

Editors: Lawrence Cabac,  
Michael Duvigneau and  
Daniel Moldt

Proceedings of the  
International Workshop on

**P**etri  
**N**ets and  
**S**oftware  
**E**ngineering  
**PNSE'12**

University of Hamburg  
Department of Informatics

These proceedings are published online by the editors as Volume 851 at

CEUR Workshop Proceedings  
ISSN 1613-0073  
<http://ceur-ws.org/Vol-851>

A printed version is published as FB-Mitteilung FBI-HH-M-346/12 by

Fachbereich Informatik  
Universität Hamburg  
Vogt-Kölln-Straße 30  
22527 Hamburg, Germany

Copyright for the individual papers is held by the papers' authors. Copying is permitted only for private and academic purposes. This volume is published and copyrighted by its editors.

## Preface

These are the proceedings of the International Workshop on *Petri Nets and Software Engineering* (PNSE'12) in Hamburg, Germany, June 25–26, 2012. It is a co-located event of *Petri Nets 2012*, the 33rd international conference on Applications and Theory of Petri Nets and Concurrency, and *ACSD 2012*, the 12th 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/pnse12/>

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)  
Didier Buchs (Switzerland)  
Lawrence Cabac (Germany) (Chair)  
Piotr Chrzastowski-Wachtel (Poland)  
Gianfranco Ciardo (USA)  
José-Manuel Colom (Spain)  
Jörg Desel (Germany)  
Raymond Devillers (Belgium)  
Michael Duvigneau (Germany) (Chair)  
Jorge C.A. de Figueiredo (Brasilia)  
Luís Gomes (Portugal)  
Stefan Haar (France)  
Xudong He (USA)  
Thomas Hildebrandt (Danmark)  
Kunihiko Hiraishi (Japan)  
Vladimir Janousek (Czech republic)  
Peter Kemper (USA)  
Hanna Kludel (France)

Radek Koci (Czech republic)  
Fabrice Kordon (France)  
Lars Kristensen (Norway)  
Johan Lilius (Finland)  
Niels Lohmann (Germany)  
Daniel Moldt (Germany) (Chair)  
Berndt Müller (Great Britain)  
Chun Ouyang (Australia)  
Wojciech Penczek (Poland)  
Laure Petrucci (France)  
Lucia Pomello (Italy)  
Heiko Rölke (Germany)  
Catherine Tessier (France)  
H.M.W. (Eric) Verbeek (Netherlands)

We received 27 high-quality contributions. The program committee has accepted eight of them for full presentation. Furthermore the committee accepted four papers as short presentations. Eight more contributions were accepted as posters.

The international program committee was supported by the valued work of Paulo Barbosa, Robin Bergenthum, Luca Bernardinello, Jean-Yves Didier, Bachir Djafri, Carlo Ferigato, Agata Janowska, Alban Linard, Edmundo Lopez, Romain Soulat, and Maciej Szreter as additional reviewers. Their work is highly appreciated.

Furthermore, we would like to thank our colleagues in the local organization team here at the University of Hamburg, Germany, for their support.

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

Thanks!

Lawrence Cabac, Michael Duvigneau, Daniel Moldt  
Hamburg, June 2012

---

## Contents

---

### Part I Invited Talks

---

<b>What Should we Teach About Petri Nets?</b> <i>Wolfgang Reisig</i> .....	11
---	----

---

### Part II Long Presentations

---

<b>Using Integer Time Steps for Checking Branching Time Properties of Time Petri Nets</b> <i>Agata Janowska, Wojciech Penczek, Agata Pótrola and Andrzej Zbrzezny</i>	15
<b>Grade/CPN: Semi-automatic Support for Teaching Petri Nets by Checking Many Petri Nets Against One Specification</b> <i>Michael Westergaard, Dirk Fahland and Christian Stahl</i> .....	32
<b>When Can We Trust a Third Party? - A Soundness Perspective</b> <i>Kees van Hee, Natalia Sidorova and Jan Martijn van der Werf</i> .....	47
<b>Modeling and Analyzing Wireless Sensor Networks with VeriSensor</b> <i>Yann Ben Maissa, Fabrice Kordon, Salma Mouline and Yann Thierry-Mieg</i> .....	60
<b>SMT-based parameter synthesis for L/U automata</b> <i>Michał Knapik and Wojciech Penczek</i> .....	77
<b>Model-Driven Middleware Support for Team-Oriented Process Management</b> <i>Matthias Wester-Ebbinghaus and Michael Köhler-Bußmeier</i> .....	93

<b>From Code to Coloured Petri Nets: Modelling Guidelines</b> <i>Anna Dedova and Laure Petrucci</i> .....	109
<b>Hierarchy of persistency with respect to the length of action's disability</b> <i>Kamila Agata Barylska and Edward Ochmański</i> .....	125
<hr/>	
<b>Part III Short Presentations</b>	
<hr/>	
<b>Local state refinement on Elementary Net Systems: an approach based on morphisms</b> <i>Luca Bernardinello, Elisabetta Mangioni and Lucia Pomello</i> .....	141
<b>Context Petri Nets: Enabling Consistent Composition of Context-dependent Behavior</b> <i>Nicolás Cardozo, Jorge Vallejos, Sebastián González, Kim Mens and Theo D'Hondt</i> .....	156
<b>MuPSi - a multitouch Petri net simulator for transition steps</b> <i>Thomas Irgang, Andreas Harrer and Robin Bergenthum</i> .....	171
<b>PetriPad – A Collaborative Petri Net Editor</b> <i>Julian Burkhart and Michael Haustermann</i> .....	182
<hr/>	
<b>Part IV Poster Abstracts</b>	
<hr/>	
<b>Agentworkflows for Flexible Workflow Execution</b> <i>Thomas Wagner</i> .....	199
<b>Cloud Transition: Integrating Cloud Calls into Workflow Petri Nets</b> <i>Sofiane Bendoukha and Thomas Wagner</i> .....	215
<b>A Concurrent Simulator for Petri Nets Based on the Paradigm of Actors of Hewitt</b> <i>Luca Bernardinello and Francesco Adalberto Bianchi</i> .....	217
<b>A Petri Net Approach to Synthesize Intelligible State Machine Models from Choreography</b> <i>Toshiyuki Miyamoto and Yasuwo Hasegawa</i> .....	222
<b>SYNOPS - Generation of Partial Languages and Synthesis of Petri Nets</b> <i>Robert Lorenz, Markus Huber, Christoph Etzel and Dan Zecha</i> .....	237

**Modeling and Simulation-Based Design Using Object-Oriented Petri Nets: A Case Study**  
*Radek Kočí and Vladimír Janoušek* ..... 253

**Porting the Renew Petri Net Simulator to the Operating System Android**  
*Dominic Dibbern* ..... 267

**SonarEditor: A Tool for Multi-Agent-Organizations Modelling**  
*Jan Bolte* ..... 269





**Invited Talks**



# What should we teach about Petri nets?

Wolfgang Reisig

Institut für Informatik, Humboldt-Universität zu Berlin  
Theory of Programming  
`reisig@informatik.hu-berlin.de`

## Abstract

I challenge the traditional choice of topics and the usual style of presentation for introductory courses on Petri nets. For such a course I suggest a number of aspects that usually are not considered fundamental. This includes faithful models, a slight revision of formalisms and terminology, specific techniques to increase the expressive power of Petri net models, aspects derived from distributed runs, and particular mathematics to specify and verify properties of Petri net models.



Long Presentations



# Using Integer Time Steps for Checking Branching Time Properties of Time Petri Nets

Agata Janowska<sup>1</sup>, Wojciech Penczek<sup>2</sup>, Agata Półrola<sup>3</sup>, and Andrzej Zbrzezny<sup>4</sup>

<sup>1</sup> Institute of Informatics, University of Warsaw, Banacha 2, 02-097 Warsaw, Poland  
janowska@mimuw.edu.pl

<sup>2</sup> Institute of Computer Science, PAS, Ordona 21, 01-237 Warsaw, Poland  
penczek@ipipan.waw.pl

<sup>3</sup> University of Łódź, FMCS, Banacha 22, 90-238 Łódź, Poland  
polrola@math.uni.lodz.pl

<sup>4</sup> Jan Długosz University, IMCS, Armii Krajowej 13/15, 42-200 Częstochowa, Poland  
a.zbrzezny@ajd.czyst.pl

**Abstract.** Verification of timed systems is an important subject of research, and one of its crucial aspects is the efficiency of the methods developed. Extending the result of Popova which states that integer time steps are sufficient to test reachability properties of time Petri nets [5, 8], in our work we prove that the discrete-time semantics is also sufficient to verify ECTL\* and ACTL\* properties of TPNs with the dense semantics. To show that considering this semantics instead of the dense one is profitable, we compare the results for SAT-based bounded model checking of ACTL<sub>X</sub> properties and the class of distributed time Petri nets.

## 1 Introduction

Verification of time-dependent systems is an important subject of research. The crucial problem to deal with is the state explosion: the state spaces of these systems are usually very large due to infinity of the dense time domain, and are likely to grow exponentially in the number of concurrent components of the system. This influences strongly the efficiency of the model checking methods.

The papers of Popova [5, 8] show that in the case of checking reachability for systems modelled by time Petri nets (i.e., while testing whether a marking of a net is reachable) one can use discrete (integer) time steps instead of real-valued ones. This reduces the state space to be searched. The aim of our work is to investigate whether the result of Popova can be extended, i.e., whether the discrete-time semantics can replace the dense-time one also while verifying a wider class of properties of dense-time Petri net systems. In this paper we present our preliminary result, i.e., prove that the discrete-time model can be used instead of the dense-time one while verifying ECTL\* and ACTL\* properties. To show that such an approach can be profitable we perform some experiments, using an implementation for SAT-based bounded model checking of ACTL<sub>X</sub>

and the class of distributed time Petri nets with the discrete-time semantics [4], as well as its modification for the dense-time case.

The rest of the paper is organised as follows: Sec. 2 discusses the related work. Sec. 3 introduces time Petri nets and their dense and discrete models. Sec. 4 presents the logics ECTL\* and ACTL\*. Sec. 5 deals with the theoretical considerations, while Sec. 6 presents the experimental results. Sec. 7 contains final remarks and sketches directions of the further work.

## 2 Related Works

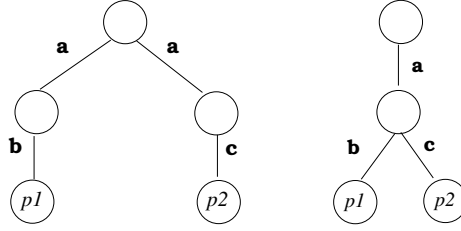
We would like to stress that in our work we are interested in branching time properties. To our best knowledge the fact that the discrete-time semantics is sufficient to verify ECTL\* or ACTL\* properties of time Petri nets (TPNs) with the dense-time semantics has never been proven before.

The topic of verification of dense-time Petri nets using integer time steps has been studied in several publications. In paper [5] it is shown how to construct a reachability graph whose vertices are reachable integer states for time Petri nets in which all the latest firing times are finite. The main theorem of [5] (Thm 3.2) states that for each run of a TPN, starting at its initial state, it is possible to find a corresponding run which starts at the initial state as well, and visits integer states only. Due to this theorem a discrete analysis of boundedness and liveness of a TPN is possible. The work [6] extends the results of [5] to arbitrary TPNs. It uses the idea of “freezing” the clock values of transitions with infinite *Lft* just as their *Eft* is reached. This way a reduced (finite) reachability graph of “essential” (integer) states is obtained.

In [8] and [7] the state space of a TPN is characterised parametrically. The main theorems (Thm 3.1 and Thm 3.2) of [8] state that for an arbitrary feasible execution path where the clocks have real values it is possible to replace these real values by integer ones and obtain another feasible path. The differences between the clocks values of each enabled transition at a given marking in the former and the latter path are always smaller than 1, and so are the differences between total times of both the executions. The main idea of the proof is as follows: the integer values are constructed out of the given assignments of real values by successive transforming all the non-integer numbers to nearby integers in  $n + 1$  steps, where  $n$  is the length of the path. According to the theorems the minimal and maximal time duration of a transition sequence are integer values. In the paper [7] an enumerative procedure for reducing the state space is introduced. The idea is to divide a problem into a finite number of smaller problems, which can be solved recursively with a methodology inspired from dynamic programming. Moreover, it extends the method of [8] to the nets with real-valued time steps (in [8] rational time steps were allowed only) and infinite latest firing times.

The authors of the above-mentioned papers claim that the knowledge of the reachable integer states is sufficient to determine the entire behaviour of the net at any point in time. However, all these papers show the trace equivalence





**Fig. 1.** Two trace equivalent models which are not (bi)similar. A formula distinguishing them is e.g.  $\varphi = \text{EF}(\text{EX}p1 \wedge \text{EX}p2)$  which holds for the model on the right only

between a continuous model and a (restricted) discrete one. It is very well known that trace equivalence preserves linear time properties, but it does not preserve branching time properties (see Fig. 1 and [1]), so the word “behaviour” should probably be understood in a way following from a fragment of [7]: “*The properties of a Petri net, both the classical one as well as the TPN, can be divided into two parts: There are static properties, like being pure, ordinary, free choice, extended simple, conservative, etc., and there are dynamic properties like being bounded, live, reachable, and having place- or transitions invariants, deadlocks, etc. While it is easy to prove the static behavior of a net using only the static definition, the dynamic behavior depends on both the static and dynamic definitions and is quite complicated to prove.*”, so as the dynamic properties listed. Moreover, the result of the papers [5, 6] does not imply bisimulation between both the models, as the construction given in these papers cannot be used to prove it. We discuss this on p. 27, showing that the relation  $\mathcal{R}$  used in our proof and derived from the result of [5, 6] cannot be used to prove bisimulation. This follows from the fact that the integer run  $\pi'$  “justifying”  $\sigma' \mathcal{R} \sigma$  (generated according to the construction of [5]) and the dense run  $\pi$  occurring in the relation do not need to “branch” in the same way. Similarly, the result of [8, 7] does not imply (bi)simulation as well. Although it is not stated directly, the construction given in these papers is based on a parametric description of the classes of the *forward-reachability graph* for a net considered (i.e., a structure in which the initial state class contains the initial state of the net and all the time successors of this state, and given a state class  $C_x$  corresponding to firing a sequence of transitions  $x$ , its successor class on a transition  $t$  contains all the concrete states which can be obtained by firing  $t$  at a concrete state  $\sigma \in C_x$  and then passing some time not disabling any enabled transition). It is well known that such a structure preserves reachability and linear time properties, but it does not preserve branching time properties. The discrete runs constructed in both the papers are retrieved from the dense ones to preserve nothing but visiting the same state classes as the runs they correspond to.

The current paper is a modified and improved version of our work [3] (published in the proceedings of a local workshop, and containing a completely different proof which does not define simulation explicitly).

### 3 Time Petri Nets

We start from introducing some basic definitions related to time Petri nets. For simplicity of the presentation we focus on 1-safe time Petri nets only. However, our result applies also to unbounded nets, which is explained in more details in the final section.

Let  $\mathbb{N}$  be the set of natural numbers (including zero), and  $\mathbb{R}$  ( $\mathbb{R}_+$ ) be the set of (nonnegative) reals. Time Petri nets are defined as follows:

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

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 *enabled* at  $m$  ( $m[t]$  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  $m[t]$  as well, if this does not lead to misunderstanding.
- $en(m) = \{t \in T \mid m[t]\}$ ;
- for  $t \in en(m)$ ,  $newly\_en(m, t) = \{u \in T \mid u \in en(m[t]) \wedge (t \bullet \cap \bullet u \neq \emptyset \vee u \bullet \cap \bullet t \neq \emptyset)\}$ .

Concerning the behaviour of time Petri nets, it is possible to consider a dense-time semantics, i.e. the one in which the time steps can be of an arbitrary (nonnegative) real-valued length, and the discrete one which considers integer time passings only. Below we define both of them.

#### 3.1 Dense-Time Semantics

In the dense-time semantics (the *dense semantics* in short) a *concrete state*  $\sigma$  of a net  $\mathcal{N}$  is defined as a pair  $(m, clock)$ , where  $m$  is a marking, and  $clock : T \rightarrow \mathbb{R}_+$  is a function which for each transition  $t \in en(m)$  gives the time elapsed since  $t$  became enabled most recently, and assigns zero to other transitions. Given a state  $(m, clock)$  and  $\delta \in \mathbb{R}_+$ , denote by  $clock + \delta$  the function defined by  $(clock + \delta)(t) = clock(t) + \delta$  for each  $t \in en(m)$ , and  $(clock + \delta)(t) = 0$  otherwise. By  $(m, clock) + \delta$  we denote  $(m, clock + \delta)$ . The *dense concrete state space* of  $\mathcal{N}$  is a structure  $(T \cup \mathbb{R}_+, \Sigma, \sigma^0, \rightarrow_r)$ , where  $\Sigma$  is the set of all the concrete states of  $\mathcal{N}$ ,  $\sigma^0 = (m^0, clock^0)$  with  $clock^0(t) = 0$  for each  $t \in T$  is the initial state of  $\mathcal{N}$ , and  $\rightarrow_r \subseteq \Sigma \times (T \cup \mathbb{R}_+) \times \Sigma$  is a timed consecution relation defined by:

- for  $\delta \in \mathbb{R}_+$ ,  $(m, clock) \xrightarrow{\delta}_r (m, clock + \delta)$  iff  $(clock + \delta)(t) \leq Lft(t)$  for all  $t \in en(m)$  (*time successor*),
- for  $t \in T$ ,  $(m, clock) \xrightarrow{t}_r (m', clock')$  iff  $t \in en(m)$ ,  $Eft(t) \leq clock(t) \leq Lft(t)$ ,  $m' = m[t]$ , and for all  $u \in T$  we have  $clock'(u) = 0$  for  $u \in newly\_en(m, t)$  and  $clock'(u) = clock(u)$  otherwise (*action successor*).

Notice that firing of a transition takes no time.

Given a set of propositional variables  $PV$ , we introduce a valuation function  $V : \Sigma \rightarrow 2^{PV}$  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  $p \in P$ , and use the same names for the propositions and the places. The function  $V$  is then defined by  $p \in V(\sigma)$  iff  $p \in m$  for each  $\sigma = (m, \cdot)$ . The structure  $M_r(\mathcal{N}) = (T \cup \mathbb{R}_+, \Sigma, \sigma^0, \rightarrow_r, V)$  is a *dense concrete model* of  $\mathcal{N}$ .

A *dense  $\sigma$ -run* of TPN  $\mathcal{N}$  is a (maximal) sequence of states:  $\sigma_0 \xrightarrow{a_0}_r \sigma_1 \xrightarrow{a_1}_r \sigma_2 \xrightarrow{a_2}_r \dots$ , where  $\sigma_0 = \sigma \in \Sigma$  and  $a_i \in T \cup \mathbb{R}_+$  for each  $i \geq 0$ . A state  $\sigma$  is reachable in  $M_r(\mathcal{N})$  if there is a dense  $\sigma^0$ -run  $\sigma_0 \xrightarrow{a_0}_r \sigma_1 \xrightarrow{a_1}_r \sigma_2 \xrightarrow{a_2}_r \dots$  such that  $\sigma = \sigma_i$  for some  $i \in \mathbb{N}$ .

### 3.2 Discrete-Time Semantics

Alternatively, one can consider integer time passings only. In such a *discrete-time semantics* (*discrete semantics* in short) a (*discrete*) *concrete state*  $\sigma_n$  of a net  $\mathcal{N}$  is a pair  $(m, clock_n)$ , where  $m$  is a marking, and  $clock_n : T \rightarrow \mathbb{N}$  is a function which for each transition  $t \in en(m)$  gives the time elapsed since  $t$  became enabled most recently, and assigns zero to the other transitions. Given a state  $(m, clock_n)$  and  $\delta \in \mathbb{N}$ , we define  $clock_n + \delta$  and  $(m, clock_n) + \delta$  analogously as in the dense-time case. The *discrete concrete state space* of  $\mathcal{N}$  is a structure  $(T \cup \mathbb{N}, \Sigma_n, \sigma_n^0, \rightarrow_n)$ , where  $\Sigma_n$  is the set of all the discrete concrete states of  $\mathcal{N}$ ,  $\sigma_n^0 = (m^0, clock_n^0)$  with  $clock_n^0(t) = 0$  for each  $t \in T$  is the initial state of  $\mathcal{N}$ , and  $\rightarrow_n \subseteq \Sigma_n \times (T \cup \mathbb{N}) \times \Sigma_n$  is a timed consecution relation defined by:

- for  $\delta \in \mathbb{N}$ ,  $(m, clock_n) \xrightarrow{\delta}_n (m, clock_n + \delta)$  iff  $(clock_n + \delta)(t) \leq Lft(t)$  for all  $t \in en(m)$  (*time successor*),
- for  $t \in T$ ,  $(m, clock_n) \xrightarrow{t}_n (m', clock_n')$  iff  $t \in en(m)$ ,  $Eft(t) \leq clock_n(t) \leq Lft(t)$ ,  $m' = m[t]$ , and for all  $u \in T$  we have  $clock_n'(u) = 0$  for  $u \in newly\_en(m, t)$  and  $clock_n'(u) = clock_n(u)$  otherwise (*action successor*).

Again, firing of a transition takes no time.

Given a set of propositional variables  $PV$ , we introduce valuation function  $V_n : \Sigma_n \rightarrow 2^{PV}$  which assigns the same propositions to the states with the same markings. Similarly as in the dense case, we assume the set  $PV$  to be such that each  $q \in PV$  corresponds to exactly one  $p \in P$ , and use the same names for the propositions and the places. The function  $V_n$  is then defined by  $p \in V_n(\sigma_n)$  iff  $p \in m$  for each  $\sigma_n = (m, \cdot)$ . The structure  $M_n(\mathcal{N}) = (T \cup \mathbb{N}, \Sigma_n, \sigma_n^0, \rightarrow_n, V_n)$  is a *discrete concrete model* of  $\mathcal{N}$ .

A *discrete*  $\sigma_n$ -run of TPN  $\mathcal{N}$  is a (maximal) sequence of states:  $\sigma_{n0} \xrightarrow{a_0} \sigma_{n1} \xrightarrow{a_1} \sigma_{n2} \xrightarrow{a_2} \dots$ , where  $\sigma_{n0} = \sigma_n \in \Sigma_n$  and  $a_i \in T \cup \mathbf{N}$  for each  $i \geq 0$ . A state  $\sigma_n$  is *reachable* in  $M_n(\mathcal{N})$  iff there is a  $\sigma_n^0$ -run of  $\mathcal{N}$   $\sigma_{n0} \xrightarrow{a_0} \sigma_{n1} \xrightarrow{a_1} \sigma_{n2} \xrightarrow{a_2} \dots$  such that  $\sigma_n = \sigma_{ni}$  for some  $i \in \mathbf{N}$ .

## 4 Temporal Logics ACTL\* and ECTL\*

In our work we deal with verification of properties of time Petri nets expressed in certain sublogics of the standard branching time logic CTL\*. Below, we define the logics of our interest.

### 4.1 Syntax and Sublogics of CTL\*

Let  $PV = \{\wp_1, \wp_2, \dots\}$  be a set of propositional variables. The language of CTL\* is given as the set of all the state formulas  $\varphi_s$  (interpreted at states of a model), defined using path formulas  $\varphi_p$  (interpreted along paths of a model), by the following grammar:

$$\begin{aligned} \varphi_s &:= \wp \mid \neg\varphi_s \mid \varphi_s \wedge \varphi_s \mid \varphi_s \vee \varphi_s \mid A\varphi_p \mid E\varphi_p \\ \varphi_p &:= \varphi_s \mid \varphi_p \wedge \varphi_p \mid \varphi_p \vee \varphi_p \mid X\varphi_p \mid U(\varphi_p, \varphi_p) \mid R(\varphi_p, \varphi_p). \end{aligned}$$

In the above  $\wp \in PV$ , A ('for All paths') and E ('there Exists a path') are path quantifiers, whereas U ('Until') and R ('Release') are state operators. Intuitively, the formula  $X\varphi_p$  specifies that  $\varphi_p$  holds in the next state of the path, whereas  $U(\varphi_p, \psi_p)$  expresses that  $\psi_p$  eventually occurs and that  $\varphi_p$  holds continuously until then. The operator R is dual to U: the formula  $R(\varphi_p, \psi_p)$  says that either  $\psi_p$  holds always or it is released when  $\varphi_p$  eventually occurs. Derived operators are defined as  $G\varphi_p \stackrel{def}{=} R(\text{false}, \varphi_p)$  and  $F\varphi_p \stackrel{def}{=} U(\text{true}, \varphi_p)$ , where  $\text{true} \stackrel{def}{=} \wp \vee \neg\wp$ , and  $\text{false} \stackrel{def}{=} \wp \wedge \neg\wp$  for an arbitrary  $\wp \in PV$ . Intuitively, the formula  $F\varphi_p$  specifies that  $\varphi_p$  occurs in some state of the path ('Finally'), whereas  $G\varphi_p$  expresses that  $\varphi_p$  holds in all the states of the path ('Globally').

Next, we define some sublogics of CTL\*:

ACTL\* : the fragment of CTL\* in which the state formulas are restricted such that negation can be applied to propositions only, and the existential quantifier E is not allowed,

ECTL\* : the fragment of CTL\* in which the state formulas are restricted such that negation can be applied to propositions only, and the universal quantifier A is not allowed,

ACTL : the fragment of ACTL\* in which the temporal formulas are restricted to positive boolean combinations of  $A(\varphi U \psi)$ ,  $A(\varphi R \psi)$ , and  $AX\varphi$  only.

ECTL : the fragment of ECTL\* in which the temporal formulas are restricted to positive boolean combinations of  $E(\varphi U \psi)$ ,  $E(\varphi R \psi)$  and  $EX\varphi$  only.

$L_{-X}$  denotes the logic **L** without the next-step operator X.

## 4.2 Semantics of CTL\*

Let  $PV$  be a set of propositions. A *model* for CTL\* is a tuple  $M = (L, S, s^0, \rightarrow, V)$ , 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<sup>5</sup>, and  $V : S \rightarrow 2^{PV}$  is a valuation function. For  $s, s' \in S$  the notation  $s \rightarrow s'$  means that there is  $l \in L$  such that  $s \xrightarrow{l} s'$ . Moreover, for  $s_0 \in S$  a *path*  $\pi = (s_0, s_1, \dots)$  is an infinite sequence of states in  $S$  starting at  $s_0$ , where  $s_i \rightarrow s_{i+1}$  for all  $i \geq 0$ ,  $\pi_i = (s_i, s_{i+1}, \dots)$  is the  $i$ -th suffix of  $\pi$ , and  $\pi(i) = s_i$ .

Given a model  $M$ , a state  $s$ , and a path  $\pi$  of  $M$ , by  $M, s \models \varphi$  ( $M, \pi \models \varphi$ ) we mean that  $\varphi$  holds in the state  $s$  (along the path  $\pi$ , respectively) of the model  $M$ . The model is sometimes omitted if it is clear from the context. The relation  $\models$  is defined inductively as follows:

$$\begin{aligned}
M, s \models \wp & \quad \text{iff } \wp \in V(s), \text{ for } \wp \in PV, \\
M, s \models \neg\wp & \quad \text{iff } M, s \not\models \wp, \text{ for } \wp \in PV, \\
M, x \models \varphi \wedge \psi & \quad \text{iff } M, x \models \varphi \text{ and } M, x \models \psi, \text{ for } x \in \{s, \pi\}, \\
M, x \models \varphi \vee \psi & \quad \text{iff } M, x \models \varphi \text{ or } M, x \models \psi, \text{ for } x \in \{s, \pi\}, \\
M, s \models \mathbf{A}\varphi & \quad \text{iff } M, \pi \models \varphi \text{ for each path } \pi \text{ starting at } s, \\
M, s \models \mathbf{E}\varphi & \quad \text{iff } M, \pi \models \varphi \text{ for some path } \pi \text{ starting at } s, \\
M, \pi \models \varphi & \quad \text{iff } M, \pi(0) \models \varphi, \text{ for a state formula } \varphi, \\
M, \pi \models \mathbf{X}\varphi & \quad \text{iff } M, \pi_1 \models \varphi, \\
M, \pi \models \varphi \mathbf{U} \psi & \quad \text{iff } (\exists j \geq 0) (M, \pi_j \models \psi \text{ and } (\forall 0 \leq i < j) M, \pi_i \models \varphi), \\
M, \pi \models \varphi \mathbf{R} \psi & \quad \text{iff } (\forall j \geq 0) (M, \pi_j \models \psi \text{ or } (\exists 0 \leq i < j) M, \pi_i \models \varphi).
\end{aligned}$$

Moreover, we assume  $M \models \varphi$  iff  $M, s^0 \models \varphi$ , where  $s^0$  is the initial state of  $M$ .

## 4.3 Equivalence Preserving ACTL\* and ECTL\*

Let  $M = (L, S, s_0, \rightarrow, V)$  and  $M' = (L', S', s'_0, \rightarrow', V')$  be two models.

**Definition 2 ([2]).** A relation  $\rightsquigarrow_{sim} \subseteq S' \times S$  is a simulation from  $M'$  to  $M$  if the following conditions hold:

- $s'_0 \rightsquigarrow_{sim} s_0$ ,
- for each  $s \in S$  and  $s' \in S'$ , if  $s' \rightsquigarrow_{sim} s$ , then  $V(s) = V'(s')$ , and for every  $s_1 \in S$  such that  $s \xrightarrow{l} s_1$  for some  $l \in L$ , there is  $s'_1 \in S'$  such that  $s' \xrightarrow{l'} s'_1$  for some  $l' \in L'$  and  $s'_1 \rightsquigarrow_{sim} s_1$ .

The model  $M'$  simulates  $M$  ( $M' \rightsquigarrow_{sim} M$ ) if there is a simulation from  $M'$  to  $M$ . The models  $M, M'$  are simulation equivalent iff  $M \rightsquigarrow_{sim}^1 M'$  and  $M' \rightsquigarrow_{sim}^2 M$  for some simulations  $\rightsquigarrow_{sim}^1 \subseteq S \times S'$  and  $\rightsquigarrow_{sim}^2 \subseteq S' \times S$ . Two models  $M$  and  $M'$  are called bisimulation equivalent if  $M' \rightsquigarrow_{sim} M$  and  $M(\rightsquigarrow_{sim})^{-1}M'$ , where  $(\rightsquigarrow_{sim})^{-1}$  is the inverse of  $\rightsquigarrow_{sim}$ .

<sup>5</sup> Totality means that  $(\forall s \in S)(\exists s' \in S) s \rightarrow s'$ .

The following theorem holds:

**Theorem 1 ([2]).** *Let  $M, M'$  be two simulation equivalent models, where the range of the valuation functions  $V, V'$  is  $2^{PV}$ . Then,  $M, s_0 \models \varphi$  iff  $M', s'_0 \models \varphi$ , for any formula  $\varphi$  over  $PV$  such that  $\varphi \in \text{ACTL}^* \cup \text{ECTL}^*$ .*

## 5 Discrete- vs. Dense-Time Verification for ACTL\* and ECTL\*

It is easy to see that both the models  $M_r(\mathcal{N})$  and  $M_n(\mathcal{N})$  can be used in ACTL\* and ECTL\* verification for a net with the semantics a given model corresponds to (as both meet the definition of the model for CTL\*). However, it is also not difficult to see that the second model is smaller and less prone to the state explosion problem. The aim of our work is then to show that both the models are equivalent w.r.t. checking ACTL\* and ECTL\* properties of time Petri nets with the dense-time semantics. In our proof we make use of the approach of Popova presented in the paper [5].

Consider the dense concrete model  $M_r(\mathcal{N}) = (T \cup \mathbb{R}_+, \Sigma, \sigma^0, \rightarrow_r, V)$  of a TPN  $\mathcal{N}$ . A state  $\sigma = (m, \text{clock}) \in \Sigma$  is called an *integer-state* if  $\text{clock}(t) \in \mathbb{N}$  for all  $t \in T$ . A *integer  $\sigma$ -run* of  $\mathcal{N}$  is a sequence of states  $\sigma_0 \xrightarrow{a_0}_r \sigma_1 \xrightarrow{a_1}_r \sigma_2 \xrightarrow{a_2}_r \dots$ , where  $\sigma_0 = \sigma \in \Sigma$  and  $a_i \in T \cup \mathbb{N}$  for each  $i \geq 0$ . Note that all the states of an integer-run which starts at an integer-state are integer-states as well. Thus, it is easy to see that the following holds:

**Lemma 1.** *For a given time Petri net  $\mathcal{N}$  the model  $M_r(\mathcal{N})$  reduced to the integer-states and the transition relation between them is equal to  $M_n(\mathcal{N})$ .*

Given a number  $x \in \mathbb{R}_+$ , let  $\lfloor x \rfloor$  denote the *floor* of  $x$ , i.e., the greatest  $a \in \mathbb{N}$  such that  $a \leq x$ , and let  $\lceil x \rceil$  denote the *ceiling* of  $x$ , i.e. the smallest  $a \in \mathbb{N}$  such that  $x \leq a$ . Moreover, let  $\text{fire}(\sigma)$  denote a set of transition that are ready to fire in the state  $\sigma \in \Sigma$ , i.e.,  $\text{fire}(\sigma) = \{t \in \text{en}(m) \mid \text{clock}(t) \in [\text{Eft}(t), \text{Lft}(t)]\}$ . We define the integer-states to be *neighbour states* of real-valued ones as follows:

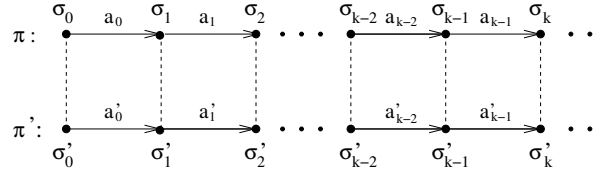
**Definition 3 (Neighbour states).** *Let  $\sigma = (m, \text{clock})$  be a state of a TPN  $\mathcal{N}$ . An integer-state  $\sigma' = (m', \text{clock}')$  is a neighbour state of  $\sigma$  (denoted  $\sigma' \sim_n \sigma$ ) iff*

- $m' = m$ ,
- for each  $t \in \text{en}(m)$ ,  $\lfloor \text{clock}(t) \rfloor \leq \text{clock}'(t) \leq \lceil \text{clock}(t) \rceil$ .

Intuitively, a neighbour state of  $\sigma$  is an integer-state of the same marking, and such that the values of its clocks, for all the enabled transitions, are “in a neighbourhood” of these of  $\sigma$ . However, it is easy to see that these values can be such that they make more transitions ready to fire than the corresponding values in  $\sigma$  do: each transition  $t$  which can be fired at a given value of  $\text{clock}(t)$  can be fired both at  $\lfloor \text{clock}(t) \rfloor$  and at  $\lceil \text{clock}(t) \rceil$  since all these three values are either equal if  $\text{clock}(t)$  is a natural number, or belong to the same (integer-bounded) interval

$[Eft(t), Lft(t)]$  if  $clock(t) \notin \mathbb{N}$ ; on the other hand, a transition  $t'$  which is not ready to fire at  $clock(t')$  can be fireable at  $\lceil clock(t') \rceil$  if  $\lceil clock(t') \rceil = Eft(t')$ . This implies  $fire(\sigma) \subseteq fire(\sigma')$ .

Let  $\pi := \sigma_0 \xrightarrow{a_0}_r \sigma_1 \xrightarrow{a_1}_r \dots$  be a  $\sigma^0$ -run in  $M_r(\mathcal{N})$ . By  $\pi_{[k]}$ , for  $k \in \mathbb{N}$ , we denote the prefix  $\sigma_0 \xrightarrow{a_0}_r \sigma_1 \xrightarrow{a_1}_r \dots \xrightarrow{a_{k-1}}_r \sigma_k$  of  $\pi$ , and by  $\pi(k)$  - the  $k$ -th state of  $\pi$ , i.e.,  $\sigma_k$ . Moreover, we assign a time  $\delta_i$  to each step  $\sigma_i \xrightarrow{a_i}_r \sigma_{i+1}$  in the run, i.e., define  $\delta_i = a_i$  if  $a_i \in \mathbb{R}$ , and  $\delta_i = 0$  otherwise. By  $\Delta_G(\sigma_i, \pi)$ , for  $i \in \mathbb{N}$ , we denote the value  $\sum_{j=0}^{i-1} \delta_j$  (i.e., the time passed along  $\pi$  before reaching  $\sigma_i$ ). Moreover, given  $k \in \mathbb{N}$  and a  $\pi(k)$ -run  $\rho := \sigma_k \xrightarrow{b_0}_r \beta_1 \xrightarrow{b_1}_r \beta_2 \xrightarrow{b_2}_r \dots$ , by  $\pi_{[k]} \cdot \rho$  we denote the run  $\sigma_0 \xrightarrow{a_0}_r \sigma_1 \xrightarrow{a_1}_r \dots \xrightarrow{a_{k-1}}_r \sigma_k \xrightarrow{b_0}_r \beta_1 \xrightarrow{b_1}_r \beta_2 \xrightarrow{b_2}_r \dots$  (i.e, the run obtained by “joining”  $\pi_{[k]}$  and  $\rho$ ). The above definitions apply to discrete runs in an analogous way. Next, we introduce the following definition (see also Fig. 2):



**Fig. 2.** Neighbour prefix of  $\pi_{[k]}$  (denoted  $\pi'_{[k]}$ ). If  $a_i \in T$ , then  $a'_i = a_i$ ; the states of  $\pi_{[k]}$  and  $\pi'_{[k]}$  related by  $\sim_n$  are linked by dashed lines.

**Definition 4 (Neighbour prefix).** Let  $\pi := \sigma_0 \xrightarrow{a_0}_r \sigma_1 \xrightarrow{a_1}_r \dots$  be a run in  $M_r(\mathcal{N})$ , and let  $\pi' := \sigma'_0 \xrightarrow{a'_0}_r \sigma'_1 \xrightarrow{a'_1}_r \dots$  be an integer run. For  $k \in \mathbb{N}$ , the prefix  $\pi'_{[k]}$  is a neighbour prefix of  $\pi_{[k]}$  (denoted  $\pi'_{[k]} \sim_n \pi_{[k]}$ ) iff for each  $i = 0, \dots, k$  it holds

- $\sigma'_i \sim_n \sigma_i$ ,
- $a_i \in T$  iff  $a'_i \in T$ , and if  $a_i, a'_i \in T$  then  $a'_i = a_i$ .

Intuitively, a neighbour prefix “visits” neighbour states of these in  $\pi_{[k]}$ , and the corresponding steps of these prefixes are either both firings of the same transition, or both passages of time (possibly of different lengths).

In order to show that  $M_n(\mathcal{N})$  can replace  $M_r(\mathcal{N})$  in ACTL\*/ECTL\* verification we shall prove the following lemma:

**Lemma 2.** *The models  $M_r(\mathcal{N})$  and  $M_n(\mathcal{N})$  are simulation equivalent.*

*Proof.* It is obvious from Lemma 1 that  $M_r(\mathcal{N})$  simulates  $M_n(\mathcal{N})$ , with the relation  $\mathcal{R}_1 \subseteq \Sigma \times \Sigma_n$  defined as  $\mathcal{R}_1 = \{(\sigma, \sigma') \mid \sigma = \sigma'\}$ .

Let  $R_r(\mathcal{N})$  and  $R_n(\mathcal{N})$  denote respectively the sets of all the dense  $\sigma^0$ -runs (discrete  $\sigma_n^0$ -runs) of the net  $\mathcal{N}$ . In order to prove that  $M_n(\mathcal{N}) \rightsquigarrow_{sim} M_r(\mathcal{N})$  we shall show that the relation  $\mathcal{R} \subseteq \Sigma_n \times \Sigma$  given by

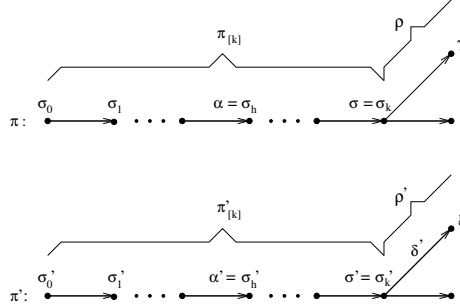
$$\mathcal{R} = \{(\sigma', \sigma) \mid \exists \pi \in R_r(\mathcal{N}) \exists \pi' \in R_n(\mathcal{N}) \exists k \in \mathbb{N} \text{ s.t.}$$

$$\sigma = \pi(k) \wedge \sigma' = \pi'(k) \wedge \pi'_{[k]} \sim_n \pi_{[k]} \wedge \forall j \leq k \Delta_G(\sigma'_j, \pi'_j) = \lfloor \Delta_G(\sigma_j, \pi_j) \rfloor\}$$

is a simulation from  $M_n(\mathcal{N})$  to  $M_r(\mathcal{N})$ . Intuitively, the states  $\sigma, \sigma'$  are related by  $\mathcal{R}$  if they both are reachable from the initial state of  $\mathcal{N}$  in  $k$  steps for some natural  $k$ , on runs  $\pi, \pi'$  such that  $\pi'_{[k]}$  is a neighbour prefix of  $\pi_{[k]}$  and for each  $j \leq k$  the total time passed along  $\pi'_{[j]}$  is the floor of that passed along  $\pi_{[j]}$ .

It is obvious that  $(\sigma_n^0, \sigma^0) \in \mathcal{R}$  due to equality of these states. Next, consider  $\sigma, \sigma'$  such that  $(\sigma', \sigma) \in \mathcal{R}$ . Assume that the runs “justifying” this relation (for some  $k$ ) are of the form  $\pi := \sigma_0 \xrightarrow{a_0} \sigma_1 \xrightarrow{a_1} \dots$  and  $\pi' := \sigma'_0 = \sigma'_0 \xrightarrow{a'_0} \sigma'_1 \xrightarrow{a'_1} \dots$  respectively, and that  $\sigma_i = (m_i, clock_i)$ ,  $\sigma'_i = (m'_i, clock'_i)$  for each  $i \in \mathbb{N}$  (which implies also the notation  $\sigma = (m_k, clock_k)$  and  $\sigma' = (m'_k, clock'_k)$  used below).

- if  $\sigma \xrightarrow{t} \gamma$  for a transition  $t \in T$  and a state  $\gamma = (m_\gamma, clock_\gamma)$ , then from  $\sigma' \sim_n \sigma$  (and therefore  $fire(\sigma) \subseteq fire(\sigma')$ ) the transition  $t$  can be fired at  $\sigma'$  as well, leading to a state  $\xi = (m_\xi, clock'_\xi)$ . Let  $\rho$  be a  $\sigma$ -run of the form  $\sigma \xrightarrow{t} \gamma \rightarrow \dots$  (i.e., a  $\sigma$ -run whose first step is  $\sigma \xrightarrow{t} \gamma$ ), and let  $\rho'$  be a  $\sigma'$ -run of the form  $\sigma' \xrightarrow{t} \xi \rightarrow \dots$  (i.e., a  $\sigma'$ -run whose first step is  $\sigma' \xrightarrow{t} \xi$ ; see Fig. 3). We shall show that  $(\pi'_{[k]} \cdot \rho')_{[k+1]} \sim_n (\pi_{[k]} \cdot \rho)_{[k+1]}$  and



**Fig. 3.** Relation between  $\pi, \pi', \rho$  and  $\rho'$  in the proof of Lemma 2.

that  $\Delta_G(\xi, \pi'_{[k]} \cdot \rho') = \lfloor \Delta_G(\gamma, \pi_{[k]} \cdot \rho) \rfloor$ .

- In order to prove  $(\pi'_{[k]} \cdot \rho')_{[k+1]} \sim_n (\pi_{[k]} \cdot \rho)_{[k+1]}$  it is sufficient to show that  $\xi \sim_n \gamma$ . It is obvious that the markings  $m_\gamma$  and  $m_\xi$  are equal, and that  $newly\_en(m_k, t) = newly\_en(m'_k, t)$ . Next, consider  $t' \in en(m_\gamma)$ . If  $t' \notin newly\_en(m_k, t)$  then the value of its clock in  $\gamma$  is the same as in  $\sigma$  (since firing of  $t$  does not influence the value of the clock of  $t'$ ). In turn, if



$t' \in \text{newly\_en}(m_k, t)$  then the values of its clock in  $\gamma$  and in  $\xi$  are equal to 0. Thus, from the fact that for  $\sigma, \sigma'$  we have  $\lfloor \text{clock}_k(t) \rfloor \leq \text{clock}'_k(t) \leq \lceil \text{clock}_k(t) \rceil$  we have also  $\lfloor \text{clock}_\gamma(t) \rfloor \leq \text{clock}'_\xi(t) \leq \lceil \text{clock}_\gamma(t) \rceil$ , which implies  $\xi \sim_n \gamma$ .

- the condition  $\Delta_G(\xi, \pi'_{[k]} \cdot \rho') = \lfloor \Delta_G(\gamma, \pi_{[k]} \cdot \rho) \rfloor$  holds in an obvious way ( $\Delta_G(\xi, \pi'_{[k]} \cdot \rho') = \Delta_G(\sigma', \pi') = \lfloor \Delta_G(\sigma, \pi) \rfloor = \lfloor \Delta_G(\gamma, \pi_{[k]} \cdot \rho) \rfloor$  as the step consisting in firing a transition is assigned the time 0).
- if  $\sigma \xrightarrow{\delta}_r \gamma$  for a time  $\delta \in \mathbb{R}_+$  and a state  $\gamma = (m_\gamma, \text{clock}_\gamma)$ , then let  $\rho$  be a  $\sigma$ -run  $\sigma \xrightarrow{\delta}_r \gamma \rightarrow_r \dots$  (i.e., a  $\sigma$ -run of the first step  $\sigma \xrightarrow{\delta}_r \gamma$ ; see Fig. 3), and let  $\pi_{[k]} \cdot \rho$  denote the run  $\sigma_0 \xrightarrow{a_0}_r \sigma_1 \xrightarrow{a_1}_r \dots \xrightarrow{a_{k-1}}_r \sigma_k \xrightarrow{\delta}_r \gamma \rightarrow_r \dots$  (i.e., the run obtained by “joining”  $\pi_{[k]}$  and  $\rho$ ). Next, assume

$$\delta' = \lfloor \Delta_G(\gamma, \pi_{[k]} \cdot \rho) \rfloor - \Delta_G(\sigma', \pi')$$

(which is an integer value due to  $\Delta_G(\sigma', \pi') \in \mathbb{N}$ ). We shall show first that the time  $\delta'$  can pass at  $\sigma'$ , leading to a state  $\xi = (m_\xi, \text{clock}'_\xi)$ .

- To show that  $\delta'$  can pass at  $\sigma'$  notice that

$$\delta = \Delta_G(\gamma, \pi_{[k]} \cdot \rho) - \Delta_G(\sigma, \pi_{[k]} \cdot \rho),$$

and that

$$\Delta_G(\sigma_i, \pi) = \Delta_G(\sigma_i, \pi_{[k]} \cdot \rho) \text{ for each } i = 0, \dots, k.$$

Moreover, we have that  $\text{clock}_\gamma(t) = \text{clock}_k(t) + \delta \leq \text{Lft}(t)$  for each  $t \in \text{en}(m_k)$ .

