrams) during development.

References

1. Helmut Balzert. *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering.* Spektrum Akademischer Verlag, 3. aufl. edition, 2009.
2. Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. The agile manifesto. http://agilemanifesto.org.
3. W.G. Bleek and H. Wolf. *Agile Softwareentwicklung: Werte, Konzepte und Methoden.* dpunkt, Heidelberg, 2008.
4. Lawrence Cabac. Multi-agent system: A guiding metaphor for the organization of software development projects. In Paolo Petta, editor, *Proceedings of the Fifth German Conference on Multiagent System Technologies*, volume 4687 of *Lecture Notes in Computer Science*, pages 1–12, Leipzig, Germany, 2007. Springer-Verlag.
5. Lawrence Cabac. *Modeling Petri Net-Based Multi-Agent Applications*, volume 5 of *Agent Technology – Theory and Applications.* Logos Verlag, Berlin, 2010.
6. Lawrence Cabac, Till Dörges, Michael Duvigneau, Christine Reese, and Matthias Wester-Ebbinghaus. Application development with Mulan. In Daniel Moldt, Fabrice Kordon, Kees van Hee, José-Manuel Colom, and Rémi Bastide, editors, *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'07)*, pages 145–159, Siedlce, Poland, June 2007. Akademia Podlaska.
7. Lawrence Cabac and Jan Schlüter. ImageNetDiff: A visual aid to support the discovery of differences in Petri nets. In *15. Workshop Algorithmen und Werkzeuge für Petrinetze, AWPN'08*, volume 380 of *CEUR Workshop Proceedings*, pages 93–98. Universität Rostock, September 2008.
8. Michael Duvigneau, Daniel Moldt, and Heiko Rölke. Concurrent architecture for a multi-agent platform. In Fausto Giunchiglia, James Odell, and Gerhard

Weiß, editors, *Agent-Oriented Software Engineering. 3rd International Workshop, AOSE 2002, Bologna. Proceedings*, pages 147–159. ACM Press, July 2002.

9. Akhil Mehra, John Grundy, and John Hosking. A generic approach to supporting diagram differencing and merging for collaborative design. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 204–213, New York, NY, USA, 2005. ACM.
10. Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big" web services: making the right architectural decision. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 805–814, New York, NY, USA, 2008. ACM.
11. C. Petrie, S. Goldmann, and A. Raquet. Agent-based project management. *Artificial intelligence today*, pages 339–363, 1999.
12. Arne Schipper, Hauke Fuhrmann, and Reinhard von Hanxleden. Visual comparison of graphical models. *Engineering of Complex Computer Systems, IEEE International Conference on*, 0:335–340, 2009.
13. Trung Hung VO. Software development process. Available at: `http://cnx.org/content/m14619/1.2/`, July 2007.

On the use of Pragmatics for Model-based Development of Protocol Software

Kent Inge Fagerland Simonsen[1,2]

[1] Department of Computer Engineering, Bergen University College, Norway
kifs@hib.no
[2] DTU Informatics, Technical University of Denmark, Denmark
kisi@imm.dtu.dk

Abstract. Protocol software is important for much of the computer based infrastructure deployed today, and will remain so for the foreseeable future. With current modelling techniques for communication protocols, important properties are modelled and verified. However, most implementations are being done by hand even if good formal models exist. This paper discusses some of the challenges in modelling and automatically generating software for protocols. The challenges are discussed using the Kao-Chow authentication protocol as a running example by outlining an approach for generating protocol software for different platforms based upon Coloured Petri Nets (CPN). The basic idea of the approach is to annotate the CPN models with pragmatics which can be used in a code generator when mapping the constructs of the CPN model onto the target platform.

1 Introduction

Much work has been done to model and verify protocols using a wide range of formalisms [8]. Petri nets [23] and Coloured Petri Nets (CPNs) [13,14] in particular are widely used formal modelling languages for behavioural modelling and verification of industrial-sized protocols [7]. There exist, however, relatively few examples [16,17,21] where CPN models have been used as a basis for automatically obtaining implementations of the modelled protocols.

This paper describes challenges with automatically generating code from protocol models and proposes some avenues for solving them. A concept of *pragmatics* is introduced for protocol models which holds information useful for generating an implementation. This paper also proposes the use of separate models to describe the configuration and platform for protocol software. This allows the protocol models to be at a high level of abstraction while specific implementations can be derived using configuration and platform models.

Figure 1 illustrates our approach to generating protocol software. The Protocol Model is a model in a language that is not yet fully designed, but it could be based on CPN or another High Level Petri Net (HLPN) language. The Configuration Model contains information on which design choices should be made for the implementation of the protocol. For example, the configuration can contain information on exactly which underlying network layer service to use for communication between protocol entities. The Platform Model is intended to contain information on how operations should be implemented on the specific platform in question. For example the details on what is needed to set up and transmit messages over the User Datagram Protocol (UDP) [9] or the Transmission Control Protocol (TCP) [10]. The Protocol Model, together with a Configuration Model and Platform Model is fed into a Generator in order to obtain an implementation of the protocol. Finding a good separation between the Protocol, Configuration and Platform Models is one of the important challenges in this approach.