Consider a transition  $t \in \text{en}(m'_k)$  (where  $\text{en}(m'_k) = \text{en}(m_k) = \text{en}(m_\gamma)$ ). Let  $h$  be an index along  $\pi_{[k]}$  pointing to a state (denoted  $\alpha$ ) at which  $t$  became enabled most recently, and let  $h'$  be an index along  $\pi'_{[k]}$  pointing to a state (denoted  $\alpha'$ ) at which  $t$  became enabled most recently. From the fact that  $\pi'_{[k]} \sim_n \pi_{[k]}$  we have  $h = h'$  (for each  $j \leq k-1$  the corresponding  $j$ -th steps of  $\pi_{[k]}$  and  $\pi'_{[k]}$  are either both firings of the same transition or both time passings, which implies that for each  $i \leq k$  a transition  $t$  becomes enabled in  $\pi(i)$  iff it becomes enabled in  $\pi'(i)$ ). From the definitions of  $\text{clock}$ ,  $\text{clock}'$  it is easy to see that

$$\text{clock}_k(t) = \Delta_G(\sigma, \pi) - \Delta_G(\alpha, \pi),$$

$$\text{clock}_\gamma(t) = \Delta_G(\gamma, \pi_{[k]} \cdot \rho) - \Delta_G(\alpha, \pi)$$

and

$$\text{clock}'_k(t) = \Delta_G(\sigma', \pi') - \Delta_G(\alpha', \pi')$$

Moreover, it holds

$$\begin{aligned} \text{clock}'_k(t) + \delta' &= \Delta_G(\sigma', \pi') - \Delta_G(\alpha', \pi') + \delta' = \\ &= \Delta_G(\sigma', \pi') - \Delta_G(\alpha', \pi') + \lfloor \Delta_G(\gamma, \pi_{[k]} \cdot \rho) \rfloor - \Delta_G(\sigma', \pi') = \\ &= \lfloor \Delta_G(\gamma, \pi_{[k]} \cdot \rho) \rfloor - \Delta_G(\alpha', \pi') \stackrel{\text{def. of } \mathcal{R} \text{ and } h=h'}{=} \lfloor \Delta_G(\gamma, \pi_{[k]} \cdot \rho) \rfloor - \Delta_G(\alpha, \pi). \end{aligned}$$

From  $clock_\gamma(t) \leq Lft(t)$  we have  $\lceil clock_\gamma(t) \rceil \leq Lft(t)$ , and from the property  $\lfloor a \rfloor - \lfloor b \rfloor \leq \lfloor a - b \rfloor$  we get  
 $clock'_k(t) + \delta' = \lfloor \Delta_G(\gamma, \pi_{[k]} \cdot \rho) \rfloor - \lfloor \Delta_G(\alpha, \pi) \rfloor \leq \lceil \Delta_G(\gamma, \pi_{[k]} \cdot \rho) - \Delta_G(\alpha, \pi) \rceil = \lceil clock_\gamma(t) \rceil \leq Lft(t)$ ;  
 Thus, we have that  $clock'_k(t) + \delta' \leq Lft(t)$  for each  $t \in en(m'_k)$ , and therefore the time  $\delta'$  can pass at  $\sigma'$ .

Next, let  $\rho'$  be a  $\sigma'$ -run of the form  $\sigma' \xrightarrow{\delta'} \xi \rightarrow_n \dots$ . We shall show that  $\xi \sim_n \gamma$ . It is obvious that the markings of these states are equal. Consider  $t \in en(m)$ . We show that  $\lfloor clock_\gamma(t) \rfloor \leq clock'_\xi(t) \leq \lceil clock_\gamma(t) \rceil$ . Let  $\alpha, \alpha', h, h'$  be defined as in the previous part of the proof (see the 8th line of the previous item). Similarly as before, from the definitions of  $clock, clock'$  we have that

- In order to prove  $(\pi'_{[k]} \cdot \rho')_{[k+1]} \sim_n (\pi_{[k]} \cdot \rho)_{[k+1]}$  it is sufficient to show that  $\xi \sim_n \gamma$ . It is obvious that the markings of these states are equal. Consider  $t \in en(m)$ . We show that  $\lfloor clock_\gamma(t) \rfloor \leq clock'_\xi(t) \leq \lceil clock_\gamma(t) \rceil$ . Let  $\alpha, \alpha', h, h'$  be defined as in the previous part of the proof (see the 8th line of the previous item). Similarly as before, from the definitions of  $clock, clock'$  we have that

$$clock_\gamma(t) = \Delta_G(\gamma, \pi_{[k]} \cdot \rho) - \Delta_G(\alpha, \pi),$$

and that

$$clock'_\xi(t) = \Delta_G(\sigma', \pi') + \delta' - \Delta_G(\alpha', \pi').$$

- \* From the property  $\lfloor a - b \rfloor \leq \lfloor a \rfloor - \lfloor b \rfloor$  (for  $a, b \in \mathbb{R}_+$  with  $a \geq b$ ) we have  
 $\lfloor clock_\gamma(t) \rfloor = \lfloor \Delta_G(\gamma, \pi_{[k]} \cdot \rho) - \Delta_G(\alpha, \pi) \rfloor \leq \lfloor \Delta_G(\gamma, \pi_{[k]} \cdot \rho) \rfloor - \lfloor \Delta_G(\alpha, \pi) \rfloor \stackrel{h=h' \text{ and def. of } \mathcal{R}}{=} \lfloor \Delta_G(\gamma, \pi_{[k]} \cdot \rho) - \Delta_G(\sigma', \pi') + \Delta_G(\sigma', \pi') \rfloor - \Delta_G(\alpha', \pi') = \lfloor \Delta_G(\gamma, \pi_{[k]} \cdot \rho) \rfloor - \Delta_G(\sigma', \pi') + \Delta_G(\sigma', \pi') - \Delta_G(\alpha', \pi') = \delta' + \Delta_G(\sigma', \pi') - \Delta_G(\alpha', \pi') = clock'_\xi(t)$ .
- \* From the property  $\lfloor a \rfloor - \lfloor b \rfloor \leq \lfloor a - b \rfloor$  we have  
 $clock'_k(t) + \delta' = \lfloor \Delta_G(\gamma, \pi_{[k]} \cdot \rho) \rfloor - \lfloor \Delta_G(\alpha, \pi) \rfloor \leq \lceil \Delta_G(\gamma, \pi_{[k]} \cdot \rho) - \Delta_G(\alpha, \pi) \rceil = \lceil clock_\gamma(t) \rceil$
- Next, we have  $\Delta_G(\xi, \pi'_{[k]} \cdot \rho') = \Delta_G(\sigma', \pi') + \delta' = \Delta_G(\sigma', \pi') + \lfloor \Delta_G(\gamma, \pi_{[k]} \cdot \rho) \rfloor - \Delta_G(\sigma', \pi') = \lfloor \Delta_G(\gamma, \pi_{[k]} \cdot \rho) \rfloor$ , which ends the proof.

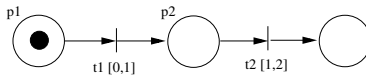
Therefore, we can formulate the following theorem:

**Theorem 2.** *Let  $M_r(\mathcal{N})$  and  $M_n(\mathcal{N})$  be respectively a dense and a discrete model for a time Petri net  $\mathcal{N}$ , and let  $\varphi$  be an ACTL\* (ECTL\*) formula. The following condition holds:*

$$M_r(\mathcal{N}) \models \varphi \text{ iff } M_n(\mathcal{N}) \models \varphi.$$

*Proof.* Follows from Theorem 1 and Lemma 2 in a straightforward way.

It should also be explained that in the case of timed systems (and therefore also TPNs) with the dense-time semantics, logics without the next-step operator are usually used, due to problems with interpreting the “next” step in the case of continuous time. However, Thm. 2 considers more general logics, in case one would interpret the next-step operator over an arbitrary passage of time.



**Fig. 4.** A net

It should be noticed that the relation  $\mathcal{R}$  used in our proof cannot be used to prove bisimulation between the models, i.e., their equivalence w.r.t. the CTL\* properties, since the integer run  $\pi'$  “justifying”  $\sigma' \mathcal{R} \sigma$  and the dense run  $\pi$  occurring in the relation do not need to “branch” in the same way. Thus, although one can prove that each transition  $t$  which can be fired at  $\sigma$  can be fired at  $\sigma'$  as well, the reverse does not hold. To see an example of the above consider the net shown in Fig. 4 and its runs:

- the dense one:
$$\pi := (p_1, (0, 0)) \xrightarrow{0.5}_r (p_1, (0.5, 0)) \xrightarrow{t_1}_r (p_2, (0, 0)) \xrightarrow{0.6}_r (p_2, (0, 0.6)) \rightarrow_r \dots$$
- and the discrete one (denoted  $\pi'$ ), built in the way shown in [5] and used in our proof in the definition of  $\mathcal{R}$  (i.e., satisfying  $\pi'_{[3]} \sim_n \pi_{[3]}$  and  $\Delta_G(\sigma'_j, \pi') = \lfloor \Delta_G(\sigma_j, \pi) \rfloor$  for each  $j \leq 3$ ):
$$\pi' := (p_1, (0, 0)) \xrightarrow{0}_n (p_1, (0, 0)) \xrightarrow{t_1}_n (p_2, (0, 0)) \xrightarrow{1}_n (p_2, (0, 1)) \rightarrow_n \dots$$

It is easy to see that in  $\pi(3)$  we have  $clock(t_2) = 0.6$ , which means that  $t_2$  cannot be fired at this state, while in  $\pi'(3)$  we have  $clock(t_2) = 1$ , which means that the transition  $t_2$  is fireable.

## 6 Experimental Results

In order to show that using discrete-time models instead of the dense ones can be profitable, we performed some tests, using as an example an implementation of SAT-based bounded model checking (BMC) for a subclass of TPNs (i.e., distributed time Petri nets) with the discrete-time semantics and the logic ACTL<sub>-X</sub> used in [4], and its modification for the dense-time case prepared for the current paper. BMC is a technique applied mainly to searching for counterexamples for universal properties, using a model truncated up to some specific depth  $k$ . The formulas used by the method are then negations of these expressing properties to be tested. So, in our case they are formulas of ECTL<sub>-X</sub>.

The first system we consider is the Generic Pipeline Paradigm Petri net model (GTPP) shown in Fig. 5. It consists of three parts: 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<sub>i</sub>Ready*), processing data (*Node<sub>i</sub>Proc*), or sending data (*Node<sub>i</sub>Send*). The example can be scaled by adding more intermediate nodes. The parameters  $a, b, c, d, e, f$  are used to adjust the time properties of Producer, Consumer, and of the intermediate Nodes. The formulas considered are  $EGEFConsReceived$ ,  $EG(ProdReady \vee ConsReady)$  and  $EFNode_1Send$ .

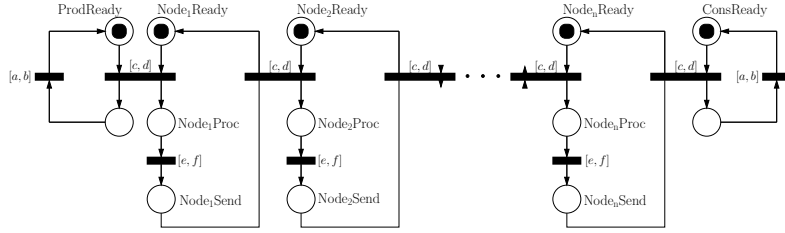


Fig. 5. A net for Generic Timed Pipeline Paradigm

The next system tested is the standard *Fischer’s mutual exclusion protocol* (Mutex). The system consists of  $n$  time Petri nets, each one modelling a process, plus one additional net used to coordinate the access of the processes to the critical sections. A TPN modelling the system for  $n = 2$  is presented in Fig. 6. In this case we have tested the formula  $EGEF(crit_1 \vee \dots \vee crit_n)$ .

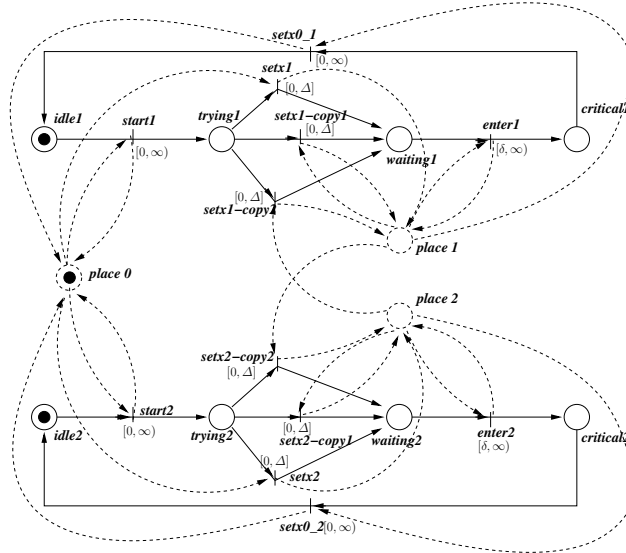


Fig. 6. A net for Fischer’s mutual exclusion protocol for  $n = 2$

The results are presented in Fig. 7–9 for GTPP, and in Fig. 10 for Mutex. It can be seen that in all the cases we are able to verify systems containing more components (indicated in the column  $n$ ) than when discrete models are used, and the total time ( $bmcT + satT$ ) and the memory required ( $\max(bmcM, satM)$ ) are usually smaller for the discrete-time case (the columns with “IN :”). In some cases the differences are quite substantial, but there are also examples in which the time and the memory used are similar for both the semantics. However, one

n	k	LL	$\mathbb{R}$ : bmcT+satT	$\mathbb{R}$ : max(bmcM,satM)	$\mathbb{N}$ : bmcT+satT	$\mathbb{N}$ : max(bmcM,satM)
1	5	7	1.41	12.00	0.20	8.00
2	7	9	9.94	25.00	1.15	12.00
3	9	11	49.45	60.00	3.55	21.00
4	11	13	154.70	146.00	9.94	38.00
5	13	15	310.18	243.00	20.90	61.00
6	15	17	708.66	313.00	41.43	94.00
7	17	19	1934.63	818.00	76.81	145.00
8	19	21	4121.60	1071.00	131.98	215.00
9	21	23	6819.25	1640.00	237.21	314.00
10	23	25	20519.20	3455.00	361.03	377.00
11	25	27	-	-	562.15	552.00

**Fig. 7.** Comparison of the results for GTPP and the formula  $EGEFConsReceived$ 

n	k	LL	$\mathbb{R}$ : bmcT+satT	$\mathbb{R}$ : max(bmcM,satM)	$\mathbb{N}$ : bmcT+satT	$\mathbb{N}$ : max(bmcM,satM)
1	5	1	0.41	7.00	0.17	7.00
2	7	1	4.14	12.00	0.76	8.00
3	9	1	32.27	29.00	2.20	9.00
4	11	1	63.28	52.00	8.57	12.00
5	13	1	200.14	151.00	21.14	17.00
6	15	1	488.59	165.00	43.18	24.00
7	17	1	870.21	342.00	105.18	38.00
8	19	1	1870.65	415.00	234.00	54.00
9	21	1	3745.33	658.00	763.84	139.00
10	23	1	7097.01	1364.00	1696.58	283.00
11	25	1	-	-	3013.98	306.00

**Fig. 8.** Comparison of the results: GTPP, the formula  $EG(ProdReady \vee ConsReady)$ 

can see that the noticeable differences occur in the cases in which the length of the witness for the formula ( $k$ ) or the number of paths required to check this formula ( $LL$ ) grow together with the size of the system, making the verification more expensive.

## 7 Conclusions and Further Work

We have shown that the result of Popova, stating that integer time steps are sufficient to test reachability of markings in time Petri nets, can be extended to testing  $ECTL^*$  and  $ACTL^*$  properties. We have focused on 1-safe TPNs for simplicity of the presentation, but it is easy to see that the result applies also to “general” time Petri nets: neither the definitions of a marking and of enabledness of a transition, nor the way multiple enabledness of transitions is handled do influence the proof.

n	k	LL	IR: bmcT+satT	IR: max(bmcM,satM)	IN: bmcT+satT	IN: max(bmcM,satM)
100	2	1	2.02	23.00	1.60	19.00
200	2	1	7.04	76.00	5.74	51.00
300	2	1	15.03	153.00	12.15	102.00
400	2	1	26.30	270.00	18.59	179.00
500	2	1	40.50	412.00	28.47	273.00
600	2	1	58.71	563.00	40.45	397.00
700	2	1	79.71	738.00	54.69	537.00
800	2	1	104.84	992.00	72.06	654.00
900	2	1	133.78	1173.00	90.48	854.00
1000	2	1	169.19	1528.00	114.86	1005.00
1100	2	1	211.16	1772.00	140.22	1168.00
1200	2	1	-	-	168.86	1506.00
1300	2	1	-	-	203.24	1604.00

**Fig. 9.** Comparison of the results: GTPP, the formula  $EFNode1Send$

n	k	LL	IR: bmcT+satT	IR: max(bmcM,satM)	IN: bmcT+satT	IN: max(bmcM,satM)
2	4	5	1.04	11.00	0.74	10.00
3	4	5	1.77	13.00	1.10	12.00
4	4	5	1.83	15.00	1.63	14.00
5	4	5	3.18	17.00	2.41	16.00
10	4	5	9.15	40.00	7.20	31.00
20	4	5	26.14	90.00	18.76	86.00
30	4	5	74.70	177.00	58.56	161.00
40	4	5	258.34	330.00	108.34	320.00
50	4	5	265.06	419.00	170.93	358.00
60	4	5	710.11	732.00	442.42	713.00
70	4	5	701.81	1092.00	728.20	1073.00
80	4	5	919.34	1001.00	2288.34	1349.00
90	4	5	780.89	1161.00	934.72	1140.00
100	4	5	4566.16	3181.00	4230.64	4549.00
110	4	5	4260.76	3414.00	4956.38	3237.00
120	4	5	-	-	4217.44	3238.00
130	4	5	-	-	2155.04	2571.00
140	4	5	-	-	5087.76	3603.00

**Fig. 10.** Comparison of the results: mutex, the formula  $EGEF(crit_1 \vee \dots \vee crit_n)$

Our experimental results show that considering the discrete semantics while verifying properties of dense-time nets can be profitable. Due to this, in our further work we are going to check whether discrete-time semantics can be used when testing other classes of properties of the dense-time Petri net systems (e.g.,  $CTL^*_X$ ).

## References

1. U. Goltz, R. Kuiper, and W. Penczek. Propositional temporal logics and equivalences. In *Proc. of the 3rd Int. Conf. on Concurrency Theory (CONCUR'92)*, volume 630 of *LNCS*, pages 222–236. Springer-Verlag, 1992.
2. O. Grumberg and D. E. Long. Model checking and modular verification. In *Proc. of the 2nd Int. Conf. on Concurrency Theory (CONCUR'91)*, volume 527 of *LNCS*, pages 250–265. Springer-Verlag, 1991.
3. A. Janowska, W. Penczek, A. Pólróla, and A. Zbrzezny. Towards discrete-time verification of time Petri nets with dense-time semantics. In *Proc. of the Int. Workshop on Concurrency, Specification and Programming (CS&P'11)*, pages 215–228. Bialystok University of Technology, 2011.
4. A. Męski, W. Penczek, A. Pólróla, B. Woźna-Szcześniak, and A. Zbrzezny. Bounded model checking approaches for verification of distributed time Petri nets. In *Proc. of the Int. Workshop on Petri Nets and Software Engineering (PNSE'11)*, pages 72–91. University of Hamburg, 2011.
5. L. Popova. On time Petri nets. *Elektronische Informationsverarbeitung und Kybernetik*, 27(4):227–244, 1991.
6. L. Popova-Zeugmann. Essential states in time Petri nets. Informatik-Bericht 96, Humboldt University, 1998.
7. L. Popova-Zeugmann. Time Petri nets state space reduction using dynamic programming. *Journal of Control and Cybernetics*, 35(3):721–748, 2006.
8. L. Popova-Zeugmann and D. Schlatter. Analyzing paths in time Petri nets. *Fundamenta Informaticae*, 37(3):311–327, 1999.

# Grade/CPN: Semi-automatic Support for Teaching Petri Nets by Checking Many Petri Nets Against One Specification

Michael Westergaard, Dirk Fahland, and Christian Stahl

Department of Mathematics and Computer Science,  
Eindhoven University of Technology, The Netherlands  
{m.westergaard,d.fahland,c.stahl}@tue.nl

**Abstract.** Grading dozens of Petri net models manually is a tedious and error-prone task. In this paper, we present Grade/CPN, a tool *supporting the grading of Colored Petri nets modeled in CPN Tools*. The tool is extensible, configurable, and can check static and dynamic properties. It automatically handles tedious tasks like checking that good modeling practise is adhered to, and supports tasks that are difficult to automatize, such as checking model legibility. We propose and support the *Britney Temporal Logic* which can be used to guide the simulator and to check temporal properties. We provide our experiences with using the tool in a course with 100 participants.

## 1 Introduction

Colored Petri nets (CPNs) [8] is a formalism useful for modeling a broad range of real-life systems, including complex network protocols [8] and business information systems [1]. It is thus natural to use CPNs or other Petri net formalisms when teaching such subjects. As modeling can only really be learned by doing, hands-on experience is a must. Larger classes can comprise more than one hundred students, and manually checking models created by students is time consuming and error-prone. This is particularly unpleasant because much of the effort is spent on checking trivial things, including whether good modeling standards are adhered to and whether formal requirements to the model are satisfied. In this paper, we aim at *supporting the grading of many models implementing the same specification* by providing with Grade/CPN an *extensible tool* for automatic assessment of such routine properties, allowing teachers to focus on more complicated tasks.

The support required for grading assignments is similar to what is needed for testing or model checking, as we need to check a model against some formal requirements. As we aim at supporting grading for all kinds of models, we here focus on the testing perspective, as a model may not be suitable for model checking due to having a large or even unbounded state space. Thus, parts of the work described here is also applicable to general testing of CPN models, but we present it here in the context in which it was developed. The significant



difference to classical testing is that for grading *a possibly large set of different models* is to be checked against *the same specification* in a uniform way.

CPN Tools [3] is a tool for editing, simulating and analysis of CPN models. It supports the user during the construction of the model due to incremental syntax checking, which gives immediate feedback about errors, and allows modelers to experiment with incomplete and even only partially correct models. This is a useful feature for inexperienced users and makes CPN Tools suitable in teaching. Furthermore, the Windows version of CPN Tools is downloaded more than 5,000 times a year, indicating that it is broadly used. The broad usage also means that CPN Tools has reached a fairly stable state, which reduces unnecessary frustrations during modeling. Finally, CPN Tools has extensive online help and video tutorials, which means it is easy for students to get started. For these reasons, we think that CPN Tools is a good choice of a tool for teaching.

There are as many ways of using models as there are teachers, so it is important that the requirements for the model can be described easily. This means that the grading tool must be *configurable*, allowing individual teachers to customize what is checked and how adhering to or deviating from each requirement is awarded or punished. In addition, it must be easily possible to *extend* the tool with new requirements. Thus our tool must have a plug-in like architecture allowing new requirements to be added with minimal effort. At the same time, we do not desire a heavy-weight framework with a steep learning curve just to add a simple custom requirement. Of course, such a tool should come with a set of reasonable built-in plug-ins so it is useful for many scenarios without requiring any programming.

To illustrate our motivation for developing such a tool, assume we want students to model a (simplified) delivery service using CPN Tools. The idea is to model that customers order products from a shop, and the shop uses a delivery service to deliver ordered products to the customers. To this end, we would provide students with a base

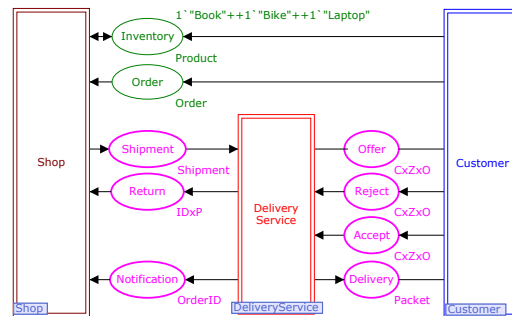


Fig. 1: Base model of a delivery service.

model as in Fig. 1. The CPN in Fig. 1 models the behavior of the customer and the shop and provides the interface between customer and delivery service (Reject, Offer, Accept, and Delivery) and the interface between shop and delivery service (Shipment, Return, and Notification). A customer can choose a product from the catalog and place an order via place Order. The shop prepares the ordered product for shipment and sends the resulting packet to the delivery service via Shipment. The delivery service shall in all tasks try to deliver packets to the respective customers via place Offer. If a customer is not at home, a token is placed on place Reject; otherwise, a token is produced on place Accept and, finally, the delivery service hands over the packet to the customer via place Delivery. Place

Return is used to send a packet back to the shop in case the packet could not be delivered. In addition, the delivery service informs the shop via place Notification that a packet has been successfully delivered. The pages Shop and Customer are given but the DeliveryService is empty and intended to be modeled by the student.

When students are given such a base model, they are asked to model the missing part(s) or to change or improve the given model. These changes must adhere to certain constraints. In our example, we would need to be able to check that the given *environment has not been changed* (as the environment constitutes a contract with the external world) and that the *model satisfies the given requirements*, which often means that behavioral properties need to be checked. Our focus on the first version of our tool has therefore been on making it easy to check these requirements.

We have also implemented checks that ensure good modeling practise, including *respecting data hiding* (i.e., student solutions are not allowed to connect to nodes of the environment other than the interface places) and *proper termination* (i.e., ensuring that tokens are not erroneously left behind), and simple *static analysis* (e.g., ensuring that communication channels are used in the correct direction, i.e., no messages are produced on an input channel).

As we cannot check all properties mechanically—for example, whether the model is readable and understandable—we have implemented functionality facilitating this. This includes generating a *view of the model* in which the student-designed parts are highlighted and the given parts from Fig. 1 are dimmed. This allows teachers to focus on the new parts without having to distinguish these parts manually.

We have earlier encountered problems with students copying solutions from one another. We would also like to detect this, so we have *checks that at least make it harder to cheat*. This includes providing each student with a unique copy of the base model from Fig. 1 with a cryptographic signature including the student ID embedded. This makes it impossible to two students to use the same base model as starting point (indicating that one got a copy from the other).

Finally, we want a *report* summarizing all findings; the report should be useful for both teachers, who should be able to grade the model based on the report only, without having to manually open the model in CPN Tools except in special cases, and for students, who should be pointed to flaws in the model, using error traces when applicable.

We have chosen to implement our tool as a vanilla Java application. The language is chosen due to its popularity and platform-independence. We have chosen not to rely on a framework for handling plug-ins, as these frameworks often demand significant overhead due to providing features we do not need (e.g., we do not need dynamic configuration of plug-ins). We have used the library Access/CPN [14] as it provides an easy way to load CPN models and programatically interact with the simulator.

To sum up, we need a tool that

1. Works with CPN Tools models,
2. Provides easy configuration,

3. Is easily extensible,
4. Contains a reasonable base set of capabilities, including:
  - (a) Detect changes to a given environment,
  - (b) Check dynamic properties using simulation, and
  - (c) Check good modeling practise, including data hiding, proper termination, and provide simple static analysis,
5. Supports the manual part of the grading process,
6. Detects attempts to defraud, and
7. Provides a report that pin-points problems, aids the teacher in grading, and allows students to understand problems.

We continue with the outline of the architecture of our tool and introduce some simple plug-ins checking basic properties in Sect. 2. In Sect. 3, we introduce a temporal logic which is powerful enough to describe most dynamic requirements while still being easy to use. In Sect. 4, we sum up our experiences using our tool in semi-automatically assessing assignments from close to 100 students. Finally, we discuss related work, conclude the paper, and provide directions for future work.

## 2 Architecture

In this section, we outline the architecture of Grade/CPN. We first give the overall architecture and explain how this solves requirements 1, 2, 3, and 7 from the introduction. Then, we provide the details of some of the built-in plug-ins, focusing on requirements 4(a), 4(c), and 6. Requirement 5 is handled partly in this section and in the next section, where we also deal with requirement 4(b) (checking dynamic properties).

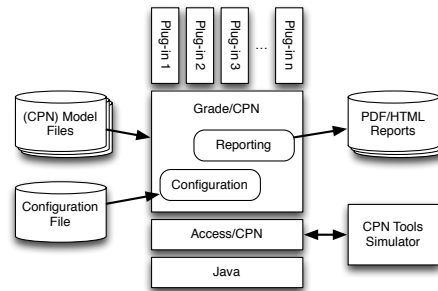


Fig. 2: Overall architecture and environment of Grade/CPN.

### 2.1 Overall Architecture

Figure 2 shows the overall architecture of Grade/CPN. We see that we build on top of Java and Access/CPN [14]. Access/CPN is a Java library making it possible to interact directly with the CPN Tools Simulator, including loading models and translating them to an object structure we can use for static analysis, and send to the simulator process also used by CPN Tools to perform syntax check and simulation of models. Grade/CPN comprises two important components, one for Configuration and one for Reporting, as well as an interface to several Plug-ins. The Configuration component is responsible for loading a configuration file and using it to instantiate and configure the appropriate plug-ins. Each plug-in returns messages useful for the Reporting component, which use this information to

generate an on-screen status view showing the overall correctness of the checked models and for generating an individual report for each student. The report can be generated as either an HTML file suitable for reading in a Web-browser or a PDF file suitable for printing or archival.

The central interface of Grade/CPN is `PlugIn`, shown in Listing 1 (ll. 1–5). Each plug-in must implement this interface. The `configure` method is a factory method to instantiate the plug-in, and takes how many points should be awarded if the plug-in succeeds and a configuration string. The format of the configuration string is defined by the plug-in, but will typically be a name identifying the plug-in and a list of named parameters. If the plug-in can be instantiated with a given configuration string, it returns a new configured instance and otherwise it returns `null`. This allows us to create an abstract factory for instantiating plug-ins from a string. Furthermore, a plug-in has a method `grade`, which is given a student ID, a base model (`base`), the student solution (`model`), and a connection to the `simulator`. The plug-in can use this information to arrive at its conclusion and return a `Message`, which comprises how many points are awarded and a descriptive message with the reason for the grade.

**Reporting.** The Reporting component of Fig. 2 is responsible for emitting a report based on the result of the `PlugIns`. All interfaced pertaining to reporting is shown in Listing 1 (ll. 7–17). The main class is `Report` (ll. 7–10), which is instantiated for each student ID and contains a set of pairs of `PlugIns` and `Messages` (produced by the `grade` method `PlugIns`).

Point range	Points	Reason
-100.00 - 0.00	0.00	The interface has not been modified incorrectly.
-5.00 - 0.00	0.00	Declarations were preserved and new ones were added (that is ok).
-5.00 - 0.00	-5.00	Generated_Task1b-solution.cpn is not a substring of Task1b-solution.cpn
-5.00 - 0.00	-5.00	Did not match with 0 < 65
0.00 - 0.03	0.03	30 Random Orders was executed successfully 10 times
0.00 - 0.03	0.03	Packet to Depot after Reject was executed successfully 10 times

Fig. 3: Report overview.

Listing 1: Plug-in interface and central components.

```

1 public interface PlugIn {
2     public PlugIn configure(double maxPoints, String configuration);
3     public Message grade(StudentID id, PetriNet base, PetriNet model,
4                           HighLevelSimulator simulator);
5 }
6
7 public class Report {
8     public Report(StudentID sid) { ... }
9     void addReport(PlugIn plugin, Message result) { ... }
10 }
11 public class Message {
12     public Message(double points, String message, Detail... details) { ... }
13 }
14 public class Detail {
15     public Detail(String header, String... details) { ... }
16     public Detail(String header, JComponent component) { ... }
17 }
18
19 public class Tester {
20     public Tester(TestSuite suite, List<StudentID> ids, PetriNet base) { ... }
21     public List<Report> test() { ... }
22 }
23 public abstract class TestSuite {
24     public TestSuite(PlugIn matcher) { ... }
25     public abstract List<PlugIn> getPlugIns();
26 }
27 public class ConfigurationTestSuite extends TestSuite {
28     public ConfigurationTestSuite(File configurationFile) { ... }
29 }

```

A `Message` (ll. 11–13) ties together a number of awarded points, a descriptive message and a list of `Details` providing in-depth reasoning leading to the outcome. Each `Detail` (ll. 14–17) consists of a descriptive header and either a list of textual details or a single graphical component, which is rendered as an image in the resulting report. For each student a report overview is generated (see Fig. 3 for an example) and supplementary details are added in separate sections.

**Configuration.** The `Configuration` component of Fig. 2 is shown in Listing 1 (ll. 19–29). The main class is a `Tester` (ll. 19–22), which given a `TestSuite`, a list of student IDs, and a base model can perform a `test` (l. 21) and yields a `Report` for each student. A `TestSuite` (ll. 23–26) has a distinguished `matcher`, which is a `Plugin` mapping models to student IDs by yielding a high score for a model and student ID pair if the model is created by the student with the given ID and a low score otherwise. A `TestSuite` can also return a list of `Plugins` for the main grading process. One implementation of a `TestSuite`, the `ConfigurationTestSuite` (ll. 27–29), is instantiated using a `configurationFile` which along with an abstract `Plugin` factory is used to instantiate the correct `Plugins` according to the configuration.

An example configuration file is shown in Listing 2. The file comprises two sections, `matcher` (ll. 1–2) and `test` (ll. 4–15), setting up the `matcher` and the actual tests graded, respectively. The intuition is that each line corresponds to a plug-in; a line starting with a `+` (ignoring white space) is considered part of the preceding line. Each line starts with a number indicating how many points are awarded for successful execution. If the number is negative, successful execution yields 0 points but a failure yields a punishment. This is followed by a colon and a configuration option recognized by the plug-in and optionally a list of named parameters. For example, in line 5 we see that the plug-in identified by `declaration-preservation` is instantiated with one named parameter. If the test fails, it yields a punishment of 5 points and if it succeeds, it yields 0 points. Lines 13–14 are merged (as line 14 starts with `+`). In the following we go into more detail with this example.

## 2.2 Simple Plug-ins for Interface Preservation

In Listing 2, we use two plug-ins to ensure that the interface to the environment and the environment itself are not modified. The `declaration-preservation` plug-

Listing 2: Example configuration file.

```

1  [matcher]
2  -5: signature , threshold=65
3
4  [tests]
5  -5: declaration-preservation
6  -100: interface-preservation , addpages=false , initmark=true , subset=deliveryservice
7  -5: matchfilename
8  0.033: btl , repeats=2 , name="Accept 10 Orders" , test=
9    + (10 * (--> Order) -> (--> Order) => failure) &
10   + (10 * (--> Receive) -> (--> Receive) => failure) &
11   + ((--> Reject) => failure) &
12   + [(--> Handle_Return) => false]
13  0.033: btl , repeats=2 , name="Only two cars of capacity 1" , test=
14   + [|Reject| + |Offer| + |Accept| < 3]

```

in (l. 5) makes sure that no declaration in the provided model is removed or changed. This ensures that it is impossible to change the type of the interface by redefining color sets. If declarations are removed or changed, this is reported as an error and if new declarations are added, they are added to the report so it is easy to see what was added without having to compare the student model with the base model manually.

The `interface-preservation` (l. 6) plug-in makes sure that students do not change the given net structure, but only add new structure. In our example from Fig. 1, students are only allowed to add new net structure, but not to modify the given environment. Here, we are given four parameters. The `addpages` parameter is set to `false`, which means that students are not allowed to add new pages. The model used here is flat, and thus introducing hierarchy is considered an error. The `initmark` option is set to `true`, which means that students are not allowed to change the initial marking of the model. Finally, the `subset` parameter contains a list of pages students are allowed to add structure to. Any page not in this list is not allowed to be changed at all. Here, we specify that the students are allowed to alter the `deliveryservice` page from Fig. 1. Any added page is listed in the report as is any modified page. If the change is illegal, the error is listed (i.e., if a node of the interface is removed or altered, this is highlighted), and if the model contains no errors, the entire environment is dimmed so only the student solution is highlighted.

### 2.3 Fraud Prevention

We have two plug-ins for matching a model to a student ID. In Listing 2, we use both to award points. We see in line 8 that we instantiate the `matchfilename` plug-in. This plug-in simply checks if the student ID is a substring of the filename (and punishes if it is not). This is fine for honest students; unfortunately, we have in earlier years encountered students copying models from one another. To catch that, we instead use the more elaborate `signature` plug-in as matcher (l. 2).

The signature matcher exploits that all elements of a CPN Tools model have a unique identifier. This is necessary, e.g., to represent that an arc is connected to a specific place and transition. While these identifiers must be unique in the file and match for nodes and arcs, the actual contents of the identifiers have no semantics. We have developed a simple signer application which, given a base model, modifies the identifiers in a predictable way. By using a cryptographic random number generator, we can generate a sequence of pseudo-random numbers using the student ID and a secret passphrase as seed. The idea is that if we know the passphrase and the student ID, we can regenerate the sequence, but using just the sequence (and optionally the student ID), it is not possible to reverse-engineer the passphrase. Now, using the generated sequence of numbers as identifiers of model elements in the file containing the environment, we create a unique signature in the base file for each student.

The `signature` plug-in can check this signature. It queries for each student ID and student model whether the two match. It regenerates the sequence of random numbers for the student ID and the provided passphrase, and check

that the identifiers are present in the file. If they are, the model is considered a match and otherwise not. The plug-in takes a parameter `threshold` which indicates how many identifiers must be present in the model. As the signing is a one-way process, students are forced to use the appropriate base model and cannot just hand in the same model (even after making cosmetic changes).

### 3 Britney Temporal Logic

An important requirement to our tool is to check dynamic properties, requirement 4(b) from the introduction. In the example in Fig. 1, we are for example interested in the behavior when a customer accepts packets ten times in a row and how many packets can be outstanding at any time. As CPN models tend to have huge or even infinite state spaces, we cannot verify such properties in general and especially not for models generated by students who have less experience with modeling. Therefore, we check such properties by guiding the simulator; that is, we apply a testing-based approach rather than exhaustive state-space exploration, yielding a sound but not necessarily complete checking mechanism.

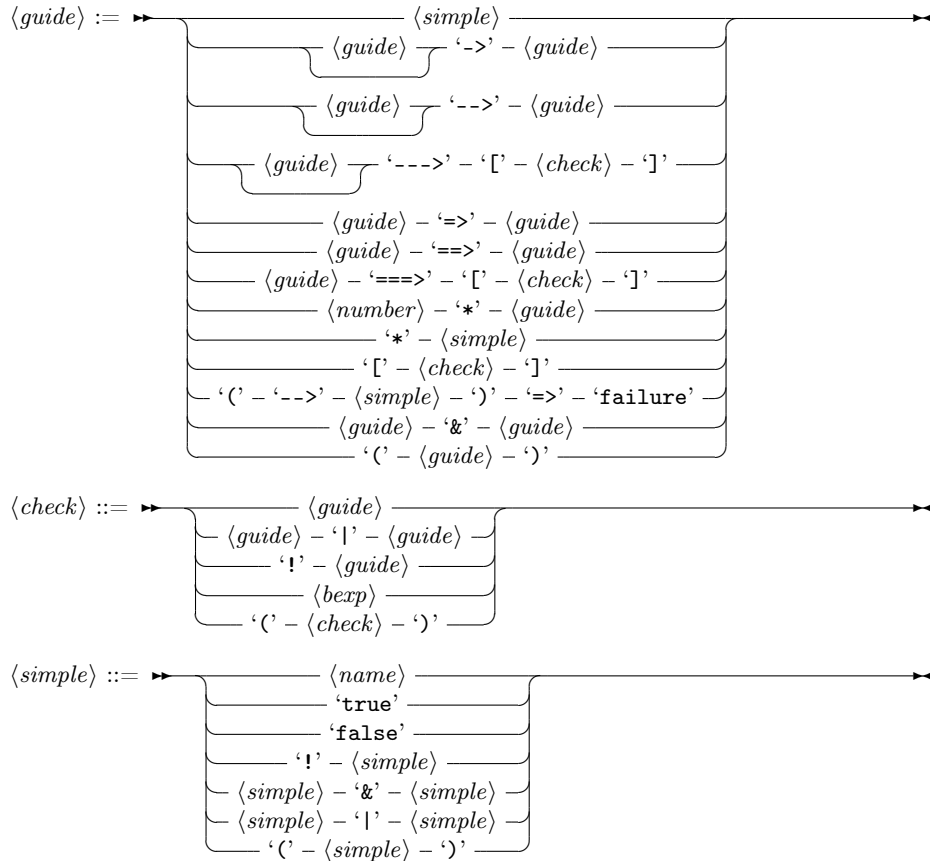
Guiding the simulator requires to specify which transition the simulator should execute. Testing whether some property holds in a state of the model requires a specification of this property. To this end, we introduce the *Britney Temporal Logic* (BTL). This logic is similar to linear-time logic (LTL) [12] but in addition to checking properties also allows guiding the simulator and to specify constraints that should hold in a state. We also adopt a syntax more similar to common descriptions of Petri net firing sequences rather than cryptic abbreviations or symbols to make it easier for practitioners to adopt the logic. The choice for an LTL-like logic reflects our wish to have existential counterexamples that can be represented by a simple firing sequence. Other kinds of counterexamples are difficult to find using simulation only. In the following, we define the syntax of BTL formulae and then their semantics based on Kripke structures [10] and structural operational semantics (SOS) [11].

#### 3.1 Syntax

A BTL formula is a *⟨guide⟩*. A guide describes how simulation should be performed; that is, it guides the simulator to a desired state. The atomic propositions of a guide are described using *⟨simple⟩*, which is an expression without temporal operators but otherwise allowing full propositional logic on transitions and place invariants. The temporal operators are various arrows emulating the arrows typically used to describe transition steps. Thus  $a \rightarrow b$  means that first  $a$  must hold and subsequently  $b$  must hold. For example,  $a$  and  $b$  can represent transitions, meaning that for the formula to hold, the corresponding transitions are executed one after the other. We lift this operator to  $a \dashrightarrow b$  meaning that  $a$  must hold and at some point afterward  $b$  must hold. Finally,  $a \dashrightarrow [b]$  means that  $a$  must hold and when the simulation stops  $b$  must hold. The brackets indicate that  $b$  is not used for guiding the simulation anymore (it has terminated after all). We

can omit  $a$ , which is an abbreviation for *true*. For each kind of arrow, we also add a double arrow version indicating that *if a holds, then b just holds at the appropriate time*. We also allow bounded and unbounded repetition using a star syntax. In contrary to a regular Kleene star, we put it in front as it improves readability for western readers. We allow conjunctions of guides using  $\&$ , but no disjunctions because for an expression like  $(a \rightarrow b) \mid (c \rightarrow d)$  it is not obvious whether to guide the simulation with an  $a$  or a  $c$  if both are enabled (as we do not know whether  $b$  or  $d$  are enabled in the next step).

A guide can also include  $\langle \textit{check} \rangle$ s, which are not used to guide the simulator but only to test assertions. They are therefore allowed to contain disjunctions and negations and general boolean expressions. Finally, a guide can include the special keyword **failure**, which is a synonym for **false** but with a very restricted syntax. This means that we try to stay clear of transitions that would violate the formula. The BTL formula in Listing 2 (ll. 8-12) guides a model to execute a transition **Order** exactly 10 times (l. 9) and a transition **Receive** exactly 10 times (l. 10) with any intermediate transitions allowed except for **Reject** (l. 11).



In addition to the syntax for guiding, we also allow boolean expressions. These are mostly for testing state properties, such as counting the tokens on a



place or testing values of the global clock. Boolean expressions are defined in the standard way and shall not be repeated here due to space limitations. Line 12 in Listing 2 tests that `Handle Return` is never executed (but does not enforce it like the guides). The formula in line 14 checks that at any point during execution, the three places `Reject`, `Offer`, and `Accept` never contain 3 or more tokens in total.

### 3.2 Semantics

We interpret formulae specified in BTL over the state space of a CPN. The state space of a CPN can be seen as a Kripke structure  $K = (Q, \delta, q_0, \Sigma, \lambda)$ , where  $Q$  is a set of states,  $q_0 \in Q$  is the initial state,  $\Sigma$  is a set of transition labels,  $\delta \subseteq Q \times \Sigma \times Q$  is the transition relation, and function  $\lambda : Q \rightarrow 2^{AP}$  maps each state  $q \in Q$  to a set of atomic propositions that hold in  $q$ . As usual,  $AP$  denotes the set of all atomic propositions. As BTL is an LTL-like logic, we also introduce the notion of a trace. A trace is a transition sequence  $q_0, \dots, q_k$  such that  $q_0$  is the initial state and for all  $0 \leq i < k$ , there exists an  $a \in \Sigma$  with  $(q_i, a, q_{i+1}) \in \delta$ . The semantics is similar to a standard finite trace semantics for LTL like the one defined in [5].

Our syntax includes a lot of conveniences. We already mentioned that avoiding the precondition for the single-arrows is a convenience for a precondition of *true* and that `failure` is semantically the same as `false`. Furthermore, all single arrows can be defined from the double arrows by forcing the precondition. The eventuality defined by  $a \Rightarrow b$  can be defined in terms of the unbounded repetition and the next operator, and bounded repetition is just a syntactical convenience:

$$\begin{aligned}
->G &\equiv \text{true} \rightarrow G \\
-->G &\equiv \text{true} \rightarrow\rightarrow G \\
--->[B] &\equiv \text{true} \rightarrow\rightarrow\rightarrow[B] \\
\text{failure} &\equiv \text{false} \\
G_1 \rightarrow G_2 &\equiv G_1 \& G_1 \Rightarrow G_2 \\
G_1 \rightarrow\rightarrow G_2 &\equiv G_1 \& G_1 \Rightarrow\Rightarrow G_2 \\
G \rightarrow\rightarrow\rightarrow[B] &\equiv G \& G \Rightarrow\Rightarrow\Rightarrow[B] \\
G_1 \Rightarrow\Rightarrow G_2 &\equiv G_1 \rightarrow (*\text{true} \rightarrow G_2) \\
n * (G) &\equiv \begin{cases} G \rightarrow (n-1) * (G) & \text{if } n \geq 1 \\ \text{true} & \text{otherwise.} \end{cases}
\end{aligned}$$

The semantics of *simple* can now be defined over the traces of a Kripke structure  $K$ . Attribute *name* in the grammar thereby refers to a transition label.

- Every trace  $(q_0, \dots, q_k)$  satisfies  $(q_0, \dots, q_k) \models \text{true}$  and  $(q_0, \dots, q_k) \not\models \text{false}$ .
- $(q_0, q_1, \dots, q_k) \models \text{name}$  iff  $q_0 \xrightarrow{\text{name}} q_1$ .
- $(q_0, q_1, \dots, q_k) \models !S$  iff  $(q_0, q_1, \dots, q_k) \not\models S$ .
- $(q_0, q_1, \dots, q_k) \models S_1 \& S_2$  iff  $(q_0, q_1, \dots, q_k) \models S_1 \wedge (q_0, q_1, \dots, q_k) \models S_2$ .

$$- (q_0, q_1, \dots, q_k) \models S_1 | S_2 \text{ iff } (q_0, q_1, \dots, q_k) \models S_1 \vee (q_0, q_1, \dots, q_k) \models S_2.$$

Boolean expressions can be evaluated over the set  $AP$  of atomic propositions and are, therefore, evaluated at a state. The remaining operators are LTL-like and are therefore defined over traces. We write  $q_0 \xrightarrow{G} q_k$  to denote that  $(q_0, \dots, q_k) \models G$ . The first is unbounded repetition. We notice that this is satisfied regardless of what we do (as zero repetitions can be performed).

$$\frac{q \xrightarrow{G} q'', q'' \xrightarrow{(*'G)} q'}{q \xrightarrow{(*'G)} q'}, \quad \frac{}{q \xrightarrow{(*'G)} q'} \quad (1)$$

Operator  $\Rightarrow$  is similar to the next operator in LTL (though here it is conditional): If  $G_1$  holds on a trace  $q_0, \dots, q_j$  then  $G_2$  must hold starting from  $q_j$ .

$$\frac{q_0 \xrightarrow{G_1} q_j \implies q_j \xrightarrow{G_2} q_k, 0 < j < k}{q_0 \xrightarrow{G_1 \Rightarrow G_2} q_k} \quad (2)$$

The following three rules define operators used for checking a property (expressed by putting the expression into squared brackets). We use them to check whether a boolean expression holds, a boolean expression holds in a final state after guiding the simulator using expression  $G$ , and whether a guide holds.

$$\frac{q_0 \models B}{q_0 \xrightarrow{[B]} q_k}, \quad \frac{q_k \models B}{q_0 \xrightarrow{G \implies [B]} q_k}, \quad \frac{\forall j \leq k : \neg(q_0 \xrightarrow{G} q_j)}{q_0 \xrightarrow{G \implies [B]} q_k}, \quad \frac{q_0 \xrightarrow{G} q_k}{q_0 \xrightarrow{[G]} q_k} \quad (3)$$

Finally we define a conjunction as usual:

$$\frac{q_0 \xrightarrow{G_1} q_k \wedge q_0 \xrightarrow{G_2} q_k}{q_0 \xrightarrow{G_1 \& G_2} q_k} \quad (4)$$

The final consideration is how we guide. This is done by defining a set of allowed transitions for each guide. For simulation, only transitions that are in this set are considered. This in particular means that if the set of allowed transitions is empty, the simulation is considered finished (and not with an error unless the formula is not satisfied). We define the set *guide* over a set  $T$  of possible transitions inductively as:

$$\begin{aligned} - \text{guide}(q, S) &= \{ \text{name} \in T \mid q \xrightarrow{\text{name}} q' \implies (q, q') \models S \}, \\ - \text{guide}(q, n \text{ } (*'G) &\begin{cases} \text{guide}(G) & \text{if } n \geq 1, \\ T & \text{otherwise,} \end{cases} \\ - \text{guide}(*G) &= T, \\ - \text{guide}(q, G_1 \Rightarrow G_2) &= T, \\ - \text{guide}(q, [B]) &= T, \\ - \text{guide}(q, G \implies [B]) &= T, \\ - \text{guide}(q, [G]) &= T, \end{aligned}$$

- $guide(q, G_1 \wedge G_2) = guide(q, G_1) \cap guide(q, G_2)$ ,
- $guide(q, (->S)) \Rightarrow failure = T \setminus guide(q, S)$ .

We more or less just allow concrete steps if they are needed to satisfy a formula or forbid a step if it would violate a formula, and otherwise allow anything when we do not care about the outcome.

### 3.3 Implementation

Our implementation of BTL uses simple formula rewriting. Our implementation implements the *guide* set for filtering enabled transitions, pick and execute one that is in the *guide* set and in the set of enabled transitions. We then rewrite the formula according to the previous rules. For efficiency, we have expanded some of the syntactical equivalences, most importantly the future temporal operator ( $a \Rightarrow b$ ). When no more transitions are in the intersection, we check if the rewritten formula is satisfied for the empty trace.

We evaluate formulae using a four-value logic similar to [2]. The idea is that we have two versions of both true and false: the value is definite and can never change and the value is true/false but may change with further execution. For example, if we have a formula  $a \rightarrow b$  and execute  $c$  we know for sure that we can never satisfy the formula (we say it is permanently false), whereas for  $a \rightarrow b$  if we execute a  $c$ , the formula is only temporarily false (we still have proof obligations but may be able to satisfy them in the future). This allows us to terminate early once a formula is permanently true or false. This has the added advantage of allowing us to provide a rewritten formula after executing a sequence of steps, which often contains hints of shortcomings of the model. Figure 4 shows such an error report, containing an error trace (which we have shortened here), the violated formula (ll. 10–14 from Listing 2), the formula at the error, and the marking at the error.

```

• Executed Trace:
System.Order{ c = "Jim", i = 0, p = "Book" }
System.Ship{ c = "Jim", id = 0, p = "Book" }
System.Order{ c = "Jim", i = 1, p = "Laptop" }
System.Put_in_Storage{ c = "Ann", id = 8, p = "Book" }
System.Put_in_Storage{ c = "Ann", id = 9, p = "Bike" }
• Initial Formula:
(10 * (-> Order) -> (-> Order) => failure) &
((-> Reject) => failure) &
(10 * (-> Receive) -> (-> Receive) => failure) &
[(-> Handle_Return) => false]
• Formula at error:
(((-> (order)) => failure) & ((-> (reject)) => failure) & ((10 * (-> (receive))) & ((10 * (-> (receive))) => ((-> (receive)) => failure)))) & (((-> (handlerreturn)) => (!(true))))
• Marking at error:
System.Accept: 1 (1,"Jim")
System.Count: 1 10

```

Fig. 4: Error report.

Figure 4 shows such an error report, containing an error trace (which we have shortened here), the violated formula (ll. 10–14 from Listing 2), the formula at the error, and the marking at the error. If a model has no error, we instead show the final state which can be manually inspected for irregularities, e.g. improper termination.

In addition to the grader used by teachers to finally grade assignments, we also have two tools for testing BTL formulas. One is used during construction to help a teacher get immediate feedback on a formula and a solution by manually or experimentally testing formulas on a proposed solution. A simplified version of this tool allows students to check that their models conform to the formulas. This tool also allows students to manually single-step through their model and watch the formula progress, aiding in finding and avoiding obvious errors before handing in.

## 4 Practical Experience

In this section, we present first experiences we made with Grade/CPN in supporting the evaluation of a CPN assignment in the course Business Information Systems at the Eindhoven University of Technology. In this assignment, students were given the base model in Fig. 1 and they had to model the delivery system according to a textual specification. Each of the 94 students had to submit two models. We received in total 130 models from 66 students.

In a first step of the assessment, we applied Grade/CPN by calling it with a student model, the base model, and a configuration file (see Listing 2). Here, we were interested whether the interface and declaration of the base model have been preserved, whether there is a suspicion of fraud, and whether six scenarios can be replayed on the model (only two are shown in the Listing). The scenarios were part of the specification of the assignment, and we specified them using BTL. As BTL refers to the interface, it is crucial that students have not changed it. The runtime of the tool was about ten minutes for all students; that is, after this time, a report had been generated for each student. The tool detected two fraud attempts, though they turned out to be caused by students handing in a subsequent assignment using the same base model as well.

In a second step, we manually checked each of the generated reports. On average, this took less than five minutes for each report. Based on the feedback provided by Grade/CPN, it was easy to check whether a model was actually correct or not. Basically, the violation of a certain scenario simplified the detection of the cause for this violation drastically. In most cases, we did not even have to look at the counterexample provided by our tool. Overall, we had to simulate only five models manually to determine the cause of an error. The tool automatically detected several subtle errors such as wrong guards and minor changes to the environment without having the need to open the respective model; it is highly unlikely we would have caught all of these completely manually. We even found subtle errors in our own solutions, yielding better results.

Based on experience from previous years, the use of Grade/CPN reduced the amount of time for grading the assignment by a factor of at least two to three. This is factoring in that we used Grade/CPN for the first time and had to both define and understand the defined logic BTL, and also did not place complete confidence in the reported results which probably increased the manual labor as well. As correcting models is a rather monotonous work, it is easily possible that one oversees an error or forgets to check some scenario. Using Grade/CPN, this is now impossible and, therefore, we think that we can provide students with a fairer (in the sense of more equal) grading on the one hand and better feedback on the other hand.

## 5 Conclusion and Future Work

We have presented Grade/CPN, a tool to semi-automatically grade CPN models. Using Access/CPN, we can support any model created using CPN Tools.

The plug-in architecture makes the tool easily extendible: to do so, one must just implement the interface in Listing 1 (ll. 1–5). The pluggable configuration with a very simple base format makes configuration simple. Configuration comprises selecting which plug-ins to use, which weight to assign them, and which parameters to instantiate them. Each plug-in only needs to consider its own options as the overall configuration format is handled by Grade/CPN. Reporting is handled by making all plug-ins return simple messages optionally annotated with more detailed reasoning (Listing 1 ll. 13–15). The information is automatically gathered by Grade/CPN and presented both as an overview in the user interface and as a detailed report. We have presented both simple plug-ins and a very powerful one implementing guided checking of Britney Temporal Logic (BTL). BTL allows us to guide the simulation toward desired scenarios and to check that the environment contracts are adhered to. All plug-ins provide categorized information explaining the score and highlighting any changes made to the model, so teachers processing the reports only have to focus on things that cannot be automatically checked. We have designed and implemented an infrastructure for detecting fraud. We have reported on our experience with the Business Information Systems course where Grade/CPN was used to grade 130 assignments from 66 students. Using Grade/CPN instead of a completely manual approach reduced the manual labor by a factor of three. Each student has to hand in a total of five models during the course, so anticipated time savings are immense.

The idea of (semi-)automatically grading assignments is not new and closely related to testing. A known testing framework is JUnit [9], which also runs a set of tests and reports the result. The advantage of our tool over JUnit is that JUnit requires programming to even get started, whereas we use simple configuration files. Also from the testing world is Jenkins (previously known as Hudson) [7], which runs tests on a central server and provides near-instantaneous feedback. The main disadvantage of Jenkins in our view is also complexity; while it does not (necessarily) require programming, setup does require complex XML configuration and extension either requires huge effort or makes it difficult to get consolidated reports. There are many tools for automatically grading programming assignments [6], for example, the tool peach<sup>3</sup> [13], which more focus on managing hand-ins, but can also run automatic tests. In contrast, we focus on the tests and CPN models directly and assume that models already exist. Our testing approach is similar to runtime LTL [2,5], but our logic also supports guiding. This is similar to hot/cold events in Live-Sequence Charts [4], but our sections are more urgent in that a guide is not only preferred, it is an immediate failure if it is not possible to follow it. This makes BTL computationally easier to check.

Future work includes loosening what is allowed in guides by having the tool try resolving, e.g., disjunctions itself (by keeping track of which branches have been explored and only reporting errors if no branch is successful). We also consider a designer for automatically building BTL formulae and full configuration files by manually guiding the simulation in a manner similar to our current

tool for testing BTL formulas. We aim to integrate Grade/CPN with Jenkins or peach<sup>3</sup> so we can combine our simplicity of configuration with the more advanced features of those systems. While BTL is designed for grading assignments using testing, it has also proved useful for finding errors in our model. It would be interesting to investigate this further, including making testing complete by exploiting the guiding perspective of our logic.

*Acknowledgements.* The authors thank Boudewijn van Dongen for fruitful discussions about the requirements for an automatic grader.

## References

1. W.M.P. van der Aalst and C. Stahl. *Modeling Business Processes – A Petri Net-Oriented Approach*. MIT Press, 2011.
2. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL Semantics for Runtime Verification. *Logic and Computation*, 20(3):651–674, 2010.
3. CPN Tools webpage. Online: [cpntools.org](http://cpntools.org).
4. Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Form. Methods Syst. Des.*, 19(1):45–80, 2001.
5. D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proc. ASE*, pages 412–416. IEEE Computer Society, 2001.
6. P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proc. International Conference on Computing Education Research*, pages 86–93. ACM, 2010.
7. Jenkins Continuous Integration webpage. Online: [jenkins-ci.org](http://jenkins-ci.org).
8. K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009.
9. JUnit webpage. Online: [junit.org](http://junit.org).
10. S.A. Kripke. A semantical analysis of modal logic: I. Normal modal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.
11. G.D. Plotkin. A Structural Approach to Operational Semantics. DAIMI-FN 19, Department of Computer Science, University of Aarhus, 1981.
12. A. Pnueli. The Temporal Logic of Programs. In *Proc. of SFCS '77*, pages 46–57. IEEE Comp. Soc., 1977.
13. T. Verhoeff. Programming Task Packages: Peach Exchange Format. *Olympiads in Informatics*, 2:192–207, 2008.
14. M. Westergaard and L.M. Kristensen. The Access/CPN Framework: A Tool for Interacting With the CPN Tools Simulator. In *Proc. of ATPN*, volume 5606 of *LNCS*, pages 313–322. Springer, 2009.

# When Can We Trust a Third Party?

## A Soundness Perspective

Kees M. van Hee, Natalia Sidorova, and Jan Martijn van der Werf

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

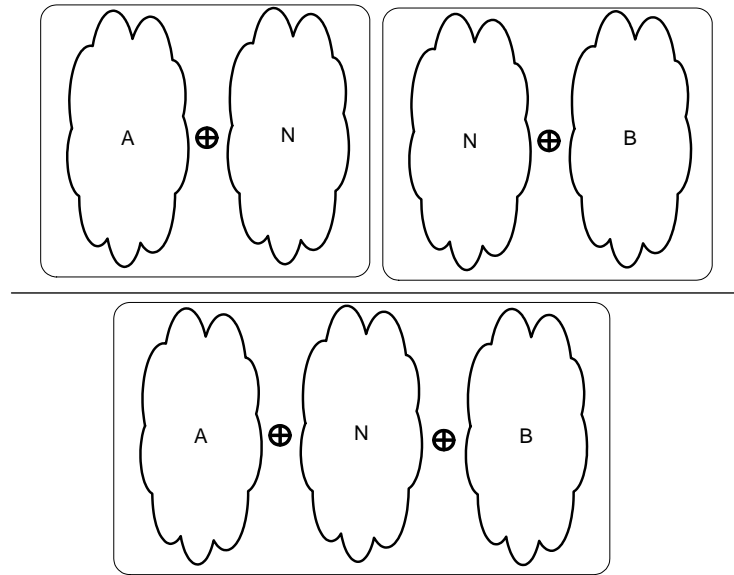
**Abstract.** Organizations often do not want to reveal the way a product is created or a service is delivered. As a consequence, if two organizations want to cooperate, they contact a trusted third party. Each specifies how it wants to communicate with the other party. The trusted third party then needs to assure that the two organizations cooperate correctly. In this paper, we study requirements on trusted third parties to ensure correct cooperation between the different organizations.

## 1 Introduction

Organizations need to anticipate on the increasing dynamicity and complexity of business markets. Therefore, organizations focus more and more on their core activities. As a result, organizations need to cooperate in large networks. The organizations in the network have as common goal the delivery of their services. Such a network is called a *virtual enterprise* [11].

Communication between the organizations is asynchronous by nature: an organization sends some data, like an inquiry, to some other organization, and eventually the latter organization sends a response. Therefore, we use Petri nets to model organizations using *components*. Components can be composed into networks of components. Such a network is again a component. A component has an initial state and a desired final state. We say that a component, or a network of components communicates correctly if it is *sound*, i.e., if in all its reachable states the component is always able to eventually reach the desired final state.

Trust is an important property in a network of cooperating organizations: organizations share business knowledge and intellectual property with other organizations within the network in order to organize the component network properly and achieve desired goals. At the same time, organizations often want to keep some intellectual property within their organization and avoid sharing it for clear reasons. A common approach used in non-virtual life is the use of trusted third parties. It becomes nowadays also quite common in the virtual world. In this paper, we consider the use of a *third party*, also called a *notary*, that is trusted by all the organizations in the network. By using a notary, each



**Fig. 1.** If the notary  $N$  communicates correctly with  $A$  and  $B$  individually, we want to conclude correctness of the network of  $A$ ,  $B$  and  $N$

of the organizations explains to the notary the way it wants to conduct business, and the notary will assure that the organizations can do business together. This requires the notary to ensure that it communicates correctly with each of the organizations, i.e., that the notary with each of the individual organizations can reach the common goals, and secondly, that the complete network with all the organizations together can reach its common goals. If this is the case, we call the notary *trusted*.

In this paper we limit ourselves to the cooperation between two organizations using a notary. Rather than to use verification to check whether the communication between the notary and the two organizations is correct, we search for conditions such that if the communication between the notary and each of the individual organizations is correct, we can automatically conclude that the communication between the three parties is correct, as depicted in Fig. 1.

This paper is structured as follows. Sec. 2 introduces the basic notions needed throughout the paper. Sec. 3 explains the concept of components and their composition. In Sec. 4 we study the conditions under which the notary is guaranteed to ensure soundness of the composition of the three parties. Sec. 5 concludes the paper.



## 2 Preliminaries

Let  $S$  be a set. The powerset of  $S$  is denoted by  $\mathcal{P}(S) = \{S' \mid S' \subseteq S\}$ . We use  $|S|$  for the number of elements in  $S$ . Two sets  $S$  and  $T$  are *disjoint* if  $S \cap T = \emptyset$ .

A *bag*  $m$  over  $S$  is a function  $m : S \rightarrow \mathbb{N}$ , where  $\mathbb{N} = \{0, 1, \dots\}$  denotes the set of natural numbers. We denote e.g. the bag  $m$  with an element  $a$  occurring once,  $b$  occurring three times and  $c$  occurring twice by  $m = [a, b^3, c^2]$ . The set of all bags over  $S$  is denoted by  $\mathbb{N}^S$ . Sets can be seen as a special kind of bag where all elements occur only once; we interpret sets in this way whenever we use them in operations on bags. We use  $+$  and  $-$  for the sum and difference of two bags, and  $=$ ,  $<$ ,  $>$ ,  $\leq$ ,  $\geq$  for the comparison of two bags, which are defined in a standard way. The projection of a bag  $m \in \mathbb{N}^S$  on some set  $U$  is a bag defined by  $m|_U(s) = m(s)$  if  $s \in U$  and  $m|_U(s) = 0$  otherwise.

A *sequence* over  $S$  of length  $n \in \mathbb{N}$  is a function  $\sigma : \{1, \dots, n\} \rightarrow S$ . If  $n > 0$  and  $\sigma(i) = a_i$  for  $i \in \{1, \dots, n\}$ , we write  $\sigma = \langle a_1, \dots, a_n \rangle$ . The length of a sequence is denoted by  $|\sigma|$ . The sequence of length 0 is called the *empty sequence*, and is denoted by  $\epsilon$ . The set of all finite sequences over  $S$  is denoted by  $S^*$ . We write  $a \in \sigma$  if  $\sigma(i) = a$  for some  $1 \leq i \leq |\sigma|$ . *Concatenation* of two sequences  $\nu, \gamma \in S^*$ , denoted by  $\sigma = \nu; \gamma$ , is a sequence defined by  $\sigma : \{1, \dots, |\nu| + |\gamma|\} \rightarrow S$ , such that  $\sigma(i) = \nu(i)$  for  $1 \leq i \leq |\nu|$ , and  $\sigma(i) = \gamma(i - |\nu|)$  for  $|\nu| + 1 \leq i \leq |\nu| + |\gamma|$ . We inductively define the projection of  $\sigma \in S^*$  on some set  $U$  by  $a; \sigma'|_U = \langle a \rangle; \sigma'|_U$  if  $a \in U$  and  $a; \sigma'|_U = \sigma'|_U$  otherwise.

**Definition 1 (Petri net [13]).** A Petri net  $N$  is a tuple  $(P, T, F)$  where (1)  $P$  and  $T$  are two disjoint sets of places and transitions respectively, we call an element of the set  $(P \cup T)$  a node of  $N$ ; and (2)  $F \subseteq (P \times T) \cup (T \times P)$  is the flow relation. An element of  $F$  is called an arc.

Let  $N = (P, T, F)$  be a Petri net. Given a node  $n \in (P \cup T)$ , we define its preset by  ${}^\bullet_N n = \{x \mid (x, n) \in F\}$ , and its postset by  $n^\bullet_N = \{x \mid (n, x) \in F\}$ . We omit the subscript if the context is clear.

Let  $N = (P, T, F)$  be a Petri net. A path from a node  $n \in P \cup T$  to a node  $m \in P \cup T$  is a sequence  $\pi \in (P \cup T)^*$  such that  $(\pi(i), \pi(i+1)) \in F$  for all  $1 \leq i < n$ . The set of all paths from  $n$  to  $m$  is denoted by  $\Pi(n, m)$ . A path is called *cyclic* if there exists a path  $\pi$  of length  $n > 0$  such that  $\pi(1) = \pi(n)$ . If  $N$  has a cyclic path, the net is called *cyclic*. If no such cycle exists, it is called *acyclic*.

To describe the semantics of a Petri net, we use *markings*. A *marking* of  $N$  is a bag  $m \in \mathbb{N}^P$ , where  $m(p)$  denotes the number of *tokens* in place  $p \in P$ . If  $m(p) > 0$ , place  $p$  is called *marked* in marking  $m$ . A Petri net  $N$  with a marking  $m$  is written as  $(N, m)$  and is called a *marked Petri net*.

Given a marked Petri net  $(N, m)$  with  $N = (P, T, F)$ , a transition  $t \in T$  is *enabled*, denoted by  $(N : m \xrightarrow{t})$ , if  ${}^\bullet t \leq m$ . If a transition is enabled in  $(N, m)$ , it can *fire*. A transition firing, denoted by  $(N : m \xrightarrow{t} m')$ , results in a new marking  $m' = m - {}^\bullet t + t^\bullet$ . We lift the firing to sequences of transitions in the standard way. A sequence  $\sigma \in T^*$  of length  $n$  is a *firing sequence* from  $m_0$  to

$m_n$ , if there exist markings  $m_i, m_{i+1} \in \mathbb{N}^P$  such that  $(N : m_i \xrightarrow{\sigma^{(i)}} m_{i+1})$  for all  $1 \leq i < |\sigma|$ . The set of reachable markings from a given marking  $m$  is denoted as  $\mathcal{R}(N, m) = \{m' \mid \exists \sigma \in T^* : (N : m \xrightarrow{\sigma} m')\}$ . We lift the set of reachable markings from a single marking to a set of markings in a standard way, i.e., given a set  $M \subseteq \mathbb{N}^P$ ,  $\mathcal{R}(N, M) = \bigcup_{m \in M} \mathcal{R}(N, m)$ .

Given a marked Petri net  $(N, m_0)$  with  $N = (P, T, F)$ , a place  $p \in P$  is called  $k$ -bounded for some  $k \in \mathbb{N}$  if  $m(p) \leq k$  for all markings  $m \in \mathcal{R}(N, m_0)$ . If all places are  $k$ -bounded, we call  $(N, m_0)$   $k$ -bounded. A transition  $t \in T$  is called *live* if for all markings  $m \in \mathcal{R}(N, m_0)$  there exist a firing sequence  $\sigma \in T^*$  and a marking  $m' \in \mathcal{R}(N, m)$  such that  $(N : m \xrightarrow{\sigma} m' \xrightarrow{t})$ . If all transitions of  $(N, m_0)$  are live,  $(N, m_0)$  is called live. A transition  $t \in T$  is called *quasi-live* if there exists a marking  $m \in \mathcal{R}(N, m_0)$  such that  $(N : m \xrightarrow{t})$ . If all transitions of  $(N, m_0)$  are quasi-live, the marked Petri net is called quasi-live. A marking  $m \in \mathcal{R}(N, m_0)$  is called a *home marking* if  $m \in \mathcal{R}(N, m')$  for all  $m' \in \mathcal{R}(N, m_0)$ . A reachable marking  $m \in \mathcal{R}(N, m_0)$  is called a *deadlock* of  $(N, m_0)$  if there is no transition  $t \in T$  with  $(N : m \xrightarrow{t})$ . Given a desired marking  $f \in \mathcal{R}(N, m_0)$ , a non-empty subset of markings  $L \subseteq \mathcal{R}(N, m_0)$  is called a *live-lock* w.r.t  $f$  if  $f \notin \mathcal{R}(N, L)$  and  $L = \mathcal{R}(N, L)$ , i.e., from  $L$  the desired marking is not reachable, and no other marking than a marking in  $L$  can be reached from  $L$ .

On Petri nets, we define two classes based on their structure: S-Nets, also called state machines, and workflow nets. A Petri net  $N = (P, T, F)$  is a *S-net* if  $|\bullet t| \leq 1$  and  $|t^\bullet| \leq 1$  for all transitions  $t \in T$ .

**Definition 2 (Workflow net, closure).** *Let  $N = (P, T, F)$  be a Petri net. It is a workflow net (WFN) if there exist two places  $i \in P$  and  $f \in P$ , called the initial place and final place respectively, such that  $\bullet i = f^\bullet = \emptyset$ , and all nodes of  $N$  are on a path from  $i$  to  $f$ . Its closure is the net  $N^* = (P, T \cup \{t^*\}, F \cup \{(t^*, i), (f, t^*)\})$ , where  $t^* \notin T$ .*

A workflow net is called *sound* if (1) it is weakly terminating, i.e., it always has the option to reach the final marking in which only the final place is marked, (2) it is properly completing, i.e., if in a marking the final place is marked, it is the only place marked, and (3) all transitions have a function, i.e., for every transition a reachable marking exists that enables the transition. Note that we use the classical soundness definition [1, 2].

**Definition 3 (Soundness).** *A workflow net  $N = (P, T, F)$  with initial place  $i$  and final place  $f$  is called sound if (1)  $[f]$  is a home marking of  $(N, [i])$ , (2) for any reachable marking  $m \in \mathcal{R}(N, [i])$ , if  $m \geq [f]$  then  $m = [f]$ , and (3)  $(N, [i])$  is quasi live.*

A WFN  $N = (P, T, F)$  with initial place  $i$  is sound if and only if the marked Petri net  $(N^*, [i])$  is live and bounded [1].

If we give a tuple a name, we subscript the elements with the name of the tuple, e.g. for  $N = (A, B, C)$  we refer to its elements by  $A_N, B_N$ , and  $C_N$ . If the context is clear, we omit the subscript.

### 3 Components and their Composition

In this paper, we use asynchronously communicating components [5,7]. We therefore model our components using Petri nets with interface places, called *open Petri nets* (OPNs) [10,14]. An OPN has two types of places: *internal places* for the inner control of the component, and *interface places* to communicate with its environment. An interface place is either an output place, i.e., it sends a message to the environment, or an input place, i.e., it requires a message from the environment. Further, a component has an initial and a final marking, defining the desired begin and end markings of the component.

**Definition 4 (Open Petri net, skeleton, open workflow net [3]).** An open Petri net (OPN) is a 6-tuple  $(P, I, O, T, F, i, f)$  where

- $((P \cup I \cup O, T, F), i)$  is a marked Petri net;
- $P$  is a set of internal places;
- $I$  is a set of input places, and  $\bullet I = \emptyset$ ;
- $O$  is a set of output places, and  $O^\bullet = \emptyset$ ;
- $P, I$  and  $O$  are pairwise disjoint;
- $\forall t \in T : |(\bullet t \cup t^\bullet) \cap (I \cup O)| \leq 1$ ; and
- $i \in \mathbb{N}^P$  is the initial marking; and
- $f \in \mathbb{N}^P$  is the final marking.

We call the set  $I \cup O$  the interface places of the OPN. Two OPNs  $N$  and  $M$  are called disjoint if  $(P_N \cup I_N \cup O_N \cup T_N) \cap (P_M \cup I_M \cup O_M \cup T_M) = \emptyset$ . An OPN  $N$  is called closed if  $I_N = O_N = \emptyset$ . We write  $\mathcal{R}(N, m)$  for  $\mathcal{R}((P_N \cup I_N \cup O_N, T_N, F_N), m)$  for  $m \in \mathbb{N}^{P_N \cup I_N \cup O_N}$ .

The skeleton of  $N$  is defined as the Petri net  $\mathcal{S}(N) = (P_N, T_N, F)$  with  $F = F_N \cap ((P_N \times T_N) \cup (T_N \times P_N))$ . For nodes  $n \in (P_N \cup T_N)$ , we write  $\overset{\circ}{N}n$  and  $t_N^\circ$  as a shorthand for  $\overset{\bullet}{\mathcal{S}(N)}t$  and  $t_{\mathcal{S}(N)}^\bullet$ , respectively.

If  $\mathcal{S}(N)$  is a workflow net with initial place  $s$  and final place  $o$ ,  $i = [s]$  and  $f = [o]$ ,  $N$  is called an open workflow net.

OPNs are composed with each other to build networks of communicating components. As a network of components can be used as a component again, the result of the composition is a component too. We say two OPNs are *composable* if the only elements shared between the two OPNs are their interface places, such that input places of the one are output places of the other and vice versa. Composition is then defined as the union of the two OPNs.

**Definition 5 (Composition of OPNs [3]).** Two OPNs  $A$  and  $B$  are composable, denoted by  $A \oplus B$ , if and only if  $(P_A \cup I_A \cup O_A \cup T_A) \cap (P_B \cup I_B \cup O_B \cup T_B) = (O_A \cap I_B) \cup (I_A \cap O_B)$ .

If  $A$  and  $B$  are composable, their composition results in an OPN  $A \oplus B = (P_A \cup P_B \cup H, (I_A \cup I_B) \setminus H, (O_A \cup O_B) \setminus H, T_A \cup T_B, F_A \cup F_B, i_A + i_B)$  with  $H = (O_A \cap I_B) \cup (I_A \cap O_B)$ .

Note that two disjoint OPNs are composable by definition. Two important properties of composition are commutativity and projection, as shown in [14].

**Corollary 6 (Commutativity, projection property [14]).** *Let  $A$  and  $B$  be two composable OPNs. Then  $N = A \oplus B = B \oplus A$ , and  $(\mathcal{S}(A) : m|_{P_A} \xrightarrow{\sigma|_{T_A}} m'|_{P_A})$  for all firing sequences  $\sigma \in T_n^*$  and markings  $m, m' \in \mathbb{N}^{P_N}$  such that  $(\mathcal{S}(N) : m \xrightarrow{\sigma} m')$ .*

The composition operator allows to create arbitrary networks of communicating components. As long as the interface places match, it is allowed to compose the components. However, it does not guarantee that the components communicate correctly. Composition is thus a syntactic check whether components are able to communicate.

Components communicate correctly if all components in the network are able to reach their desired final marking, and no messages are pending in one of the interface places. Further, we do not want to have transitions that are unreachable in the composition. To express this property, we use the notion of *soundness* for components: a component is sound if, ignoring the communication with other components in the network, all components can reach their final marking, no tokens are left in the network, and for each transition in the network, a marking should be reachable in which the transition is enabled.

**Definition 7 (Soundness).** *An OPN  $N$  is sound if:*

1.  $\forall m \in \mathcal{R}(\mathcal{S}(N), i_N) : f_N \in \mathcal{R}(\mathcal{S}(N), m)$  (weak termination);
2.  $\forall m \in \mathcal{R}(\mathcal{S}(N), i_N) : m \geq f_N \implies m = f_N$  (proper completion); and
3.  $\forall t \in T_N : \exists m \in \mathcal{R}(\mathcal{S}(N), i_N) : {}^o t \leq m$  (quasi liveness).

Note that this soundness definition is stronger than the soundness notion used in [3], where soundness has been defined as the combination of weak termination and proper completion.

A direct consequence of the projection property and soundness is that if in a composition between  $A$ ,  $B$  and  $C$ , such that  $A$  and  $C$  are disjoint, and  $A$  and  $B$  are composable, as well as  $B$  and  $C$ , and  $B$  is in its final marking, then the other two components can reach their final marking as well.

**Lemma 8.** *Let  $A$ ,  $B$  and  $C$  be three pairwise composable OPNs such that  $A$  and  $C$  are disjoint, and  $A \oplus B$  and  $B \oplus C$  are sound. Define  $L = A \oplus B \oplus C$ . Then  $f_L \in \mathcal{R}(\mathcal{S}(L), m)$  for all markings  $m \in \mathcal{R}(\mathcal{S}(L), i_L)$  such that  $f_B \leq m$ .*

*Proof.* Define  $K = A \oplus B$ . By the projection property,  $(\mathcal{S}(K) : i_K \xrightarrow{\sigma|_{T_K}} m|_{P_K})$ . Since  $f_B \leq m$ , and  $K$  is sound, there exists a firing sequence  $\mu \in T_A^*$  such that  $(\mathcal{S}(K) : m|_{P_K} \xrightarrow{\mu} f_K)$ , and hence  $(\mathcal{S}(L) : m \xrightarrow{\mu} m')$  for some  $m' \in \mathbb{N}^{P_L}$  with  $f_K \leq m'$ . Applying the same argument for  $B \oplus C$ , there exists a firing sequence  $\nu \in T_B^*$  such that  $(\mathcal{S}(L) : m' \xrightarrow{\nu} f_L)$ , which proves the statement.  $\square$

### 4 Soundness Using Trusted Third Parties

Organizations have to cooperate more and more in order to do their business. However, they often do not want to share the way they operate, for example to hide internal business knowledge or intellectual property. An often proposed solution is a third party that is trusted by all organizations within the network. This third party, the notary, needs to ensure that it knows how the organizations within the network want to operate. On the one hand the notary needs to ensure that it works correctly with each individual organization, and on the other hand, that the network of all organizations, including the notary, is correct.

As the main purpose of a notary is to ensure correct behavior of the communication between the two organizations that want to cooperate, we model the notary by an OWN. The main actions of the notary are the sending and receiving of messages of the different components. Therefore, each transition that is communicating is labeled with the sending or receiving of a message, or as silent if the transition represents an internal step of the notary. We restrict the notary to state machines, i.e., each notary is sound by its structure [9].

**Definition 9 (Notary).** *Let  $A$  and  $B$  be two disjoint components. A notary, between  $A$  and  $B$  is an OWN  $N$  such that (1)  $A$  and  $N$ , as well as  $B$  and  $N$  are composable, (2)  $\mathcal{S}(N)$  is an S-net, (3) each transition is connected to at most one interface place, i.e.,  $|(\bullet t \cup t \bullet) \cap (I \cup O)| \leq 1$  for all  $t \in T$ , and, (4) each interface place is connected to exactly one transition, i.e.,  $|\bullet x \cup x \bullet| = 1$  for all  $x \in I \cup O$ .*

*As each transition is connected to at most one component, we introduce the communication function  $C_N : T \rightarrow \{A, B, \tau\}$  such that  $C_N(t) = X$  if  $\bullet t \cap O_X \neq \emptyset$  for  $X \in \{A, B\}$  and  $C_N(t) = \tau$  otherwise.*

#### 4.1 Acyclic Notaries

We first consider the case of an acyclic notary. If a notary is acyclic, then its set of possible firing sequences is finite. For acyclic notaries, soundness can be guaranteed, as shown in this section.

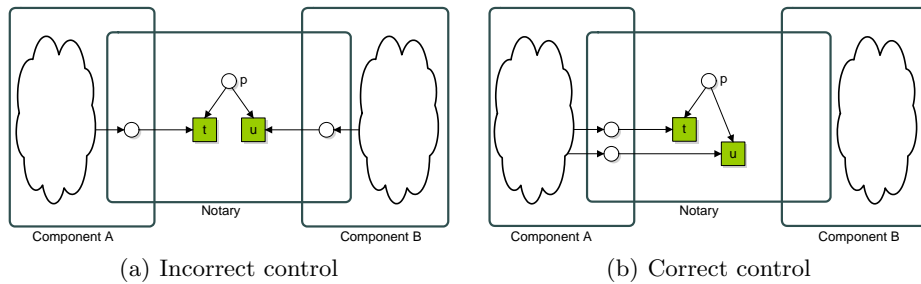


Fig. 2. Conflicts in a notary

One source of possible erroneous behavior lies in the control of conflicts: if in a notary two transitions share a place in their presets, then the transitions should either be both controlled by the same component, or by the notary. Consider the examples of Fig. 2. Taking the composition  $A \oplus N$  of Fig. 2(a), then transition  $u$  is always enabled if transition  $t$  is enabled, whereas in Fig. 2(b), component  $A$  controls the conflict in the composition of  $A$  and  $N$ . If the composition is sound, then the example of Fig. 2(a) is not possible, as shown in the next lemma.

**Lemma 10 (Conflict control).** *Let  $A$  and  $B$  be two components such that  $A$  and  $C$  are disjoint, and let  $N$  be an acyclic notary between  $A$  and  $B$ . If  $A \oplus N$  and  $N \oplus B$  are sound, then for all places  $p \in P$  and transitions  $t, u \in p^\bullet$  we have  $C_N(t) = C_N(u)$ .*

*Proof.* Let  $p \in P$  and  $t, u \in p^\bullet$ . Suppose  $C_N(t) \neq C_N(u)$ . This implies that at least one of the transitions  $t$  and  $u$  is controlled by a component (otherwise we would have  $C_N(t) = C_N(u) = \tau$ ). Without loss of generality, assume this transition to be  $t$  and component to be  $A$ , i.e.,  $C_N(t) = A$ . Then there exists a place  $q \in I_N \cap O_A$ , with  $q \in \bullet t$ .

Define  $M = A \oplus N$ . Since  $N$  is an S-net,  ${}_M^\circ u \subset {}_M^\circ t$ . Since  $M$  is sound, there exists a reachable marking  $m \in \mathcal{R}(\mathcal{S}(M), i_M)$  with  $(\mathcal{S}(M) : m \xrightarrow{t})$ , and thus  $m(q) > 0$ . Note that transition  $u$  is also enabled in  $m$ . Hence, we can fire transition  $u$  and obtain a marking  $m' \in \mathbb{N}^{P_M}$ :  $(\mathcal{S}(M) : m \xrightarrow{u} m')$ , where  $m' = m - {}_M^\circ u + u_M^\circ$  and  $m'(q) = m(q)$  (since  $q \notin {}_M^\circ u$ ).

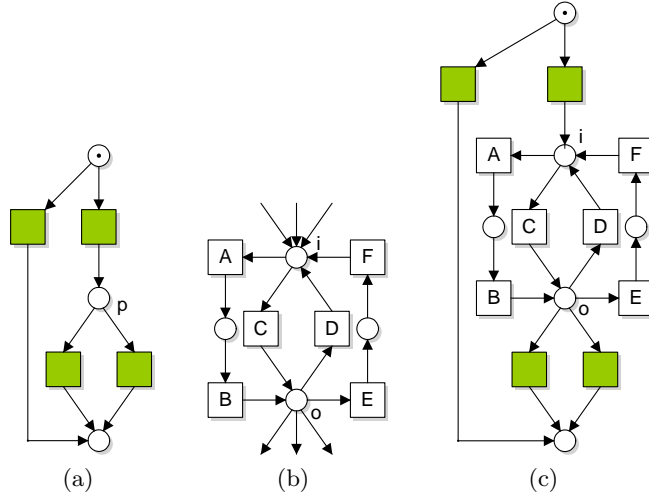
As  $N$  is acyclic and  $t$  is the only transition consuming from  $q$ , the token from  $q$  will never be consumed by any sequence firing from  $m'$ . Thus,  $M$  is not sound, which is a contradiction. Hence, the lemma holds.  $\square$

The lemma shows that conflicts (choices) in the notary component are always controlled correctly, either by a single component or the notary itself. Consequently, if the composition of the components of  $A$  and  $B$  with  $N$  individually is sound, the composition of the three is sound as well, as proven in the next theorem.

**Theorem 11.** *Let  $A$  and  $B$  be two OPNs such that  $A$  and  $B$  are disjoint. Let  $N$  be an acyclic notary between  $A$  and  $B$ . If  $A \oplus N$  and  $N \oplus B$  are sound, then  $A \oplus N \oplus B$  is sound.*

*Proof.* Define  $M = A \oplus N \oplus B$ . Suppose  $M$  is not sound. Then there exist a marking  $m \in \mathcal{R}(\mathcal{S}(M), i_M)$  and a firing sequence  $\sigma \in T_M^*$  such that  $(\mathcal{S}(M) : i_M \xrightarrow{\sigma} m)$ ,  $f_M \notin \mathcal{R}(\mathcal{S}(M), m)$ . Moreover, since  $N$  is acyclic, there exists such an  $m$ , and, additionally  $\gamma|_{T_N} = \emptyset$  for all firing sequences  $\gamma \in T_M^*$  such that  $(\mathcal{S}(M) : m \xrightarrow{\gamma})$ . Now consider the following two possible cases:

1. Notary  $N$  is in its final place in this marking  $m$ , i.e.  $m \geq f_N$ , but  $A$  or  $B$  are not in their final markings;
2. Notary  $N$  is not in its final place in marking  $m$ , i.e.  $m \not\geq f_N$ .



**Fig. 3.** An acyclic S-net (a), a single-entry-single-exit loop (b), and the refinement of place  $p$  in (a) with loop (b)

The first case contradicts Lm. 8. Consider the second case. No transitions of  $N$  will be enabled in  $(\mathcal{S}(M), m)$ , and  $N$  is not in its final marking. As  $f_N \neq m|_{P_N}$  and  $\mathcal{S}(N)$  is an S-net, there exist a place  $p \in P_N$  of the notary such that  $m(p) > 0$ .  $p^\bullet$  cannot contain any transition  $t$  with  $C_N(t) = \tau$ , since this transition would be enabled in  $m$ . Due to Lm. 10, all transitions  $t$  from  $p^\bullet$  have the same value for  $C_N(t)$ . Without the loss of generality, we suppose it to be  $A$ . Since  $A \oplus N$  is sound, there is a firing sequence  $\sigma; t$  from marking  $m|_{A \oplus N}$  in  $A \oplus N$  such that  $\sigma \in T_A^*$  and  $t \in T_N$ . This firing sequence is then also a firing sequence of  $M$ , but this contradicts the statement that no transition from  $T_N$  can fire starting from  $m$  in  $M$ .

Therefore,  $A \oplus N \oplus B$  is sound. □

### 4.2 Simple Cyclic Notaries

Acyclic notaries ensure the correctness of the composition of two components if these components communicate correctly with the notary. Often, cyclic behavior between components is needed. For example, in order to agree on some quote, several cycles may be involved. In this section, we extend acyclic nets with single-entry-single-exit (SESE) loops.

A SESE loop consists of an entry place and an exit place, not being the same, and all nodes inside the loop are on a path from entry to exit or vice versa, on a path from exit to entry. Furthermore, we require each place in the loop to have exactly one transition in its preset and one in its postset, except for the entry and exit place. An example of a SESE loop is depicted in Fig. 3(b).

An S-net is a *Simple Cyclic S-net* (SCS-net) if all loops in the net are disjoint, i.e., each place and transition of the net belongs to at most one loop. In simple

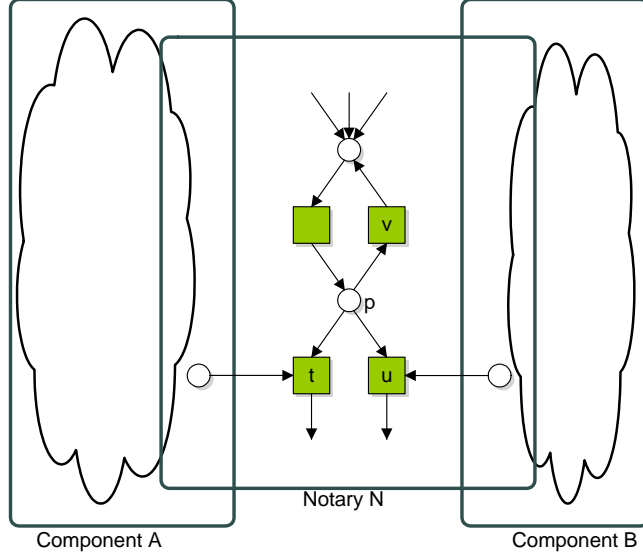


Fig. 4. Controlling conflicts in a simple-cyclic notary

cyclic nets, each loop can be replaced by a place, which results in an acyclic S-net (see Fig. 3).

**Definition 12 (SESE Loop, simple-cyclic S-net).** Let  $(P, T, F)$  be an S-net. A set  $L \subseteq P \cup T$  is called a single-entry-single-exit loop (SESE loop) with entry place  $e \in L \cap P$  and exit place  $o \in L \cap P$  if all nodes  $n \in L$  are on a path from  $e$  to  $o$  or on a path from  $o$  to  $e$ ,  ${}^\bullet e \setminus L \neq \emptyset$ ,  $e^\bullet \subseteq L$ ,  $o^\bullet \setminus L \neq \emptyset$ , and  ${}^\bullet o \subseteq L$ , and for all places  $p \in L \cap P$ , if  $|{}^\bullet p| > 1$  then  $p = e$  and if  $|p^\bullet| > 1$  then  $p = o$ .

Let  $(P, T, F)$  be an S-net. It is called a simple-cyclic S-net (SCS-net) if all loops are SESE loops and pairwise disjoint, i.e., for all loops  $L_1, L_2 \subseteq P \cup T$  if  $L_1 \cap L_2 \neq \emptyset$  then  $L_1 = L_2$ .

Note that in the definition of SCS-nets, we require each node to be in at most one loop. By the definition of the SESE loop, we have that if a node contains multiple elements in its preset or postset, it is either the entry or the exit of the loop. As a consequence, all SESE loops are simple: there is one path from entry to exit and one path from exit to entry. The basis of an SCS-net is an acyclic S-net. Consequently, each loops will be entered at most once.

Whereas in an acyclic notary every conflict is controlled by a single component, this is not the case in the simple-cycled case, as shown in Fig. 4. Choices still need to be controlled by a single component, but silent loops, i.e., no transition in the loop is connected with an interface places, are allowed.

Similarly to the acyclic case, if the skeleton of a notary is a simple-cyclic S-net, soundness of the three parties is assured if the notary composed with each of the organizations individually is sound. Whereas in the acyclic case every



conflict is controlled by a single component, if the notary is cyclic, the moment of control can be postponed. As a consequence, we need to also consider the possibility of live-locks in which the notary is involved.

**Theorem 13.** *Let  $A$  and  $B$  be two disjoint OPNs and let  $N$  be an simple-cyclic notary such that  $A \oplus N$  and  $N \oplus B$  are sound. Then  $A \oplus N \oplus B$  is sound.*

*Proof.* Define  $K = A \oplus N$ ,  $L = N \oplus B$  and  $M = A \oplus N \oplus B$ . Suppose  $M$  is not sound. Then there exists a marking  $m \in \mathcal{R}(\mathcal{S}(M), i_M)$  such that  $f_M \notin \mathcal{R}(\mathcal{S}(M), m)$ .

1. Notary  $N$  is in its final marking, but  $A$  or  $B$  cannot reach its final marking, or tokens are left in the interface places;
2. Notary  $N$  reaches a deadlock different from the final marking;
3. Notary  $N$  reaches a live lock with respect to the final marking;

The first case contradicts with Lm. 8.

Next, consider the second case. Suppose a marking  $m \in \mathcal{R}(\mathcal{S}(M), i_M)$  not being the final marking exists that is a deadlock. Then, a place  $p \in P_N$  exists with  $p \neq f_N$ ,  $m(p) = 1$  and  $\bullet t \not\leq m$  for all transitions  $t \in p^\bullet$ . Let  $t \in p^\bullet$ . Then  $C_N(t) \neq \tau$ . If  $C_N(t) = C_N(u)$  for all  $t, u \in p^\bullet$ , then either  $A \oplus N$  or  $N \oplus B$  would not be sound. Hence, transitions  $t, u \in p^\bullet$  exist such that  $C_N(t) \neq C_N(u)$ . Without loss of generality assume  $C_N(t) = A$  and  $C_N(u) = B$ .

Since  $\mathcal{S}(N)$  is an SCS-net, place  $p$  is either an entry or exit point of a loop, or outside a loop. Suppose place  $p$  is not the exit of a loop. Then in  $K$ , transition  $u$  is enabled in  $m|_{P_{A \oplus N}}$ . Since  $K$  is sound, it must be possible to fire transition  $t$ . Thus, a marking  $m' \in \mathcal{R}(\mathcal{S}(K), i_L)$  exists such that  $(\mathcal{S}(K) : m' \xrightarrow{t})$ . As transition  $u$  is also enabled in  $m'$ ,  $u$  has to be in a loop, since otherwise  $K$  the token from the places  $q \in \bullet t \cap O_A$  would never be consumed. Similarly, transition  $t$  has to be in a loop, since otherwise the token from the places  $q \in \bullet u \cap O_B$  would never be consumed. Hence, a deadlock cannot occur.

Last, consider the case in which  $N$  reaches a live-lock, i.e, it entered a loop  $L$  with entry  $i$  and exit  $o$  such that it cannot leave the loop. Hence,  $C_N(t) \neq \tau$  for all transitions  $t \in o^\bullet \setminus L$ . If  $C_N(t) = C_N(u)$  for all  $t, u \in p^\bullet$ , then either  $A \oplus N$  or  $N \oplus B$  would not be sound. Hence, transitions  $t, u \in p^\bullet$  exist such that  $C_N(t) \neq C_N(u)$ . Without loss of generality assume  $C_N(t) = A$  and  $C_N(u) = B$ . Again due to liveness of  $K$  and  $L$ , this is not possible.

As all cases lead to a contradiction,  $A \oplus N \oplus B$  is sound. □

## 5 Conclusions

We studied in this paper the problem of ensuring correctness of networks of cooperating organizations. By introducing a trusted third party, called a notary, organizations do not need to share their knowledge with the other organizations within the network. The notary needs to ensure that firstly it works correctly with each of the organizations individually, and secondly that all organizations

in the network, including the notary itself, work correctly together. In this paper, we showed for two organizations and a notary that if the notary is an acyclic state machine, or it contains only single-entry-single-exit (SESE) loops, then the notary ensures soundness if it is sound with each of the organizations individually.