The evaluation of our approach will be based on the software we are able to generate using our approach. If we are able to generate software for a wide range of protocols with high quality, this will be considered a success. We will also evaluate the confidence we can gain that the generated software maintains the properties of the Protocol Model.

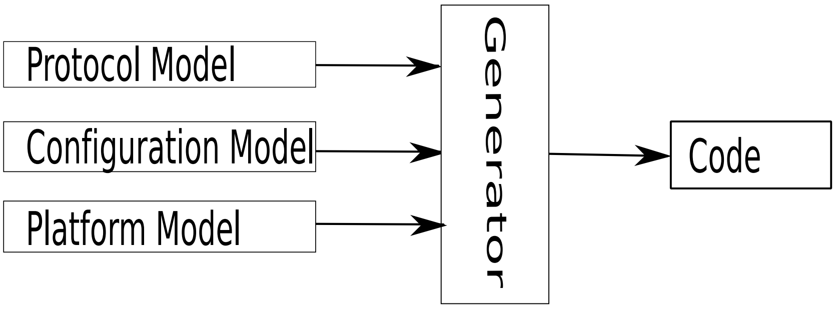


Fig. 1: Protocol software generation approach.

In order to include information that will help with code generation, we introduce the concepts of *pragmatics* and *scope* to the Protocol Model. Pragmatics assign special behaviour and meaning to model elements. This means that we are able to differentiate between transitions, places and data according to their function in the protocol. In the protocol models, pragmatics are encapsulated in ≪ and ≫. We will provide more details on these pragmatics in the following.

This paper is structured as follows. Section 2 focuses on elements that are missing from CPNs in order to model and generate code for protocols. This section also introduces the Kao-Chow (KC) authentication protocol [15] which is used as a running example throughout this paper. The concepts of pragmatics and scope are also introduced in this section. Section 3 discusses the need for configuration and platform models and identifies some elements that should be contained in those models. Finally, Section 4 discusses future work and identify criteria for evaluating our approach to model based development of protocol software. The reader is assumed to be familiar with the basic concepts of CPNs.

2 Protocol Model

To illustrate our approach we use the KC protocol. KC is a protocol that makes it possible for two entities, A and B, to authenticate each other using uncertified symmetric key [3] cryptography and an authentication server, S. The authentication server is assumed to have pre-shared keys with each of the authenticating entities. Listing 1 shows the basic sequence of messages exchanged in KC using Alice and Bob notation [20]. First some entity, A, wants to authenticate with another entity, B. To this end, A sends its and Bs identity together with a nonce[4], Na, to the authentication server, S (1). S then generates a session key, Kab, for use between A and B. This session key and A and B's identities, together with A's nonce is encrypted with the pre-shared key, Kbs, between B and S. S also creates a copy of the same data encrypted with the pre-shared key, Kas, between A and S and sends both copies to B (2). B then sends the part of the message it got from S encrypted with the key, Kas, shared by A and S to A together with As nonce encrypted with the session key, Kab, and a new nonce Nb (3). Finally, A responds to B with B's nonce, Nb, encrypted with the session key, Kab, (4). A considers B to be authenticated when the nonce, Na, it receives from A encrypted with Kab is identical to the Na which A created at the beginning of the exchange. Similarly, B considers A to be authenticated when B receives its nonce, Nb, encrypted with Kab from A.

Listing 1: Kao-Chow message sequece from [24]

```
1. A -> S: A, B, Na
2. S -> B: {A, B, Na, Kab}Kas, {A, B, Na, Kab}Kbs
3. B -> A: {A, B, Na, Kab}Kas, {Na}Kab, Nb
4. A -> B: {Nb}Kab
```

CPNs and other types of Petri Nets are widely used and have a well documented capability for modelling and verifying protocols and aiding in the implementation of protocol software [1, 7]. Our approach is to use HLPNs, and CPNs in particular, as a starting point for modelling protocols.

The top page of a CPN model of KC is shown in Fig. 2. Here the participants, A, B and Server, in the protocol are modelled as substitution transitions on the top level module. The places make it explicit that A send messages to Server, Server send messages to B, and that A and B send messages to each other.

Another effort to model KC using Petri Nets is presented in [3]. In this paper KC is first defined in the Security Protocol Language [6] and then translated into S-nets [4]. KC is also modelled using several different tools and languages in [5].

[3] Uncertified keys are keys that are not accompanied with information such as proof of who is the keys owner and issuer and how the key should be used.

[4] A nonce is a number or bit-string that is used only once.

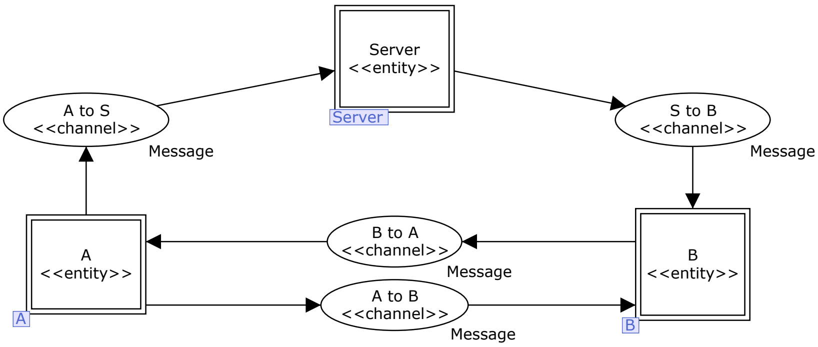


Fig. 2: Top level module of the Kao-Chow model.