In literature, different approaches exist. For example, in the approach of [4], the authors use contracts, implemented as public views. Organizations then need to implement their public views as a private view. If each of the private views agrees on the public view, the network is guaranteed to be correct. In [8], an interactive Petri net is designed, modeling the communication between different organizations.

The disadvantage of these approaches is that each of the organizations need to implement a private view, whereas often organizations already have existing components. In these approaches, the organizations have to re-engineer the existing components, and prove that these re-engineered components adhere to the views defined in the contract using e.g. accordance [12] or contract theory [6]. In the approach described in this paper, organizations can reuse existing components, as the approach requires an organization to cooperate correctly with the notary.

The setting in this paper is comparable with the more general setting of decentralized controllability [15], which is shown to be undecidable [16]. We limited ourselves to two organizations with a notary which is either acyclic or only contains SESE loops. Although these requirements are quite strong, they are needed to ensure soundness. Future work will be to search for more liberal notaries and to extend the results to service trees [3]. As shown in [14], soundness is not compositional, and additional requirements are needed.

## References

1. W.M.P. van der Aalst. Verification of Workflow Nets. In *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 407 – 426. Springer-Verlag, Berlin, 1997.
2. W.M.P. van der Aalst, K.M. van Hee, A.H.M. ter Hofstede, N. Sidorova, H.M.W. Verbeek, M. Voorhoeve, and M.T. Wynn. Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Computing*, pages 1–31, 2010.
3. 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 *Applications and Theory of Petri Nets 2009*, volume 5606 of *Lecture Notes in Computer Science*, pages 283 – 302. Springer-Verlag, Berlin, 2009.
4. W.M.P. van der Aalst, N. Lohmann, P. Massuthe, C. Stahl, and K. Wolf. Multi-party Contracts: Agreeing and Implementing Interorganizational Processes. *The Computer Journal*, 53(1):90–106, 2010.
5. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services – Concepts, Architectures and Applications*. Springer-Verlag, Heidelberg, 2004.
6. S.S. Bauer, A. David, R. Hennicker, K.G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Moving from specifications to contracts in component-based design. In *FASE 2012*, pages 43–58, 2012.

7. M. Beisiegel, K. Khand, A. Karmarkar, S. Patil, and M. Rowley. Service Component Architecture Assembly Model Specification Version 1.1, 2010.
8. G. Decker and M. Weske. Local Enforceability in Interaction Petri Nets. In *Business Process Management*, volume 4714 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, Berlin, 2007.
9. K.M. van Hee, N. Sidorova, and M. Voorhoeve. Soundness and Separability of Workflow Nets in the Stepwise Refinement Approach. In *Application and Theory of Petri Nets 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 335 – 354. Springer-Verlag, Berlin, 2003.
10. P. Massuthe, W. Reisig, and K. Schmidt. An Operating Guideline Approach to the SOA. *Annals of Mathematics, Computing & Teleinformatics*, 1(3):35–43, 2005.
11. N. Mehandjiev and P.W.P.J. Grefen, editors. *Dynamic Business Process Formation for Instant Virtual Enterprises*. Springer-Verlag, Berlin, 2010.
12. A.j. Mooij, C. Stahl, and M. Voorhoeve. Relating fair testing and accordance for service replaceability. *J. Log. Algebr. Program.*, 79(3-5):233–244, 2010.
13. W. Reisig. *Petri Nets: An Introduction*, volume 4 of *Monographs in Theoretical Computer Science: An EATCS Series*. Springer-Verlag, Berlin, 1985.
14. J.M.E.M. van der Werf. *Compositional design and verification of component-based information systems*. PhD thesis, Technische Universiteit Eindhoven, 2011.
15. K. Wolf. Does My Service Have Partners? In *Transactions on Petri Nets and Other Models of Concurrency II*, Lecture Notes in Computer Science, pages 152 – 171. Springer-Verlag, Berlin, 2009.
16. K. Wolf. Decidability Issues for Decentralized Controllability of Open Nets. In *17th German Workshop on Algorithms and Tools for Petri Nets*, volume 643, pages 124–129. CEUR-WS, 2010.

# Modeling and Analyzing Wireless Sensor Networks with VeriSensor

Yann Ben Maissa<sup>1,2</sup>, Fabrice Kordon<sup>2</sup>, Salma Mouline<sup>1</sup>, and Yann  
Thierry-Mieg<sup>2</sup>

<sup>1</sup> LRIT – CNRST URAC29, Université Mohammed V-Agdal  
4, Avenue Ibn Battouta, B.P. 1014 RP, Rabat Maroc,  
mouline@fsr.ac.ma,

<sup>2</sup> LIP6 – CNRS UMR7606, Université P. & M. Curie 4, place Jussieu, 75005 Paris,  
France

Yann.Ben-Maissa@lip6.fr, Fabrice.Kordon@lip6.fr, Yann.Thierry-Mieg@lip6.fr

**Abstract.** A Wireless Sensor Network (WSN), made of distributed autonomous nodes, is designed to monitor physical or environmental conditions. WSNs have many application domains such as environment or health monitoring. Their design must consider energy constraints, concurrency issues, node heterogeneity, while still meeting the quality requirements of life-critical applications. Formal verification helps to obtain WSN reliability, but usually requires a high expertise, which limits its adoption in industry.

This paper presents *VeriSensor*, a domain specific modeling language (DSML) for WSNs offering support for formal verification. *VeriSensor* is designed to be used by WSN experts. It can be automatically translated into a formal specification for model checking. We present the language, its translation, show how they work on a simple case study, and illustrate how several metrics and properties relevant to the domain can be evaluated.

**Keywords:** wireless sensor networks, domain specific modeling languages, model driven engineering, formal verification

## 1 Introduction

**Context** Wireless sensor networks (WSNs) are composed of distributed autonomous nodes, containing programs and sensors to monitor physical or environmental conditions. Each node is a small physical device embedding sensors, a small CPU, a battery, a wireless transceiver and an antenna for communication. WSNs are useful in many contexts, such as environment or health monitoring, thus being a hot topic [14, 8].

The design of WSNs is complex and error-prone due to their numerous constraints:

- *lifetime* is a crucial preoccupation (even more important than quality of service [3]). Overall lifetime of the WSN usually depends on sensor nodes lifetime because nodes have limited battery power.

- *concurrency and asynchrony* lead to important issues such as interleaving of actions and race conditions.
- *heterogeneity*, because WSNs may contain various types of nodes, each having different characteristics (embedded sensors, wireless range, battery capacity, etc.).
- *limited resources*, because nodes have limited CPU and memory capacities.

**Problem** When WSNs are intended to handle critical functions, verification and validation must be performed to reach a significant confidence in such systems [12, 7]. Several propositions in that direction have emerged in recent years.

*Case studies using Formal Verification.* Formal methods have been applied on case studies to verify some relevant properties for WSNs. For instance, [12] uses Real-Time Maude, [11] uses the language IF and the model-checker Kronos, [19] uses UPPAAL.

While these studies show the practical and industrial relevance of performing formal analysis on WSNs, they use ad-hoc modeling of the system by an expert in both WSNs and formal verification. This increases the design and verification costs of WSNs. Moreover, complex verification “tricks” must be elaborated to achieve the verification goals, creating a gap between the formal specifications and the real system.

*Language-based approaches.* Current trends in software engineering show the emergence of model-driven engineering (MDE): a model of the system is expressed using a domain specific modeling language (DSML) providing concepts of the domain. Then, using model transformation technologies, executable code or simulation models can be automatically produced.

Such DSMLs dedicated to WSNs ease their modeling by domain experts. However, they currently do not support formal analysis. VisualSense [4] for instance only allows simulation which is useful during the early design stages, but may not catch rare unexpected events. Baobab [2], Matilda [21], and Medwsa [20] provide code generators that produce executable artifacts to be deployed on the physical system.

Unfortunately, these DSMLs often have a very detailed level of specification (such as wireless signal propagation characteristics), including non-linear parts that can only be simulated in practice. So, if they are adapted for code generation or simulation, they generate a high combinatorial explosion and are thus not suitable for verification.

**Contribution** This paper presents VeriSensor, a DSML for WSNs and its mapping to a formal language for verification and analysis. VeriSensor has the features of an architectural description language (ADL [10]) adapted to a modular description of WSNs.

VeriSensor offers “natural” modeling of a WSN to domain experts by providing high-level concepts that capture the main use cases of such systems – periodic data collection, query-based processing, etc. [22, 9]. VeriSensor can be transformed into a discrete formal model supporting analysis: Instantiable Transition Systems (ITS) [18]. At this stage, VeriSensor is not intended for code generation.

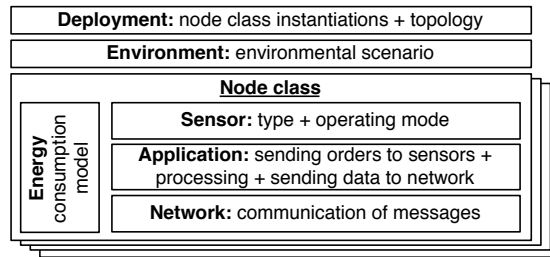


Fig. 1: Structure of a VeriSensor specification

To illustrate its capabilities, we use VeriSensor on a small example. Analysis is performed using our own tools, based on the ITS library.

**Contents** Section 2 gives an overview of VeriSensor. Section 3 presents the language concepts together with the biomedical area network (BAN) case study [13] used as a running example. Section 4 explains the mapping of VeriSensor into ITS. Then, section 5 presents the analysis results we compute on the case study.

## 2 Overview of VeriSensor

A VeriSensor specification is composed of the definition of the nodes themselves, a description of the physical environment in which the nodes evolve, and the deployment of the system (see Fig. 1).

**Description of the nodes** There can be several classes of nodes in a WSN (e.g. in a heterogeneous network), each one having its own characteristics such as:

- its sensors (which physical quantities to be measured and how they are captured),
- its application operating mode (periodic data collection, query-based processing, etc.) and the way it manipulates data,
- its interface with the network (wireless range, routing, etc.),
- the energy consumption model.

These characteristics are described through four orthogonal *dimensions*: sensor, application, network and energy. Dimensions describe *independent* aspects of the system.

Several node classes can share common dimensions and a node class can be instantiated several times when several nodes have the same characteristics.

**Environment Model** It defines physical quantities as a function of space  $(x, y, z)$  and time  $(t)$ . Thus, the designer may describe a particular scenario in which the WSN evolves. These scenarios are used to test qualitative properties of models on given problem instances. A given environment represents a particular situation in which a given behavior of the WSN is expected.

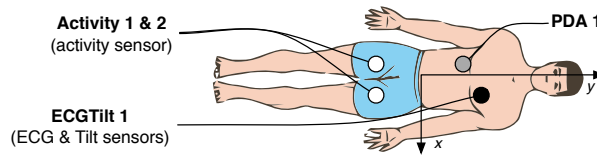


Fig. 2: The Body Area Network (BAN)

**Deployment Model** It defines how instances of node classes are spread in the physical environment and may change position over time<sup>3</sup>. Engineers use this model to define the topology of the system (number of instances per class and their coordinates) as well as the logical routing of messages among the nodes.

**Structuration in VeriSensor** The various dimensions are defined separately in VeriSensor to support *modularity* and *reusability* of WSNs components. The deployment model of a system is the entry point of a VeriSensor model. It defines the Environment model and instantiates all nodes from the definition of their classes.

### 3 Modeling with VeriSensor

This section presents VeriSensor through the specification of a case study.

#### 3.1 The Body Area Network (BAN)

Our case study takes place in the context of home medical monitoring of patients who need constant care but can stay out of hospitals. Home medical monitoring allows to avoid hospitalization, which is as good for medical staff as for their patients.

The Body Area Network [13] is part of a wireless health monitoring system. It is composed of (see Fig. 2): *i*) a set of sensor nodes capable of sensing, processing and communicating vital signs to a personal server ; *ii*) a Portable Digital Assistant (PDA) that forwards patient data to a medical center through internet (3G or WIFI).

The BAN monitors the vital signs of patients recovering from a heart attack. It checks whether a patient is exercising regularly as recommended by the doctors. WSNs, due to their small size and wireless nature, reduce system intrusiveness in patient's lives.

As shown in Fig. 2, two redundant activity nodes detect periods of physical exercise (when the body activity level is above  $8 \text{ Watts.kg}^{-1}$ ) while a third one periodically collects both heartbeat with an electrocardiogram (ECG) sensor and the tilt (i.e. upper body orientation) in terms of the absolute angle relative to a vertical position.

The system designer (i.e. the end-user of VeriSensor) wants to assess some critical aspects of his system. To do so, he needs to evaluate properties such as:

<sup>3</sup> We do not yet support mobility in our approach but this is a natural extension that is semantically possible in VeriSensor.

```

System BAN {environment => HumanBody;
  ECGTilt    => ecgtilt1 (x=0.1, y=0.4, nextHop = pda1);
  Activity   => activity1 (x=-0.3, y=0.1, nextHop = pda1),
             activity2 (x=-0.3, y=-0.1, nextHop = pda1);
  PDA        => pda1 (x=-0.1, y=0.3, nextHop = null);}

```

Fig. 3: Deployment of the BAN system

- $p_1$  evaluate which node limits the system lifetime according to a given scenario,
- $p_2$  identify scenarios leading to undesirable situations that should be avoided,
- $p_3$  check that the system behaves as expected by “replaying” existing situations identified by doctors,
- $p_4$  compare alternative hardware solutions according to their characteristics (energy consumption of sensors, processing duration, etc.),

### 3.2 Modeling the BAN in VeriSensor

This section illustrates the VeriSensor syntax and structure through the modeling of the BAN case study. Here, we follow a “path” going from the more general aspects of the system (its elements) up to implementation of some nodes and the description of its environment.

**The Deployment Model** Fig. 3 shows the deployment parameters of the BAN system. Each node instance is parameterized by its position (shown on fig. 3) and next hop. For instance, the only node of class `ECGTilt` is located at position  $\langle 0.1, 0.4, 0 \rangle$  (when a position parameter is unspecified, its value is 0) and routes messages to the `pda1` instance. Distances are expressed in meters.

**The Node Class Model** A node class specifies the physical characteristics of a node to be instantiated. It relates the data dispatched in the four dimensions: sensing, application, network, and energy (Fig. 4, left).

In the case study, we only consider static routing based on the `nextHop` parameter defined in the deployment model. The `XNetwork` dimension reflects this choice and is used by all nodes of the BAN as specified in Fig. 3.

Fig. 4 (right) describes the sensors of `ECGTilt`. In our study, this node class samples the upper body orientation (Tilt) and the heartbeat (Heartbeat). Sensors are described through their main technical characteristics: the measured

```

NodeClass ECGTilt {
  sensing => ECGTsensing;
  application => ECGTApplication;
  network => XNetwork;
  energy => ECGTEnergy;}
include types.def;
sensing ECGTsensing {
  sensor ECGSensor (
    physical_quantity = Heartbeat,
    startup_time = 1,
    capture_time = 1);}
sensor TiltSensor (
  physical_quantity = Tilt,
  startup_time = 0,
  capture_time = 1);

```

Fig. 4: ECGTilt, the node class (left) and its sensing dimension (right)



```

application (collectNode) ECGTApplication {
  physical_quantity HeartBeat (sensing_period = 13,
    processing_period = 13, sending_period = 13);
  physical_quantity Tilt (sensing_period = 4,
    processing_period = 8, sending_period = 16);}
energy ECGTEnergy {
  initial = 1000;
  reception = 4;
  emission = 5;
  processing = 3;
  sensing = 2 ;}

```

Fig. 5: ECGTilt, its application (left) and energy dimensions (right)

physical quantity startup time (i.e. the time for the sensor to be operational after being turned on), and its capture time (i.e. the time for the sensor to sense the value). For instance, ECGSensor measures the heartbeat and starts-up in 1 time unit. Physical quantities are defined in a dedicated model (see Fig. 6, left) contained in the file `types.def`.

Each physical quantity  $q$  is connected to the environment which must provide a function returning the values of  $q$  at the coordinates of the node instance and for the current time (at any time). So, in Fig. 4 (right), when ECGSensor samples a value, it invokes the corresponding function returning the `Heartbeat` from the Environment dimension. There is one such function per physical quantity of the system.

Fig. 5 (left) shows the application dimension of ECGTilt. In VeriSensor, nodes have several typical behaviors provided as a parameter of the definition. Here, ECGTilt behaves in “collect” mode (keyword `collectNode` in the figure): this periodic data collection is parameterized by a sensing period (i.e. the time between two samples), a processing period (i.e. the time between two processing of the sampled data), and a sending period (i.e. time between two emissions of the processed data). For instance, `Tilt` is sampled every 4 time units, processed every 8 time units, and sent every 16 time units.

Fig. 5 (right) shows the energy dimension that describes the initial power stored in the battery (`initial`) and defines the consumption of dedicated actions: reception (message reception), emission (message sending), processing (processing of sampled data), and sensing (sample acquisition).

**The Physical quantities Model** This model describes physical quantities as discrete ranges of values (see Fig. 6 left). The underlying semantics is the one of discrete event systems, so, continuous values must be mapped to an integer

```

type HeartBeat is 0..200; include types.def;
type Tilt is 0..180;      environment cyclic HumanBody {context {}}
type Activity is 0..15;   body {
  cycle 60; // cyclic behavior (in time units)
  HeartBeat function HeartBeatFunc (x,y,z,t) {
    if (0 <= t and t < 10) then return(95);
    elseif (10 <= t and t < 14) then return(40);
    else return (75);}
  Tilt function TiltFunc (x,y,z,t) {...}
  Activity function ActivityFunc (x,y,z,t) {...}}

```

Fig. 6: Physical quantities definition (left) and an example of Environment Model (right)

range. This mapping is user-defined; the designer must evaluate the trade-off between precision of quantities units and the analysis complexity.

**The Environment Model** It defines the evolution of each physical quantity in the model (see Fig. 6, right ) in a given scenario. Thus, it provides a function that is bound to each sensor sampling the corresponding physical quantity (e.g `HeartBeatFunc` is bound to the `ECGSensor` defined in Fig. 4).

In our example of environment model, values of `HeartBeat` depend on time only. Since there are two `Activity` sensors, `ActivityFunc` can use the node coordinates to provide different values to each sensor in nodes `activity1` and `activity2`. Here, values in `HeartBeatFunc` are deduced from the thresholds of the application: for instance, any value above 95 is considered a situation where the patient exercises; then, these values are abstracted to the threshold constant. Our simplified example illustrates a common situation where WSNs designers generate such a function from observed traces of the system activity. No parsing of traces is provided since these are too “system dependent”. To extend an existing trace, a cyclic behavior may be specified.

For some properties of interest such as worst case scenarios, instead of using the user supplied environment we can use a “free” unconstrained environment, which might return any value at any time. The *clear separation* between the input conditions (environment) and the system specification is important in the analysis phase described below.

## 4 Formal Analysis of VeriSensor specifications

Formal analysis by model checking of a system is a powerful technique that allows to capture subtle defects as well as to reason about worst case scenarios and occurrence of rare events by exhaustively analyzing all possible behaviors. However it is limited in the scale of the systems it can analyze due to the combinatorial state space explosion characteristic of concurrent asynchronous systems. To partly overcome this problem, techniques and tools have emerged such as SAT solvers [6] or shared decision diagrams [5].

However formal models are usually limited to a low level specification of the system transition relation, that describes the state space generator.

Since WSNs are highly time driven and complex, we need a tool supporting a large amount of concurrency, some notion of time constraints and able to tackle combinatorial explosion. To achieve this, we rely on our own preexisting tool: Instantiable Transition Systems (ITS) [18] and their recent extension that supports discrete time [15]. The ITS model checker is general and efficient: it relies on a powerful decision diagram library to cope with the complexity of large systems. ITS also provide a way to define a structured and hierarchical specification of a system and a notion of behavior instantiation. They were previously experimented to analyze UML activity diagrams through a model transformation approach [17] similar to the one outlined here.

### 4.1 The Underlying Formal Model

This section first gives an informal overview of the underlying formal notations for VeriSensor. There are two formalisms involved: labeled time Petri nets to describe elementary behavior and ITS to structure the specification. We only provide here an intuitive definition (see [18, 15] for a formal presentation).

**Instantiable Transition Systems** ITS allow hierarchical and compositional modeling, through a notion of type and instance and an application of the composite design pattern at a behavioral level. A type has an interface, defined as a set of action labels, and some definition of its internal behavior. Similarly to component oriented models, an ITS *composite* is a type that contains *instances* of ITS *types*.

Figure 7 shows a simple example of a composite ITS type. The system offers one interface, **begin**, that is synchronized with the **start** interfaces of the nested components (Client and Server). This system contains a local transition ( $\epsilon$ ) that only has a local effect and is built on the synchronization of **send** and **get** interfaces. Client and Server are elementary components that contain an automaton where local transitions are labeled by  $\epsilon$  too. In practice, we use labeled time Petri nets to define elementary ITS types.

**Labeled Time Petri Nets** In a Petri net, places (circles) contain tokens representing resources that are consumed by transitions (rectangles) when they fire, producing new tokens. A state of a Petri net assigns to each place of the net an integer representing the number of tokens it contains. In a given state, a transition is enabled if all its input places (connected by an arc from place to transition) contain enough tokens. Each arc may be labeled by an integer that indicates how many tokens are consumed or produced (the value 1 is assumed if there is no annotation). When firing, a transition produces tokens in the places connected by outgoing arcs.

Time Petri nets (TPNs) add a notion of clock to each transition, constrained by an earliest and latest firing time noted  $[\alpha, \beta]$ . As soon as a transition is enabled, the associated clock starts. This transition cannot fire before  $\alpha$  time units have elapsed and must occur if the transition’s clock reaches  $\beta$ . Hence a transition with  $[0, \infty]$  can occur at any date if it is enabled, like normal Petri nets. This is assumed to be the default values and is not explicitly shown in the figures.

The time model is discrete: a special transition *elapse* represents the evolution of time by one unit. All clocks evolve simultaneously when *elapse* is fired.

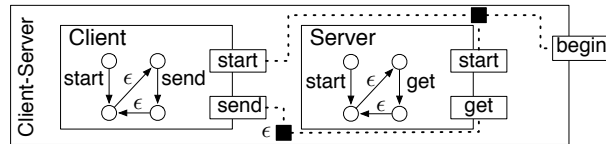


Fig. 7: Small example of composite-ITS

Labels add a notion of interface to Petri nets, where some transitions (represented with thick borders) are called *public* and allow communication with the outside world. These transitions define the ITS interface. *Private* transitions can occur locally, independently from any situation outside the net, and typically represent an autonomous control flow.

#### 4.2 Mapping VeriSensor to a Formal Specification

The mapping of VeriSensor into formal specifications relies on patterns associated to its syntactic elements. It is also based on a set of automatically computed abstractions that help containing the combinatorial explosion due to large datatypes.

**The Transformation process** To automatically transform the specification into a formal model we define a set of “generic ITS”, modeling behavioral patterns that correspond to the VeriSensor execution semantics.

Thus, the transformation process takes parameters in a VeriSensor specification to customize such patterns. Each dimension has its own generic pattern that is hierarchically defined, thus taking benefits from the ITS mechanisms. The final model is obtained by assembling and instantiating these patterns according to the deployment model.

Figure 8 shows two examples of generic ITS. The first one (Fig. 8a) represents the environment as seen by a given node. To obtain this behavior, the environment function  $q(x, y, z, t)$  is projected over the coordinates of the node, yielding a function  $q(t)$  of time only that is specific to the considered node sensor. This function is finally discretized, and encoded as a series of plateau values that have a certain duration  $d_i$ . Each public transition is labeled by a possible value of the physical quantity. The time bound on local transitions ( $\epsilon$ ) represents the evolution of  $q(t)$  as time progresses. The last transition  $\epsilon_n$  can be added to represent a cyclic environment. This ITS is parameterized by  $n$ , the number of values sent in the cycle, and by  $d_i$  for  $i \in [1..n]$ , the duration for sending these values. Its ITS interface is the set of possible values  $\text{sendV}_i\text{ToSens}$  of the physical quantity.

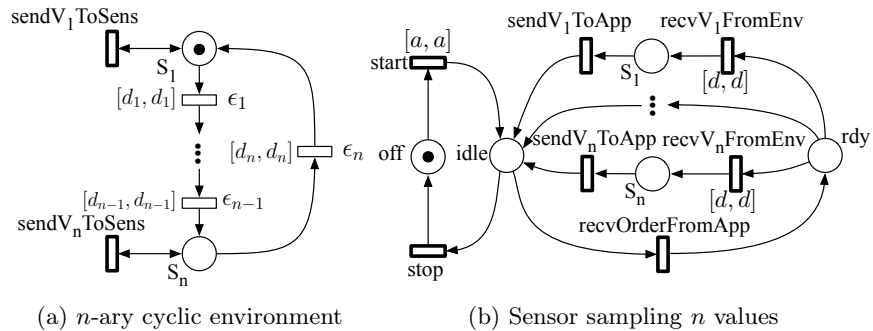


Fig. 8: Two generic ITS, interfaces transitions are outlined in bold

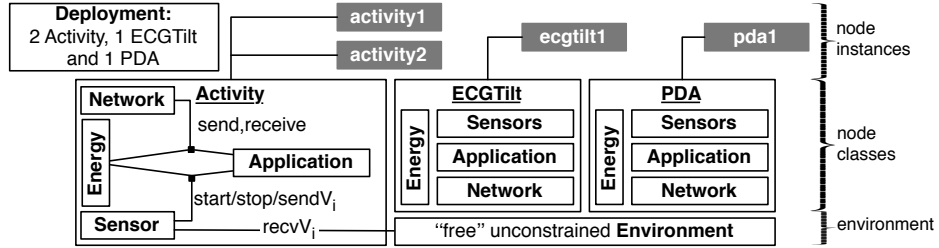


Fig. 9: Structure of the BAN specification according to the deployment of Fig. 3

Fig. 8b represents the behavior of a sensor (as for the BAN system in Fig. 4 right) and is parameterized by  $n$ , the number of potential values in the physical quantity,  $a$ , the startup time, and  $d$ , the capture time. Its ITS interface is composed of the control commands (`start`, `stop`, `recvViFromEnv`, `sendViToApp`).

Although the model size grows with the number of potential values, we control this combinatorial explosion by reducing the domains of physical quantities to the minimum set of representative values that impact the system control flow (see paragraph *Abstraction* below). Moreover, our ITS tool only encapsulates on P/T nets.

The transmission of a value  $V_k$  from the environment to the sensor is represented by a synchronization between `sendVkToSens` and `recvVkFromEnv`. The transition `sendViToApp` transmits sampled values back to the application dimension. Because these definitions of the sensor and the environment are clearly separated we can easily associate the specification to any arbitrary environment instead of a fixed scenario. This is done in the deployment model.

Similarly, each dimension has its own parameterized pattern. Some dimension, such as the application dimension of a node class has one pattern per operating mode (data collection, query processing, etc.).

The Energy dimension is modeled by a one-place Petri net. This place's initial marking depends on the initial energy of the node. Transitions (capture, process, send, receive, etc.) consume the number of tokens corresponding to the energy cost of the associated operation. Since operations in a node are synchronized to the energy dimension, the lack of tokens in the energy dimension stops the corresponding node. When all nodes are out of energy, the system cannot execute anymore and reaches a deadlock.

The full node is then defined as a composite ITS that assembles the projection of the environment with the various ITS corresponding to each dimension (sensors, application, network, energy). This composite ITS has an interface allowing transmission of network messages to other nodes. The nodes are then finally instantiated and connected according to the logical network topology.

Figure 9 illustrates the overall elaboration of the final formal specification for the BAN case study based on the VeriSensor architecture. The assembling of a node class is illustrated for the activity node class. We show how the dimensions

interfaces are synchronized one to another. For example, the public transition `start` is a synchronization between the application, energy and sensor dimensions. This is similar with `recvVi` between the sensor and environment dimensions. Let us remind that transitions like `recvVi` are instantiated as many times as there are relevant values in the parameters to be exchanged.

Then, each node class is instantiated according to the deployment model. In Fig. 9, there are two activity nodes, one `ECGTilt` and one `PDA`. Context variables of each node, describing the node characteristics and its coordinates are initialized according to the deployment model.

**Abstractions** The final assembling, as described, generates complex models since each potential value of a physical quantity  $q$  makes the overall model larger. To avoid this, a structural analysis of the `VeriSensor` specification allows to automatically abstract the domains of physical quantities to the minimum set of representative values that impact the system control flow. Such techniques are derived from automatic symmetry detection [16] or symbolic trajectory evaluation [1]. The complexity of these techniques is low, since it relies on the size of the specification instead of the size of the state space.

Deriving such abstractions automatically is important because: *i*) they are then correct by construction *ii*) using abstractions does not imply any end-user knowledge of the underlying techniques. For instance, activity nodes only detect whether the patient is exercising (i.e. `activity > 8`, see subsection 3.1) or not, so the domain of the physical quantity `activity` (i.e. 0 to 15, see Fig. 6 left) is automatically reduced to 2 values: 0 for no physical exercise, 1 for physical exercise.

**About the Final Model** The resulting model for the BAN case study is composed of 17 ITS-types of which 13 are elementary. The enclosed Petri nets contain 100 places and 81 transitions of which 43 are time constrained. Thus, each state is a vector of 143 variables (places marking + transition clocks). Explicit storage with no optimization of such a state would need 143 integer values, and thus 1.12 Kbyte with a 64 bit representation.

## 5 Analyzing the Case study

This section discusses the analysis we performed on our case study. The ITS representation was generated according to the rules defined in the previous section (a tool is being implemented). From the ITS model this procedure produces, we evaluated the properties identified in section 3.1. All experimentations were done with our own tool based on our ITS library [15].

Prior to this, we discuss the efficiency of this translation with regards to the analysis scalability (based on the parameters values). All experiments were run on a Xeon 64 bits at 2.6 GHz processor.

**Analysis Scalability** The model of the BAN case study is associated with a “free” unconstrained environment providing all possibles situations from the environment point of view. Thus leading to the analysis of possible situations in

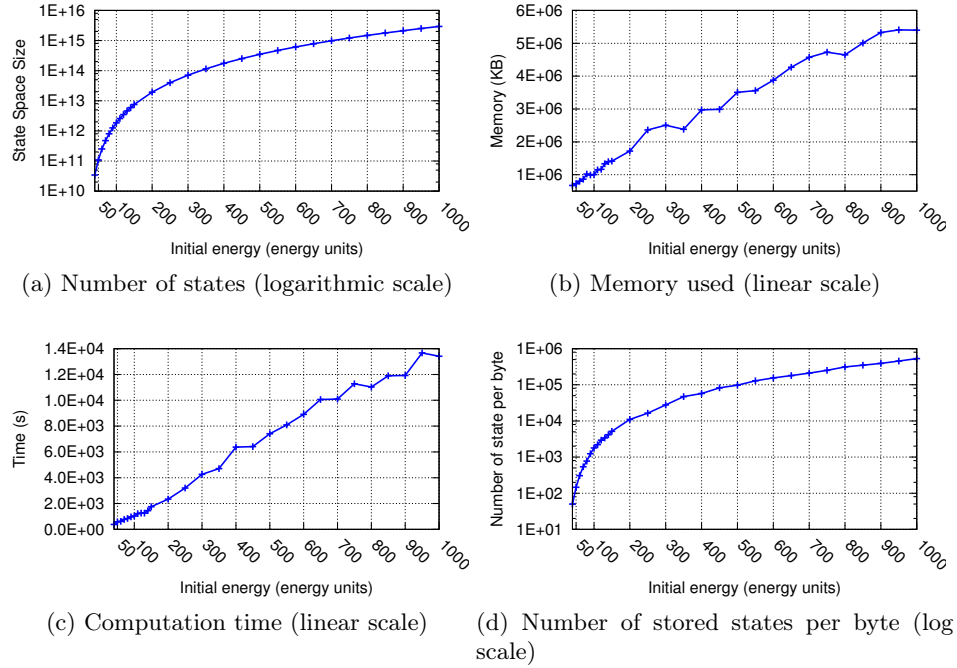


Fig. 10: Evolution of the state space complexity when the initial energy allocated for each node increases (activity sensor period=18 time units for node activity1 and 9 time units for activity2, ECG sensor period=10 time units, Tilt sensor period=29 time units, message emission for every node=5 energy units, message reception for every node=4 energy units, sensing cost for every sensor = 2 energy units, processing sampled data = 3 energy units for every nodes)

the system. Figure 10 shows the evolution of the corresponding state space, its computation time and the memory required to build it, according to the initial energy allowed to the system. In these scenarios, a time unit lasts 1 minute and an energy unit is 50 microjoules. Such interpretation is decided by the designer of the WSN.

As seen in Fig. 10a, the state space grows exponentially, the end of the curve tending to a line in a logarithmic scale. Its representation in memory, as well as the computation time, evolves in a much more favorable way, thus validating the choice of ITS, based on decision diagrams, that already proved its efficiency for such systems [18].

Figures 10b and 10c show the evolution of memory and time required for state space construction according to the initial energy allocated to each node. As shown, we can scale this energy up to 1000 units and still have a reasonable CPU and memory consumption (5.4 GBytes and 3.7 hours). From an industrial point of view, it becomes feasible to process larger values on current high-performance servers.

Considering the memory required to store a state and the size of the state space, our translation into ITS, even if it is yet at a prototype stage, shows encouraging results (up to  $5.3 \times 10^5$  states per byte as shown in Fig. 10d). Moreover, no particular optimization has been done besides the abstraction automatically computed during the translation, thus avoiding the need for expertise in the underlying formal tools.

This experiment on the BAN shows a good scalability potential for the overall approach. In particular, it shows the verification complexity of reachability properties (e.g.  $p_2$  in section 3.1) that are a reliable way to detect “rare events”, difficult to track using classical simulation-based techniques. However, if a Yes/No answer for a reachability property is provided within a reasonable time, we measured that computation of a counterexample takes significantly more time and memory.

**Information about the System Lifetime ( $p_1$ )** Exhibiting the energy consumption of the WSN in the worst case scenario allows the end-user to evaluate a lower bound of the system lifetime. Figure 11a shows the worst case lifetime evolution of the BAN nodes.

To do so, we associate the BAN model with an *unconstrained environment* allowing any action. We thus compute a superset of all the possible behaviors from which we can obtain a worst case scenario. In this model, we search for  $S_{end}$ , the set of states where at least one node cannot communicate anymore (its energy is below a constant  $Min$ , the minimum energy to send a message). Then, our tool computes the shortest path (i.e. shortest transition sequence) leading from the initial state to a state in  $S_{end}$ . To get the corresponding lifetime, we count the occurrences of the *elapse* transition (that let time elapse for 1 time unit). This is the minimal time from the initial state to a state where an observed node cannot communicate anymore.

The objective is not to provide quantitative information since the initial number of energy units allocated to nodes is not sufficient (Fig. 11a shows a system duration in hours, while, at least, weeks would be needed). However, a designer can get an idea of the most critical component (i.e. the one that fails

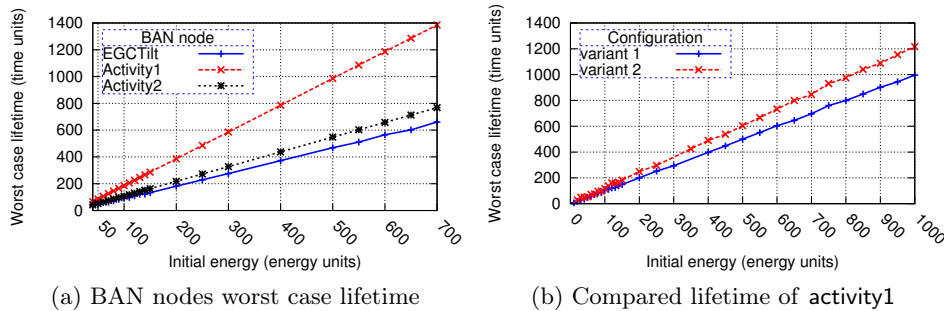


Fig. 11: Lifetime analysis on the BAN case study



first) according to various scenarios. This result is complementary of simulation that can tackle longer duration but not in an exhaustive way.

Let us note that, in Fig. 11a, ECGTilt and Activity1 are the ones to lack energy first. Typically the difference between the lifetime of Activity1 and Activity2 can be analyzed and several parameters can be studied to overcome this situation. Later in this section, we show how two alternative designs for Activity1 can be explored.

**Reachability Properties ( $p_2$ )** A typical and interesting reachability property deals with unexpected deadlocks in the system (expected ones being those where nodes have no more energy). This can reveal real deadlocks in the system, or allow the identification of crucial nodes whose activity is required to keep the system working. Such a situation can be detected using the following reachability formula, computed with no additional cost with respect to state space generation:

$$dead \wedge \bigwedge_{i \in Nodes} (energy(i) > Min_i) \quad (1)$$

Where  $Min_i$  corresponds to the minimum energy required by node  $i$  to send a message and  $dead$  is the boolean meaning that the current state of the state space has no successor. On the BAN case study, this property is verified. It was computed with the unconstrained environment and with a configuration providing up to 500 energy units (it took 1 hour 38 minutes and 2.8 Gbytes).

**Checking Behavior for Existing Situations ( $p_3$ )** Such properties usually require causal formulas expressed by means of temporal logic.

For the BAN system, a typical property is to ensure that the system generates neither a false negative (i.e. a heart attack is not detected) nor a false positive (i.e. a heart attack is detected by mistake in the system). To get this equivalence relation, we use the CTL formula 2 to detect the presence of a false negative and the CTL formula 3 to detect the presence of a false positive.

$$AG(occurs_{heart\ attack} \implies AF(detected_{heart\ attack})) \quad (2)$$

$$AG(\neg occurs_{heart\ attack} \implies AF(\neg detected_{heart\ attack})) \quad (3)$$

In this formula the  $AG$  and  $AF$  operators respectively mean “in all cases” and “in all futures”.  $occurs_e$  is either true or false for a given environment  $e$ . In the BAN case study, a given environment corresponds to a patient behavior which is annotated by the doctors as being sick or healthy.  $detected_e$  is a state property. In our case ( $e = heart\ attack$ ) it involves the PDA and corresponds to the detection of low activity (gathered from the activity sensors) and bradycardia detected by ECGTilt.

On the BAN case study, this property is verified (this was computed up to the system with 500 energy units). This was tested for several environments representing different patients. Such a computation is less complex in time and memory than the worst case lifetime analysis since the system is more constrained. Formulas were computed for 500 initial energy units. Formula 2 was

Parameter	value in <i>config</i> <sub>1</sub>	value in <i>config</i> <sub>2</sub>
sensing frequency	11 TU	20 TU
acquisition time	3 TU	1 TU
acquisition energy	3 EU	4 EU
processing time	1 TU	2 TU
processing energy	4 EU	6 EU
emission time	2 TU	3 TU
emission energy	6 EU	8 EU
reception time	2 TU	3 TU
reception energy	5 EU	7 EU

Fig. 12: Data for the two studied variants in time units (TU) or energy units (EU)

computed in 1 hour 45 minutes and 2.8 Gbyte memory. Formula 3 was computed in 1 hour 28 minutes and 2.7 Gbyte memory.

**Comparing Alternative Solutions** ( $p_4$ ) The choice of a given component may have an impact on a WSN lifetime or on some important characteristics of the system. VeriSensor can be useful to compare two possible solutions. To do so, the designer may either change the characteristics of the nodes to be replaced (if only those change) or replace the node by an instance of another node class.

For the BAN case study, we want to evaluate the impact of two configurations on the system lifetime (e.g. when at least one node cannot communicate anymore). These configurations differ with the characteristics of the activity nodes. The first configuration (*config*<sub>1</sub>) embeds a node that samples often but performs light computation. The second one (*config*<sub>2</sub>) uses a node that performs less samples but more computations.

To evaluate these configurations, we provide two variations of the activity node specification, following the information displayed in Fig. 12. Then, the obtained specification is linked to the “free” unconstrained environment used to evaluate the worst case lifetime of the system. This work leads to the results displayed in Fig. 11b.

## 6 Conclusion

This paper presented VeriSensor, a domain specific modeling language for wireless sensor networks (WSNs), designed to be used by WSNs experts and offering support for modeling and formal verification. The objective is to evaluate both quantitative results (e.g. estimation of the system’s lifetime or average consumption per time unit) as well as qualitative results (e.g. detection of unexpected situations to be avoided).

VeriSensor enables the modeling of a WSN by providing high-level concepts that support the main use cases of WSNs. Thus, specifying WSNs consists in defining the node characteristics, how nodes are deployed and the physical environment in which the system evolves. The physical environment may model all possible situations (the “free” unconstrained environment), thus leading to

the evaluation of the WSN in the worst possible condition. It may also model a dedicated scenario for which the WSN behavior has to be verified.

Instantiable Transition Systems (ITS) and time Petri nets are the underlying formal techniques used for verification. They show encouraging scalability capabilities, thus enabling the analysis of reasonable systems with significant parameters.

The main advantage of the overall approach is to make formal specification and verification more accessible to the end-users (i.e. the designers of WSNs).

Even if we focus on the verification aspects, our approach does not exclude simulation. In fact, since VeriSensor has a formal semantic, it is executable and thus, can be simulated. Then, the environment dimension still allows to select one situation where the system has to be plugged in.

## References

1. S. Adams, M. Björk, T. F. Melham, and C.-J. H. Seger. Automatic abstraction in symbolic trajectory evaluation. In *Formal Methods in Computer-Aided Design*, pages 127–135. IEEE Computer Society, 2007.
2. B. Akbal-Delibas, P. Boonma, and J. Suzuki. Extensible and precise modeling for wireless sensor networks. In *UNISCON*, pages 551–562, 2009.
3. I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *Communications Magazine, IEEE*, 40(8):102–114, Aug. 2002.
4. P. Baldwin, S. Kohli, E. A. Lee, X. Liu, Y. Zhao, C. H. Brooks, N. V. Krishnan, S. Neuendorffer, C. Zhong, and R. Zhou. Visualsense: Visual modeling for wireless and sensor network systems. Technical report, U.C. Berkeley, 2005.
5. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. Hwang. Symbolic model checking:  $10^{20}$  states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33. IEEE Computer Society Press, 1990.
6. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In E. Brinksma and K. Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 241–268. Springer Berlin / Heidelberg, 2002.
7. S. C. Ergen, M. Ergen, and T. J. Koo. Lifetime analysis of a sensor network with hybrid automata modelling. In *WSNA*, pages 98–104, 2002.
8. O. Gnawali and M. Welsh. Sensor networks architectures and protocols. In *Emerging Wireless Technologies and the Future Mobile Internet*, pages 125–153. Cambridge University Press, 2011.
9. A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *1st ACM Int. workshop on Wireless sensor networks and applications (WSNA)*, pages 88–97. ACM, 2002.
10. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26:70–93, January 2000.
11. L. Mounier, L. Samper, and W. Znaidi. Worst-case lifetime computation of a wireless sensor network by model-checking. In *4th ACM workshop on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks (PE-WASUN)*, pages 1–8. ACM, 2007.

12. P. C. Ölveczky and S. Thorvaldsen. Formal modeling and analysis of the ogdc wireless sensor network algorithm in real-time maude. In *9th Int. conf. on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, pages 122–140. Springer, 2007.
13. C. Otto, A. Milenković, C. Sanders, and E. Jovanov. System architecture of a wireless body area sensor network for ubiquitous health monitoring. *J. Mob. Multimed.*, 1:307–326, January 2005.
14. K. Sohraby, D. Minoli, and T. Znati. *Wireless sensor networks: technology, protocols and applications*. Wiley Interscience, 2007.
15. Y. Thierry-Mieg, B. Bérard, F. Kordon, D. Lime, and O. H. Roux. Compositional Analysis of Discrete Time Petri nets. In *1st workshop on Petri Nets Compositions (CompoNet 2011)*, volume 726, pages 17–31, Newcastle, UK, June 2011. CEUR.
16. Y. Thierry-Mieg, C. Dutheliet, and I. Mounier. Automatic symmetry detection in well-formed nets. In *Proc. of ICATPN 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 82–101. Springer Verlag, June 2003.
17. Y. Thierry-Mieg and L.-M. Hillah. UML behavioral consistency checking using Instantiable Petri nets. *ISSE*, 4(3):293–300, 2008.
18. Y. Thierry-Mieg, D. Poitrenaud, A. Hamez, and F. Kordon. Hierarchical Set Decision Diagrams and Regular Models. In *15th Int. conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 of *LNCS*, pages 1–15. Springer, March 2009.
19. S. Tschirner, L. Xuedong, and W. Yi. Model-based validation of QoS properties of biomedical sensor networks. In *8th ACM Int. conf. on Embedded software (EMSOFT)*, pages 69–78. ACM, 2008.
20. C. Vicente-Chicote, F. Losilla, B. Álvarez, A. Iborra, and P. Sánchez. Applying mde to the development of flexible and reusable wireless sensor networks. *Int. J. Cooperative Inf. Syst.*, 16(3/4):393–412, 2007.
21. H. Wada, P. Boonma, J. Suzuki, and K. Oba. Modeling and executing adaptive sensor network applications with the Matilda UML virtual machine. In *11th IASTED Int. conf. on Software Engineering and Applications (SEA)*, pages 216–225. ACTA Press, 2007.
22. Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31, 2002.

# SMT-based Parameter Synthesis for L/U Automata

Michał Knapik<sup>1,2</sup> and Wojciech Penczek<sup>1,3</sup>

<sup>1</sup> Institute of Computer Science, Polish Academy of Sciences,  
J.K. Ordonia 21, 01-237 Warszawa, Poland

<sup>2</sup> International PhD Project in Intelligent Computing (MPD, FNP)  
mknapik@ipipan.waw.pl

<sup>3</sup> Institute of Informatics, University of Natural Sciences and Humanities,  
3 Maja 54, 08-110 Siedlce, Poland  
penczek@ipipan.waw.pl

**Abstract** We present and evaluate a straightforward method of representing finite runs of a parametric timed automaton (PTA) by means of formulae accepted by satisfiability modulo theories (SMT)-solvers. Our method is applied to the problem of parametric reachability, i.e., the synthesis of a set of the parameter substitutions under which a state satisfying a given property is reachable. While the problem of parametric reachability is not decidable in general, we provide algorithms for under-approximation of the solution to this problem for a certain class of PTA, namely for the lower/upper bound automata.

## 1 Introduction

Model checking of real-time systems (RTS), performed by the analysis of their models is a very important subject of research. This is highly motivated by an increasing demand to verify safety critical systems, i.e., time-dependent systems, failure of which could cause dramatic consequences for both people and hardware. These systems include robotic surgery machines, nuclear reactor control systems, railway signalling, braking systems, air traffic control systems, flight planning systems, rocket range launch safety systems, and many others. Humans already benefit a lot from a variety of real-time systems, being often unaware of this. Parametric model checking [2,10] aims at extending the successful developments of model checking of RTS. In this case, the model contains free variables, called parameters. Such a situation typically arises at the initial stages of a system design, when some of the details might be unknown.

Timed automata (TA) [1] constitute the most popular and applied class of RTS. The introduction of free variables into timed automata leads to parametric timed automata [2]. It was proven in [12] that the emptiness problem: “Is there a parameter valuation for which an automaton has an accepting run” for PTA is

---

W. Penczek is partly supported by the Polish National Science Centre under grant No. DEC-2011/01/B/ ST6/01477.

undecidable. Several tools have been implemented ([3,5,9,12,14,16]), which allow to verify certain properties of PTA (or related phase automata in [9]). All these tools except for [9] have one thing in common: they aim at fully describing the set of parameter substitutions under which the given property holds. Unfortunately, this means that the process of parametric model checking does not need to stop, consuming time and memory resources. These approaches usually employ extensions of classical (non-parametric) model checking methods such as: parametric difference bound matrices [5,12], partition refinement [16], compositional model-checking [9], and CEGAR and CEGAR-inspired methods [3,4,11].

In [17] a theoretical basis for bounded model checking for PTA was introduced. We have presented the counterpart of the region graph, which allows for a synthesis of a part of the set of constraints under which the given existential CTL property holds. The proposed method ensures that the process of verification stops with correct (but usually not complete) results. In the current paper we continue this work in order to ensure its feasibility. To this aim we consider lower bound/upper bound (L/U) automata [12] in which each parameter occurs either as a lower- or as an upper bound in the timing constraints. Despite this limitation, L/U automata are still interesting in practice, as for example they can be used to model the Fischer's Mutual Exclusion Algorithm, the Root Contention Protocol [12] and other well known systems<sup>1</sup>. Hune et al. [12] showed that the emptiness problem for L/U automata with respect to finite runs is decidable, whereas Bozelli and Torre [8] proved that it is also decidable w.r.t. infinite accepting runs. Similarly, the universality problem: "Does an automaton have an accepting run for each parameter valuation" is decidable for L/U automata. The above decidability results do not solve the problem of finding the valuations of the parameters in case the answer to the emptiness problem is positive and the answer to the universality problem is negative. This means that in case an automaton does have an accepting run only for some parameter valuations, it is not known how to compute them. The above observation is the main motivation of our work, which aims at offering a symbolic method for the synthesis of parameter valuations for which an L/U automaton satisfies some reachability property, i.e., it has a finite accepting run.

From the practical point of view the synthesis of all the parameter valuations is usually not needed: an analyst would typically be satisfied with a possibility of obtaining just a part of them. The direct analysis of parametric region graph is not feasible due to its typically large size, therefore the new methods of unwinding of the state space are needed. To this end, in this paper we offer a direct translation from an unwinding of the the concrete model of an L/U automaton to an SMT instance. This allows for synthesizing a subset of the parameter valuations for which an automaton satisfies some reachability property.

The rest of the paper is organized as follows. In the next section we briefly present the theory of parametric timed automata and formulate the task of parametric synthesis. In Section 3 we present how to encode all the finite runs of a

<sup>1</sup> IEEE Computer Society. IEEE Standard for a High Performance Serial Bus. Std 1394-1995, 1996.

given length as an SMT formula, and in Section 4 the algorithm for a state space exploration and a parameter synthesis for L/U automata is presented. Section 5 contains the preliminary evaluation of our method, as applied to two benchmarks: Fischer’s Mutual Exclusion Protocol and a version of Generic Timed Pipeline Paradigm. We conclude with a brief discussion in Section 5.

## 2 Theory of parametric timed automata

In this section we introduce all the main notions and define timed automata, parametric timed automata, and L/U automata.

Parametric timed automata, to be defined later, employ two sets of variables: the set  $X = \{x_1, \dots, x_n\}$  of real time variables, called *clocks*, and the set  $P = \{p_1, \dots, p_m\}$  of integer variables, called *parameters*. Both types of the variables are used in the clock constraints of parametric timed automata.

The clock constraints are built using *linear expressions*, i.e., expressions of the form  $\sum_{i=1}^m t_i \cdot p_i + t_0$ , where  $t_i \in \mathbb{Z}$ . Clocks or differences of clocks compared with linear expressions, formally, the expressions of the form  $x_i \prec e$  or  $x_i - x_j \prec e$ , where  $i \neq j$ ,  $\prec \in \{\leq, <\}$  and  $e$  is a linear expression, are called *simple guards*. The conjunctions of simple guards are called *guards*. By  $G$  we denote the set of all guards. By  $G'$  we mean the subset of  $G$  consisting of the guards built only of the simple guards of type  $x_i \prec e$ , where  $x_i \in X$  and  $\prec \in \{\leq, <\}$ .

The clocks range over the nonnegative reals ( $\mathbb{R}_{\geq 0}$ ) while the parameters range over the naturals ( $\mathbb{N}$ , including 0). The function  $v : P \rightarrow \mathbb{N}$  is called a *parameter valuation* and  $\omega : X \rightarrow \mathbb{R}_{\geq 0}$  is called a *clock valuation*. Sometimes it is convenient to perceive  $v$  and  $\omega$  as points in, respectively,  $\mathbb{N}^m$  and  $\mathbb{R}_{\geq 0}^n$ .

The value obtained by substituting the parameters in a linear expression  $e$  according to the parameter valuation  $v$  is denoted by  $e[v]$ . If  $\omega(x_i) - \omega(x_j) \prec e[v]$  ( $\omega(x_i) \prec e[v]$ ) holds, then we write  $\omega \models_v x_i - x_j \prec e$  (resp.,  $\omega \models_v x_i \prec e$ ). This notion is naturally extended to the guards.

Two operations can be executed on the clocks: incrementation and reset. Let  $\omega$  be a clock valuation and  $\delta \in \mathbb{R}$ , then by  $\omega + \delta$  we denote such a clock valuation that  $(\omega + \delta)(x_i) = \omega(x_i) + \delta$  for all  $1 \leq i \leq n$ . A set of the expressions of the form  $x_i := b_i$ , where  $b_i \in \mathbb{N}$ , and  $1 \leq i \leq n$  is called a *reset*, and the set of all resets is denoted by  $R$ . Let  $\omega$  be a clock valuation and  $r$  be a reset, then by  $\omega[r]$  we denote such a clock valuation that  $\omega[r](x_i) = b_i$  if  $x_i := b_i \in r$ , and  $\omega[r](x_i) = \omega(x_i)$  otherwise. Intuitively, resetting a clock valuation amounts to setting the selected clocks to some fixed values, while leaving the remaining clocks intact. The *initial clock valuation*  $\omega_0$  satisfies  $\omega_0(x_i) = 0$  for all  $x_i \in X$ .

### 2.1 Parametric timed automata

Timed automata [1] are an established formalism for modelling the behavior of real-time systems. The clock constraints are expressed as the restrictions imposed on clocks or differences of clocks. Parametric timed automata [2] are an extension of timed automata, where linear expressions containing parameters are allowed in the clock constraints.

**Definition 1.** A parametric timed automaton is a seven-tuple  $\mathcal{A} = \langle Q, l_0, A, X, P, \rightarrow, I \rangle$ , where:

- $Q$  is a finite set of locations,
- $l_0 \in Q$  is the initial location,
- $A$  is a finite set of actions,
- $X$  and  $P$  are, respectively, finite sets of clocks and parameters,
- $I : Q \rightarrow G'$  is an invariant function,
- $\rightarrow \subseteq Q \times A \times G \times R \times Q$  is a transition relation.

A transition  $(q, a, g, r, q') \in \rightarrow$  is typically denoted by  $q \xrightarrow{a, g, r} q'$ .

Notice that the co-domain of the invariant function is the conjunction of upper bounds on clocks. This assumption is taken from [12] in order to ensure that the set of time delays under which the automaton can stay in a given location is connected and contains 0 (if nonempty) for each parameter valuation.

The *concrete semantics* of a parametric timed automaton under a parameter valuation  $v$  is defined in the form of a labelled transition system.

**Definition 2 (Concrete semantics).** Let  $\mathcal{A} = \langle Q, l_0, A, X, P, \rightarrow, I \rangle$  be a parametric timed automaton and  $v$  be a parameter valuation. The labelled transition system for  $\mathcal{A}$  under  $v$  is defined as the tuple  $\llbracket \mathcal{A} \rrbracket_v = \langle S, s_0, \mathbb{R}_{\geq 0} \cup A, \xrightarrow{d} \rangle$ , where:

- $S = \{(l, \omega) \mid l \in Q, \text{ and } \omega \text{ is a clock valuation such that } \omega \models_v I(l)\}$ ,
- $s_0 = (l_0, \omega_0)$  (we assume that  $\omega_0 \models_v I(l_0)$ ),
- Let  $(l, \omega), (l', \omega') \in S$ . The transition relation  $\xrightarrow{d} \subseteq S \times S$  is defined as follows:
  - if  $d \in \mathbb{R}_{\geq 0}$ , then  $(l, \omega) \xrightarrow{d} (l', \omega')$  iff  $(l = l' \text{ and } \omega' = \omega + d)$ ,
  - if  $d \in A$ , then  $(l, \omega) \xrightarrow{d} (l', \omega')$  iff  $l \xrightarrow{d, g, r} l'$ , for some  $g \in G, r \in R$ , where  $\omega \models_v g$ , and  $\omega' = \omega[r]$ .

The elements of  $S$  are called the concrete states of  $\llbracket \mathcal{A} \rrbracket_v$ .

After substituting the parameters in  $\mathcal{A}$  according to a parameter valuation  $v$  we obtain the timed automaton, denoted by  $\mathcal{A}_v$ . The concrete semantics of  $\mathcal{A}_v$  is usually denoted by  $\llbracket \mathcal{A}_v \rrbracket$  and it is straightforward [12] to observe that  $\llbracket \mathcal{A}_v \rrbracket = \llbracket \mathcal{A} \rrbracket_v$ .

For  $k \in \mathbb{N}$  by a  $k$ -run  $\rho^k$  in  $\llbracket \mathcal{A} \rrbracket_v$  we mean a sequence of states and transitions:  $\rho^k = s_0 \xrightarrow{d_0} s'_0 \xrightarrow{act_1} s_1 \xrightarrow{d_1} s'_1 \xrightarrow{act_2} \dots \xrightarrow{d_{k-1}} s'_{k-1} \xrightarrow{act_k} s_k \xrightarrow{d_k} s'_k$ , where  $d_i \in \mathbb{R}_{\geq 0}$  and  $act_i \in A$  for all  $0 \leq i \leq k$ . By a *run* we mean any  $k$ -run for  $k \in \mathbb{N}$ . We say that  $k$  is the *length* of  $\rho^k$  and  $s'_k$  is  $k$ -reachable in  $\llbracket \mathcal{A} \rrbracket_v$ .

## 2.2 Parametric reachability and synthesis

The original definition of parametric timed automata [2] contains a distinguished subset of the locations, called *final locations*. A run is *accepted* under a given valuation of the parameters if it ends with a final state. The question of the



emptiness of a set of the parameter valuations under which there exists an accepting run was shown in [2] to be undecidable.

Following [12,17] we present the results in the setting typical for model checking, where we distinguish the model and the property to be verified.

**Definition 3.** Let  $\mathcal{A} = \langle Q, l_0, A, X, P, \rightarrow, I \rangle$  be a parametric timed automaton. The state formulae are defined by the following grammar:

$$\phi = l \mid x_i < b \mid x_i - x_j < b \mid \phi \wedge \psi \mid \neg\phi,$$

where  $l \in Q$ ,  $x_i, x_j \in X$ ,  $< \in \{\leq, <\}$  and  $b \in \mathbb{N}$ .

We also refer to a state formula as to a property. Let  $v$  be a parameter valuation,  $(l, \omega) \in \llbracket \mathcal{A} \rrbracket_v$ , and let  $\phi, \psi$  be state formulae. We define the validity of a state formula in a global states, denoted  $(l, \omega) \models \phi$ , inductively as follows:

- $(l, \omega) \models l' \text{ iff } l = l'$ ,
- $(l, \omega) \models x_i < b \text{ iff } \omega \models_v x_i < b$ , and  $(l, \omega) \models x_i - x_j < b \text{ iff } \omega \models_v x_i - x_j < b$ ,
- $(l, \omega) \models \phi \wedge \psi \text{ iff } (l, \omega) \models \phi \text{ and } (l, \omega) \models \psi$ ,
- $(l, \omega) \models \neg\phi \text{ iff not } (l, \omega) \models \phi$ .

Let  $v$  be a parameter valuation and  $\phi$  be a state formula. Let  $k \in \mathbb{N}$ ,  $d_j \in \mathbb{R}_{\geq 0}$  and  $act_j \in A$  for all  $1 \leq j \leq k$ . Let  $\rho^k = s_0 \xrightarrow{d_0} s'_0 \xrightarrow{act_1} s_1 \xrightarrow{d_1} s'_1 \xrightarrow{act_2} \dots \xrightarrow{d_{k-1}} s'_{k-1} \xrightarrow{act_k} s_k \xrightarrow{d_k} s'_k$  be a  $k$ -run in  $\llbracket \mathcal{A} \rrbracket_v$ . If for some  $k \in \mathbb{N}$  there exists a run  $\rho^k$  in  $\llbracket \mathcal{A} \rrbracket_v$  such that  $s'_k \models \phi$ , then we say that a state satisfying  $\phi$  is reachable in  $\mathcal{A}$  under  $v$  and write  $\llbracket \mathcal{A} \rrbracket_v \models EF\phi$ . The  $EF$  modality originates from Computation Tree Logic (CTL), where  $EF\phi$  stands for “there exists a path such that eventually  $\phi$  holds”.

The task of parametric reachability, otherwise called the parameter synthesis problem, is formulated as follows.

Let  $\mathcal{A}$  be parametric timed automaton and let  $\phi$  be a state formula.  
Automatically describe the set  $\Gamma(\mathcal{A}, \phi) = \{v \mid \llbracket \mathcal{A} \rrbracket_v \models EF\phi\}$ .

As mentioned earlier, there is no decision procedure for checking whether  $\Gamma(\mathcal{A}, \phi)$  is empty or contains all the parameter valuations, therefore we can not expect to be able to fully solve the parameter synthesis problem, at least in the general case.

### 2.3 L/U automata

Hune et al. have identified in [12] an important class of parametric timed automata for which the problem of the emptiness of  $\Gamma(\mathcal{A}, \phi)$  is decidable. These are the lower/upper bound automata (L/U automata, for short), where additional constraints on the parameters are used.

In what follows if  $f$  is a function and  $B$  a subset of its domain, then  $f|_B$  stands for the restriction of  $f$  to  $B$ .

**Definition 4.** A lower/upper bound automaton is a parametric timed automaton  $\mathcal{A} = \langle Q, l_0, A, X, P, \rightarrow, I \rangle$  satisfying the following conditions

- $P = L \cup U$ , where  $L = \{\lambda_1, \dots, \lambda_l\}$ ,  $U = \{\mu_1, \dots, \mu_u\}$ , and  $L \cap U = \emptyset$ .
- Each linear expression in the guards or the invariants of  $\mathcal{A}$  can be written in form  $\sum_{i=1}^l l_i \cdot \lambda_i + \sum_{j=1}^u u_j \cdot \mu_j + t_0$ , where  $l_i, u_j, t_0 \in \mathbb{Z}$  and  $l_i \leq 0$ ,  $u_j \geq 0$  for all  $1 \leq i \leq l$  and  $1 \leq j \leq u$ .

The elements of  $L$  are called the lower parameters while the elements of  $U$  are called the upper parameters.

Intuitively, in an L/U automaton the clock constraints can be uniformly relaxed by decreasing the values assigned to the lower parameters and increasing the values assigned to the upper parameters.

Let  $\mathcal{A}$  be an L/U automaton and  $v$  be a parameter valuation. Define  $\lambda = v|_L$  and  $\mu = v|_U$ . If  $v'$  is also a valuation of the parameters,  $\lambda' = v'|_L$ ,  $\mu' = v'|_U$ , and  $\forall \lambda_i \in L \lambda'(\lambda_i) \leq \lambda(\lambda_i)$  and  $\forall \mu_j \in U \mu(\mu_j) \leq \mu'(\mu_j)$ , then we write  $v \leq v'$ .

When it is convenient to define  $v$  in terms of  $\lambda$  and  $\mu$ , we write  $v = (\lambda, \mu)$ .

**Proposition 1 ([12]).** Let  $\mathcal{A}$  be an L/U automaton,  $\phi$  a state formula, and  $v, v'$  be parameter valuations such that  $v \leq v'$ . Then,  $\llbracket \mathcal{A} \rrbracket_v \models EF\phi$  implies  $\llbracket \mathcal{A} \rrbracket_{v'} \models EF\phi$ .

Assume that  $\mathcal{A}$  is an L/U automaton. From the above lemma it follows that the problem of the emptiness of  $\Gamma(\mathcal{A}, \phi)$  can be reduced to the problem of reachability of a state satisfying  $\phi$  in the automaton obtained from  $\mathcal{A}$  by substituting lower parameters with 0 and removing each guard or invariant containing at least one upper parameter. The latter, in terms of [12], is equivalent to substituting  $\infty$  for each upper parameter.

### 3 Translation to SMT

The idea of encoding of system's behavior using the translation to propositional formulae originates from [7]. The techniques for SAT-based verification of various extensions of timed automata have evolved in parallel with these based on difference bound matrices. Usually, it is possible to translate only a part of a model to a logical formula, hence this method is applied for bounded model checking: a technique especially suited for seeking for bugs and unwanted behaviors.

SMT-solvers extend the capabilities of SAT-solvers by allowing for formulae of the first order over several built-in theories as an input. In our considerations, we use SMT-solvers to obtain the satisfiability and example models (valuations of the parameters) for formulae expressed using boolean variables and operators together with real-valued variables, linear arithmetic operators and relations. As we have decided to make the translation as straightforward as possible, in this experimental phase we have chosen to use SMT-lib ver. 2.0 [6] compliant solvers and rich logics allowing for linear arithmetic over real sort (e.g. QF\_LRA).

Let  $\mathcal{A} = \langle Q, l_0, A, X, P, \rightarrow, I \rangle$  be a parametric timed automaton. In this section we show how to encode, for a given  $k \in \mathbb{N}$ , all the  $k$ -runs of  $\mathcal{A}_v$  for all the parameter valuations  $v$ , as the formula acceptable by SMT solvers such as CVC3. We start with the description of the sorts (types), the variables, and the additional predicates used.

### 3.1 Sorts and Predicates

We encode the locations of  $\mathcal{A}$  by means of enumerating them using propositional expressions. For each  $i \in \mathbb{N}$  let  $\mathcal{BV}^i = \{bv_1^i, bv_2^i, \dots, bv_{\lceil \log |Q| \rceil}^i\}$  be a set of propositional variables. Let  $\mathcal{BE}^i$  denote the set of all the propositional formulae over  $\mathcal{BV}^i$ . It is straightforward to notice for each  $i \in \mathbb{N}$  we can define the function  $loc\_enc^i : Q \rightarrow \mathcal{BE}^i$  assigning to each of the locations from  $Q$  the conjunction of the literals (variables or their negations) from  $\mathcal{BV}^i$  such that  $loc\_enc^i(l) \wedge loc\_enc^j(l')$  is false iff  $i \neq j$  or  $l \neq l'$ . Intuitively, for  $l \in Q$  and  $i \in \mathbb{N}$ , the formula  $loc\_enc^i(l)$  can be interpreted as an encoding of the location  $l$  at the  $i$ -th step ( $i \leq k$ ) of the  $k$ -runs, using variables from  $\mathcal{BV}^i$ .

Recall that  $X = \{x_1, x_2, \dots, x_n\}$  is a set of the clocks. For each  $i \in \mathbb{N}$  let  $X^i = \{x_1^i, x_2^i, \dots, x_n^i\}$  be a set of real variables, where  $X^i \cap X^j = \emptyset$  for  $i \neq j$ , and similarly, let  $T = \{t_0, t_1, \dots\}$  be a set of real variables. As previously, the variables from  $X^i$  are used to encode the clock valuations in the  $i$ -th steps of the  $k$ -runs with the help of the variables from  $T$ , which record the time delays between the consecutive actions.

Recall that  $P = \{p_1, p_2, \dots, p_m\}$  is a set of the parameters ranging over  $\mathbb{N}$ . With a slight notational abuse we treat  $P$  as a set of the variables of real sort. In the current version, SMT-lib standard does not allow for typecasts between reals and integers, therefore we need to use the predicate *is\_int* to ensure that the variables from  $P$  hold integer values only (e.g., *is\_int*(7.0) evaluates to true, while *is\_int*(4.3) is false).

Summarizing, when considering the  $k$ -runs, we declare  $Vars^k = \bigcup_{i=1}^k X^i \cup T \cup P$  to be real variables and  $Bvars^k = \bigcup_{i=1}^k \mathcal{BV}^i$  to be boolean variables. Additionally, we define the formula

$$TypeCut^k = \bigwedge_{v \in Vars^k} v \geq 0 \wedge \bigwedge_{p \in P} is\_int(p)$$

which ensures that all used variables range over the appropriate sets.

### 3.2 Encoding the Transitions

In what follows, if  $\eta$  is a formula containing the free variables  $a_1, a_2, \dots, a_n$ , then by  $\eta[a_1 \leftarrow a'_1, a_2 \leftarrow a'_2, \dots, a_n \leftarrow a'_n]$  we denote the formula obtained by substituting  $a'_1$  for  $a_1$ ,  $a'_2$  for  $a_2$ , etc. in  $\eta$ . Additionally, we assume that there are no two transitions having the same label in  $\mathcal{A}$ . This assumption is not essential

for the translation<sup>2</sup>, and it is used only to simplify the presentation of the results and the associated proofs.

Let  $tr = l \xrightarrow{act, g, r} l'$  be a transition, where  $l, l'$  are respectively the source and the target location,  $act$  is the action label,  $g$  is the guard, and  $r$  is the reset. It is convenient to use the following notations:  $source(tr) = l$ ,  $target(tr) = l'$ ,  $guard(tr) = g$ , and  $reset(tr) = r$ . Now, let  $i \in \mathbb{N}$ . We define  $guard^i(tr) = guard(tr)[x_1 \leftarrow x_1^i, \dots, x_n \leftarrow x_n^i]$ , i.e., the encoding of  $guard(tr)$  using the variables introduced earlier. Define  $reset^i(tr)$  as the smallest set such that  $x_j^i := a + t_i \in reset^i(tr)$  if  $x_j := a \in reset(tr)$  and  $x_j^i := x_j^{i-1} + t_i \in reset^i(tr)$  otherwise, for each  $1 \leq j \leq n$ . Intuitively,  $reset^i(tr)$  models the new value of each clock after the consecutive reset and delay. Let  $s \in Q$  be a location, we define  $inv^i(s) = I(s)[x_1 \leftarrow x_1^i, \dots, x_n \leftarrow x_n^i]$ , i.e., the encoding of the invariant of  $s$ . The above notions are combined to define the encoding of the transition  $tr$  as follows:

$$tr\_enc^i(tr) = loc\_enc^i(source(tr)) \wedge guard^i(tr) \wedge reset^{i+1}(tr) \\ \wedge inv^{i+1}(target(tr)) \wedge loc\_enc^{i+1}(target(tr)).$$

The correctness of the above construction is stated in the following lemma.

**Lemma 1.** *Let  $tr = l \xrightarrow{act, g, r} l'$  be a transition in  $\mathcal{A}$ ,  $v$  be a parameter valuation,  $(l, \omega)$  be a concrete state in  $\llbracket \mathcal{A} \rrbracket_v$  and  $i \in \mathbb{N}$ . Then,  $(l, \omega) \xrightarrow{act} (l', \omega[r]) \xrightarrow{d} (l', \omega[r] + d)$  in  $\llbracket \mathcal{A} \rrbracket_v$  iff for some valuation  $V$  of all the variables in  $Vars^k$  satisfying  $V \models tr\_enc^i(tr) \wedge TypeCut^{i+1}$  we have that:*

- $v = V|_P$ ,
- $\omega = V|_{X^i}[x_1^i \leftarrow x_1, \dots, x_n^i \leftarrow x_n]$ ,
- $d = V(t_{i+1})$ ,
- $\omega[r] + d = V|_{X^{i+1}}[x_1^{i+1} \leftarrow x_1, \dots, x_n^{i+1} \leftarrow x_n]$ .

*Proof.* Observe that the locations are uniquely represented by their encodings, thus we can focus on nonboolean variables only.

( $\Leftarrow$ ) Let  $V$  be a valuation of the variables such that  $V \models tr\_enc^i(tr) \wedge TypeCut^{i+1}$ . Let  $\omega = V|_{X^i}[x_1^i \leftarrow x_1, \dots, x_n^i \leftarrow x_n]$  and  $v = V|_P$ . Denote  $\omega^i = V|_{X^i}$ , then from  $V \models guard^i(tr)$  we obtain  $\omega^i \models_v guard^i(tr)$ , which in turn yields that  $\omega \models_v guard(tr)$ . Let  $d = V(t_{i+1})$ , denote  $\omega^{i+1} = V|_{X^{i+1}}$  and notice that from  $V \models reset^{i+1}(tr)$  it follows that  $\omega^{i+1}(x_j^{i+1}) = \omega^i[r](x_j^i) + d$  for all  $1 \leq j \leq n$ . Thus, if we denote  $\omega' = V|_{X^{i+1}}[x_1^{i+1} \leftarrow x_1, \dots, x_n^{i+1} \leftarrow x_n]$ , then  $\omega' = \omega[r] + d$ . Now, observe that from  $V \models inv^{i+1}(target(tr))$  we can infer that  $\omega^{i+1} \models_v inv^{i+1}(target(tr))$ , from which  $\omega' \models_v I(target(tr))$ , i.e.,  $\omega[r] + d \models_v I(target(tr))$ . As  $d \geq 0$  and in the view of the assumption that the invariants admit only upper bounds on clocks, we have also that  $\omega[r] \models_v I(target(tr))$ . This, together with the fact that  $V \models TypeCut^{i+1}$  assures that the used variables range over the correct sets, concludes this part of the proof.

( $\Rightarrow$ ) The implication in the other direction follows easily from the basic definitions of the transitions in  $\llbracket \mathcal{A}_v \rrbracket$ .

<sup>2</sup> We can always relabel the labels.

### 3.3 Encoding $k$ -runs and reachability testing

Our aim is to encode all the  $k$ -runs of  $\mathcal{A}_v$  for each parameter valuation  $v$ . Recall that  $n$  is the number of the clocks. To this end we define the following formula:

$$\begin{aligned} model\_enc^k(\mathcal{A}) = & TypeCut^k \wedge (\bigwedge_{i=1}^n (x_i^0 = t_0) \wedge loc\_enc^0(l_0) \wedge inv^0(l_0)) \\ & \wedge \bigwedge_{i=0}^{k-1} \bigvee_{tr \in \rightarrow} tr\_enc^i(tr). \end{aligned}$$

The first component ensures that all variables range over the proper values. The second component sets all the initial clocks to some arbitrary common value (the assumption that the invariants represent the upper bounds on the clocks is significant here), encodes the initial state, and makes sure that its invariant is satisfied. The last component encodes all the possible transitions in the  $k$ -runs.

Let  $\phi$  be a state formula and  $i \in \mathbb{N}$ . Let  $\{l_1, \dots, l_m\}$  be a set of all the locations present in  $\phi$ . We define the encoding of  $\phi$  as follows:

$$pr\_enc^i(\phi) = \phi[x_1 \leftarrow x_1^i, \dots, x_n \leftarrow x_n^i, l_1 \leftarrow loc\_enc^i(l_1), \dots, l_m \leftarrow loc\_enc^i(l_m)],$$

i.e., we simply substitute in  $\phi$  each clock with its  $i$ -th variable counterpart, and each location with its encoding using boolean variables from  $\mathcal{BV}^i$ .

We obtain the formula to be used for testing and parameter synthesis by combining the encodings of the  $k$ -runs and the property, as presented in the following lemma.

**Lemma 2.** *Let  $\mathcal{A}$  be a parametric timed automaton,  $\phi$  be a state formula,  $v$  be a valuation of the parameters, and  $k \in \mathbb{N}$ . A state satisfying  $\phi$  is  $k$ -reachable in  $\llbracket \mathcal{A} \rrbracket_v$  iff there exists a valuation  $V$  of all the variables in  $Vars^k$  such that  $V \models model\_enc^k(\mathcal{A}) \wedge pr\_enc^k(\phi)$  and  $V|_P = v$ .*

*Proof.* Due to the presence of  $TypeCut^k$  in  $model\_enc^k(\mathcal{A})$  we know that all the variables range over the proper sets.

Let  $l$  be the (unique) location such that  $V|_{\mathcal{BV}^k} \models loc\_enc^k(l)$  and  $\omega^k = V|_{X^k}[x_1^k \leftarrow x_1, \dots, x_n^k \leftarrow x_n]$ . First, we prove that  $V \models model\_enc^k(\mathcal{A})$  iff the state  $(l, \omega^k)$  is  $k$ -reachable in  $\llbracket \mathcal{A}_v \rrbracket$  for  $v = V|_P$ .

If  $k = 0$ , then  $\bigwedge_{i=1}^n (x_i^0 = t_0) \wedge loc\_enc^0(l_0) \wedge inv^0(l_0)$  is satisfied by the valuation  $V$  iff  $V$  is such that if we denote  $\omega^0 = V|_{X^0}[x_1^0 \leftarrow x_1, \dots, x_n^0 \leftarrow x_n]$  and  $V|_P = v$ , then for some  $t^0 = V(t_0)$  we have that  $\omega^0 = \omega_0 + t^0$  and  $\omega^0 \models_v I(l_0)$ . This corresponds to the set of states to which  $\llbracket \mathcal{A}_v \rrbracket$  can progress by the time transitions

For the inductive step observe that  $model\_enc^k(\mathcal{A}) = model\_enc^{k-1}(\mathcal{A}) \wedge \bigvee_{tr \in \rightarrow} tr\_enc^{k-1}(tr) \wedge TypeCut^k = \bigvee_{tr \in \rightarrow} (model\_enc^{k-1}(\mathcal{A}) \wedge tr\_enc^{k-1}(tr)) \wedge TypeCut^k$  and apply Lemma 1 and the inductive assumption.

To conclude, notice that  $pr\_enc^k(\phi)$  simply encodes all the concrete states for which  $\phi$  holds, using the variables from  $\mathcal{BV}^k \cup X^k$ .

## 4 Parameter set approximations

We already know how to write, for a given parametric timed automaton and a property, the formula encoding  $k$ -reachable states for which the property holds together with the associated valuations of the parameters. It might be beneficial to verify this formula as it is, if we wish to obtain the answer to the question whether the property is satisfied by  $k$ -reachable states. We can also rely on the SMT-solver to obtain an exemplary witness, i.e., a correct valuation of the parameters. Our task is, however, to systematically explore the space of the admissible parameters, with a hope for painting a part of the picture from which an analyst can make further generalizations.

Let  $\mathcal{A} = \langle Q, l_0, A, X, P, \rightarrow, I \rangle$  be an L/U automaton and  $P = L \cup U$ , where  $L$  and  $U$  are disjoint sets of the upper and the lower parameters, respectively. Assume that  $L = \{\lambda_1, \dots, \lambda_l\}$  and  $U = \{\mu_1, \dots, \mu_u\}$ .

Let  $\phi$  be a property and  $v$  be such a valuation of the parameters that there exists a reachable state in  $\llbracket \mathcal{A} \rrbracket_v$  satisfying  $\phi$ . Recall (Proposition 1) that in the class of the L/U automata this means that a state satisfying  $\phi$  is also reachable in  $\llbracket \mathcal{A} \rrbracket_{v'}$  for each  $v'$  such that  $v \leq v'$ .

Define the *complementing clause* with respect to  $v$  as follows

$$\text{ComplClause}(v) = \bigvee_{i=1}^l (\lambda_i > v(\lambda_i)) \vee \bigvee_{i=1}^u (\mu_i < v(\mu_i)),$$

and observe that  $v' \models \text{ComplClause}(v)$  iff  $v \leq v'$  is not true.

We employ  $\text{ComplClause}(v)$  to block the SMT-solver from seeking for parameter valuations which can be inferred from the L/U automata properties and the set of the parameters that has been already synthesized.

The following algorithm attempts to synthesise parameter valuations for which there exists a  $k$ -reachable state satisfying the property  $\phi$ . If the search is successful, the user is presented with a newly synthesised parameter valuation  $v$  and asked whether the procedure should be continued. If so, a new blocking  $\text{ComplClause}(v)$  is added to the main formula and the loop takes another turn.

---

### Algorithm 1 $\text{ReachSynth}(\mathcal{A}, \phi, k)$

---

**Input:** an L/U automaton  $\mathcal{A}$ , a property  $\phi$ , a depth value  $k \in \mathbb{N}$

**Output:** a set  $\text{Res}$  of valuations of the parameters

- 1:  $\text{Res} := \emptyset$
  - 2:  $\text{reachFormula} := \text{model\_enc}^k(\mathcal{A}) \wedge \text{pr\_enc}^k(\phi)$
  - 3: **while** user requests to expand  $\text{Res}$  and  $\text{reachFormula}$  is satisfiable **do**
  - 4:   let  $V$  be such that  $V \models \text{reachFormula}$  and  $v := V|_P$
  - 5:   let  $\text{Res} := \text{Res} \cup \{v\}$
  - 6:   let  $\text{reachFormula} := \text{reachFormula} \wedge \text{ComplClause}(v)$
  - 7: **end while**
  - 8: **return**  $\text{Res}$
-

Note that in the above algorithm the testing for satisfiability (line 3), and extraction of the witness valuation  $v$  of the parameters (line 4) is performed by means of a call to an external SMT-solver.

**Lemma 3.** *Let  $\mathcal{A}$  be an L/U automaton,  $\phi$  be a property, and  $k \in \mathbb{N}$ . For each valuation of the parameters  $v'$  such that there exists  $v \in \text{ReachSynth}(\mathcal{A}, \phi, k)$  satisfying  $v \leq v'$  we have that  $\llbracket \mathcal{A} \rrbracket_{v'} \models EF\phi$ .*

*Proof.* It follows immediately from Lemma 2 combined with the properties of  $\text{ComplClause}(v)$ .

The  $\text{ReachSynth}$  algorithm can be used as the main building block of a bounded parametric model checking process. The input consists of an L/U automaton  $\mathcal{A}$  and a property  $\phi$ . Initially, we can employ the results from [12] to solve the *emptiness problem* for  $\phi$  and  $\mathcal{A}$ , i.e., to check whether there exists a parameter valuation  $v$  such that  $\llbracket \mathcal{A} \rrbracket_v \models EF\phi$ . If the existence of such a valuation is confirmed, then the *universality problem*, i.e., the question whether  $\llbracket \mathcal{A} \rrbracket_v \models EF\phi$  for all parameter valuations  $v$ , can be checked as a dual to the emptiness. If the answer to the universality problem is false, then  $\text{ReachSynth}(\mathcal{A}, \phi, k)$  is called, starting from  $k = 0$  and incrementing the value of  $k$  whenever the previous call returned empty set or the loop was stopped by the user.

## 5 Evaluation

In this section we present some preliminary results on parametric analysis of two selected models, namely Fischer's Mutual Exclusion Protocol and a version of Generic Timed Pipeline Paradigm [15]. Both of them are well established and scalable benchmarks specified in a form of networks of parametric timed automata.

**Definition 5.** *Let  $\mathcal{U} = \{\mathcal{A}^i = \langle Q^i, l_0^i, A^i, X^i, P^i, \rightarrow^i, I^i \rangle \mid 1 \leq i \leq m\}$  be a set (a network) of parametric timed automata such that  $X^i \cap X^j = \emptyset$  for each  $1 \leq i, j \leq m$  and  $i \neq j$ . Let  $\mathcal{L}(a) = \{1 \leq i \leq m \mid a \in A^i\}$  be a function associating with each action  $a \in \bigcup_{1 \leq i \leq m} A^i$  the indices of the automata recognizing  $a$ . We define the product automaton  $\mathcal{A} = \langle Q, l_0, A, X, P, \rightarrow, I \rangle$  of the network  $\mathcal{U}$ , where:*

- $Q = \prod_{i=1}^m Q^i$ ,
- $l_0 = (l_0^1, \dots, l_0^m)$ ,
- $A = \bigcup_{i=1}^m A^i$ ,
- $X = \bigcup_{i=1}^m X^i$ ,
- $P = \bigcup_{i=1}^m P^i$ ,
- $I((l_1, \dots, l_m)) = \bigwedge_{i=1}^m I^i(l_i)$  for each  $(l_1, \dots, l_m) \in Q$ ,

and the transition relation  $\rightarrow$  is such that:

- $(l_1, \dots, l_m) \xrightarrow{a, g, r} (l'_1, \dots, l'_m)$  iff for each  $i \in \mathcal{L}(a)$  we have  $l_i \xrightarrow{a, g_i, r_i} l'_i$ , and  $g = \bigwedge_{i \in \mathcal{L}(a)} g_i$ ,  $r = \bigcup_{i \in \mathcal{L}(a)} r_i$ , and  $l_i = l'_i$  for all  $i \in \{1, \dots, m\} \setminus \mathcal{L}(a)$ .

In addition to a network, the user is expected to supply an *experiment's plan* file. Such a plan consists of a sequence of pairs  $(k, \text{No})$  of natural numbers, where  $k$  is the length of the runs to be considered, and  $\text{No}$  is a maximal number of parameter valuations to be synthesised should the verified property be found satisfiable. Our tool goes through the pairs in accordance with increasing  $k$ , incrementally building the formulae to be tested as it was presented in earlier sections.

All the experiments have been performed on Intel P6200 2.13GHz dual core machine with 3GB memory, running Linux operating system.

### 5.1 Fischer's Mutual Exclusion Protocol

The timed automata network presented in Figure 1 models one of the possible solutions to the classical problem of mutual exclusion, i.e., ensuring that only one of the competing processes is able to gain an access to the critical section.

The system in question consists of  $n$  independent processes synchronised via the shared variable  $X$ . The model contains two parameters, i.e., the lower bound parameter  $\delta$  and the upper bound parameter  $\Delta$ . It is well known that no two processes are able to simultaneously get to their critical sections iff  $\Delta \leq \delta$ , thus we have chosen to verify the reachability of  $\phi_1 = \text{critical}_1 \wedge \text{critical}_2$ . Intuitively, this means that we aim to synthesise values of the parameters  $\delta$  and  $\Delta$  under which the system behaves incorrectly, allowing two competing processes to jointly enter their critical sections.

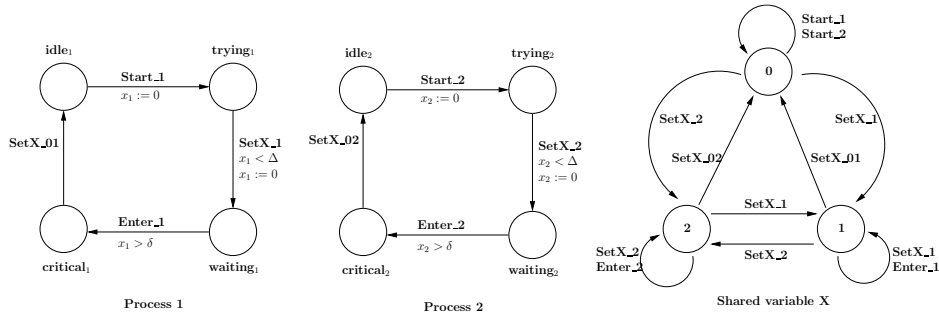


Figure 1: Fischer's Mutual Exclusion Protocol, 2 processes

As it turns out, for each test with the positive outcome, the set of the returned valuations consists of the pairs  $(\delta = i, \Delta = i + 1)$  for  $i$  from 0 to the limit  $\text{No} = 10$  given in the experiment's plan. Clearly, from the point of view of an analyst and in light of Lemma 3 this result indeed justifies an educated guess that the mutual exclusion property is violated if  $\Delta > \delta$ .



$n$	$k$	SAT? (Y/N)	param vals found	max. form. size (MB)	form. bldg. time (sec.)	total CVC3 time (sec.)	peak CVC3 mem. (MB)
7	1-5	N	-	1.54	1.7	1.8	20
7	6	Y	10	3.51	2.25	37.2	41
8	1-5	N	-	2.95	3.95	3.6	30
8	6	Y	10	7.13	5.45	75.3	70
9	1-5	N	-	5.48	7.1	6.5	48
9	6	Y	10	13.71	11.86	187.8	119
10	1-5	N	-	9.43	13.1	11.33	69
10	6	Y	10	24.62	24.06	245.75	198
11	1-5	N	-	15.25	23.29	20.71	108
11	6	Y	10	41.84	46.23	504.35	331
12	1-5	N	-	24.94	40.13	25.11	170
12	6	Y	10	71.17	85.07	726.51	560
13	1-5	N	-	38.89	66.1	40.78	256
13	6	Y	10	115.8	149.24	1315.87	1000
14	1-5	N	-	57.76	105.98	67.50	384
14	6	Y	10	180.71	253.99	3192.47	1600

Table 1: Fischer’s Mutual Exclusion parametric verification results

Legend:  $n$ : a number of competing processes,  $k$ : runs’ lengths,  $SAT?$ : satisfiability,  $4$ -th column: the number of the synthesised parameter valuations,  $5$ -th column: the maximal (if  $k$  is an interval) size of the generated SMT-lib v2 formula,  $6$ -th column: the time spent on incrementally building the formulae,  $7$ -th column: the total time spent on verifying the formulae,  $8$ -th column: the maximal memory used by CVC3 solver.

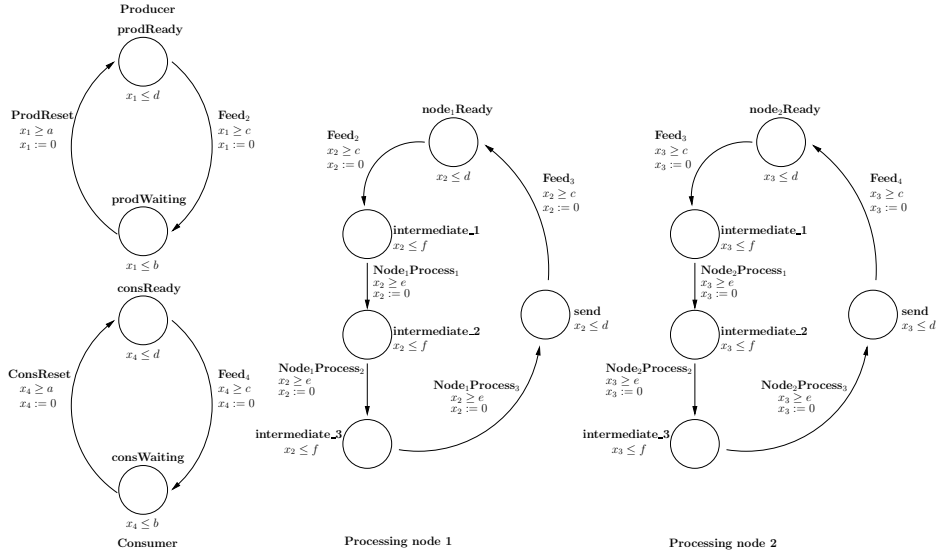


Figure 2: Generic Timed Pipeline Paradigm, 2 processing nodes of length 3

## 5.2 Generic Timed Pipeline Paradigm

The network presented in Figure 2 models the system consisting of the Producer feeding the Consumer with data sent through a sequence of nodes with additional processing capabilities.

The model is scalable with respect to the number  $n$  of the processing nodes and the length  $m$  of each processing node and it contains three lower ( $a, c, e$ ) and three upper ( $b, d, f$ ) parameters.

We have decided to add one dummy clock, called  $x_{total}$ , to the above system. It is straightforward to see that such an alteration does not change the behaviour of the model, and that  $x_{total}$  can be used to measure the total time passed along a given run. With the help of the new clock we have analysed the reachability of  $\phi_2 = ConsWaiting \wedge ProdReady \wedge x_{total} \geq 5$ . Again, the limit No is set to 10 for all  $k$ .

$n$	$m$	$k$	SAT? (Y/N)	param vals found	max. form. size (MB)	form. bdg. time (sec.)	total CVC3 time (sec.)	peak CVC3 mem. (MB)
2	10	1-12	N	—	0.01	0.01	1.27	7
2	10	13	Y	10	0.02	0.003	9.25	17
2	15	1-17	N	—	0.02	0.015	3.04	12
2	15	18	Y	10	0.02	0.004	22.29	29
2	20	1-22	N	—	0.03	0.02	6.40	18
2	20	23	Y	10	0.03	0.004	57.73	60
3	1	1-5	N	—	0.01	0.008	0.26	4
3	1	6	Y	10	0.02	0.004	16.91	26
3	2	1-7	N	—	0.02	0.014	0.51	5
3	2	8	Y	10	0.03	0.006	86.55	85
3	3	1-9	N	—	0.04	0.023	0.89	10
3	3	10	Y	10	0.05	0.008	13.68	17
3	4	1-11	N	—	0.06	0.034	1.48	10
3	4	12	Y	10	0.08	0.01	39.87	32
3	5	1-13	N	—	0.08	0.049	2.55	8
3	5	14	Y	10	0.11	0.014	2472.87	458

Table 2: Generic Timed Pipeline Paradigm parametric verification results  
Legend: see Table 1,  $m$ : a number of processing nodes

Note that the generated SMT formulae are rather small in this case. This probably reflects the power of a concise representation by means of SMT instances rather than the size of model's state space [13].

Table 3 contains some exemplary parameter valuations, synthesised for several instances of the model. This illustrates the power of approximation-based approach, where the collected data may be used in search for general pattern.

$n$	$m$	$k$	$a$	$b$	$c$	$d$	$e$	$f$
2	15	18	0	0	0	0	0	0
			0	0	1	2	0	0
			1	1	0	1	0	1
			0	0	1	17	1	1
			0	0	2	16	0	0
			1	1	1	2	0	1
			0	0	2	19	1	1
			0	0	3	17	0	0
			1	1	0	15	1	1
			0	0	3	6	0	0
2	20	23	0	0	0	0	0	0
			0	0	1	2	0	0
			1	1	0	1	0	1
			0	0	1	22	1	1
			0	0	2	21	0	0
			1	1	0	20	1	1
			0	0	2	4	0	0
			1	1	1	2	0	1
			0	0	2	24	1	1
			0	0	3	22	0	0
3	1	6	0	0	0	1	0	0
			0	1	1	4	0	0
			1	1	0	1	0	0
			0	2	1	6	1	1
			1	1	1	4	0	0
			2	2	0	3	1	1
			0	1	0	2	1	1
			2	2	0	1	0	0
			1	1	0	3	1	1
			2	2	0	2	0	0
3	2	8	0	0	0	0	0	0
			0	1	1	3	0	0
			1	1	0	1	0	0
			0	3	1	7	1	1
			1	2	2	6	0	0
			2	2	0	1	0	1
			1	1	1	4	0	0
			1	1	1	3	0	0
			3	3	0	2	0	1
			0	2	0	4	1	1
3	4	12	0	0	0	0	0	0
			1	1	0	1	0	1
			2	2	0	2	0	2
			3	3	0	3	0	3
			4	4	0	4	0	4
			5	5	0	5	0	5
			6	6	0	6	0	6
			7	7	0	7	0	7
			8	8	0	8	0	8
			9	9	0	9	0	9

Table 3: Generic Timed Pipeline Paradigm: exemplary parameter valuations

## 6 Conclusions

We have proposed a simple translation for a direct representation of finite runs of a parametric timed automaton in form of SMT instances. This translation coupled with blocking clauses allowed us for an underapproximation of the set of the parameter valuations under which the given reachability property holds in L/U automata. To the best of our knowledge this is the first such application of SMT solvers, and this is at the same time a proof-of-concept as well as a practical tool for exploring the spaces of parameter valuations.

In the future we plan to extend our work to the parametric verification of properties more complex than reachability, e.g., the existential fragment of CTL\*. Additionally, we plan to investigate the possibilities of an automated inference of more general (or even complete) constraints under which the given property holds using partial knowledge on the space of the parameter valuations obtained using the methods presented in this paper.

## References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. R. Alur, T. Henzinger, and M. Vardi. Parametric real-time reasoning. In *Proc. of the 25th Ann. Symp. on Theory of Computing (STOC'93)*, pages 592–601. ACM, 1993.
3. É. André. Imitator ii: A tool for solving the good parameters problem in timed automata. In Yu-Fang Chen and Ahmed Rezzine, editors, *INFINITY*, volume 39 of *EPTCS*, pages 91–99, 2010.
4. E. André, T. Chatain, E. Encrenaz, and L. Fribourg. An inverse method for parametric timed automata. *International Journal of Foundations of Computer Science*, 20(5):819–836, Oct 2009.
5. A. Annichini, A. Bouajjani, and M. Sighireanu. TREX: A tool for reachability analysis of complex systems. In *Proc. of the 13th International Conference on Computer Aided Verification, CAV '01*, pages 368–372, 2001.
6. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at [www.SMT-LIB.org](http://www.SMT-LIB.org).
7. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of the 5th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, 1999.
8. L. Bozzelli and S. La Torre. Decision problems for lower/upper bound parametric timed automata. *Formal Methods in System Design*, 35(2):121–151, 2009.
9. H. Dierks and J. Tapken. MOBY/DC – A tool for model-checking parametric real-time specifications. In H. Garavel and J. Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *LNCS*, pages 271–277. Springer, 2003.
10. L. Doyen. Robust parametric reachability for timed automata. *Inf. Process. Lett.*, 102:208–213, May 2007.