2.1 Scope Pragmatics

Implicit in the KC model is information about different protocol entities that have different roles. Our approach is to make this information more explicit is to add the entity pragmatic to substitution transitions that indicate that the module is an entity in the protocol. In this approach, the top level of a model typically consists only of substitution transitions and network nodes, which is the case in the KC model in Fig. 2.

An alternate approach could be to tag all model elements that are part of the same protocol entity or to encompass all elements inside some form of field that delimits the entities from each other. One problem with this approach is that since several parties can exist on the same module some elements may interact without going through a network node. On one hand this could make models more error-prone. On the other hand such back-channels may be used to represent out of band communication that is relevant to the protocol and not properly network traffic. Since such out of band traffic could also be represented by non-network nodes in the top level anyway, this is not a strong argument against the chosen approach as explained in the previous paragraph.

2.2 Communication Channel Places

Network places, which have the channel pragmatic, represent the network and firing adjacent transitions corresponds to sending some data over the network. The sender and recipients are identified by the transitions on either side of the network places. Pragmatics on network places could, for example, include constraints on the network channel which corresponds to the assumptions made on the network used by the protocol. Such assumptions could be that package are guaranteed to arrive in order without duplicates, as TCP channels guarantee, or that there are no such guarantees, as is the case with UDP. Another example may be a constraint indicating that the channel is secure from an attacker being able to read the data which is provided by the Transport Layer Security (TLS) protocol [11]. How the communication channel should be initialized and used specifically should be specified in the configuration and platform models.

2.3 API Pragmatics

Figures 3 and 4 show the two modules for the A entity in the KC model. The behaviour of A is somewhat complex despite the simplicity of the protocol, because many steps are taken for each message. In the figures, pragmatics have been added to several elements in the model. Figure 3 shows how the protocol is initiated by placing a token on the place Init at the top of the figure. The token contains the addresses for A and B. These addresses are then combined with a nonce from the place Nonce and put on the place A to S which represent sending the message to the authentication server. At the same time, a copy of the nonce is placed on the WaitAuthenticate place. This place represents a state where A is waiting for a response from B.

When a message is placed on the place B to A, the transitions on the submodule associated with the Authenticate substitution transition become enabled. This submodule is shown in Fig. 4. Here the transition Receive Authentication (when enabled) stores As original Nonce and Bs Nonce in StoreA and StoreB, and then places a token on the Wait Decrypt place. Then the Decrypt Key transition can use the key in the place Key Store to decrypt the session key and nonce. The Authenticate transition is now able to perform the actual authentication of B. In the KC protocol, the authentication involves simply to check that the stored and the received and decrypted nonces are identical. The model does not explicitly show what should be done if the nonces are not the same. In practice, this would typically cause an error to be raised and the protocol would terminate. This is left out of the model in this paper for simplicity. Assuming that the authentication step is successful, Bs nonce is encrypted with the session key, which is generated by the authentication server and stored at the place Session Key Store, at the Encrypt Nonce transition, and finally put on the A to B place.

In the upper part of Fig. 3 there is a transition with the API pragmatic. This pragmatic symbolizes an entry point where the outside environment can interact with the protocol software. For target languages in the object-oriented paradigm this would typically be translated into a method with public access. In the KC example, the API pragmatic is the starting point of the protocol. It is given the name `kcAuthenticate` and takes two arguments; `toAddr` and `fromAddr`. Listing 2 shows an example of how the API transition could be translated into the signature of a Java method.

Listing 2: API method signature

```
public void kcAuthenticate(Id fromAddr, Id toAddr)
```

2.4 Operation Pragmatics

The operation pragmatic means the implementation should performing an operation such as printing data to the screen or calling a system library when this