11. G. Frehse, S. K. Jha, and B. H. Krogh. A counterexample-guided approach to parameter synthesis for linear hybrid automata. In *Proc. of the 11th international workshop on Hybrid Systems: Computation and Control*, HSCC '08, pages 187–200, Berlin, Heidelberg, 2008. Springer-Verlag.
12. T. Hune, J. Romijn, M. Stoelinga, and F. Vaandrager. Linear parametric model checking of timed automata. *J. Log. Algebr. Program.*, 52-53:183–220, 2002.
13. M. Knapik, W. Penczek, M. Szreter, and A. Pólrola. Bounded parametric verification for distributed time Petri nets with discrete-time semantics. *Fundam. Inform.*, 101(1-2):9–27, 2010.
14. D. Lime, O. H. Roux, C. Seidner, and L. M. Traonouez. Romeo: A parametric model-checker for Petri nets with stopwatches. In S. Kowalewski and A. Philippou, editors, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 54–57. Springer, 2009.
15. D. Peled. All from one, one for all: On model checking using representatives. In *Proc. of the 5th Int. Conf. on Computer Aided Verification (CAV'93)*, volume 697 of *LNCS*, pages 409–423. Springer-Verlag, 1993.
16. R. F. Lutje Spelberg and W. J. Toetenel. Splitting trees and partition refinement in real-time model checking. In *HICSS*, page 278, 2002.
17. W. Penczek and M. Knapik. Bounded Model Checking for Parametric Timed Automata. *T. Petri Nets and Other Models of Concurrency*, 5, to appear, 2012.

# Model-Driven Middleware Support for Team-Oriented Process Management

Matthias Wester-Ebbinghaus and Michael Köhler-Bußmeier

University of Hamburg, Department of Informatics  
Theoretical Foundations of Informatics  
{wester,koehler}@informatik.uni-hamburg.de  
<http://www.informatik.uni-hamburg.de/TGI>

**Abstract.** Management of collaborative processes involving multiple parties is one of the dominant topics in contemporary information system research. While the process perspective is quite well understood and supported by a wide range of modeling approaches, it is necessary to go beyond the process perspective alone. We specifically address the following question: If we consider the involved parties of a collaborative process as a *team*, then (1) which are the general *formation rules* for such a team together with the collaborative process it carries out and (2) to which concrete underlying *organizational structure* do these rules apply? To address this question, we present the organizational modeling approach SONAR. The accompanying models are rather high-level and illustrative but at the same time they are rich enough in order to generate executable models and other kinds of code that together form the core of a middleware implementation for team-oriented process management.

## 1 Introduction

Management of collaborative processes involving multiple parties is one of the most dominant topics in contemporary information system research, especially in the field of business process management (BPM) but also on a smaller scale in the field of computer-supported cooperative work (CSCW) or community support. The process perspective itself is quite well understood and there exists a wide range of more or less similar process modeling approaches (differing in specific aspects), including workflow nets and their descendants [1,2], the Business Process Modeling Notation (BPMN) [16], the Web Service Business Process Execution Language (BPEL) [4], Event-driven Process Chains (EPCs) [13] and the Yet Another Workflow Language (YAWL) [3]. However, there remains the question of organizational structures behind a given set of processes, which is not addressed in a thorough and systematic way by these approaches. We want to formulate this question a bit more vividly in the following way: If we consider the involved parties of a collaborative process as a *team*, then (1) which are the general *formation rules* for such a team together with the collaborative process it carries out and (2) to which concrete underlying *organizational structure* do these rules apply?

To answer this question, a more comprehensive modeling approach is necessary, encompassing both a system's processes and structure in an integrated manner. While this is to some extent addressed in the field of enterprise architecture management (EAM), EAM is a rather high-level discipline and is at least not necessarily concerned with models whose primary purpose is to be directly transferred into software artifacts (although this may be true for some parts, especially for process models). Contrary to that, the field of organization-oriented multi-agent systems is primarily concerned with rather comprehensive organizational models that exhibit a close gap to software-technical deployment [5,8]. Here we find models that encompass multiple integrated organizational perspectives (e.g. structure, function, interactions, norms). But despite this multi-perspective approach, we typically still find approaches where either a structural or a process perspective dominates. In approaches like *S-MOISE*<sup>+</sup> [12] or *TEAMCORE/KARMA* [17], a structural perspective dominates and a process perspective has to be inferred from certain functional specifications or from normative requirements concerning action execution. In approaches like *ISLANDER* [9], a process perspective dominates and a structural perspective has to be inferred from decompositions of the process models.

In this context of process and overall organizational modeling, we present the Petri net-based organizational modeling approach **SONAR (Self-Organizing Net ARchitecture)**.<sup>1</sup> It explicitly addresses the question from above concerning structure and process perspectives in teamwork modeling. We provide a way to capture the whole context of team-oriented process management: from the underlying organizational structure over team formation up to process execution by the team.

We have laid specific emphasis on achieving the following combination: (1) SONAR models are simple enough to be easily understood and analyzed (by means of standard Petri net tools). (2) SONAR models are rich enough to capture the interplay of various organizational concepts in such detail that we can automatically generate executable models and other kinds of code from them. In this context, Figure 1 gives an overview of the results we present in this paper. We concentrate on the modeling approach and in which ways model parts are

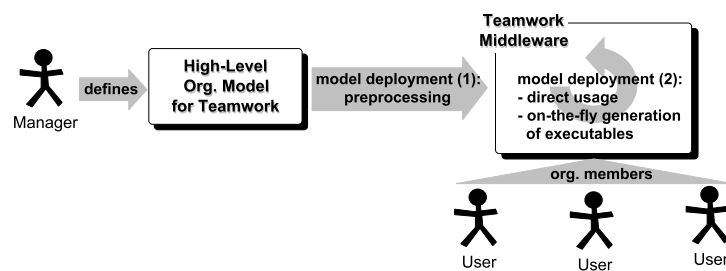


Fig. 1: Model-driven support for teamwork, general overview

<sup>1</sup> We are not dealing with the “self-organizing” part in this paper, but see the conclusion.

made executable or other code is generated from them. The figure can be regarded as capturing the most fundamental way of applying our SONAR approach. In the outlook of the paper we address several extensions that build upon it, but here we just regard the basic case. A manager creates a high-level organizational model that is void of execution details. The different model parts undergo a preliminary deployment phase where they are pre-processed and then passed on to a middleware layer for team-oriented processes management. Here, models are actually deployed and support the different phases/aspects of teamwork. Some of the pre-processed model parts are persistent and directly used while others serve an on-demand generation of temporary executables. Users access the middleware layer and act as organizational members that participate in teamwork (this users might be social but also artificial/software-technical actors).

In the remainder of the paper, we flesh out this rather abstract description. In Section 2, we introduce the SONAR modeling approach. In Section 3 we elaborate on transformations of original SONAR models in order to obtain code from them and in Section 4 we describe how this code is embedded in an agent-based middleware for teamwork. We conclude our work in Section 5 and give an outlook to advanced and future topics of our research.

Note that both the SONAR modeling approach and the middleware implementation rest on our previous work (cf. especially [14,15]). Several extensions, simplifications and improvements have been introduced over the years and in this paper, we present the consolidated current state with original contributions concerning both the modeling approach and the middleware support.

## 2 Organizational Models Based on SONAR

For organized activities two fundamental (and opposing) requirements have to be taken into account, the *division of labour* into various tasks and the *coordination* of carrying out these tasks. For SONAR, this can be rephrased more concretely and with reference to the terminology used in the introduction of the paper. *Coordinated carrying out of tasks* corresponds to a team executing a distributed (multi-party) workflow (DWF). *Division of labor* corresponds to the formation of such a team together with a DWF definition. Formation takes place according to general formation rules and a specific organizational structure to which these rules are applied. Consequently, SONAR models center around the duality of DWF (process) and organizational structure models. Both sides have to be coherently related with one another.

SONAR is based on Petri nets which offer both a graphical representation and formal semantics. In [14] we present SONAR in a formal way with theorems and proofs. However, in this paper we present a new version of SONAR, where the differences concern mainly a more readable and better structured organizational structure model. We will avoid formal specifications and instead give a rather illustrative introduction of the SONAR modeling approach. We just assume a general understanding of Petri nets (cf. [10]).

We will consider a running example throughout the paper. As SONAR models are based around the duality of distributed workflow (DWF) and organizational structure models, we could start with either of them. Here we begin with the workflow perspective. Figure 2 shows a DWF for collaboratively submitting a paper. The DWF is distributed in the sense that it encompasses multiple roles, here

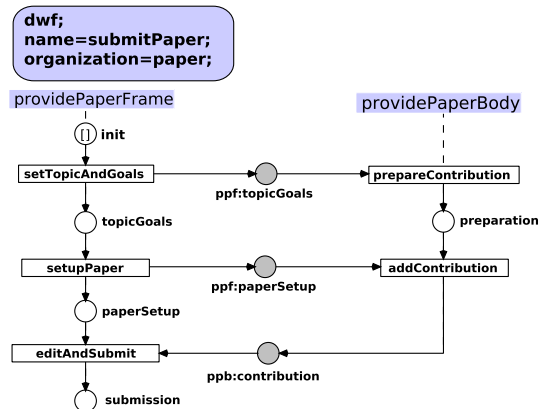


Fig. 2: Workflow with multiple roles for submitting paper

`providePaperFrame` and `providePaperBody`. Each action of the DWF is mapped onto exactly one role. Actions are modeled as transitions. They are connected by places. Places connecting actions belonging to the same role form the *DWF life line* of that role and we arrange such a life line vertically in our models.<sup>2</sup> Places connecting actions of different roles can best be considered as *message transfers between roles* and we draw them as horizontal connections. One can consider the places between the transitions of different roles as the *interface* between these roles. Places and transitions of a DWF model are named. Names of message places are prefixed with a key for the role sending that message (for example `ppf:` for the role `providePaperFrame`). Such message place names have to be unique across the whole set of DWF models of an overall SONAR model (see below for the reason). If a DWF model is decomposed into role parts and there exists an interface between two roles, each role part gets its own copy of the corresponding interface places.

Figure 3 shows another DWF. More exactly, it shows a DWF fragment. This fragment consists of two roles `supervisePaperSubmission` and `writeIntroAndConclusion`. These two roles can be used to refine the role `providePaperFrame` from

<sup>2</sup> Of course, we do not rely on graphical arrangements in order to determine the different role parts of a DWF. We are currently working on an action inscription language for DWF transitions. So far, such an inscription does at least contain the name of the role that the transition belongs to. This is even more important when DWF life lines are not just sequences as in the rather simple examples in this paper. They may include forks, joins and concurrency. However, we have omitted the transition inscriptions in the DWF figures of this paper as the different role parts should be easily identified.



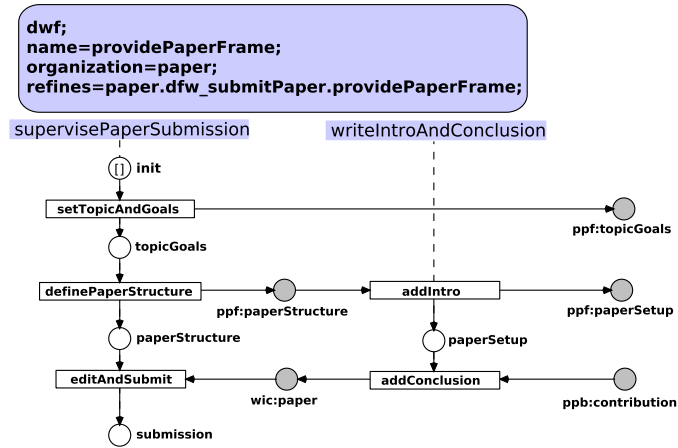


Fig. 3: Refined workflow part for the `providePaperFrame` role from Figure 2

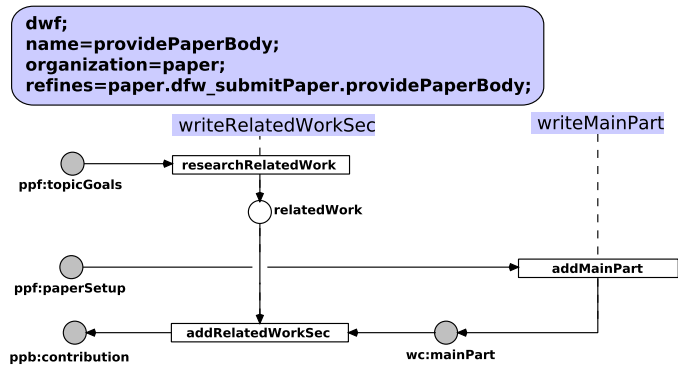


Fig. 4: Refined workflow part for the `providePaperBody` role from Figure 2

Figure 2. Note that on the right side of Figure 3, the two roles `supervisePaperSubmission` and `writeIntroAndConclusion` *in combination* share the same interface as the role `providePaperFrame` in Figure 2 in terms of message places (whose names have been carried over and uniquely identify them). In fact, it is possible to substitute the two combined roles `supervisePaperSubmission` and `writeIntroAndConclusion` for the role `providePaperFrame` and obtain the same input/output behavior to the outside, i.e. from the viewpoint of the partner role `providePaperBody`.

Likewise, Figure 4 shows a DWF fragment, where the two roles `writeRelatedWorkSec` and `writeMainPart` can be used to refine/substitute the role `providePaperBody` from Figure 2 while obtaining the same input/output behavior from the viewpoint of the partner role `providePaperFrame`.

Following this line of thought, it is of course also possible to substitute both roles `providePaperFrame` and `providePaperBody` with the combined roles from Figures 3 and 4 respectively as both refinements respect the original input/output

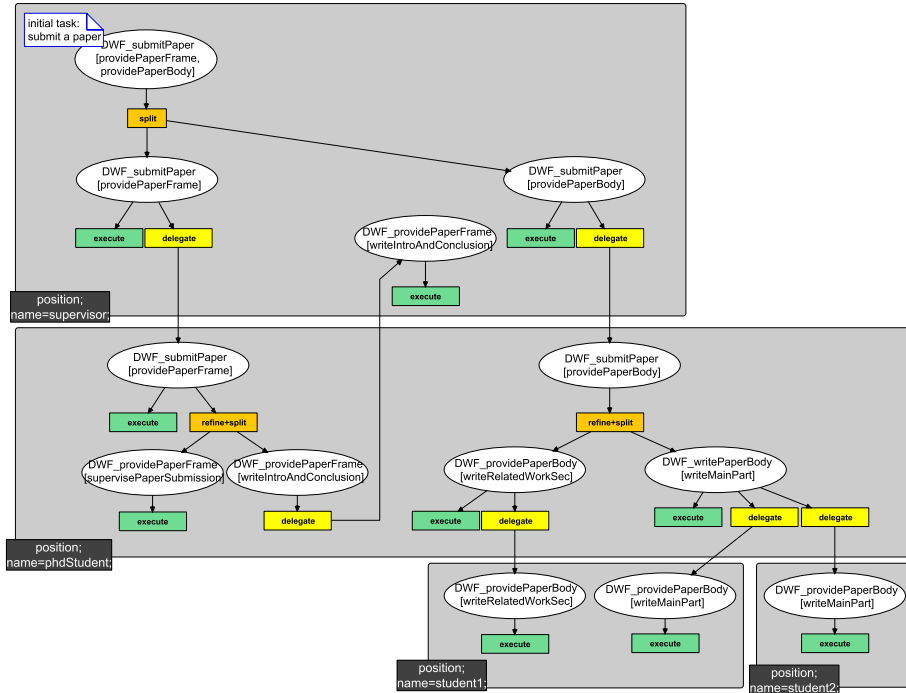


Fig. 5: Delegation model for jointly submitting a paper

behavior of the substituted roles and the overall composition thus fits together. To conclude, we arrive at basically four possible DWFs for jointly submitting a paper. Further models of role refinements would lead to more possibilities of DWF composition. It remains to supplement such a set of DWFs and DWF fragments with a model that determines not only when to compose which DWF parts but also who takes on which roles in a finally composed DWF. This is where SONAR organizational structure models come into play.

Organizational structures in SONAR are basically modeled as delegation structures. Figure 5 shows such a delegation net for the running example of joint paper submission.<sup>3</sup> A SONAR delegation net comprises multiple *positions* that are abstractions of actors (that occupy these positions when a SONAR organization is deployed). Positions are modeled as grey boxes that partition an underlying *task structure*. In Figure 5, we have as positions a *supervisor*, a *phd student* and two *students* that distribute tasks among themselves in order to jointly submit a paper. The underlying task structure is modeled as a Petri net. A place models a *task* and a transition models the *implementation* of a task. For this purpose, each transition has exactly one place in its preset. Task implementation can take on multiple forms and transitions are named accordingly:

<sup>3</sup> Delegation nets have been over-hauled compared to previous publications, cf. [14,15]. The explicit inscription of transitions with the implementation type that they represent leads to slightly larger but much more readable models.

1. *Execute*: The task is directly executed.
2. *Delegate*: The task is delegated.
3. *Refine*: Sub-tasks for a task are determined.
4. *Split*: A task is split into (already determined) sub-tasks.

The latter two cases are typically combined. All the implementation cases appear in Figure 5. Delegations are the kind of task implementation that relates two positions while refines, splits and executions are internal to positions.

The intertwining of a SONAR delegation net with DWF (fragment) models lays in the nature of the tasks. Each task in a delegation net corresponds to one or more roles in a DWF. Consequently, the places in Figure 5 are named according to the pattern  $DWF_a[role_1, \dots, role_n]$ , meaning that the task corresponds to implementing the roles  $role_1, \dots, role_n$  from DWF  $DWF_a$ . Combining a delegation model with a set of DWF models leads to a straightforward notion of well-formedness of an overall SONAR model: (1) Delegation has to start with an initial task that corresponds to all roles of a complete DWF model (not a DWF fragment) and (2) task refinements must map onto associated role refinements in the set of DWF (fragment) models. Consequently, Figure 5 shows *one* possible delegation model for the DWF models from Figures 2 – 4 (likewise, other sets of DWF models may fit to the delegation model).

This way, the process perspective represented by DWF modeling is supplemented with an organizational structure perspective that guides both the formation of teams (where positions represent the team members) and the associated team DWFs. For example, the delegation model from Figure 5 allows multiple teams to be formed. An interesting fact is that team formation actually corresponds to the possible firings of the delegation net, its Petri net processes, cf. [11]. Each Petri net process of a delegation net model corresponds to a possible team.

Using Petri nets as the basis for SONAR modeling allows us to take advantage of well-known analysis techniques. As we rely on simple place/transition (P/T) nets, there exist standard techniques and tools for checking the soundness of workflow net models or the free-choice nature of delegation models (cf. [1,7] and the ProM framework<sup>4</sup>). The interleaving of delegation and DWF models and especially the notion of role refinement of course goes beyond P/T net analysis. But especially the tool set from <http://www.service-technology.org> promotes a service-oriented perspective on Petri net models where Petri nets with interface places (open nets) are characterized in terms of their possible partners, cf. [20]. For our purpose, this allows to analyse whether a role and its refinement in terms of multiple roles really have the same input/output behavior and can be substituted with one another in DWF models.

### 3 Model Deployment

The models presented so far have been on a relatively high level. They are basically P/T nets, where some naming conventions have to be followed. There

<sup>4</sup> <http://www.promtools.org/>

are no execution details, except for the fact that Petri nets inherently have an operational semantics. It is not even necessary to model all possible DWFs that can occur during the execution of a SONAR organization. Instead, it is sufficient to model some initial DWF models and then just add models for selective role refinements.

In order to utilize the models in the context of a SONAR-based middleware layer for teamwork, some deployment steps are necessary. Based on the preceding section, it is now possible to be more specific on the model-driven support for teamwork illustrated in Figure 1 from the introduction. Figure 6 shows a SONAR-based re-interpretation of the figure.

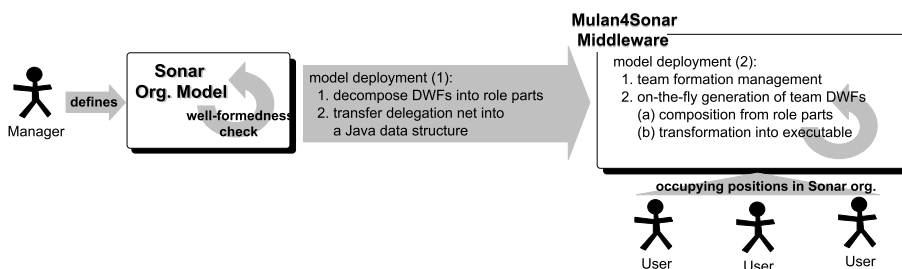


Fig. 6: Model-driven support for teamwork based on SONAR

After checking well-formedness of all aspects of an overall SONAR model (see the previous section), the first phase of model deployment is pre-processing. One immediate question is whether to use the Petri net models themselves (enriched/extended for deployment)<sup>5</sup> or whether to transfer them into other artifacts. Although we do not deal with run-time re-organization in this paper, this aspect has a strong impact on answering this question. Changing the Petri net models and re-deploying them at run-time can get quite cumbersome and costly. Currently, we have decided to use the DWF models directly in their Petri net form and to transfer the delegation model into a Java data structure. We treat DWF models and thus *how things are basically done* as rather fixed and the role parts as the basic behavioral building blocks. Fundamentally changing the DWF models is often better done by starting from scratch (however, instead of changing DWF models, they can be extended by further role refinements). The delegation model on the other hand and thus *the context leading to the actual behavior* is prone to quite frequent and light-weight re-organization efforts: adding/removing positions, adding/removing delegation relationships, adding/removing execution etc. Consequently, we prefer a data structure that handles changes easier. In addition, such a data structure is helpful to share (communicate about) and process knowledge in the context of team formation (determining eligibility of task implementations, possible delegation partners or whole sub-teams etc.). To conclude, the pre-processing phase of model deployment comprises two parts.

<sup>5</sup> This is what we did for our previous versions of a SONAR middleware layer, cf. [15].

1. All DWF models are decomposed into their singular role parts, which makes it easy to dynamically compose team DWFs later on.<sup>6</sup>
2. From the delegation model, a Java data structure is generated. Figure 7 shows the according class diagram in UML style. More specifically, it is a *concept diagram* [6] and is supported by the tool suite that we use in the context of our multi-agent framework MULAN that we briefly address in the following section.<sup>7</sup> The class hierarchy resulting from a concept diagram comes with the handy feature that all objects of these classes have FIPA<sup>8</sup>-conform String representations, which allows to directly include them as message contents in agent communication. According to the class hierarchy from Figure 7, each SONAR delegation model is transferred into an *organization object* that contains all other information.

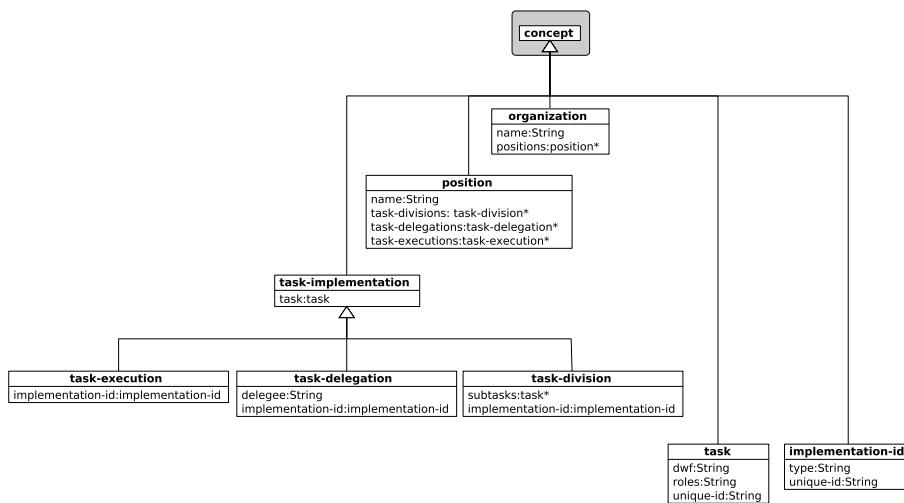


Fig. 7: Concept (or class) diagram for SONAR delegation models

In the second phase of model deployment, the pre-processed models are used by the MULAN4SONAR middleware layer to enable and frame teamwork among members of a SONAR organization. Members are (social or artificial) actors that access the middleware layer and occupy positions of a SONAR delegation model. We will elaborate on the MULAN4SONAR middleware layer and on how to access it in the next section.

<sup>6</sup> It might of course be possible to keep role compositions that always have to appear together in a team DWF, like `supervisePaperSubmission` and `writeIntroAndConclusion` from the running example. But as we intend to have dynamic re-organizations of SONAR models at run-time (see the conclusion), further role refinements might be introduced. Thus it is simpler to keep track of each singular role part in the first place.

<sup>7</sup> see also <http://www.paose.net>.

<sup>8</sup> <http://www.fipa.org>

Basically, the Java data structure of the delegation model is used to manage task delegation and thus the team formation process. As soon as a team is formed, there is a unique team DWF associated with it: It is the composed DWF that consists of the role parts that are implemented by *execute* (instead of *refine*, *split* or *delegate*) transitions during the delegation process. The well-formedness of an overall SONAR model ensures that these role parts fit together. For example, the composed DWF from Figure 8 is the team DWF for a team where the delegation process has lead to the maximum level of task (and thus role) refinement for the running example of joint paper submission.

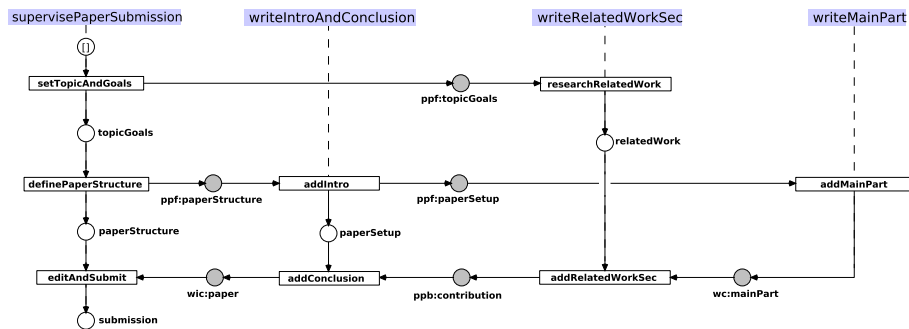


Fig. 8: A multi-role workflow for submitting a paper composed from role fragments

Consequently, team formation leads to an on-the-fly generation of the corresponding team DWF. However, for such a team DWF to be executable in the context of the middleware layer, a further refinement and enrichment has to be carried out. This is also done by automatic generation. Basically, each action transition of a team DWF has to be enriched with execution inscriptions and has to be divided into a *call* and *return* part. Figure 9 exemplifies the substitution rule applied to a DWF transition for the *addConclusion* action of the team DWF from Figure 8. The transition is split into two transitions for call and return. The names of places lead to the generation of variable names that are bound to work-item and result objects of the action. The action call is parametrized with the role name, action name and a set of incoming work-items. The surrounding engine for the execution of team DWFs has to take care of forwarding the call to the position holder that implements this role for this team. In addition, the engine generates a unique action ID that can be used to associate action call and return. The action return is parametrized with a result object. We omit details on handling erroneous or aborted execution of DWF actions here.

Figure 10 shows the class diagram for content objects used in the context of executable team DWFs. More specifically, it shows the generic part. Concrete SONAR models are intended to extend the concept *dwf-action-content* with customized concepts. In fact, we are working on a high-level action inscription language for SONAR DWF models. Such a language can for example be used to attach pre-conditions, post-conditions and effects for/of DWF actions based on

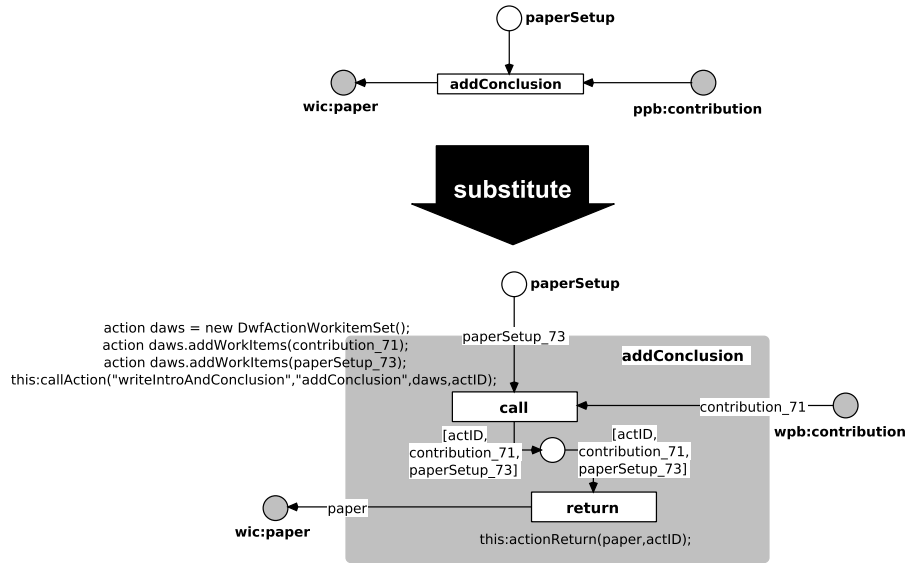


Fig. 9: Substitution rule (by example) for generating executable workflow models

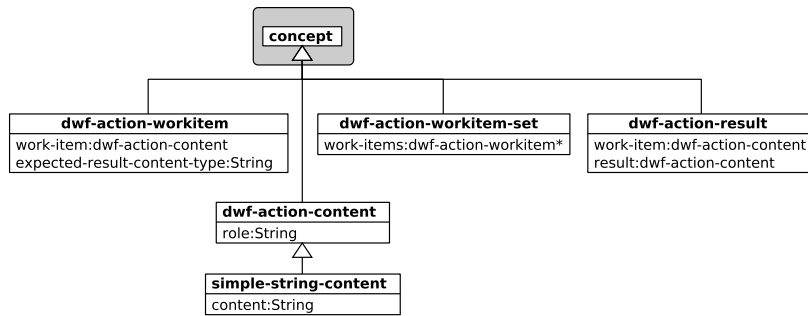


Fig. 10: Concept (or class) diagram for executable team DWF contents

the content objects and their attributes that are involved in the action. For this purpose it is necessary to explicitly define the according custom concepts.

To conclude this section, the illustrative and rather high-level Petri net models that a modeler has to create for a SONAR organization are rich enough to allow the generation of different kinds of executable artifacts for computer-supported teamwork. In the next section, we give an overview of a middleware implementation that utilizes these artifacts.

#### 4 MULAN4SONAR: Agent-Based Teamwork Engine

We present a middleware implementation for teamwork support that is based on SONAR models and their deployment. There exist of course multiple possibilities and here we present our current approach, called MULAN4SONAR. This name

stems from the fact that it is based on the multi-agent system (MAS) framework MULAN (cf. [6] and [www.paose.net](http://www.paose.net)). This framework provides the possibility to combine Java programming with Petri net modeling and simulation for realizing MAS and is consequently perfectly suited for our purpose. More concretely, MULAN relies on the high-level Petri net formalism of *Java reference nets* that is supported by the RENEW tool ([www.renew.de](http://www.renew.de)).<sup>9</sup>

Figure 11 shows our general proposal for multi-agent system deployment of SONAR models. We have an *organization agent* that represents the SONAR or-

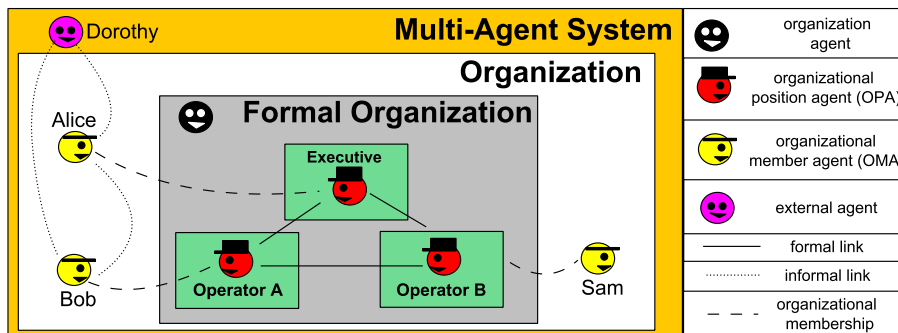


Fig. 11: Basic concept for SONAR-based multi-agent systems

ganization as a whole. It is responsible for initialization and for keeping a global perspective. With each position of a SONAR model we associate one dedicated agent, called an *organizational position agent* (OPA). From a conceptual point of view, the resulting OPA network (together with the organization agent) embodies a *formal organization* as each OPA represents an *organizational artifact* and not a *member/employee* of the organization. From a technical point of view, the OPA network is an agent-based middleware layer for supporting teamwork according to SONAR models. Consequently, each OPA represents a connection point for an *organizational member agent* (OMA). Each OMA interacts with its associated OPA to carry out organizational tasks and to make decisions where required. An OPA both enables and constrains organizational behaviour of its OMA. The OMA can effect the organization only in a way that is in conformance with the OPA's specification. In return, the OPA relieves its OMAs of a considerable amount of organizational overhead by automating coordination and administration. Conceptually speaking, OMAs implement/occupy the formal positions and thus represent the *informal part* of the organization. Note that an OMA can be an artificial as well as a human agent. OMAs might of course only be partially involved in an organization and have relationships to multiple other agents than their OPA or even to agents completely external to

<sup>9</sup> An example for using Java inscriptions in the context of a Petri net model was already shown in Figure 9. The other way round is equally possible, namely having Java objects monitoring, triggering or even controlling parts of the execution (simulation) of a Petri net model.



the organization. From the perspective of the organization, all other ties than the OPA-OPA and OPA-OMA links are considered as informal connections.

Our current implementation of MULAN4SONAR follows this general proposal. However, it does not (yet) feature OPAs as distinct agents. Instead, our current implementation features a central *organization agent* that manages the teamwork processes of a SONAR organization but also utilizes separate *OPA shells* for each position. Consequently, we have already prepared the implementation in way to be able to single out the shells as OPAs and thus to obtain a more distributed implementation. Figure 12 shows the three-level architecture of the organization agent in its current form. This architecture is actually realized based on the

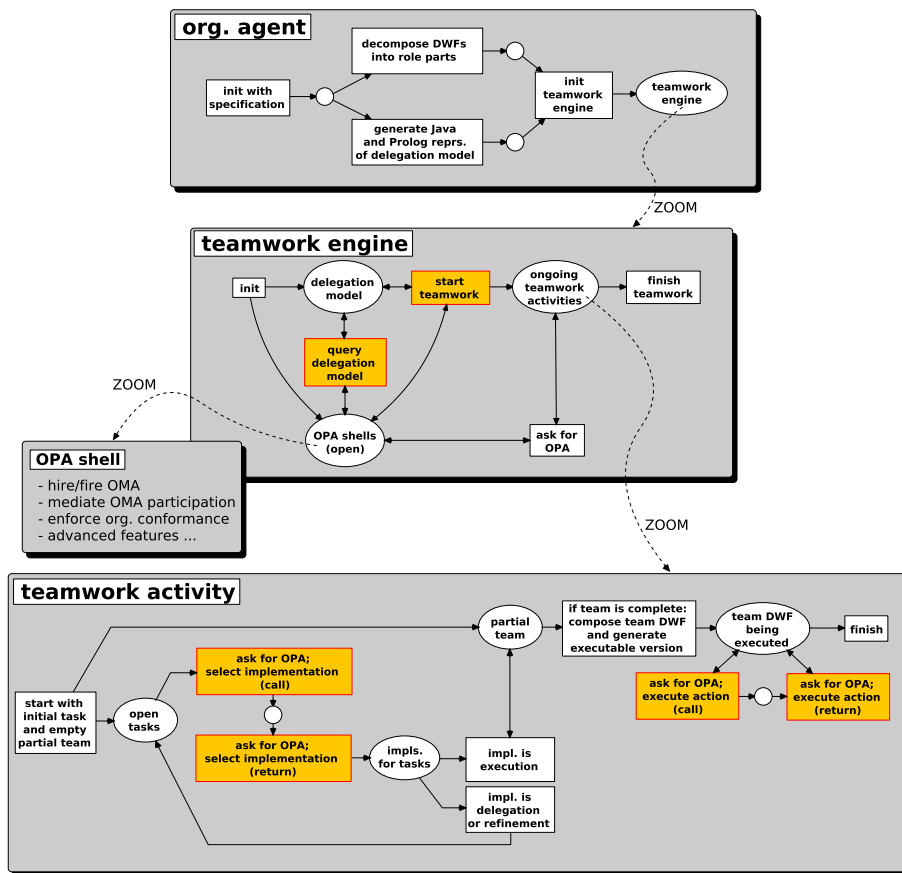


Fig. 12: Three-level architecture of the SONAR teamwork engine agent

high-level Petri net concept of *nets-within-nets* [18] where Petri nets can have other Petri nets as tokens and cross-net communication is possible. Nets-within-nets modeling and simulation is supported by the RENEW tool and is one of the fundamental features of MAS development with MULAN. The figure only shows

a high-level overview of the organization agent's architecture but it represents the actual implementation quite precisely.

On the top level of the *organization agent*, the organization is initialized and the pre-processing of the original SONAR models is carried out, just as described in the previous section. Afterwards, the *teamwork engine* is initialized with the pre-processed models as input and represents the second level.

The teamwork engine sets up *OPA shells* for all positions and makes the delegation model available to them. We will not go into detail concerning the manifold responsibilities of the OPA shells. We just assume that basically, they take care of binding users as members (OMAs) into the organization and manage the inclusion of the OMAs' actions and decisions in conformance to the organizational specifications. New *teamwork activities* can be started and represent the third level. Teamwork activities and OPA shells stay connected via the teamwork engine level.

A teamwork activity basically comprises the delegation process for team (DWF) formation and afterwards the execution of the composed team DWF by the team members that take on roles in the DWF. All of this is managed by the teamwork activity level, together with the involved OPA shells that can be consulted via the joint teamwork engine super-level.

## 5 Conclusion and Outlook

Starting from the question for an integrated treatment of structure and process perspectives in modeling collaborative systems, we have presented our SONAR approach. It provides a way to capture the whole context of team-oriented process management: from the underlying organizational structure over team formation up to process execution by the team. The accompanying models are rather high-level and illustrative but at the same time they are rich enough in order to generate executable models and other kinds of code that together form the core of the MULAN4SONAR middleware implementation for team-oriented process management.

Regarding our future research efforts, there is a range of topics that we and other people from our research group are working on.

- *Collaborative Agent Platform (deployment)*: Our group has the long-term goal of developing an agent-based platform for computer-supported collaboration. We envision SONAR-based organizational models to be used for specific teamwork applications *on top of the platform* as well as for supporting the infrastructure *of the platform itself*, managing the various platform tasks and processes.
- *Self-organization*: Instead of the manager from Figure 1 we promote an alternative approach where a SONAR model has multiple management levels and the team processes on one level lead to the transformation of the specifications of the lower levels, cf. [15].
- *Hierarchy/holism*: While the idea of self-organization introduces multiple management levels in the context of *one* SONAR organization, we also address

the concept of having multiple levels of nested SONAR organizations. The basic idea is to have positions being occupied by *organizational units* that are SONAR organizations themselves. This allows to model inter-organizational scenarios and so-called *multi-organization systems*. The refinement concept for roles and tasks inherent to SONAR directly supports such an extension. For a thorough report of our research on modeling organizational units and multi-organization systems (not limited to SONAR), we refer to [19] (in German).

- *Simulation*: We are also interested in organizational simulation. We intend to enrich the models with quantitative information and apply routines to evaluate simulation runs with respect to certain criteria. Especially in combination with hierarchic models we are interested in studying the fit of different (types of) organizational units to one another (in terms of nesting relationships as well as in terms of cooperation effectiveness on the same level).

## References

1. van der Aalst, W.: Verification of workflow nets. In: Application and Theory of Petri Nets 1997. Lecture Notes in Computer Science, vol. 1248, pp. 407–426. Springer (1997)
2. van der Aalst, W.: Interorganizational workflows. Systems Analysis - Modelling - Simulation 34(3), 335–367 (1999)
3. van der Aalst, W., ter Hofstede, A.: YAWL: Yet another workflow language. Information Systems 30(4), 245–275 (2005)
4. Alves et al., A.: OASIS web services business process execution language (WS-BPEL) v2.0. OASIS Standard, 11. April 2007 (2007)
5. Boissier, O., Hübner, J., Sichman, J.S.a.: Organization oriented programming: From closed to open organizations. In: O’Hare, G., Ricci, A., O’Grady, M., Dikenelli, O. (eds.) Engineering Societies in the Agents World VII. Lecture Notes in Computer Science, vol. 4457, pp. 86–105. Springer (2007)
6. Cabac, L.: Modeling Petri Net-Based Multi-Agent Applications, Agent Technology: Theory and Application, vol. 5. Logos (2010)
7. Desel, J., Esparza, J.: Free Choice Petri Nets, Cambridge Tracts in Theoretical Computer Science, vol. 40. Cambridge University Press (1995)
8. Dignum, V.: The role of organization in agent systems. In: Dignum, V. (ed.) Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models, pp. 1–16. Information Science Reference (2009)
9. Esteva, M., de la Cruz, D., Sierra, C.: ISLANDER: An electronic institutions editor. In: The First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, Proceedings. pp. 1045–1052. ACM (2002)
10. Girault, C., Valk, R. (eds.): Petri Nets for Systems Engineering: A Guide to Modelling, Verification and Applications. Springer (2003)
11. Goltz, U., Reisig, W.: The non-sequential behaviour of Petri nets. Information and Control 57(2–3), 125–147 (1983)
12. Hübner, J.F., Sichman, J.S.a., Boissier, O.: Using the *Moise*<sup>+</sup> for a cooperative framework of MAS reorganisation. In: Bazzan, A., Labidi, S. (eds.) Advances in Artificial Intelligence – SBIA 2004. Lecture Notes in Computer Science, vol. 3171, pp. 481–517. Springer (2004)

13. Keller, G., Nüttgens, M., Scheer, A.W.: Semantische prozessmodellierung auf der grundlage „ereignisgesteuerter prozessketten (epk)“. In: Scheer, A.W. (ed.) Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi), Universität des Saarlandes. Heft 89 (1992)
14. Köhler-Bußmeier, M., Moldt, D., Wester-Ebbinghaus, M.: A formal model for organisational structures behind process-aware information systems. In: van der Aalst, W., Jensen, K. (eds.) Transactions on Petri Nets and Other Models of Concurrency II: Special Issue on Concurrency in Process-Aware Information Systems, Lecture Notes in Computer Science, vol. 5460, pp. 98–115. Springer (2009)
15. Köhler-Bußmeier, M., Wester-Ebbinghaus, M., Moldt, D.: Generating executable multi-agent system prototypes from SONAR specifications. In: de Vos, M., Fornara, N., Pitt, J., Vouros, G. (eds.) Coordination, Organizations, Institutions and Norms in Agent Systems VI. Lecture Notes in Artificial Intelligence, vol. 6541, pp. 21–38. Springer (2010)
16. OMG: Business process modeling notation (BPMN) version 1.0. OMG Final Adopted Specification, Object Management Group (2006)
17. Pynadath, D., Tambe, M.: An automated teamwork infrastructure for heterogeneous software agents and humans. *Autonomous Agents and Multi-Agent Systems* 7(1–2), 71–100 (2003)
18. Valk, R.: Object petri nets: Using the nets-within-nets paradigm. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets: Advances in Petri Nets, Lecture Notes in Computer Science, vol. 3098, pp. 819–848. Springer (2004)
19. Wester-Ebbinghaus, M.: Von Multiagentensystemen zu Multiorganisationssystemen – Modellierung auf Basis von Petrinetzen. Dissertation, Universität Hamburg, Fachbereich Informatik. Elektronische Veröffentlichung im Bibliothekssystem der Universität Hamburg: <http://www.sub.uni-hamburg.de/opus/volltexte/2011/4974/> (2010)
20. Wolf, K.: Does my service have partners? *Transactions on Petri Nets and Other Models of Concurrency II, Special Issue on Concurrency in Process-Aware Information Systems* 5460, 152–171 (2009)

# From Code to Coloured Petri Nets: Modelling Guidelines

Anna Dedova and Laure Petrucci

LIPN, CNRS UMR 7030, Université Paris XIII  
99, avenue Jean-Baptiste Clément, F-93430 Villetaneuse, France  
{Anna.Dedova,Laure.Petrucci}@lipn.univ-paris13.fr

**Abstract.** This paper presents a method for designing a coloured Petri net model of a system starting from its high-level object oriented source code. The entire process is divided into two parts: grounding and code analysis. For each part detailed step-by-step guidelines are given. The approach is illustrated with an industrial application case study, the NEO protocol.

## 1 Introduction

The modelling problem has been being under investigation for many years. It has a lot of particular cases depending on 1) the nature of the description of the system to be modelled and 2) which formalism is chosen for the final model. According to the first criteria there are three basic groups of modelling approaches:

1. Starting from an informal description of a problem;
2. Starting from a detailed specification of a system;
3. Starting from the source code.

Some recent works tackle the first group of approaches. For example, in [3] the authors propose a modular design method and illustrate it on a model railway case study. One of the main points of [3] is using properties of the system at the modelling stage. In [4] an approach aggregating different views of the system is given. This method assumes that the system can be observed from several points of view: pre/post, process and lifeline views expressing respectively pre- and post-conditions of events, sequences of events, and sequences of states. Thus, steps in a process view correspond to system events and can be modelled by transitions in a Petri nets formalism. Similarly, steps of a lifeline view correspond to the states of the system and can be modelled by places of a Petri net. Then, by identifying the elements of these different representations of systems, places and transitions are glued together in order to get a complete Petri net.

The second group of modelling approaches includes various attempts to deliver a formal model from UML diagrams [8, 6, 5]. The advantage of these methods is that most developers are familiar with the UML and an automatic transformation of their diagrams into formal models and model-check them, would

greatly simplify the software quality control. The difficulty is that UML diagrams allow for much more freedom for the designer than formal models and the automatic translation is not trivial.

This paper addresses the third group of modelling approaches, which is not covered by a wide range of methods in the literature [9]. Such approaches are dedicated to systems for which the source code already exists, in order to guarantee it satisfies some requirements. They often do not support a complete language, but are restricted to some subset of it. Moreover, to the best of our knowledge, no work addressed a high level object oriented language, such as python.

Hence, what are the particular difficulties encountered by reverse-engineering from the source code? If a program is rather small (tens of lines) one can simply suppose that the operators are the system events and correspond to transitions, places between them model the intermediate states of the system, and some additional places model the states of variables used. But this approach is no more applicable when the system under consideration is as large as 3 MBytes of object oriented code. Of course it is possible to model all operators as in the previous case, but then the model becomes so huge that there is no means to analyse it and it becomes useless. Thus, it is necessary to choose an appropriate *level of abstraction* for the system. If it is too low and the model contains too many details, the same problem as above arises. If the level of abstraction is too high, there are too many hypotheses and assumptions and it may happen that nothing is left worth checking. The model is then trivial and its behaviour is completely correct while the system contains drawbacks that are hidden due to the modelling assumptions.

The paper is organised as follows. Section 2 gives detailed guidelines on how to derive step-by-step a coloured Petri net from the source code. Then Section 3 shows how this method was applied in practice to the NEO protocol. Finally, Section 4 draws some conclusions.

We assume the reader is familiar with coloured Petri nets [7].

## 2 Modelling Guidelines

This section discusses the guidelines to follow in order to deliver a coloured Petri net from high level object oriented source code.

### 2.1 Grounding

Before the start of the modelling process some preparation work is required. It mainly concerns the deep understanding of project structure and expected properties of the system. This helps a lot during the modelling by saving the time devoted to the consideration of unnecessary elements or restructuring model hierarchy. It is always possible to skip this stage and proceed directly to modelling the most interesting piece of code, but then the risk of choosing an inadequate abstraction level is very high. The main steps of grounding are listed below. They

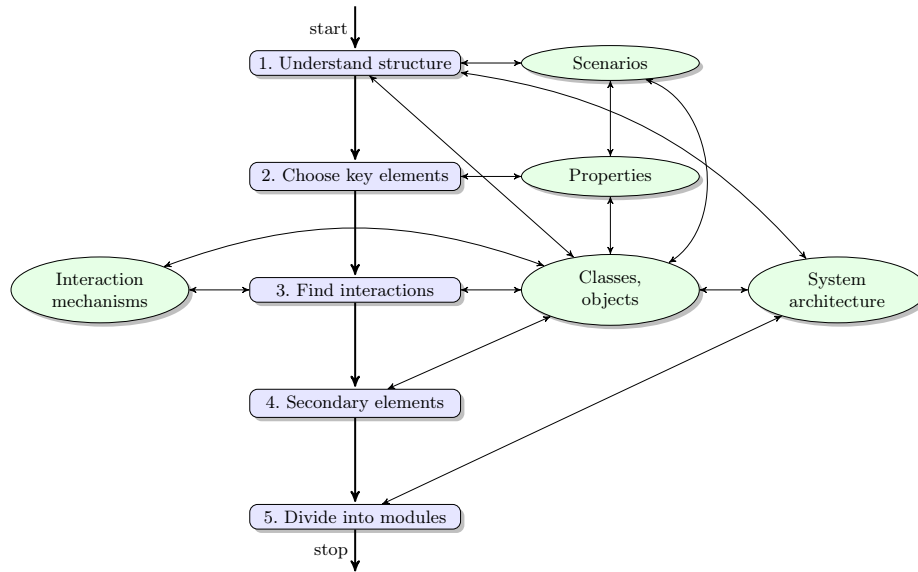


Fig. 1. Schema of the grounding process

are depicted in Figure 1 together with the elements that are to be considered or impacted.

1. **Understand the structure.** First of all, we should pay attention to the architecture of the project. The key elements (classes) should be found as well as their roles in the whole system. It can be very useful to find the most common scenarios of the system use (or maybe scenarios that should be verified later). We can look for the parts (classes or objects) of the system that are impacted by these scenarios. We also need to understand the class structure of the project (with particular attention to inheritance and polymorphism). During this step the most important result is to understand globally how the system works from the inside.
2. **Choose key elements.** The second step focusses on the system properties to be checked. Properties can be proposed by developers, clients or anyone else. Then, they must be considered one by one in order to choose those that are the most crucial for the system. Selecting them before starting the modelling process is very important since this choice can influence a lot the model structure that will not be so easy to change later on. Once the properties are selected, we look for the scenarios they concern. Moreover the classes and methods used within these scenarios are selected, according to the project structure from the previous step. Thus, the main pieces of code that are going to be modelled are defined.
3. **Find interactions.** We should keep in mind that objects of chosen classes can be verified separately from one another. But the ultimate goal is usually

to model-check the whole system altogether. Separate parts can be subjected to traditional testing techniques while the complexity and the size of the system makes their application to the entire project impossible.

So, when modelling something larger than an isolated object, the interactions between them must be identified. They can be of very different natures: message passing; shared or global variables (e.g. between different methods inside a class); sometimes a class is composed of other auxiliary classes; a method of an external class can be called. All interactions should be investigated and the corresponding elements added to our modelling selection.

4. **Secondary elements.** Here we need to look at auxiliary classes that are used by the selected key classes. They can be classes of data structures, or classes providing message exchange capabilities. On the one hand, operations and/or interconnections of key elements are impossible without them. On the other hand, if we model them in detail, the model will be too bulky to perform any analysis. Such elements usually describe the work of the system on a low level of abstraction and can be verified separately. So, the idea is to model them as simple as possible, but without loss of essence.

In the end of this step we should know which abstractions are going to be used: some algorithms could be modelled as a single transition, some complex data structures encoded with natural numbers, etc.

5. **Divide into modules.** All the scenarios and methods that have been chosen for modelling are used to design a modular structure for the future model. Of course, it can be changed later during the modelling process.

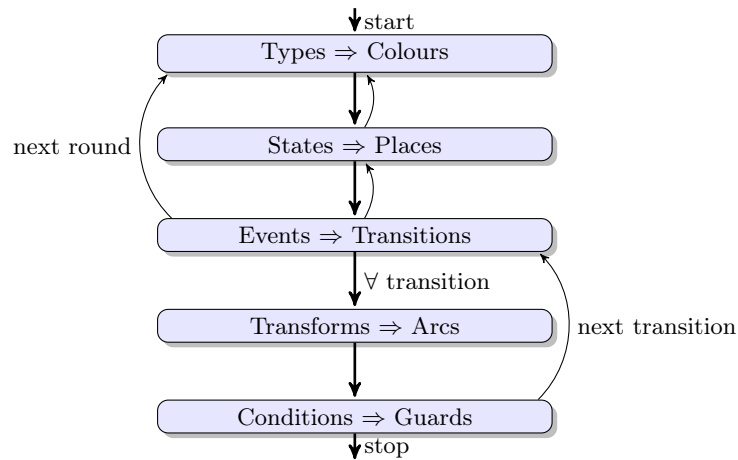
It is rather natural to associate a submodule with a class or a method. It is also important to pay attention to auxiliary elements and decide whether they are worth a separate module or not.

## 2.2 Code analysis

During this stage, two processes are carried out in parallel: the analysis of the source code and the construction of the model. In order to streamline these processes it is proposed to divide them into five main activities, according to the scheme on Figure 2. Each activity requires to look for some elements in the source code as well as interpreting them in terms of the modelling formalism (in our case coloured Petri nets). At each step, the source code is looked at from different points of view in order to extract the different components of the model. In practice, it is usually needed to go through the cycle several times but at the start it is hard to tell how many times it should be done. It is also possible that some activities are skipped on later rounds, since a new element cannot be extracted from the source code. From one round to another the understanding of the chosen abstraction level is more and more accurate and the model is more and more complete.

Since the module hierarchy of the CPN can be different from the initial structure of classes and methods, the work within the five activities can be organised in different ways.





**Fig. 2.** Schema of the modelling process

- Consider the modules of the future model (found at the fifth step of grounding) one by one. For each module examine scenarios, classes and methods it concerns and analyse them via all activities.
- Consider scenarios or methods (found at the second step of grounding) one by one. For each scenario/method perform each activity that will give refinement for different modules of the model.
- Consider activities one by one and look at the system as a whole, analysing different parts of code and changing different models, but from the chosen point of view.

In practice third approach is difficult to apply unless the model is almost ready and it can be grasped at a glance. The first approach is the most effective one, but sometimes the second one may also prove useful by focusing on a particular behaviour. In this case, the behaviour is either described by an execution scenario, or the details of a method are tracked step-by-step.

**Data structures** It is important to start from this activity because it forms the basis of the future model. It is natural to start with colour domains in order to use them (and may be supplemented later with new details) during further activities.

In general, data structures of the source code *should* be expressed in terms of colour domains. However, it is often not that simple. In object oriented code, data structures are usually integrated together with their storing, loading and treatment methods. Colour domains syntax does not allow to do this, so, it may be needed to model a “simple” object with a separate CPN. Such cases can be left to further activities nevertheless providing basic types for future CPNs.

This phase provides as a result a preliminary list of colour domains and variables needed in the model.

**States and conditions on objects** This is the first activity that assumes the modeller thinks in terms of parts of CPN that have no strict correspondence in the source code, namely the places. It may be difficult to deliver them in the situation of “blank sheet”, but the model with places make other activities become much simpler or even possible (e.g. construction of arcs).

Hence, this phase aims at creating the set of places of the CPN which usually represent the states of the system or its parts (objects, variables, etc.). To begin with, the system flow of operations can be represented as a finite state automaton. The set of states of this automaton can be a first approximation of the set of places of the CPN. Then conditions required to proceed from one state to another are considered. These conditions often concern the states of some objects or variables. They should also be added to the set of places. Finally, a colour domain (defined during the previous activity) is associated with each place.

**Events and actions** This activity is in general simpler than the previous one. Each operator or method call in the source code can be considered as an action and thus be modelled as a transition. The main hindrance here is a tendency to model every operator with a transition. To avoid this we can apply information obtained during grounding (2nd and 4th steps).

The purpose is to select actions, essential for the processes to be verified. To start with, consider the changes of variables and data structures that are implicitly mentioned in the properties. If the properties are not formalised yet, main constructions of the system can serve as a basis. As for previous phases, on the first round only a preliminary view of transitions in the model can be given. After going through other activities it will be completed and refined.

**Transformation of data** During this phase, the modeller considers for each transition the three following questions, and performs the corresponding net construction:

1. What is taken as input? (Connect corresponding places with input arcs);
2. What is produced as output? (Connect corresponding places with output arcs);
3. How are the tokens transformed? (Provide input and output arc expressions).

If there is a special input format, it can be reflected in the input arc inscription. If the output is somehow calculated from input variables, the corresponding output arc must be assigned with a formula, representing these calculations in terms of CPN. Often, the formulae from the source code cannot be applied directly and need to be adapted w.r.t. the chosen level of abstraction.

**Conditions on events** Here, as in the previous phase, we consider the set of transitions. The focus this time is on the special conditions under which a transition can be fired. In practice the conditions for most transitions are modelled by the tokens in input places. In this case the transition has a guard `true` that can be omitted in the model. But sometimes for better readability of the model, and also to prevent having too large sets of places and transitions, it can prove better to formulate such a condition as a guard of the transition.

The goal of this activity is to find such cases and to figure out the guards. It can happen that some condition is not possible to express on the selected level of data abstraction. If so, the colour domains created in the first activity must be revised, as well as their occurrences in parts of the CPN that have already been built. Thus, a next round of activities is started.

### 3 Application of the Guidelines to the NEO Protocol

This section illustrates modelling guidelines with examples from modelling process of the NEO protocol. The protocol, designed to handle a large distributed database over a cluster of machines, was described in [1, 2]. Its main characteristics are shortly summarised in Section 3.1. This specification was part of an industrial project which aimed at validating the protocol and its prototype implementation both designed and developed by the NEXEDI company. It was implemented in Python, but our approach is not specific to this language.

#### 3.1 Brief Description of the NEO Protocol

A more extensive description and analysis of the NEO protocol can be found in [1] and [2].

Different kinds of nodes play dedicated roles in the protocol, as depicted by the architecture in Figure 3(a):

**storage nodes** handle the database itself. Since the database is distributed, each storage node cares for a part of the database, according to a *partition table*. To avoid data loss in case of a node failure, data is duplicated, and is thus handled by at least two storage nodes.

**master nodes** handle the transactions requested by the client nodes and forward them to the appropriate storage nodes. A distinguished master node, called *primary master*, handles the operations. *Secondary masters* (i.e. the other master nodes) are ready to become primary master in case of a failure of this node. They also inform other nodes of the identity of the primary master.

**the administration node** is used for manual setup if needed.

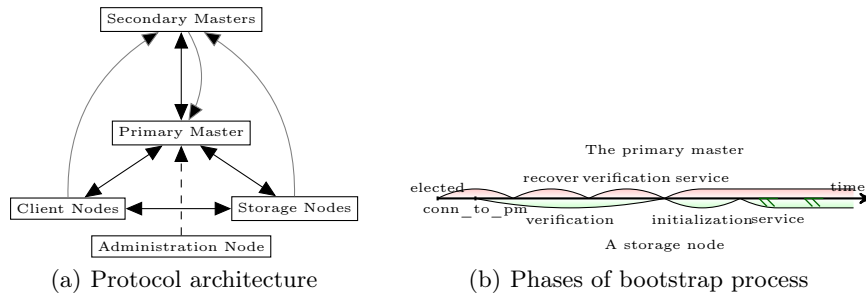
**client nodes** correspond to the machines running applications concerned with the database objects. Thus, they request either read or write operations. They first ask the primary master which storage nodes are concerned with their data, and can then contact them directly.

The lifecycle of the NEO rotocol is depicted in Figure 3(b).

At the system startup, the primary master is *elected* among all master nodes. The primary master maintains the key information for the protocol to operate.

After the election of a primary master, the system goes through various stages with the purpose of checking that all transactions were completely processed, and thus that the database is consistent across the different storage nodes (*bootstrap* protocol).

Finally, the system enters its *operational state*. Clients can then access the database through the elected primary master.



**Fig. 3.** Architecture and lifecycle of the NEO Protocol

### 3.2 Grounding

Each step described in Section 2 is now applied.

*Understand structure* This step is difficult to illustrate on a real example since it implies working on extensive code. The conclusions cannot be confirmed by a small piece of code. Nevertheless, for the NEO protocol, at this stage we can state the following, and confirm the brief description from Section 3.1.

The main entities are nodes of the cluster, of four types: master, storage, client and admin nodes. For each of these types there is a corresponding function in the source code.

The life cycle of nodes leads them through different phases implemented by an auxiliary class (RecoveryManager, VerificationManager) or a method of the corresponding node class (ElectPrimay, VerifyData, Initialize, DoOperation). Also, depending on the phase of the protocol, a node changes its message handlers.

*Choose key elements* Based on the conclusions of the previous step and on the verification issues, we decided to focus on master and storage nodes. This paper does not get into the details of the numerous properties to check, a good part of which can be found in [1] and [2]. Many properties were provided as an informal statement by the code developers. For example, only a single node is elected as

a primary master ; all shared information (partition tables, identifiers) has been made consistent for the service phase to take place.

Most attention is paid to the election of the primary master and to the bootstrap process (everything between election and operational state). Later on in this paper we focus on bootstrap phase.

Therefore, we chose the following fragments of code for detailed analysis:

1. master node application

```
1 def __init__(self, config)
2 def run(self)
3 def playPrimaryRole(self)
4 def runManager(self, manager_class)
5 def changeClusterState(self, state)
```

2. recovery manager class

```
1 def __init__(self, config)
2 def run(self)
3 def buildFromScratch(self)
```

3. verification manager class

```
1 def __init__(self, app)
2 def _askStorageNodesAndWait(self, packet, node_list)
3 def run(self)
4 def verifyData(self)
5 def verifyTransaction(self, tid)
```

4. storage node application

```
1 def __init__(self, config)
2 def run(self)
3 def connectToPrimary(self)
4 def verifyData(self)
5 def initialize(self)
```

*Find interactions* The nodes in the cluster need to communicate with one another. For this purpose they use a class `EventManager`. It describes the mechanism for sending and receiving messages. To treat them, each node has its own handlers, different for different phases of the protocol. Thus, they should be added to our list of pieces of code.

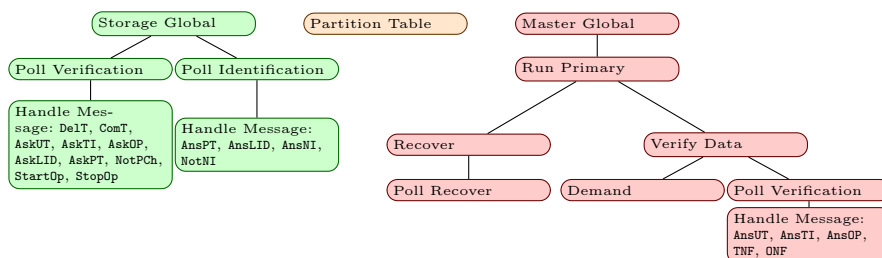
Another means for nodes collaboration in the cluster is a partition table. It is implemented as a class that keeps the distribution of data between storage nodes. This class is another key element and should also be added to the analysis list.

The master and storage applications also use some global variables to allow their methods to know the state of the application (primary, operational, has\_pt — partition table). This information should be kept aside to be used during modelling.

*Secondary elements* When this stage occurs, all significant parts of the project and their communications are identified. It is then time to make rather crude abstractions on objects that could not be eliminated from the model, but must be simplified because of the abstraction level.

For example, for the NEO protocol we made following abstractions:

1. The complex message structure, defined in a class package is modelled as an integer number;
2. Connection, described as a group of classes, is modelled as a pair of nodes, that are considered to be connected;
3. Transaction and object, that have a lot of fields, such as serial number, history, data, etc., are modelled by their identification numbers.



**Fig. 4.** Hierarchy of models

*Divide into modules* Figure 4 presents the sub-module structure of the bootstrap model. First of all, there are two main global level models: “Storage Global” and “Master Global”. Aside, a “Partition Table” model is referenced by the both storage and master nodes.

Then, in “Storage Global” there are two substitution transitions, corresponding, respectively, to “Poll Verification” and “Poll Identification” sub-modules, and presented by two instances of “Poll Storage”. Each of them contains a “Handle Message” sub-model, but with different messages inside.

In the model “Master Global” there is only one sub-module “Run Primary”. It itself has two sub-nets: “Recover” and “Verify Data”, that correspond to recovery and verification managers run methods. The “Recover” module contains one sub-module, “Poll Rec” together with the handlers of two messages. The “Verify Data” module contains two sub-models occurring several times in the model : “Demand” (twice) and “Poll Verification” (four times). “Poll Verification” includes a “Handle Message” with different message handlers.

### 3.3 Code Analysis

In this subsection we will give some detailed examples of application the guidelines to the source code of the NEO protocol.

*Data structures* In order to show how the colour domains can be constructed from data structures, let us take the piece of code in Figure 5. It is a fragment of the partition table class definition where the internal fields are declared. First,

```

1 class Cell(object):
2     def __init__(self, node, state = CellStates.UP_TO_DATE):
3         self.node = node
4         self.state = state
5     ...
6
7 class PartitionTable(object):
8     def __init__(self, num_partitions, num_replicas):
9         self._id = None
10        self.np = num_partitions
11        self.nr = num_replicas
12        self.num_filled_rows = 0
13        self.partition_list = [[] for _ in xrange(num_partitions)]
14    ...

```

**Fig. 5.** Fragment of the source code declaration of partition table class

the class for a cell of the partition table is declared. It has two attributes: a storage node and a state. Knowing that a partition cell has two possible states, we can declare a colour domain `PSTATE` as a set of these two values and a colour domain `PT_CELL` as a product of storage node type and cell state.

```

colset PSTATE = with UTD | OOD; (* the set of states of partition *)
colset PT_CELL = product SN*PSTATE; (* a cell of partition table *)

```

Now let us consider the beginning of the partition table class constructor. It starts with assigning the values of variables for number of replicas and number of partitions. Then it creates a two-dimension list. In one dimension its size is equal to the number of partitions, in the other dimension the size is not specified. So, we declare three auxiliary colour domains: a set of partitions, a list of partition cells and a partition row, that is a product of a partition and a list of cells. Finally, a partition table colour domain consists of a list of partition rows.

```

colset PART = index np with 0..NP (* the set of partitions *)
colset PT_CELLlist = list PT_CELL with 0..NR+1; (* a cell list *)
colset PT_ROW = product PART*PT_CELLlist; (* a partition table row *)
colset PT = list PT_ROW with 0..NP; (* the partition table type *)

```

The following colour domain definitions will be used later on:

```

colset SN = index sn with 0..N; (* the set of storage nodes *)
colset MN = index mn with 0..M; (* the set of storage nodes *)
colset NODE = union s1:SN + m1:MN; (* the set of all nodes *)
colset CSTATE = with VER | REC | RUN | STP; (* the set of cluster states *)
colset MTYPE = with StopOp | StartOp | AskUT | AskPT | AskNI | AskLID |
                AskTI | AskOP | AnsUT | AnsNI | AnsPT | AnsLID |
                AnsTI | AnsOP | NotNI | NotPCh | DelT | ComT;
                (* the set of message types *)
colset MESS = product MTYPE*NODE*NODE*INT; (* the set of messages *)
colset SNlist = list SN with 0..N; (* a list of storage nodes *)
colset MESSlist = list MESS 0..1000; (* a list of messages *)

```

*States and conditions on objects* As an illustration of the next four steps, let us consider the beginning of the verification phase from the primary master point of view. The corresponding source code is listed in Figure 6.



```

1 def run(self):
2     self.app.changeClusterState(ClusterStates.VERIFYING)
3     self.verifyData()
4     ...
5
6 def verifyData(self):
7     em, nm = self.app.em, self.app.nm
8     neo.lib.logging.debug('waiting_for_the_cluster_to_be_operational')
9     while not self.app.pt.operational(): em.poll(1)
10    neo.lib.logging.info('start_to_verify_data')
11    self._askStorageNodesAndWait(Packets.AskUnfinishedTransactions(),
12    [x for x in self.app.nm.getIdentifiedList() if x.isStorage()])
13    ...
14
15 def _askStorageNodesAndWait(self, packet, node_list):
16    poll = self.app.em.poll
17    operational = self.app.pt.operational
18    uuid_set = self._uuid_set
19    uuid_set.clear()
20    for node in node_list:
21        uuid_set.add(node.getUUID())
22        node.ask(packet)
23    while True:
24        poll(1)
25        if not uuid_set:
26            break


```

Fig. 6. Fragment of the source code of verification phase



First three lines come from the run method of the verification manager class. We can see that the primary master changes cluster state to VERIFYING and calls verifyData method. So, we can start by defining two places:

-  *start\_verif* with colour domain MN (the state of the primary master at the start of the verification manager);
-  *c\_state* with colour domain CSTATE (the current state of the cluster).


Then, the primary master waits until the partition table becomes operational (line 9). We define a new place, corresponding to this state of the primary master.

-  *wait\_pt* with colour domain MN.

After that, it calls a method `_askStorageNodesAndWait`, where it sends requests about unfinished transactions to storage nodes, and waits until the list `uuid_set` becomes empty. This waiting period can be modelled as a new place. In order to send messages to other nodes we need a channel place. According to the protocol, it must be a FIFO list. Hence, two places are added:


-  *network* with colour domain MESSlist;
-  *wait\_ut* with colour domain MN.

Finally, the primary master starts the verification of transactions one by one. This code is out of scope of our example, but we can at least give the next state of the primary master by adding a new place:


-  *verifying\_trans* with colour domain MN.





*Events and actions* Now, we need to extract the important actions from the same piece of code. The first method call `changeClusterState` can be considered as one of them. So, we add the first transition:

–  *change\_c\_state*.


When getting to the next lines, we see that line 7 contains nothing but shortcuts and line 8 writes the current state to the log. The next important action is `em.poll(1)` that is executed while the primary master waits for the partition table to be operational. Here, it is supposed to treat different messages. For the sake of readability of the model, we decide to organise message handlers in separate sub-nets. A new transition is added, coloured in black to symbolise there is a net behind.

–  *poll\_pm\_verif*.

Line 10 is not important since it writes a log. Then the `_askStorageNodesAndWait` method is called with a list of all identified storage nodes as an argument. Inside this method some shortcuts occur (lines 16–18) and the list is cleared `uuid_set`. Then, considering the storage nodes from the input list one by one, they are added to `uuid_set` and sent a request of unfinished transaction (which is also given as input parameter, defined in line 11). An additional place is needed to store the identified storage nodes. So, we go to the previous step, add this place, and return to add a new transition:

–  *s\_iden* with colour domain SN.  
–  *ask\_ut*.

Then the primary master is waiting once again, executing `poll` (line 24). So, we duplicate the corresponding transition. Finally, we add a transition that models the exit of this process.

–  *got\_ut*.

*Transformations of data* Let us consider the transitions we have up to now one by one in order to build arcs and provide their expressions. To do so, some variables should first be declared. Let `pm`: MN; `cst`: CSTATE. The whole net can be seen in Figure 7. Transition *change\_c\_state* moves the primary master token from place *start\_verif* to *wait\_pt* and replaces current cluster state token with a VERIF one.

Transition *got\_ut* moves the primary master from place *wait\_ut* to *verifying\_trans*. But it can fire only when all the answers of storage nodes are received. Here an additional place, similar to the variable `uuid_set` (line 25), is created, that will contain all storage nodes, answers from which the primary master is waiting. Transition *got\_ut* must fire if and only if this place is empty. However, it is not possible to check if the multiset is empty without inhibitor arc. One of the solutions is to change the colour domain to `SNlist`, since a list can be checked for emptiness.

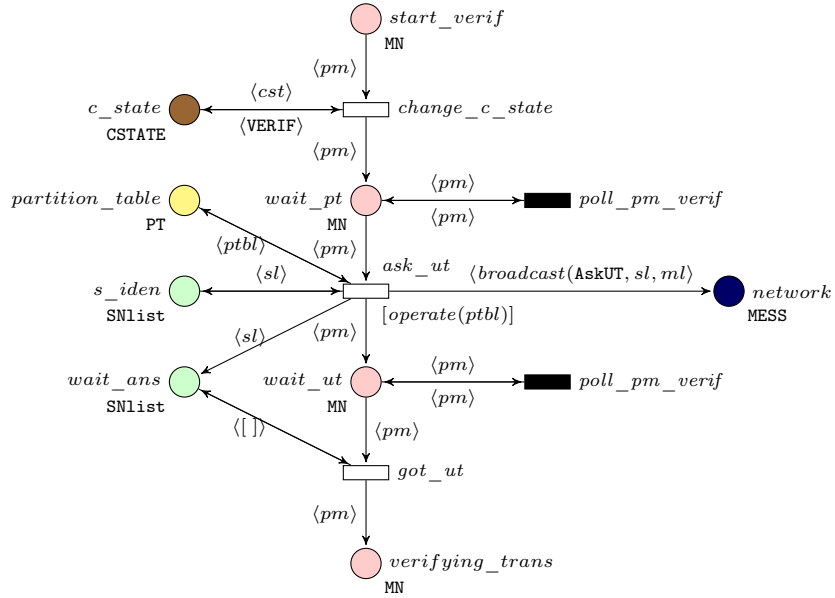


Fig. 7. Model of verification manager - part 1

- *wait\_ans* with colour domain SNlist.

A new variable *sl*: SNlist is also declared.

Transition *poll\_pm\_verif* is replaced by a sub-net. Here it simply takes the primary master token and puts it back. Handlers of messages, that are hidden behind them, can change the state of some variables, e.g. *uuid\_set*, and, respectively, the content of place *wait\_ans*.

Transition *ask\_ut* moves the primary master token from place *wait\_pt* to *wait\_ut*. It also sends messages to all storage nodes from place *s\_iden*. Here we see that it could be convenient to change the colour domain of *s\_iden* to SNlist. In this case we can directly put this list into place *wait\_ans*. Also we can write an SML function *broadcast*, that sends the same message to each node from the list.


```

fun broadcast (msgType, l) =
  List.foldr (fn (sNode, tokens) =>
    1'(msgType, sl (sNode), pm, 0) ++ tokens) [] l

```

A new variable declaration is needed: *ml*: MESSlist.

*Conditions on events* In this example, there is only a single transition that requires an auxiliary condition to fire. It is *ask\_ut*, since it can fire only if the partition table is operational (see lines 9, 11, 22 of figure 6). To make this check, first of all, we need to add an additional place:

–  *partition\_table* with colour domain PT,

together with a new variable `ptbl`: PT. A guard must also be added. The partition table is operational if and only if there is at least one up-to-date cell for each partition. So, we can write the following SML function.

```
1 fun operational pt = List.all (fn (_, row) =>
2   List.exists (fn (_, st) => st = UP) row) pt
```

### 3.4 Analysis and feedback

The properties the protocol should satisfy were model-checked. The outcome of this analysis was suspicious scenarios. The design approach allowed for tracing back the execution sequence in the source code, and thus the engineers could check their validity.

Some scenarios were due to a too coarse abstraction level, but the assumptions made during modelling did hold and guarantee the appropriate behaviour of the code. An interesting erroneous scenario pointed out the possibility of a livelock in the primary master election process. However, this never happened in practice, as the developers found out it was prevented by a side-effect of a Python function. Nevertheless, they could fix it, such a side effect being undesirable, in case it doesn't happen in a future version of Python.

## 4 Conclusion

In this paper we gave detailed directions on how to construct a coloured Petri net model from a high level object oriented source code and illustrated it with a real case example. The modelling process is divided into 2 main parts: grounding and code analysis.

Now the following questions could be raised. Can this process be automated? The most complicated part of the modelling process is to choose objects and actions that are important for the goals of verification and separate them from those that are not as useful. If a programming language could provide some kind of priorities to data structures and methods, it may simplify the automation process. During the industrial project in which the NEO protocol was analysed, a code-tagging approach to facilitate both the modelling and the interpretation of the verification results was envisioned for future work. Moreover, part of our group works on a tool-supported Petri net model design method from a natural language description. Further work could include the integration of both approaches.

Another interesting question is could these modelling guidelines be applied elsewhere? Even if not directly, but with some refinements, they could be applied to any reverse-engineering process providing a coloured Petri net or a similar model of concurrency.

## References

1. C. Choppy, A. Dedova, S. Evangelista, S. Hong, K. Klai, and L. Petrucci. The NEO protocol for large-scale distributed database systems: Modelling and initial verification. 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 145–164. Springer Verlag, June 2010.
2. C. Choppy, A. Dedova, S. Evangelista, K. Klai, L. Petrucci, and S. Youcef. Modelling and formal verification of the NEO protocol. *Transactions on Petri Nets and Other Models of Concurrency*, 2012. To appear.
3. C. Choppy, L. Petrucci, and G. Reggio. A modelling approach with coloured Petri nets. In *Proc. 13th International Conference on Reliable Software Technologies / ADA-Europe, Venice, Italy, LNCS 5026*, pages 73–86. Springer-Verlag, 2008.
4. J. Desel and L. Petrucci. Aggregating views for Petri net model construction. In *Proc. workshop on Petri Nets and Distributed Systems (PNDS08, associated with Petri Nets 2008), Xi'an, China*, pages 17–31, 2008.
5. U. Farooq, C. P. Lam, and H. Li. Transformation methodology for uml 2.0 activity diagram into colored petri nets. In *Proceedings of the third conference on IASTED International Conference: Advances in Computer Science and Technology, ACST'07*, pages 128–133, Anaheim, CA, USA, 2007. ACTA Press.
6. Elhillali Kerkouche, Allaoua Chaoui, El-Bay Bourennane, and Ouassila Labbani. A uml and colored petri nets integrated modeling and analysis approach using graph transformation. *Journal of Object Technology*, 9(4):25–43, 2010.
7. Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.*, volume Basic Concepts. Springer-Verlag, 1992.
8. John Anil Saldhana and Sol M. Shatz. Uml diagrams to object petri net models: An approach for modeling and analysis. In *In International Conference on Software Engineering and Knowledge Engineering*, pages 103–110, 2000.
9. JB. Voron and F. Kordon. Transforming Sources to Petri Nets : A Way to Analyze Execution of Parallel Programs. In *International Workshop on Petri Nets Tools and Applications (PNTAP)*, pages 1–10. ACM, 2008.

# Hierarchy of persistency with respect to the length of action's disability

Kamila Barylska<sup>1\*</sup> and Edward Ochmański<sup>1,2</sup>  
{khama,edoch}@mat.umk.pl

<sup>1</sup> Faculty of Mathematics and Computer Science, Nicolaus Copernicus University,  
Chopina 12/18, 87-100 Toruń, Poland

<sup>2</sup> Institute of Computer Science, Polish Academy of Sciences,  
Jana Kazimierza 5, 01-248 Warszawa, Poland

**Abstract.** The notion of persistency, based on the rule "no action can disable another one" is one of the classical notions in concurrency theory. We recall two ways of generalization of this notion: the first is "no action can kill another one" and the second "no action can kill another enabled one". Afterwards we present an infinite hierarchy of computations in which one action disables another one for no longer than a specified number of steps ( $e/l$ -persistency). We prove that if an action is disabled, and not killed, by another one, it can not be postponed indefinitely. Finally we deal with decision problems about  $e/l$ -persistency. We show that this kind of persistency is decidable with respect to steps, markings and nets.

**Keywords:** Petri nets, concurrency, persistency, decision problems

## 1 Introduction

The notion of persistency is one of the most frequently discussed issues in the Petri net theory - [4,7,8,11,12] and many others. It is being studied not only in terms of theoretical properties, and also as a useful tool for designing and analyzing software environments [3]. In software engineering, persistency is a desirable property and many systems may not work properly without it.

We say that an action of a processing system is persistent if, whenever it becomes enabled, it remains enabled until executed. A system is said to be persistent if all its actions are persistent. This classical notion is introduced by Karp/Miller [10]. Also interesting, with practical applications, is the notion of weak persistency introduced and investigated in [16,15,9]. In section 3, we bring to mind two generalizations of the classical notion (defined in [2]):  $l/l$  persistency and  $e/l$ -persistency. An action is said to be  $l/l$ -persistent if it remains live until executed, and is  $e/l$ -persistent if, whenever it is enabled, it cannot be killed by

---

\* The study is cofunded by the European Union from resources of the European Social Fund. Project PO KL "Information technologies: Research and their interdisciplinary applications", Agreement UDA-POKL.04.01.01-00-051/10-00.

another action. For uniformity, we name the traditional persistency notion  $e/e$ -persistency. Next, we recall that those kinds of persistency are decidable in the class place/transition nets.

In section 4, we extend the hierarchy mentioned above with an infinite hierarchy of  $e/l$ -persistent steps. A step  $MaM'$  is said to be  $e/l-k$ -persistent for some  $k \in \mathbb{N}$  if the execution of an action  $a$  pushes the execution of any other enabled action away for at most  $k$  steps. We prove that if an action is disabled by another one, it can not be postponed indefinitely. We show that if a p/t-net is  $e/l$ -persistent, then it is  $e/l-k$ -persistent for some  $k \in \mathbb{N}$  (Theorem 3) and such a  $k$  can be effectively found (Theorem 9). We also point, that the above-cited result does not hold for nets which do not have the monotonicity property (for example for inhibitor nets).

Afterwards, we investigate the set of markings in which two actions are enabled simultaneously, and also the set of reachable markings with that feature. We show that the minimum of the latter is finite and effectively determined. We also prove that if some action pushes the enabledness of another one away for more than  $k$  steps, then it also needs to happen in some minimal reachable marking enabling these two actions.

Finally, we show that  $e/l-k$ -persistency is decidable with respect to steps (Theorem 1), markings (Theorem 2) and nets (Theorem 7).

The concluding section contains some questions and plans for further investigations.

## 2 Basic Notions

### 2.1 Denotations

The set of non-negative integers is denoted by  $\mathbb{N}$ . Given a set  $X$ , the cardinality (number of elements) of  $X$  is denoted by  $|X|$ , the powerset (set of all subsets) by  $2^X$ , the cardinality of the powerset is  $2^{|X|}$ . Multisets over  $X$  are members of  $\mathbb{N}^X$ , i.e. functions from  $X$  into  $\mathbb{N}$ .

### 2.2 Petri Nets and Their Computations

The definitions concerning Petri nets are mostly based on [5].

**Definition 1 (Nets).** Net is a triple  $N = (P, T, F)$ , where:

- $P$  and  $T$  are finite disjoint sets, of places and transitions, respectively;
- $F \subseteq P \times T \cup T \times P$  is a relation, called the flow relation.

For all  $a \in T$  we denote:

- $\bullet a = \{p \in P \mid (p, a) \in F\}$  - the set of entries to  $a$
- $a \bullet = \{p \in P \mid (a, p) \in F\}$  - the set of exits from  $a$

Petri nets admit a natural graphical representation. Nodes represent places and transitions, arcs represent the flow relation. Places are indicated by circles, and transitions by boxes.

The set of all finite strings of transitions is denoted by  $T^*$ , the length of  $w \in T^*$  is denoted by  $|w|$ , number of occurrences of a transition  $a$  in a string  $w$  is denoted by  $|w|_a$ , two strings  $u, v \in T^*$  such that  $(\forall a \in T) |u|_a = |v|_a$  are said to be *Parikh equivalent*.

**Definition 2 (Place/Transition Nets).** Place/transition net (*shortly, p/t-net*) is a quadruple  $S = (P, T, F, M_0)$ , where:

- $N = (P, T, F)$  is a net, as defined above;
- $M_0 \in \mathbb{N}^P$  is a multiset of places, named the initial marking; it is marked by tokens inside the circles, capacity of places is unlimited.

Multisets of places are named *markings*. In the context of p/t-nets, they are mostly represented by nonnegative integer vectors of dimension  $|P|$ , assuming that  $P$  is strictly ordered. The natural generalizations, for vectors, of arithmetic operations  $+$  and  $-$ , as well as the partial order  $\leq$ , all defined componentwise, are well known and their formal definitions are omitted.

In this context, by  $\bullet a(a^\bullet)$  we understand a vector of dimension  $|P|$  which has 1 in every coordinate corresponding to a place that is an entry to (an exit from, respectively)  $a$  and 0 in other coordinates.

A transition  $a \in T$  is *enabled* in a marking  $M$  whenever  $\bullet a \leq M$  (all its entries are marked). If  $a$  is enabled in  $M$ , then it can be executed, but the execution is not forced. The execution of a transition  $a$  changes the current marking  $M$  to the new marking  $M' = (M - \bullet a) + a^\bullet$  (tokens are removed from entries, then put to exits). The execution of an action  $a$  in a marking  $M$  we call a (sequential) step. We shall denote  $Ma$  for the predicate "a is enabled in  $M$ " and  $MaM'$  for the predicate "a is enabled in  $M$  and  $M'$  is the resulting marking".

This notions and predicates we extend, in a natural way, to strings of transitions:  $M \varepsilon M$  for any marking  $M$ , and  $MvaM''$  ( $v \in T^*, a \in T$ ) iff  $MvM'$  and  $M'aM''$ .

If  $MwM'$ , for some  $w \in T^*$ , then  $M'$  is said to be *reachable from  $M$* ; the set of all markings reachable from  $M$  is denoted by  $[M)$ . Given a p/t-net  $S = (P, T, F, M_0)$ , the set  $[M_0)$  of markings reachable from the initial marking  $M_0$  is called the *reachability set* of  $S$ , and markings in  $[M_0)$  are said to be *reachable* in  $S$ .

A transition  $a \in T$  is said to be *live in a marking  $M$*  if there is a string  $u \in T^*$  such that  $ua$  is enabled in  $M$ . A transition  $a \in T$  that is not live in a marking  $M$  is said to be *dead in a marking  $M$* . If  $MaM'$  and a transition  $b \neq a$  is enabled (live) in  $M$  and not enabled (not live) in  $M'$ , then we say that (the execution of)  $a$  *disables (kills)  $b$* .

**Definition 3 (Inhibitor nets).** Inhibitor net is a quintuple  $S = (P, T, F, I, M_0)$ , where:

- $(P, T, F, M_0)$  is a p/t-net, as defined above;
- $I \subseteq P \times T$  is the set of inhibitor arcs (depicted by edges ended with a small empty circle). Sets of entries and exits are denoted by  $\bullet a$  and  $a^\bullet$ , as in p/t-nets; the set of inhibitor entries to  $a$  is denoted by  ${}^\circ a = \{p \in P \mid (p, a) \in I\}$ .

A transition  $a \in T$  (of an inhibitor net) is enabled in a marking  $M$  whenever  $\bullet a \leq M$  (all its entries are marked) and  $(\forall p \in {}^\circ a) M(p) = 0$  - all inhibitor entries to  $a$  are empty. The execution of  $a$  leads to the resulting marking  $M' = (M - \bullet a) + a^\bullet$ .

The following well-known fact follows easily from Definitions 1 and 2.

**Fact 1 (Diamond and big diamond properties)** Any place/transition net possesses the following property:

Big Diamond Property:

If  $MuM' \ \& \ MvM'' \ \& \ u \approx v$  (Parikh equivalence), then  $M' = M''$ .

Its special case with  $|u| = |v| = 2$  is called the Diamond Property:

If  $MabM' \ \& \ MbaM''$ , then  $M' = M''$ .

**Definition 4 ( $\omega$ -extension).** Let  $\Omega = \mathbb{N} \cup \{\omega\}$ , where  $\omega$  is a new symbol (denoted infinity). We extend, in a natural way, arithmetic operations:  $\omega + n = \omega$ ,  $\omega - n = \omega$ , and the order:  $(\forall n \in \mathbb{N}) n < \omega$ . The set of  $k$ -dimensional vectors over  $\Omega$  we shall denote by  $\Omega^k$ , and its elements we shall call  $\omega$ -vectors. Operations  $+$ ,  $-$  and the order  $\leq$  in  $\Omega^k$  are componentwise.

For  $X \subseteq \Omega^k$ , we denote by  $Min(X)$  the set of all minimal (wrt  $\leq$ ) members of  $X$ , and by  $Max(X)$  the set of all maximal (wrt  $\leq$ ) members of  $X$ . Let  $v, v' \in \Omega^k$  be  $\omega$ -vectors such that  $v \leq v'$ , then we say that  $v'$  covers  $v$  ( $v$  is covered by  $v'$ ).

Let us recall the well known important fact known as the Dickson's Lemma.

**Lemma 1 ([6]).** Any subset of incomparable elements of  $\Omega^k$  is finite.

**Definition 5.** We say that a Petri net  $S = (P, T, F, M_0)$  has the monotonicity property if and only if  $(\forall w \in T^*)(\forall M, M' \in \mathbb{N}^{|P|}) Mw \wedge M \leq M' \Rightarrow M'w$ .

**Fact 2** P/t-nets have the monotonicity property.

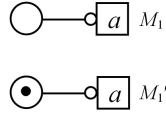
*Proof.* Obvious, since in p/t-nets the tokens of  $M' - M$  can be regarded as frozen (disactive) tokens.

**Fact 3** Inhibitor nets do not have the monotonicity property.

*Proof.* Let us look at the example of Fig. 1. It can be easily seen that  $M_1 < M'_1$ .  $M_1a$  holds but  $M'_1a$  doesn't hold.

**Remark:** In the paper we will use the notions of *reachability graph (tree)* and *coverability graph (tree)*. We assume that the notions are known to the reader. Their definitions can be found in any monograph or survey about Petri nets (see [5,13] or arbitrary else).





**Fig. 1.** Non-monotonic inhibitor net

### 3 Three Kinds of Persistency

The notion of persistency is one of the classical notions in concurrency theory. The notion is recalled in [2] (named in the sequel *e/e*-persistency). Some of its generalizations: *l/l*-persistency and *e/l*-persistency are also introduced there.

Let us sketch the notions informally. The classical *e/e*-persistency means "no action can disable another one", the *l/l*-persistency means "no action can kill another one" and the *e/l*-persistency means "no action can kill another enabled one". Let us go on to formal definitions.

**Definition 6 (Three kinds of persistency).** *Let  $S = (P, T, F, M_0)$  be a place/transition net.*

*If  $(\forall M \in [M_0]) (\forall a, b \in T)$*

- $Ma \wedge Mb \wedge a \neq b \Rightarrow Mab$ , then  $S$  is said to be *e/e*-persistent;*
- $Ma \wedge (\exists u)Mub \wedge a \neq b \Rightarrow (\exists v)Mavb$ , then  $S$  is said to be *l/l*-persistent;*
- $Ma \wedge Mb \wedge a \neq b \Rightarrow (\exists v)Mavb$ , then  $S$  is said to be *e/l*-persistent.*

*The classes of *e/e*-persistent (*l/l*-persistent, *e/l*-persistent) p/t-nets will be denoted by  $P_{e/e}$ ,  $P_{l/l}$  and  $P_{e/l}$ , respectively.*

It is shown in [2] that the following decision problems are decidable:

**Instance:** A p/t net  $(N, M_0)$

**Questions:**

**EE Net Persistency Problem:** Is the net  $S$  *e/e*-persistent?

**LL Net Persistency Problem:** Is the net  $S$  *l/l*-persistent?

**EL Net Persistency Problem:** Is the net  $S$  *e/l*-persistent?

### 4 Hierarchy of *e/l*-persistency

In the previous section we defined three kinds of persistency. Now, we extend the hierarchy mentioned above with an infinite hierarchy of *e/l*-persistent steps.

**Definition 7 (E/l-persistent steps - an infinite hierarchy).**

Let  $S = (P, T, F, M_0)$  be a p/t-net, let  $M$  be a marking. We call a step  $MaM'$ :

- e/l-0-persistent iff it is e/e-persistent (the execution of an action  $a$  does not disable any other action);
- e/l-1-persistent iff  $(\forall b \in T, b \neq a) Mb \Rightarrow [Mab \vee (\exists c \in T)Macb]$  (the execution of an action  $a$  pushes the execution of any other enabled action away for at most 1 step);
- e/l-2-persistent iff  $(\forall b \in T, b \neq a) Mb \Rightarrow (\exists w \in T^*) [|w| \leq 2 \wedge Mawb]$  (the execution of an action  $a$  pushes the execution of any other enabled action away for at most 2 steps);
- ...
- e/l-k-persistent for some  $k \in \mathbb{N}$  iff  $(\forall b \in T, b \neq a) Mb \Rightarrow (\exists w \in T^*) [|w| \leq k \wedge Mawb]$  (the execution of an action  $a$  pushes the execution of any other enabled action away for at most  $k$  steps);
- ...
- e/l- $\infty$ -persistent iff  $(\forall b \in T, b \neq a) Mb \Rightarrow (\exists w \in T^*) Mawb$  (the execution of an action  $a$  pushes the execution of any other enabled action away).

**Remark:** Note that e/l- $\infty$ -persistent steps are exactly e/l-persistent steps.

Directly from Definition 7 we get the

**Fact 4** Let  $S = (P, T, F, M_0)$  be a p/t-net, let  $M$  be a marking. If the step  $MaM'$  is e/l-k-persistent for some  $k \in \mathbb{N}$ , then it is also e/l-i-persistent for every  $i \geq k$ .

**Definition 8.** Let  $S = (P, T, F, M_0)$  be a p/t-net,  $M$  be a marking and  $k \in \mathbb{N}$ . Marking  $M$  is e/l-k-persistent iff for every action  $a \in T$  that is enabled in  $M$  the step  $Ma$  is e/l-k-persistent. P/t-net  $S = (N, M_0)$  is e/l-k-persistent iff every marking reachable in  $S$  is e/l-k-persistent. We denote the class of e/l-k-persistent p/t-nets by  $P_{e/l-k}$ .

The direct conclusion from Fact 4 and Definition 8 is as follows:

**Fact 5** Let  $S = (P, T, F, M_0)$  be a p/t-net,  $M$  be a marking, and  $k \in \mathbb{N}$ . If the marking  $M$  is e/l-k-persistent, then it is also e/l-i-persistent for every  $i \geq k$ . If the net  $S$  is e/l-k-persistent, then it is also e/l-i-persistent for every  $i \geq k$ .

Now we can formulate the problem:

**EL-k Step Persistency Problem**

**Instance:** P/t-net  $S$ , marking  $M$ , action  $a \in T$  enabled in  $M$ .

**Question:** Is the step  $Ma$  e/l-k-persistent?

**Theorem 1.** The EL-k Step Persistency Problem is decidable (for any  $k \in \mathbb{N}$ ).

*Proof.* An algorithm of checking if a step  $Ma$  is  $e/l$ - $k$ -persistent (for some  $k \in \mathbb{N}$ ) for a given net  $S = (N, M_0)$ :

Let us build the part of the depth of  $k+1$  (we call it the  $(k+1)$ -component) of the reachability tree of  $(N, M')$ , where  $M'$  is a marking obtained from  $M$  by execution of  $a$ . The step  $Ma$  is  $e/l$ - $k$ -persistent if for every action  $b \in T$ , such that  $a \neq b$  and  $b$  is enabled in  $M$ , there is a path in the  $(k+1)$ -component of the reachability tree of  $(N, M')$  containing an arc labeled by  $b$ .

Let us introduce another problem:

### EL- $k$ Marking Persistency Problem

**Instance:**  $P/t$ -net  $S = (N, M_0)$ , marking  $M$ .

**Question:** Is the marking  $M$   $e/l$ - $k$ -persistent?

**Theorem 2.** *The EL- $k$  Marking Persistency Problem is decidable (for any  $k \in \mathbb{N}$ ).*

*Proof.* For every action  $a \in T$  that is enabled in a marking  $M$ , we check if a step  $Ma$  is  $e/l$ - $k$ -persistent (for some  $k \in \mathbb{N}$ ) for a given net  $S = (N, M_0)$ , using the algorithm of Theorem 1.

Let us recall the well-known fact, that follows from the Dickson's Lemma 1.