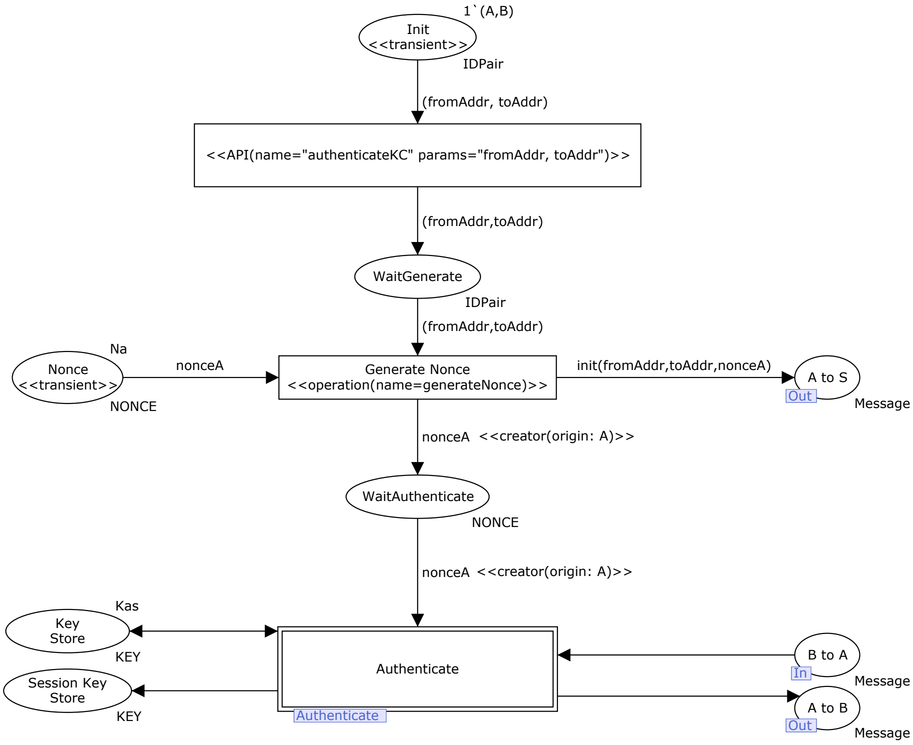


Fig. 3: Module A of Kao-Chow model.

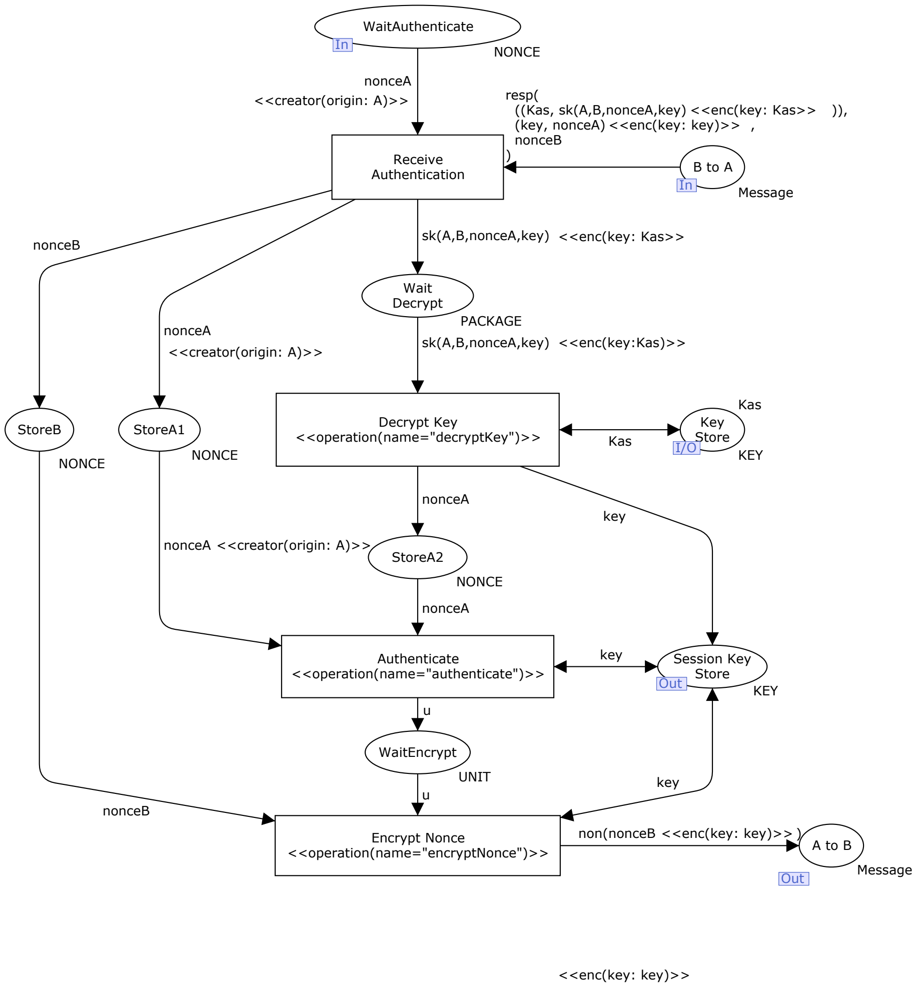


Fig. 4: Module Authenticate of Kao-Chow model.

pragmatic is encountered. Operation pragmatics are typically attached to transitions. The specific code that results from an operation in the Protocol Model is defined by the configuration and platform models.

In the KC protocol for entity A, there are four operation pragmatics. These are to generate a nonce (Generate Nonce in 3), encrypt (Encrypt Nonce in 4), decrypt (Decrypt Key in 4) and authenticate (Authenticate in 4). These pragmatics help to make explicit what operations are to be done for the transition with this pragmatic. Also it makes it possible for the Generator (see Fig. 1) to know how to generate code for these transitions, even if they are not modelled at the same level of detail as the implementation.

In an implementation, the Encrypt Nonce operation in the lower part of Figure 4 could be translated into what is shown in listing 3 on the Java platform where the encrypt method is already defined.

Listing 3: Encryption operation

```
String nonceReply = serverNonce.nonce.toString();
nonceReply = encrypt( nonceReply.bytes, sessionKey);
```

2.5 Transient Entities

Two places with the transient pragmatic are present in Figure 3. Model elements with the transient pragmatics are elements that are not considered by the generator, but may be useful for other uses of the model such as simulation and verification. The transient places in Figure 3 provide an initial state in the model which is necessary for simulation of the CPN model.

2.6 Data Pragmatics

In Fig. 4 several pieces of data have an enc pragmatic, for example on the arc between the Wait Decrypt place and the Decrypt Key transition in the middle of the figure. This pragmatic indicates that the data is encrypted with a given key. Encrypted data should only be used (read or manipulated) in transitions where the encryption key is available. In the KC example, encrypted data is only available after passing through a transition with a decryption operation where the correct key is available.

The enc pragmatic as shown here only takes symmetric encryption schemes into account. However, extending the pragmatic to also be able to model asymmetric encryption should be relatively simple. The enc pragmatic is an example of a domain specific pragmatic which is specific to the area of security protocols.

3 Configuration and Platform Models

Pragmatics in the model bring the model closer to an implementation by adding information that is useful for generating an implementation. Still the model is too abstract to generate code without making many assumptions about design choices and the underlying platform. We propose to use configuration and platform models to provide information so that the generator can generate an implementation.

The configuration model contains information about how to implement the protocol. It is likely to be highly dependent on both the protocol model and the platform model. It therefore seems possible that configuration models will not be reusable for other protocols or platforms. A typical design choice that will be represented in the configuration model is the choice of underlying network service to be used for communication between protocol entities. For example if for a protocol that has no constraints on the network layer service, a configuration would be whether to use UDP or TCP for the implementation.

The platform model should hold specific implementation details. In the example with the underlying network layer service, the platform model would hold information on how to set up, send and receive messages over UDP and TCP. The platform models are general in the sense that a platform model can be used to generate implementations of several protocols for the specific platform. In order to achieve this, the platform models, of course, need to support a wide range of features for different protocols and configurations.

Separating the configuration and platform models in this way makes it possible to reuse the models. Protocol models can be reused for different platforms and configurations. Platform models can also be reused to create protocol software for different protocols with different configurations for a specific platform.

4 Discussion

This paper has discussed some initial ideas for generating protocol software from models in a general way by annotating the model with pragmatics and adding configuration and platform information. This paper has also introduced a few specific pragmatics for protocol models that are exemplified by a model of the KC protocol. The list of pragmatics is by no means exhaustive, but provides a starting point for creating the first generation of technologies for protocol software modelling and generation using our approach. Additional information to be specified in configuration and platform models has also been introduced and argued for.

4.1 Related Work

In [19] a method for annotating CPNs is described. This method makes it possible to add auxiliary information to tokens in CPNs in layers of annotations. This approach is similar to the pragmatics presented in this paper in that both add information to CPNs. The approaches are different in that the pragmatics are added directly to the CPNs whereas the annotations in [19] are created and maintained separate from the underlying CPN model. Another difference is that the annotations are only concerned with tokens, while pragmatics can be added to places and transitions as well.

In [18] a restricted version of CPNs, called Colored Control Flow Nets (CCFN), are used to generate Java programs. This is done by first translating the CCFN to an intermediate model called a Annotated Java Workflow Net (AJWN) which is annotated by Java snippets derived from arc inscriptions in the corresponding CCFN.

In [17] a subclass of CPNs called Process-Partitioned CPNs (PP-CPNs) is introduced and used to automatically generate an implementation of the Dynamic MANET On-demand (DYMO) [12] routing protocol. The approach in [12] to generate code is to first translate the PP-CPN model into a control flow graph. The control flow graph is then used to construct an abstract syntax tree (AST) for an intermediate language which in turn is used to generate the AST of the target language. One difference to our approach is that in [17] information about the target platform and how translate model concepts to target language is contained in the generator instead of configuration and platform models. The method of [17] also defines a new subclass of CPNs instead of extending CPNs with annotations such as the pragmatics described here.

The notion of using different models for different layers of abstraction is also present in the Model Driven Architecture (MDA) [22] methodology of software engineering. In MDA three models are defined for a system. A Computation Independent Model (CIM) defines what a system is supposed to do and roughly corresponds to the protocol model as described in this paper. A Platform Independent Model (PIM) describes behaviour and structure of a system independent of the platform it is implemented on and a Platform Specific Model (PSM) combines the information in the PIM with all the details that are needed to generate an implementation of the system for the specified platform. The PIM and PSM are quite different from the configuration and platform models in this paper which do not include information on the software system itself, but rather design choices and how to implement these choices on the target platform for the given protocol model.

4.2 Future work

In the near future, we plan to use the KC model and manually simulate the code generation and then compare the implementation that is obtained through this simulation to an implementation that we have already created independently from the model. After that we will produce the first set of tools to automatically generate protocol software from HLPNs using the concepts of pragmatics and scope discussed in this paper as well as configuration and platform models.

Code generation will be done by model transformations. A significant challenge will be to gain confidence in the output of the generator. Formal verification of the generator will likely not be possible, but it is critical that we can maintain a high degree of confidence in the generated software. One technique that can be used to validate both the generator and the software it produces it to generate test suits based on the state space of the protocol code. Another technique is to rigorously test and examine several generated protocol implementations from several different protocol domains.

The protocol model itself should be verifiable. One approach to verifying protocols using CPNs has been described in [2]. We will study whether this and other approaches are applicable to protocol models with pragmatics as described in this paper. We will also look into how pragmatics can be used to help verify more properties about a protocol such as verifying that secret data is never places on a network channel in plain text and that the correct keys are always present to decrypt encrypted data.

References

1. Jonathan Billington, Michel Diaz, and Grzegorz Rozenberg, editors. *Application of Petri Nets to Communication Networks, Advances in Petri Nets*, volume 1605 of *Lecture Notes in Computer Science*. Springer, 1999.
2. Jonathan Billington, Guy Edward Gallasch, and Bing Han. A coloured petri net approach to protocol verification. In *Lectures on Concurrency and Petri Nets*, pages 210–290, 2003.
3. Roland Bouroulet, Raymond R. Devillers, Hanna Klaudel, Elisabeth Pelz, and Franck Pommereau. Modeling and analysis of security protocols using role based specifications and petri nets. In Kees M. van Hee and Rüdiger Valk, editors, *Petri Nets*, volume 5062 of *Lecture Notes in Computer Science*, pages 72–91. Springer, 2008.
4. Roland Bouroulet, Hanna Klaudel, and Elisabeth Pelz. A semantics of security protocol language (spl) using a class of composable high-level petri nets. In *ACSD*, pages 99–110. IEEE Computer Society, 2004.
5. Manuel Cheminod, Ivan Cibrario Bertolotti, Luca Durante, Riccardo Sisto, and Adriano Valenzano. Tools for cryptographic protocols analysis: A technical and experimental comparison. *Computer Standards & Interfaces*, 31(5):954–961, 2009.
6. Federico Crazzolara and Glynn Winskel. Events in security protocols. In *ACM Conference on Computer and Communications Security*, pages 96–105, 2001.
7. CPnets – Industrial Use. `http://cs.au.dk/cpnets/industrial-use/`.
8. G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International Editions, 1991.
9. Internet Engineering Task Force. *RFC768: User Datagram Protocol*, August 1980. `http://tools.ietf.org/html/rfc768`.
10. Internet Engineering Task Force. *RFC793: Transmission Controll Protocol*, September 1981. `http://tools.ietf.org/html/rfc793`.
11. Internet Engineering Task Force. *RFC5246: The Transport Layer Security (TLS) Protocol, Version 1.2*, August 2008. `http://tools.ietf.org/html/rfc5246`.
12. Internet Engineering Task Force. *Dynamic MANET On-demand (DYMO) Routing*, July 2010. `http://datatracker.ietf.org/doc/draft-ietf-manet-dymo/`.
13. K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009.
14. K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3-4):213–254, 2007.

15. I-Lung Kao and Randy Chow. An efficient and secure authentication protocol using uncertified keys. *Operating Systems Review*, 29(3):14–21, 1995.
16. L.M. Kristensen, P. Mechlenborg, L. Zhang, B. Mitchell, and G.E. Gallasch. Model-based Development of COAST. *STTT*, 10(1):5–14, 2007.
17. L.M. Kristensen and M. Westergaard. Automatic structure-based code generation from coloured petri nets: A proof of concept. In *Proc. of Int. Workshop on Formal Methods for Industrial Critical Systems*, volume 6371 of *Lecture Notes in Computer Science*, pages 215–230. Springer, 2010.
18. K. B. Lassen and S. Tjell. Translating colored control flow nets into readable java via annotated java workflow nets. In *Proc. 8th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2007)*, pages 39–58, 2007.
19. B. Lindstrøm and L. Wells. Annotating coloured petri nets. In *Proc. of the Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 39–58, 2002.
20. Sebastian Mödersheim. Algebraic properties in alice and bob notation. In *ARES*, pages 433–440. IEEE Computer Society, 2009.
21. K.H. Mortensen. Automatic Code Generation Method Based on Coloured Petri Net Models Applied on an Access Control System. In *Proc. of ATPN'00*, volume 1825 of *LNCS*, pages 367–386. Springer, 2000.
22. OMG Model Driven Architecture. *Web Site*. `http://www.omg.org/mda/`.
23. W. Reisig. *Petri Nets - An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.
24. Security Protocols Open Repository. Kao chow authentication v.1. `http://www.lsv.ens-cachan.fr/Software/spore/kaoChow1.html`.

Part IV

Poster Abstracts

A Goal Based Approach on top of Petri Nets

Nejm Saadallah and Benoit Daireaux

IRIS 4068 Stavanger Norway nejm.saadallah@iris.no , IRIS 4068 Stavanger Norway
benoit.daireaux@iris.no

Abstract. This poster presents ongoing work and mainly proposes a way to model goals on a Petri net model. We consider that the basic functioning of many machines can be captured in a Petri net model, while the environment where the machine is deployed is often too complex to be modelled in Petri nets. We handle the influence of the environment on the choice of operations by the so-called external agents, and show how these agent's goals could be studied before real deployment.

Keywords: *Control System, Agents, Supervisory Control, Petri Net, Model building*

1 Problem Formulation

The basic idea of our approach is to use Petri nets [6] to model the dynamic of machines [1, 3, 5, 4], regardless of the environments in which they are deployed, and use the notion of agents'goals to model the influences of the environments on the machine's behaviour. Our approach is motivated by two facts. The first fact is that basic functioning of many machines can be captured in a Petri net model. The second fact is that the environments in which a given machine is deployed are often hard to model within Petri nets. In contrast with synchronous Petri nets [2] which include sensory data into transition firing rules, we consider sensory data as inputs to external agents that need to interact with the machine.

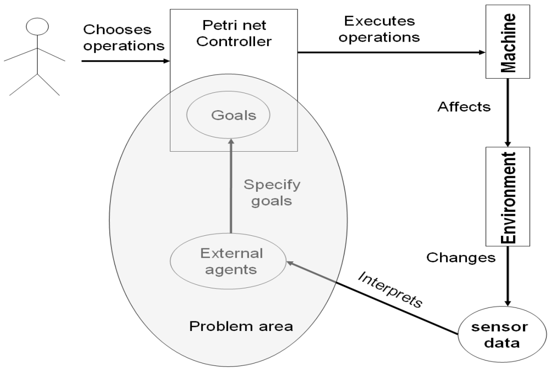


Fig. 1. Actions are chosen according to a logic control that is captured using the concept of control interpreted Petri nets. The machine commands affect the surrounding environment, which is reflected in sensory data. An external agent could be a software or human agent continuously analyses sensory data, and triggers actions.

In this work we are interested in finding a systematic approach for studying the behaviour of agents, based on their respective goals. Given a set of agents acting on a machine that is modelled in Petri net, how can we classify the behaviour of these agents? To the system designer, this approach is intended to give an understanding of the system prior to its implementation. Modelling the execution policy, that is, how the interaction between the agents and the Petri net model could be implemented is not addressed in this paper, but is shortly discussed in Section 3.

2 Modelling Goals

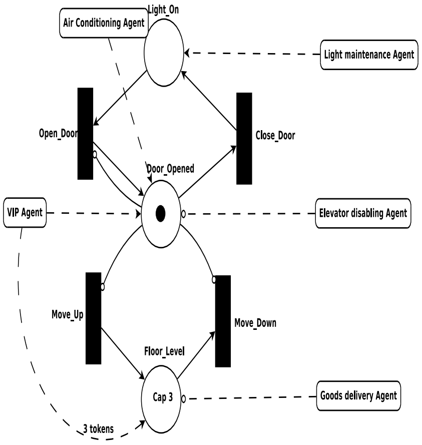


Fig. 2. Shows a simple elevator enhanced with 5 agents. **Light maintenance agent** will trigger every period of time to check whether the light lamp is still working or not. **Elevator disabling agent** will trigger if there is a fire in the building and no person in the elevator, to keep the door closed. **Goods delivery agent** moves the elevator to the floor 0. **VIP agent** requests the elevator to be at the third floor with the door open. **Air conditioning agent**, when triggered the door of the elevator stays open for a period of time.

The five agents shown in Figure 2, are categorised according to four relations as illustrated in Table 1. The four relations are defined as follows:
Distinctly Inclusive goals: We say that a goal g_a distinctly includes g_b if by achieving g_a, g_b is also achieved.
Mutually Inclusive goals: We say that two goals g_a and g_b are mutually inclusive if g_a distinctly includes g_b, and g_b distinctly includes g_a.
Partially Inclusive goals: We say that g_a partially include g_b, when only some markings that achieve g_a also achieve g_b but not all of them.
Mutually Exclusive goals: We say that two goals g_a and g_b are mutually exclusive when they can not be achieved together.

3 Conclusion and Future Work

In this paper we have addressed some aspects that are related to machines deployed in complex environments. We believe that the basic functioning of many

Table 1. Summary of the relations between the five goals of Figure 2

Goals	Mutual Exclusion	Mutual Inclusion	Distinct Inclusion	Partial Inclusion
Light maintenance g_1	g_4, g_5	g_2	g_2	g_3 via M_1
Elevator disabling g_2	g_4, g_5	g_1	g_1	g_3 via M_1
Goods delivery g_3	g_4	No	No	g_5 via M_0, g_1 and g_2 via M_1
VIP g_4	g_1,g_2,g_3	No	g_5	No
Air conditioning g_5	g_1,g_2	No	No	g_3 via M_0, g_4 via M_7

machines can be captured in a Petri net model, while the environments where the machines are deployed are often too complex to model in Petri nets. We consider that the influence of the environment on the machine is handled by agents, and raise the the following question: how agents acting on a machine can be categorised? We answer the question by introducing four properties, and use these properties to analyse some behavioural aspects, prior to system implementation as illustrated in Table 1.

We focused on the off-line analyses of agents acting on a machine, but we have not studied the on-line problematic, in other words the execution policy. When several agents are involved, which one should have the priority to execute? Another interesting question would be to find the set of necessary agents that guaranty some safety levels. To be more precise, could we provide a Petri net modelled machine with a set of safety agents, such that even in the presence of other faulty agents, the system guaranties the specified safety level? We believe that this work could be done within the Petri net formalism, and will be the subject of our future efforts.

References

1. Christos G. Cassandras and Stephane Lafortune. Introduction to discrete event systems. 2006.
2. René David and Hassane Alla. *Discrete, Continuous, and Hybrid Petri Nets.* Springer, 1 edition, November 2004.
3. B. Hrz and M. C. Zhou. *Modeling and Control of Discrete-event Dynamic Systems: with Petri Nets and Other Tools.* Springer Publishing Company, Incorporated, 2nd edition, 2007.
4. Vedran Kordic, editor. *Petri Net, Theory and Applications.* I-Tech Education and Publishing, 2007.
5. Tadao Murata. Petri nets: Properties, analysis and applications. pages 541–580, April 1989. NewsletterInfo: 33Published as Proceedings of the IEEE, volume 77, number 4.
6. Carl Adam Petri. *Communication with automata.* PhD thesis, Univ. Hamburg, 1966.

PNTM – Integration of Petri Nets and Transactional Memory

Weiyi Wu, Yao Zhang, Shengyuan Wang, and Yuan Dong

Department of Computer Science and Technology , Tsinghua University, Beijing, 100084, China
w1w2y3@gmail.com wwssyy@tsinghua.edu.cn

PNTM demonstrates a new concurrent programming model, providing explicit concurrency among cooperative transactions with correctness. It integrates a special Petri net and transactional memory, and improves the performance of transactions by decreasing the rate of conflicts. The GUI part of PNTM environment is based on PNK [1], the compiler is a modified GJC [2], and the runtime is based on DSTM2 [3].

Editor The IDE provides a simple GUI with a code editor and a net editor. The editor for Petri net system is modified from PNK. All elements can have extra fields and be edited visually. The extra fields such as resources in places and code in transitions all correspond special variables and functions in the code, as in Table 1.

Table 1. Extra fields in Petri nets' elements and corresponding elements in code

code in transition	`petrinet` function name
resource in place	`resource` variable
inscription in arc	`resource` variable

Virtual Machine The code editor can compile code along with Petri nets. Before compilation, the Petri nets are interpreted to internal representation for static check. In order to guarantee correctness at the level of Petri nets, the Petri net must meet constraints below:

1. All resources must have different names.
2. One resource should appear at no more than one place at any time.
3. The input arcs to one transition should have no common resource.
4. The output arcs from one transition should have no common resource. Besides, the resources in output arcs must be the subset of resources in input arcs.

After checking the correctness at the level of Petri nets, the editor will append a piece of code for building Petri Net simulator at runtime. Hence the simulator is created and initialized at runtime. The runtime provide 5 APIs as below:

1. `AddTransition(`transition`,` code`)`   adds transition.
2. `AddPlace(`place`,` resource`)`   adds place.
3. `AddArc(`source`,` target`,` inscription`)`   adds arc.
4. `Start()`   starts simulation.
5. `Join()`   terminates simulation.

The runtime is a Petri nets VM using DSTM2. The first 3 API can build up a simulator of a Petri net and the last 2 API control the simulator. The simulator allocate a DSTM2 `Thread` for every transition in the net. The `Threads` are all waiting for notification. Only notified `Thread` can consume resources, call function and produce new resources. A global lock is used to protect all resources in order to make manipulation on resources atomic and prevent deadlock. Some optimizations accelerate the check-and-consume process. Hence the overhead is relatively low. When the transition consumes resources and is ready to fire, corresponding `petrinet` function is called using reflection. If all `global` variables protected by STM successfully commit, the transition will produce new resources. Otherwise the transition will revert all state and return consumed resources.

Compilation At the early stage of compilation, the compiler will recognize and mark new keyword `petrinet`, `resource`, `global`. The `global` variables need to transform to instances of pre-defined interfaces in order to meet requirement of DSTM2's APIs. For example, equivalent code to transformed "`global int a;`" is shown in Table 2. `AInt` refers to "**A**tomic **Int**eger".

Table 2. Equivalent code to transformed code

original code	equivalent code	pre-defined interface

`global int a;`	`AInt a =`
	`factory_AInt.create();`
		`@atomic interface AInt {`
		`int getValue ();`
		`void setValue (int value);`
		`}`
		`Factory<AInt> factory_AInt =`
		`Thread.makeFactory(AInt.class);`

After AST is built, the compiler will check the semantic correctness. The functions with `petrinet` modifier and Petri nets themselves should meet constraints below:

1. `petrinet` function must have function body.
2. `petrinet` function should have no parameter.
3. Only `resource`, `global` and local variables can be used in a `petrinet` function.
4. Only `petrinet` function can be called in transition.
5. Resources in incoming arcs of a transition must be a superset of all `resource` variable used in corresponding `petrinet` function.

In addition, all references of `global` variables and all left-values consisted of `global` variables must be transformed to proper getter and setter in order to meet requirement of DSTM2's API. Equivalent code to getter and setter is shown in Table 3.

Table 3. Equivalent code to getter and setter

transform type	original code	equivalent code
initializer	`global int a = 0;`	`...;a.setValue(0);`
reference	x = a + 1;	x = a.getValue() + 1;
left-value	a = x + 1;	a.setValue(x + 1);
self-operation	a++;	a.setValue(a.getValue() + 1);

The rest of compilation is the same with GJC.

Future Work We are making efforts to some basic performance evaluation.

References

1. INA:Integrated Net Analyzer, at http://www.informatik.hu-berlin.de/lehrstuehle/automaten/ina.
2. GJC available at : http://www.sun.com/software/communitysource/j2se
3. Maurice Herlihy, Victor Luchangco, Mark Moir, A Flexible Framework for Implementing Software Transactional Memory, In Preceedings of OOPSLA'06, Pages 253-262, 2006.