**Fact 6** *Every infinite sequence of markings contains an infinite increasing (not necessarily strictly) subsequence of markings.*

Recall also that  $p/t$ -nets have the monotonicity property - Fact 2.

Let us define the notion of  $k$ -enabledness.

**Definition 9 (k-enabledness).** *Let  $S = (P, T, F, M_0)$  be a  $p/t$ -net, let  $M$  be a marking. For  $k \in \mathbb{N}$  we say that the action  $a \in T$  is  $k$ -enabled in the marking  $M$  if and only if  $\exists w \in T^*$ , such that  $|w| \leq k \wedge Mwa$ .*

Now, we can show:

**Lemma 2.** *Let  $S$  be a  $p/t$ -net. For an arbitrary  $a \in T$  there exists a natural number  $k_a \in \mathbb{N}$ , such that in every marking  $M$  the transition  $a$  is  $k_a$ -enabled or it is dead.*

*Proof.* Suppose that the lemma does not hold for some action  $a \in T$ . It means that for each  $k \in \mathbb{N}$  there is a marking  $M$  such that  $M$  is not  $k$ -enabled but not dead. This means that  $M$  is  $k'$ -enabled for some  $k' > k$ . Thus, there exists an infinite sets of markings  $M_1, M_2, \dots$  and integers  $k_1 < k_2 < \dots$ , such that the action  $a$  is live in each marking  $M_i$  and it is not  $k_i$ -enabled in  $M_i$  for all  $i = 1, 2, \dots$ . Let us choose (by Fact 6) an infinite increasing sequence of markings  $M_{i1} \leq M_{i2} \leq \dots$ . Since the action  $a$  is live in  $M_{i1}$ , it is  $k$ -enabled in  $M_{i1}$ , for some  $k \in \mathbb{N}$ . As the strictly increasing sequence  $k_1 < k_2 < \dots$  is infinite,  $k < k_{ij}$  for some  $j$ . By the monotonicity property (Fact 2), the action  $a$  is  $k$ -enabled, hence  $k_{ij}$ -enabled in the marking  $M_{ij}$ . Contradiction.

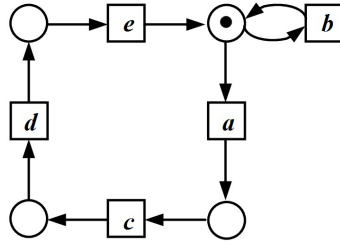
**Remark:** Note that the proof of Lemma 2 is purely existential, it does not present any algorithm for finding  $k$ .

Now, we are ready to formulate the main theorem of the chapter:

**Theorem 3.** *If a p/t-net is e/l-persistent, then it is e/l-k-persistent for some  $k \in \mathbb{N}$ .*

*In words: Whenever an action is disabled by another one, it is pushed away for not more than k-steps.*

*Proof.* If the net is e/l-persistent, then no action kills another enabled one. From the Lemma 2 we know, that if an action  $a \in T$  is not dead then it is  $k_a$ -enabled. Let us take  $K = \max\{k_a | a \in T\}$ , for the numbers  $k_a$  from the Lemma 2. One can see that every action in the net that is not dead, is  $K$ -enabled. Thus, the execution of any action may postpone the execution of an action  $a$  for at most  $K$  steps. So we have the implication: if a p/t-net is e/l-persistent, then it is e/l- $K$ -persistent, for  $K$  defined above.



**Fig. 2.** A p/t-net that is e/l-3 persistent but not e/l-2 persistent

*Example 1.* Let us look at the example of Fig. 2. The only possible situation for temporary disabling an action by another one is the execution of  $a$  that disables  $b$ . And then  $b$  could be enabled again after the execution of the sequence  $cde$ , so after 3 steps. Hence, the net is e/l-3-persistent, and obviously not e/l-2-persistent.

The following example shows that Theorem 3 does not hold for nets without the monotonicity property.

*Example 2.* Let us look at the example of Fig. 3. We can see an inhibitor net and its computation such that for every  $k \in \mathbb{N}$  one can push an action away for a distance greater than  $k$  steps.

This net is life, hence it is e/l-persistent, but it is not e/l- $k$ -persistent for any  $k \in \mathbb{N}$ .

In the infinite computation  $acbcd a e c b c d d a e e c b c d d d a e e e c b \dots$  the first  $a$  pushes  $b$  away for 1 step, the second - for 2 steps and every  $k$ -th  $a$  - for  $k$  steps.

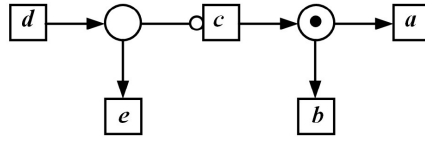


Fig. 3. An inhibitor net and its infinite computation

#### 4.1 EL-k Net Persistency Problem

In the previous section, we established that an action can not postpone another action indefinitely (Theorem 3). We proved, that if a p/t-net is e/l-persistent, then it is e/l-k-persistent for some  $k \in \mathbb{N}$ . We showed that such a  $k$  exists but we did not present any algorithm for finding this  $k$ .

In view of the statements above, let us consider the following problem:

##### EL-k Net Persistency Problem

**Instance:** P/t-net  $S = (N, M_0)$ ,  $k \in \mathbb{N}$ .

**Question:** Is the net  $S$  e/l-k-persistent?

To solve this problem we must prove a set of auxiliary facts.

From this moment, let  $S = (N, M_0)$  be an arbitrary p/t-net.

Let us define the following set of markings:

$E_{a,b} = \{M \in \mathbb{N}^{|P|} \mid Ma \wedge Mb\}$ - the set of markings enabling actions  $a$  and  $b$  simultaneously.

Let us define  $\min E_{a,b} \in \mathbb{N}^{|P|}$ , the minimum marking enabling actions  $a$  and  $b$  simultaneously: if  $(\bullet a[i] = 1 \vee \bullet b[i] = 1)$  then  $\min E_{a,b}[i] := 1$  else  $\min E_{a,b}[i] := 0$  (for  $i = \{1, \dots, |P|\}$ ).

Note that  $\min E_{a,b} = \min E_{a,b} + \mathbb{N}^{|P|}$ .

Let us formulate an auxiliary problem:

##### Mutual Enabledness Reachability Problem

**Instance:** P/t-net  $S = (N, M_0)$ , actions  $a, b \in T$ .

**Question:** Is there a marking  $M$  such that  $M \in E_{a,b}$  and  $M \in [M_0)$  ?

(Is there a reachable marking  $M$  such that actions  $a$  and  $b$  are both enabled in  $M$ ?)

**Theorem 4.** *The Mutual Enabledness Reachability Problem is decidable.*

*Proof.* Let  $M = \min E_{a,b}$ . We build a coverability graph for the p/t-net  $S$ . We check whether in the graph exists a vertex corresponding to an  $\omega$ -marking  $M'$  such that  $M'$  covers  $M$ . If so, then actions  $a$  and  $b$  are simultaneously enabled in some reachable marking of the net  $S$ . Otherwise, those transitions are never enabled at the same time.

Let  $\text{Min}[M_0\rangle$  be a set of minimal (wrt  $\leq$ ) reachable markings of the net  $S$ . As members of  $\text{Min}[M_0\rangle$  are incomparable, the set  $\text{Min}[M_0\rangle$  is finite, by Lemma 1.

Let us denote by  $\text{RE}_{a,b}$  the set of all reachable markings of the net  $S$  enabling actions  $a$  and  $b$  simultaneously:  $\text{RE}_{a,b} = \{M \in [M_0\rangle \mid Ma \wedge Mb\} = E_{a,b} \cap [M_0\rangle$ .

Let  $\text{Min}(\text{RE}_{a,b})$  be a set of all minimal reachable markings of the net  $S$  enabling action  $a$  and  $b$  simultaneously.

Let us recall the Set Reachability Problem.

#### Set Reachability Problem

**Instance:** P/t-net  $S = (N, M_0)$  and a set  $X \subseteq \mathbb{N}^{|P|}$ .

**Question:** Is there a marking  $M \in X$ , reachable in  $S$ ?

**Theorem 5.** *If  $X \subseteq \mathbb{N}^k$  is a rational convex set, then the  $X$ -Reachability Problem is decidable in the class of p/t-nets.*

*Proof.* In [2].

**Proposition 1.** *The set  $\text{Min}(\text{RE}_{a,b})$  can be effectively constructed.*

*Proof.* Sketch of the proof: We put into work the theory of residue sets of Valk/Jantzen [14]. By Valk's/Jantzen's Theory, to show that the set of minimal elements of the set  $\text{RE}_{a,b}$  is effectively computable, it is enough to demonstrate that the set  $\text{RE}_{a,b} \uparrow$  has the property RES (where  $\text{RE}_{a,b} \uparrow = \bigcup \{x \uparrow \mid x \in \text{RE}_{a,b}\}$  and  $x \uparrow = \{z \in \mathbb{N}^{|P|} \mid x \leq z\}$ ). We show it using decidability Theorem 5).

*Example 3.* The set of all minimal reachable markings of the net depicted in Fig.4 enabling action  $a$  and  $b$  simultaneously, is  $\text{Min}(\text{RE}_{a,b}) = \{[1, 1, 1], [2, 0, 1]\}$ .

**Proposition 2.** *If there exists a marking  $M \in \text{RE}_{a,b}$  such that the execution of an action  $a$  in  $M$  pushes the execution of an action  $b$  away for more than  $k$  steps (for some  $k \in \mathbb{N}$ ), then there exists some minimal marking  $M' \in \text{Min}(\text{RE}_{a,b})$  such that the execution of an action  $a$  in  $M'$  pushes the execution of an action  $b$  away for more than  $k$  steps, too.*

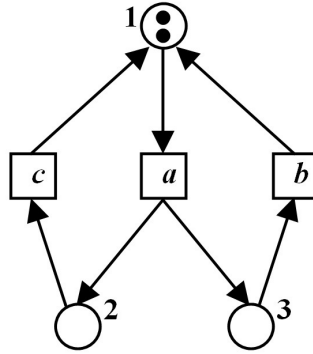


Fig. 4. A p/t-net.

*Proof.* Let  $M$  be a marking, such that the execution of an action  $a$  in  $M$  pushes the execution of an action  $b$  away for more than  $k$  steps (for some  $k \in \mathbb{N}$ ). Let  $M' \in \text{Min}(\text{RE}_{a,b})$  such that  $M' \leq M$ . Such a marking has to exist. Suppose that there is a string  $w \in T^*$ ,  $|w| \leq k$  such that  $M'awb$ . Then obviously also  $Mawb$  (from the monotonicity property - Fact 2). We obtain a contradiction. Hence, the execution of an action  $a$  in  $M'$  postpones the execution of  $b$  for more than  $k$  steps.

Now, we are ready to introduce the following problem:

**EL-k Transition Persistency Problem**

**Instance:** P/t-net  $S = (N, M_0)$ , ordered pair  $(a, b) \in T \times T, b \neq a, k \in \mathbb{N}$ .

**Question:** Is there a reachable marking  $M \in [M_0]$  such that  $Ma \wedge Mb \wedge \neg[(\exists w \in T^*)|w| \leq k \wedge Mawb]$ ?  
 (Does  $a$  postpone  $b$  for more than  $k$  steps?)

**Theorem 6.** *The EL-k Transition Persistency Problem is decidable.*

*Proof.* We introduce an algorithm of deciding if an action  $a$  pushes the execution of an action  $b$  away for more than  $k$  steps in some reachable marking  $M$ .

1. We check whether any markings from the set  $E_{a,b}$  is reachable.
  - (a) If not, we answer NO.
  - (b) Otherwise:
    - i. We build the set  $\text{Min}(\text{RE}_{a,b})$ .
    - ii. For all markings  $M_1 \in \text{Min}(\text{RE}_{a,b})$ :  
 $M_2 := M_1a$ .  
 We build a part of the depth of  $k+1$  (the  $(k+1)$ -component) of the reachability tree of  $(N, M_2)$ . If the piece has an edge labeled by  $b$ , we answer NO. Otherwise we answer YES.

And now the proof of decidability of the EL-k Net Persistency Problem is ready.

**Theorem 7.** *The EL-k Net Persistency Problem is decidable (for any  $k \in \mathbb{N}$ ).*

*Proof.* S is e/l-k-persistent iff the algorithm solving EL-k Transition Persistency Problem answers NO for all ordered pairs  $(a, b) \in T \times T$ ,  $a \neq b$ .

Finally, let us bring to mind decision another problems defined in [2]:

#### Transitions Persistency Problems

**Instance:** p/t-net  $S = (N, M_0)$ , and transitions  $a, b \in T$ ,  $a \neq b$ .

**Questions (informally):**

EE-Persistency Problem: Does  $a$  disable an enabled  $b$ ?

LL-Persistency Problem: Does  $a$  kill a live  $b$ ?

EL-Persistency Problem: Does  $a$  kill an enabled  $b$ ?

From [2] we know that the problems are decidable.

**Theorem 8.** *For a given p/t net  $S = (N, M_0)$  and a pair of transitions  $a, b \in T$  one can calculate a minimum number  $k_{a,b} \in \mathbb{N}$  such that  $a$  postpones an enabled  $b$  for at most  $k_{a,b}$  steps (if such a number exists).*

*Proof.* We ask whether  $a$  kills an enabled  $b$  (EL-Persistency Problem).

If YES then  $k_{a,b}$  does not exist ( $a$  kills  $b$ )

else:

We compute a set  $\text{Min}(\text{RE}_{a,b})$ .

We build the reachability tree as long as from every  $M \in \text{Min}(\text{RE}_{a,b})$  a path leads to a vertex  $M'$  (it can be an empty path) such  $M'b$ . The maximum length of such paths is the desired number  $k_{a,b}$ .

**Theorem 9.** *If a p/t-net  $S = (N, M_0)$  is e/l-persistent, then it is e/l-k-persistent for some  $k \in \mathbb{N}$  and such a  $k$  can be effectively computed.*

*Proof.* For every pair  $(a, b)$  of transitions we find  $k_{a,b}$  defined above. The number we are looking for is  $k = \max(k_{a,b} : a, b \in T)$ .

## 5 Questions for Further Work

It is shown in [1] that if we change the firing rule in the following way: only e/e-persistent computations are permitted, then we get a new class of nets (we call them *nonviolence nets*) which are computationally equivalent to Turing machines. We plan to investigate net classes, with firing rules changed (only e/l-k-persistent computations are allowed) and answer the question:

#### Question 1:

What is the computational power of nets created this way?

We investigated p/t-nets because they are easy to examine (mainly due to their convenient properties such as the monotonicity property). We would like to study the hierarchy of e/l-k-persistency in more difficult extensions of p/t-nets.



## 6 Acknowledgments

The authors are very grateful to the anonymous referees, for their very sound remarks and suggestions.

## References

1. Kamila Barylska, Lukasz Mikulski, and Edward Ochmanski. On persistent reachability in petri nets. In Susanna Donatelli, Jetty Kleijn, Ricardo Jorge Machado, and João M. Fernandes, editors, *ACSD/Petri Nets Workshops*, volume 827 of *CEUR Workshop Proceedings*, pages 373–384. CEUR-WS.org, 2010.
2. Kamila Barylska and Edward Ochmanski. Levels of persistency in place/transition nets. *Fundam. Inform.*, 93(1-3):33–43, 2009.
3. E. Best and J. Esparza. Model checking of persistent Petri nets. *Lecture Notes in Computer Science*, 626:35–??, 1992.
4. Eike Best and Philippe Darondeau. Decomposition theorems for bounded persistent petri nets. In Kees M. van Hee and Rüdiger Valk, editors, *Petri Nets*, volume 5062 of *Lecture Notes in Computer Science*, pages 33–51. Springer, 2008.
5. Jorg Desel and Wolfgang Reisig. Place or transition petri nets. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets, Vol. I: Basic Models, Advances in Petri Nets*, volume 1491 (Volume I) of *Lecture Notes in Computer Science (LNCS)*, pages 122–173. Springer-Verlag (New York), Dagstuhl, Germany, September 1996, revised paper 1998.
6. Leonard E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with  $n$  distinct prime factors. *Amer. J. Math.*, 35:413–422, 1913.
7. Jan Grabowski. The decidability of persistence for vector addition systems. *Information Processing Letters*, 11(1):20–23, 29 August 1980.
8. M. Hack. Decidability questions for petri nets. Technical Report TR-161, MIT Lab. for Comp. Sci., June 1976.
9. Hiraishi and Ichikawa. On structural conditions for weak persistency and semilinearity of petri nets. *TCS: Theoretical Computer Science*, 93, 1992.
10. R. Karp and R. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3:147–195, 1969.
11. Landweber and Robertson. Properties of conflict-free and persistent petri nets. *JACM: Journal of the ACM*, 25, 1978.
12. Ernst W. Mayr. Persistence of vector replacement systems is decidable. *Acta Informatica*, 15:309–318, 1981.
13. P. Starke. *Petri-Netze (in German)*. VEB Deutscher Verlag der Wissenschaften, East Berlin, GDR, 1980.
14. Valk and Jantzen. The residue of vector sets with applications to decidability problems in petri nets. *ACTAINF: Acta Informatica*, 21, 1985.
15. Yamasaki. Normal petri nets. *TCS: Theoretical Computer Science*, 31, 1984.
16. Hideki Yamasaki. On weak persistency of Petri nets. *Information Processing Letters*, 13(3):94–97, 13 December 1981.



Short Presentations



# Local state refinement on Elementary Net Systems: an approach based on morphisms

Luca Bernardinello, Elisabetta Mangioni, and Lucia Pomello

Dipartimento di Informatica Sistemistica e Comunicazione,  
Università degli studi di Milano - Bicocca,  
Viale Sarca, 336 - Edificio U14 - I-20126 Milano, Italia  
[mangioni@disco.unimib.it](mailto:mangioni@disco.unimib.it)

**Abstract.** In the design of concurrent and distributed systems, modularity and refinement are basic conceptual tools. We propose a notion of refinement/abstraction of local states for a basic class of Petri Nets, associated with a new kind of morphisms. The morphisms, from a refined system to an abstract one, associate suitable subnets to abstract local states. The main results concern behavioural properties preserved and reflected by the morphisms. In particular, we focus on the conditions under which reachable markings are preserved or reflected, and the conditions under which a morphism induces a bisimulation between net systems.

**Keywords:** Elementary Net Systems, morphisms, local state refinement

## 1 Introduction

Refinement and composition of modules are among the basic conceptual tools of a system designer. Several formal approaches are available. One of the main challenges consists in developing languages and methods allowing to derive properties of the refined system from properties of the abstract one.

We propose an approach based on Petri nets, where the refinement of a model is supported by so-called  $\alpha$ -morphisms on the class of Elementary Net Systems. We focus on the refinement of local states. Given a net  $N_2$ , interpreted as an abstract description of a system, the local states of  $N_2$  are replaced by subnets, giving a new net, say  $N_1$ , so that there is an  $\alpha$ -morphism from  $N_1$  to  $N_2$ .

Using morphisms to formalize the relation between a refined net and a more abstract one is not new. Most approaches, in Petri net theory, are based on transition refinement and, less frequently, on place refinement; for a survey, see [5]. Another survey paper, [9], describes a set of techniques which allow to refine transitions in Place/transition nets, so that the relation between the abstract net and its refinement is given by a morphism. There, the emphasis is on refinement rules that preserve specific behavioural properties, within the wider context of general transformation rules on nets.

A very general class of morphisms, interpreted as abstraction of system requirements, with less focus on strict preservation of behavioural properties, is defined in [6].

The approach we present in this paper is similar in spirit to the refinement operation proposed in [8]. In that approach, refinement is defined on transition systems, but is strictly related to refinement of local states in nets, through the notion of region.

$\alpha$ -morphisms can be seen as a special case of the morphisms introduced by Winskel in [13], as it will be formally shown in Section 5. Other morphisms introduced in the literature on the same line of Winskel morphisms, are the ones given in [12] and [1].

Our approach is motivated by the attempt to define a refinement operation preserving behavioural properties on the basis of structural and only local behavioural constraints. The additional restrictions, with respect to general morphisms, aim, on one hand, to capture typical features of refinements, and on the other hand to ensure that some behavioural properties of the abstract model still hold in the refined model.

Moreover, in [2], we use  $\alpha$ -morphisms as a means supporting a composition operator defined through an interface, following the same approach proposed in [4].

In the rest of this section, the main ideas of refinement and related morphisms are explained by means of a simple example. In Section 2 we collect preliminary definitions related to Petri nets which are used in the rest of the paper. Section 3 contains the definition of  $\alpha$ -morphisms and the main results of the paper: in particular, we show that reachable markings are preserved, we characterize the local conditions under which reachable markings are reflected, i.e.: under which the counterimage of reachable markings are reachable markings, and such that morphisms induce a bisimulation between the related net systems. In Section 5 we compare  $\alpha$ -morphisms with Winskel's morphisms. Finally, in Section 6 we discuss some critical issues in our approach and suggest possible developments.

Most proofs are given in an extended version of the present paper [3].

### 1.1 An example

The example presented in this section aims at explaining, informally, how  $\alpha$ -morphisms support refinement of local states in Elementary Net Systems. The morphism maps nodes of a refined system,  $N_1$ , on a more abstract one,  $N_2$ .

The Elementary Net System shown in Fig. 1 represents an abstract view of the interaction between a student and an University secretariat office. A student may ask the office either to emit an English proficiency certificate or to admit her to the final exam. Note that, at this level of abstraction, the model does not distinguish a positive answer from a negative one. Suppose that the local state `inspect_request` corresponds to the actual inspection of the request by a Faculty board, which delivers the decision to the secretariat.

We might want to refine `formal_check`, in order to distinguish two cases: positive answer and negative answer.

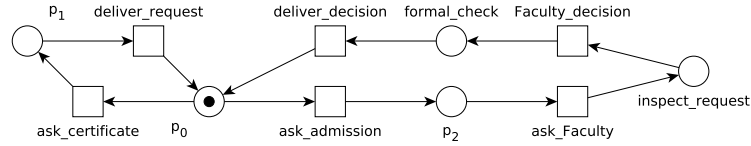


Fig. 1: Abstract view ( $N_2$ )

The actual decision has been taken in state `inspect_request`, so the refinement of `formal_check` requires splitting the event `Faculty_decision`, thus reflecting the choice between the two answers. The result of the refinement is shown in Fig. 2,

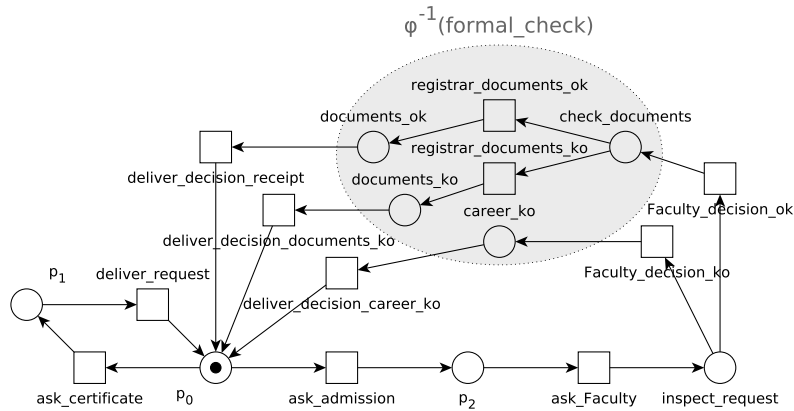


Fig. 2: Refined model ( $N_1$ )

where the subnet refining `formal_check` is enclosed in a shaded oval. Note that the operation has required also splitting the outgoing transitions, in order to reflect the alternative outcomes.

## 2 Preliminary definitions

In this section, we recall the basic definitions of net theory, in particular Elementary Net Systems [11], and bisimulation [7].

We will use the symbol  $\downarrow$  to denote the restriction of a function on a subset of its domain.

### 2.1 Petri Nets

In net theory, models of distributed systems are based on objects called nets which specify local states, local transitions and the relations among them. A *net* is a triple  $N = (B, E, F)$ , where  $B$  is a set of *conditions* or local states,  $E$  is a

set of *events* or transitions such that  $B \cap E = \emptyset$  and  $F \subseteq (B \times E) \cup (E \times B)$  is the *flow relation*.

We adopt the usual graphical notation: conditions are represented by circles, events by boxes and the flow relation by arcs. The set of elements of a net will be denoted by  $X = B \cup E$ ; we allow nets with isolated elements.

The *preset* of an element  $x \in X$  is  $\bullet x = \{y \in X \mid (y, x) \in F\}$ ; the *postset* of  $x$  is  $x^\bullet = \{y \in X \mid (x, y) \in F\}$ ; the *neighbourhood* of  $x$  is given by  $\bullet x^\bullet = \bullet x \cup x^\bullet$ . These notations are extended to subsets of elements in the usual way.

For any net  $N$  we denote the *in-elements* of  $N$  by  $\circ N = \{x \in X_N \mid \bullet x = \emptyset\}$  and the *out-elements* of  $N$  by  $N^\circ = \{x \in X_N \mid x^\bullet = \emptyset\}$ .

A net is *simple* if for all  $x, y \in X$ , if  $\bullet x = \bullet y$  and  $x^\bullet = y^\bullet$ , then  $x = y$ .

A net  $N' = (B', E', F')$  is a *subnet* of  $N = (B, E, F)$  if  $B' \subseteq B$ ,  $E' \subseteq E$ , and  $F' = F \cap ((B' \times E') \cup (E' \times B'))$ . Given a subset of elements  $A \subseteq X$ , we say that  $N(A)$  is the *subnet of  $N$  identified by  $A$*  if  $N(A) = (B \cap A, E \cap A, F \cap (A \times A))$ .

A *State Machine* is a connected net such that each event  $e$  has exactly one input condition and exactly one output condition:  $\forall e \in E, |\bullet e| = |e^\bullet| = 1$ .

Elementary Net (EN) Systems are a basic system model in net theory. An *Elementary Net System* is a quadruple  $N = (B, E, F, m_0)$ , where  $(B, E, F)$  is a net such that  $B$  and  $E$  are finite sets, self-loops are not allowed, isolated elements are not allowed, and the *initial marking* is  $m_0 \subseteq B$ .

The elements in the initial marking are interpreted as the conditions which are true in the initial state.

A subnet of an EN System  $N$  identified by a subset of conditions  $A$  and all its pre and post events,  $N(A \cup \bullet A^\bullet)$ , is a *Sequential Component* of  $N$  if  $N(A \cup \bullet A^\bullet)$  is a State Machine and if it has only one token in the initial marking.

An EN System is *covered* by Sequential Components if every condition of the net belongs to at least a Sequential Component. In this case we say that the system is *State Machine Decomposable (SMD)*.

The behaviour of EN Systems is defined through the firing rule, which specifies when an event can occur, and how event occurrences modify the holding of conditions, i.e. the state of the system.

Let  $N = (B, E, F, m_0)$  be an EN System,  $e \in E$  and  $m \subseteq B$ . The event  $e$  is *enabled* at  $m$ , denoted  $m[e]$ , if  $\bullet e \subseteq m$  and  $e^\bullet \cap m = \emptyset$ ; the occurrence of  $e$  at  $m$  leads from  $m$  to  $m'$ , denoted  $m[e]m'$ , iff  $m' = (m \setminus \bullet e) \cup e^\bullet$ .

Let  $\epsilon$  denote the empty word in  $E^*$ . The firing rule is extended to sequences of events by setting  $m[\epsilon]m$  and  $\forall e \in E, \forall w \in E^*, m[ew]m' = m[e]m''[w]m''$ ;  $w$  is called *firing sequence*.

A subset  $m \subseteq B$  is a *reachable marking* of  $N$  if there exists a  $w \in E^*$  such that  $m_0[w]m$ . The *set of all reachable markings* of  $N$  is denoted by  $[m_0]$ .

An EN System is *contact-free* if  $\forall e \in E, \forall m \in [m_0]: \bullet e \subseteq m$  implies  $e^\bullet \cap m = \emptyset$ . An EN System covered by Sequential Components is contact-free. An event is called *dead* at a marking  $m$  if it is not enabled at any marking reachable from  $m$ . A reachable marking  $m$  is called *dead* if no event is enabled at  $m$ . An Elementary Net System is *deadlock-free* if no reachable marking is dead.



## 2.2 Unfoldings

The semantics of an EN System can be given as its *unfolding*. The unfolding is an acyclic net, possibly infinite, which records the occurrences of its elements in all possible executions.

**Definition 1.** Let  $N = (B, E, F)$  be a net, and let  $x, y \in X$ . We say that  $x$  and  $y$  are in conflict, denoted by  $x \#_N y$ , if there exist two distinct events  $e_x, e_y \in E$  such that  $e_x F^* x$ ,  $e_y F^* y$ , and  $\bullet e_x \cap \bullet e_y \neq \emptyset$ .

**Definition 2.** An occurrence net is a net  $N = (B, E, F)$  satisfying:

1. if  $e_1, e_2 \in E, e_1 \bullet \cap e_2 \bullet \neq \emptyset$  then  $e_1 = e_2$ ;
2.  $F^*$  is a partial order,
3. for any  $x \in X, \{y : y F^* x\}$  is finite;
4.  $\#_N$  is irreflexive,
5. the minimal elements with respect to  $F^*$  are conditions.

A branching process of  $N$  is an occurrence net whose elements can be mapped to the elements of  $N$ .

**Definition 3.** Let  $N = (B, E, F, m_0)$  be an EN System, and  $\Sigma = (P, T, G)$  be an occurrence net. Let  $\pi : P \cup T \rightarrow B \cup E$  be a map.

The pair  $(\Sigma, \pi)$  is a branching process of  $N$  if:

- $\pi(P) \subseteq B, \pi(T) \subseteq E$ ;
- $\pi$  restricted to the minimal elements of  $\Sigma$  is a bijection on  $m_0$ ;
- for each  $t \in T, \pi$  restricted to  $\bullet t$  is injective and  $\pi$  restricted to  $t \bullet$  is injective;
- for each  $t \in T, \pi(\bullet t) = \bullet(\pi(t))$  and  $\pi(t \bullet) = (t \bullet)$ .

The unfolding of an EN System  $N$ , denoted by  $Unf(N)$ , is the maximal branching process of  $N$ , namely the unique branching process such that any other branching process of  $N$  is isomorphic to a subnet of  $Unf(N)$ . The map associated to the unfolding will be denoted  $u$  and called *folding*.

## 2.3 Bisimulations

Bisimulation relations have been introduced as equivalence notions with respect to event observation [7]. We define the observability of events of a system by using a labelling function which associates the same label to different events, when viewed as equal by an observer, and the label  $\tau$  to unobservable events.

**Definition 4.** Let  $N = (B, E, F, m_0)$  be an EN System,  $l : E \rightarrow L \cup \{\tau\}$  be a labelling function where  $L$  is the alphabet of observable actions and  $\tau \notin L$  the unobservable action. Let  $\epsilon$  denote the empty word both of  $E^*$  and  $L^*$ . The function  $l$  is extended to a homomorphism  $l : E^* \rightarrow L^*$  in the following way:

$$l(\epsilon) = \epsilon$$

$$\forall e \in E, \forall w \in E^*, l(ew) = \begin{cases} l(e)l(w) & \text{if } l(e) \neq \tau \\ l(w) & \text{if } l(e) = \tau \end{cases}$$

The pair  $(N, l)$  is called Labelled EN System.

Let  $m, m' \in [m_0]$  and  $a \in L \cup \{\epsilon\}$  then:

- $a$  is enabled at  $m$ , denoted  $m(a)$ , iff  $\exists w \in E^* : l(w) = a$  and  $m[w]$ ;
- if  $a$  is enabled at  $m$ , then the occurrence of  $a$  can lead from  $m$  to  $m'$ , denoted  $m(a)m'$ , iff  $\exists w \in E^* : l(w) = a$  and  $m[w]m'$ .

We define weak bisimulation as a relation between reachable markings of Labelled EN Systems [10].

**Definition 5.** Let  $N_i = (B_i, E_i, F_i, m_0^i)$  be an EN System for  $i = 1, 2$ , with the labelling function  $l_i : E_i \rightarrow L \cup \{\tau\}$ . Then  $(N_1, l_1)$  and  $(N_2, l_2)$  are weakly bisimilar, denoted  $(N_1, l_1) \approx (N_2, l_2)$ , iff  $\exists r \subseteq [m_0^1] \times [m_0^2]$  such that:

- $(m_0^1, m_0^2) \in r$ ;
- $\forall (m_1, m_2) \in r, \forall a \in L \cup \{\epsilon\}$  it holds

$$\forall m'_1 : m_1(a)m'_1 \Rightarrow \exists m'_2 : m_2(a)m'_2 \wedge (m'_1, m'_2) \in r$$

and (vice versa)

$$\forall m'_2 : m_2(a)m'_2 \Rightarrow \exists m'_1 : m_1(a)m'_1 \wedge (m'_1, m'_2) \in r$$

Such a relation  $r$  is called weak bisimulation.

For short in the rest of the paper we will use the term *bisimulation* instead of *weak bisimulation*.

### 3 A class of morphisms

In this section we present the formal definition of  $\alpha$ -morphisms for State Machine Decomposable Elementary Net Systems (SMD-EN Systems), and discuss some of their properties, particularly with respect to the preservation of both structural and behavioural properties.

We start by defining a more general class of morphisms, and then present the more specific restrictions.

**Definition 6.** Let  $N_i = (B_i, E_i, F_i, m_0^i)$  be a SMD-EN System, for  $i = 1, 2$ . An  $\omega$ -morphism from  $N_1$  to  $N_2$  is a total surjective map  $\varphi : X_1 \rightarrow X_2$  such that:

1.  $\varphi(B_1) = B_2$ ;
2.  $\varphi(m_0^1) = m_0^2$ ;
3.  $\forall e_1 \in E_1$ , if  $\varphi(e_1) \in E_2$ , then  $\varphi(\bullet e_1) = \bullet \varphi(e_1)$  and  $\varphi(e_1 \bullet) = \varphi(e_1) \bullet$ ;
4.  $\forall e_1 \in E_1$ , if  $\varphi(e_1) \in B_2$ , then  $\varphi(\bullet e_1 \bullet) = \{\varphi(e_1)\}$ ;

We require that the map is total and surjective because  $N_1$  refines the abstract model  $N_2$ , and any abstract element must be related to its refinement.

In particular, a subset of nodes can be mapped on a single condition  $b_2 \in B_2$ ; in this case, we will call *bubble* the subnet identified by this subset, and denote it by  $N_1(\varphi^{-1}(b_2))$ ; if more than one element is mapped on  $b_2$ , we will say that  $b_2$  is *refined* by  $\varphi$ .

**Definition 7.** Let  $N_i = (B_i, E_i, F_i, m_0^i)$  be a SMD-EN System, for  $i = 1, 2$ . An  $\alpha$ -morphism from  $N_1$  to  $N_2$  is an  $\omega$ -morphism satisfying

5.  $\forall b_2 \in B_2$ 
  - (a)  $N_1(\varphi^{-1}(b_2))$  is an acyclic net;
  - (b)  $\forall b_1 \in \circ N_1(\varphi^{-1}(b_2)), \varphi(\bullet b_1) \subseteq \bullet b_2$  and  $(\bullet b_2 \neq \emptyset \Rightarrow \bullet b_1 \neq \emptyset)$ ;
  - (c)  $\forall b_1 \in N_1(\varphi^{-1}(b_2))^\circ, \varphi(b_1 \bullet) = b_2 \bullet$ ;
  - (d)  $\forall b_1 \in \varphi^{-1}(b_2) \cap B_1,$   
 $(b_1 \notin \circ N_1(\varphi^{-1}(b_2)) \Rightarrow \varphi(\bullet b_1) = \{b_2\})$  and  $(b_1 \notin N_1(\varphi^{-1}(b_2))^\circ \Rightarrow \varphi(b_1 \bullet) = \{b_2\})$ ;
  - (e)  $\forall b_1 \in \varphi^{-1}(b_2) \cap B_1,$  there is a sequential component  $N_{SC}$  of  $N_1$  such that  $b_1 \in B_{SC}$  and  $\varphi^{-1}(\bullet b_2 \bullet) \subseteq E_{SC}$ .

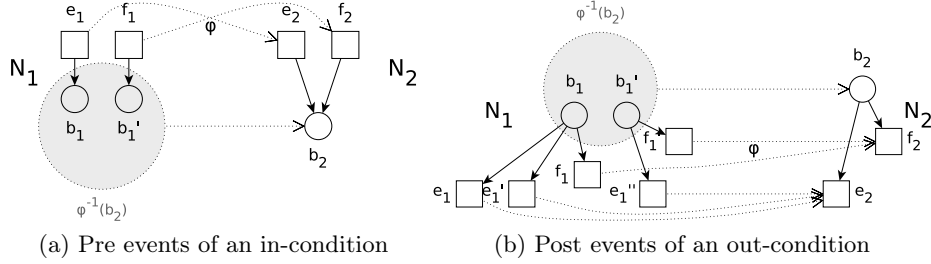


Fig. 3: Pre and post event of a bubble

As we can see in Fig. 3a and 3b, in-conditions and out-conditions have different constraints, 5b and 5c respectively. As required by 5c, we do not allow that choices, which are internal to a bubble, constrain a final marking of that bubble: i.e., each out-condition of the bubble must have the same choices of the condition it refines. Instead, pre-events do not need this strict constraint (5b): hence it is sufficient only that pre-events of any in-condition are mapped on a subset of the pre-events of the condition it refines. For example, in this particular case, we know that the choice between  $e_1$  and  $f_1$  of Figure 3a is made before the bubble, and this is implied also by the requirement 5e) on sequential components. Moreover, the conditions that are internal to a bubble must have pre-events and post-events which are all mapped to the refined condition  $b_2$ , as required by 5d.

By requirement 5e, events in the neighbourhood of a bubble are not concurrent, as their images. Within a bubble, there can be concurrent events; however, post events are in conflict, and firing one of them will empty the bubble, as shown in Lemma 1 below.

The  $\alpha$ -morphisms are closed by composition, the identity function on  $X$  is an  $\alpha$ -morphism, and the composition is associative. Hence, the family of SMD-EN Systems together with  $\alpha$ -morphisms forms a category.

The partition of elements of  $N_1$  induced by an  $\alpha$ -morphism  $\varphi : N_1 \rightarrow N_2$  defines the structure of a net:

**Definition 8.** Let  $N_i = (B_i, E_i, F_i, m_0^i)$  be a SMD-EN System, for  $i = 1, 2$ . Let  $\varphi$  be an  $\alpha$ -morphism from  $N_1$  to  $N_2$ .

Then  $\varphi$  defines an equivalence relation on  $X_1$ , where the equivalence class of  $x \in X_1$  is  $[x] = \{y \in X_1 \mid \varphi(y) = \varphi(x)\}$ .

The quotient of  $N_1$  with respect to  $\alpha$  is  $N_1/\varphi = (B_1/\varphi, E_1/\varphi, F_1/\varphi, m_0^1/\varphi)$ , where

- $B_1/\varphi = \{[x] : x \in X_1, \varphi(x) \in B_2\}$ ;
- $E_1/\varphi = \{[x] : x \in X_1, \varphi(x) \in E_2\}$ ;
- $F_1/\varphi = \{([x], [y]) : x, y \in X_1, x \neq y, \exists(x, y) \in F_1\}$ ;
- $m_0^1/\varphi = \{[x] : x \in m_0^1\}$ .

By a simple verification [3], the quotient of  $N_1$ ,  $N_1/\varphi$ , is a SMD-EN System isomorphic to  $N_2$ .

## 4 Properties preserved and reflected by $\alpha$ -morphisms

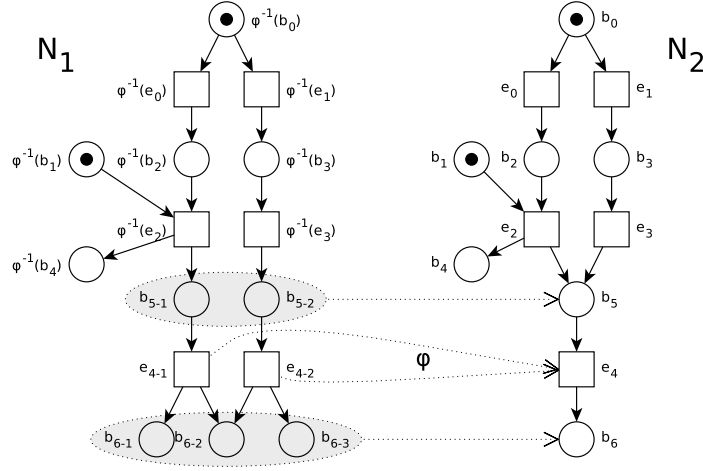
Since we consider SMD-EN Systems, it is natural to ask whether  $\alpha$ -morphisms preserve and reflect sequential components. Let  $\varphi$  be an  $\alpha$ -morphism from  $N_1$  to  $N_2$ . We know that, if a condition  $b_2$  belongs to a sequential component, then also its pre- and post-events belong to the same sequential component. Hence, if  $b_2$  is refined by a bubble  $N_1(\varphi^{-1}(b_2))$ , by the requirement 5e) of  $\alpha$ -morphisms any condition of the bubble belongs to a sequential component containing any event in  $\varphi^{-1}(\bullet b_2 \bullet)$ . This allows one to say that the sequential components of  $N_2$  are reflected by  $\varphi$ , in the sense that the inverse image of a sequential component is covered by sequential components.

**Lemma 1.** Let  $\varphi : N_1 \rightarrow N_2$  be an  $\alpha$ -morphism.

Let  $N_{SC2}$  be a sequential component of  $N_2$ . Then  $\varphi^{-1}(N_{SC2})$  is covered by sequential components, each one containing all the inverse image of the neighbourhood of each condition of  $N_{SC2}$ .

Sequential components are not preserved, as we can see in Fig. 4. The sequential component of  $N_1$  generated by  $\{\varphi^{-1}(b_1), b_{5-1}, b_{6-1}\}$  is such that its image  $\{b_1, b_5, b_6\}$  is not a sequential component of  $N_2$ .

The idea driving our interpretation of bubble is that the subnet corresponding to a condition “behaves” in the same way as the condition it refines. In a SMD-EN System, each condition at any time can be true or false. It is not possible that


 Fig. 4: Two SMD-EN Systems related by an  $\alpha$ -morphism

this condition is partially true or partially false; hence, also the bubble should behave like this. The next lemma states that firing an output event of a bubble empties the bubble, and that no input event of a bubble is enabled whenever a token is inside the bubble.

**Lemma 2.** *Let  $\varphi : N_1 \rightarrow N_2$  be an  $\alpha$ -morphism. Then:*

1. *Let  $e_1 \in E_1, b_2 \in B_2: e_1 \in \varphi^{-1}(b_2^\bullet)$ ;  $m_1, m'_1 \in [m_0^1]: m_1[e_1]m'_1$ , then  $m'_1 \cap \varphi^{-1}(b_2) = \emptyset$ .*
2. *Let  $e_1 \in E_1, b_2 \in B_2: e_1 \in \varphi^{-1}(\bullet b_2)$ ;  $m_1, m'_1 \in [m_0^1]: m_1[e_1]m'_1$  then  $m_1 \cap \varphi^{-1}(b_2) = \emptyset$ .*

*Proof.* Take a marking  $m_1$  in which a condition  $b_1 \in \varphi^{-1}(b_2)$  is marked.

We know by Def. 7, point 5e) that there exists a sequential component  $SC$  of  $N_1$  such that  $b_1 \in B_{SC}$  and  $\varphi^{-1}(\bullet b_2^\bullet) \subseteq E_{SC}$ .

1. By contradiction, take  $e_1 \in \varphi^{-1}(b_2^\bullet)$  such that  $b_1 \notin \bullet e_1$  and  $m_1[e_1]$ ; hence all its preconditions are marked. Since  $SC$  contains  $e_1$ , one of its preconditions belongs to  $SC$  as well as  $b_1$ , this is a contradiction because the sequential component has only one token.
2. By contradiction, take  $e_1 \in \varphi^{-1}(\bullet b_2)$  such that  $m_1[e_1]$ ; hence all its preconditions are marked. Since  $SC$  contains  $e_1$ , one of its preconditions belongs to  $SC$  as well as  $b_1$ , and this is a contradiction because the sequential component has only one token.

□

Our morphisms can be seen like a special case of Winskel morphisms [13], as we shall prove in Section 5. Then, since Winskel morphisms preserve reachable markings, also  $\alpha$ -morphisms do, as stated in the following.

**Proposition 1.** *Let  $\varphi : N_1 \rightarrow N_2$  be an  $\alpha$ -morphism.  
 Then if  $m_1 \in [m_0^1]$  and  $m_1 [e] m'_1$  then  $\varphi(m_1) \in [m_0^2]$  and*

- *if  $\varphi(e) \in E_2$  then  $\varphi(m_1) [\varphi(e)] \varphi(m'_1)$  else*
- *(if  $\varphi(e) \in B_2$  then)  $\varphi(m_1) = \varphi(m'_1)$ .*

As for other morphisms in the literature,  $\alpha$ -morphisms do not reflect reachable markings. This happens either when a condition is refined by a subnet leading to a block before reaching a marking enabling out-events, or whenever the refinements of conditions “interfere” with each other so that, even if in each bubble a “final” local marking is reached, the global marking doesn’t enable any event. The second case is shown in Fig. 5: any event in each bubble can fire, but  $N_1$  has two deadlocks:  $\{p_3, p_6\}$  and  $\{p_4, p_5\}$ . The two above cases suggest to require

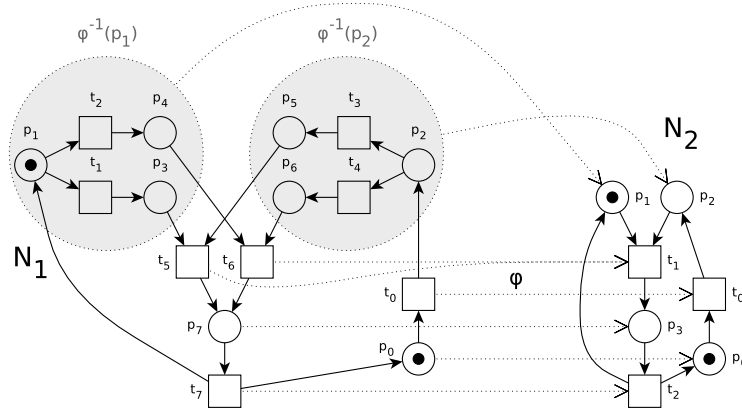


Fig. 5: An  $\alpha$ -morphism.

both that any condition is refined by a subnet such that, when a final marking is reached, this one enables events which correspond to the post-events of the refined condition; and also that different refinements do not “interfere” with each other. The non interference is guaranteed when any event of  $N_2$  has at most a unique condition in its neighbourhood that is properly refined in  $N_1$ .

In order to reflect the reachable markings we have to introduce local behavioural constraints and this by considering the unfolding of subnets related to the bubbles. Then, we need to define the following auxiliary construction. Given an  $\alpha$ -morphism  $\varphi : N_1 \rightarrow N_2$ , and a condition  $b_2 \in B_2$  with its refinement  $N_1(\varphi^{-1}(b_2))$ , we define two new SMD-EN Systems. The first one, denoted  $S_1(b_2)$ , contains (a copy of) the subnet  $N_1(\varphi^{-1}(b_2))$ , its pre and post-events in  $E_1$  and two new conditions:  $b_1^{in}$ , which is pre of all the pre-events, and  $b_1^{out}$ , which is post of all the post-events. The initial marking of  $S_1(b_2)$  will be  $\{b_1^{in}\}$ . The second system, denoted  $S_2(b_2)$  contains  $b_2$ , its pre- and post-events and two new conditions:  $b_2^{in}$ , which is pre of all the pre-events, and  $b_2^{out}$ , which is post of all the post-events. The initial marking of  $S_2(b_2)$  will be  $\{b_2^{in}\}$ .

In Fig. 6 and 7 we show the two systems  $S_1(b_2)$  and  $S_2(b_2)$  for the nets showed in the initial example (Fig. 1 and 2), in Section 1, with  $b_2 = \text{formal\_check}$ .

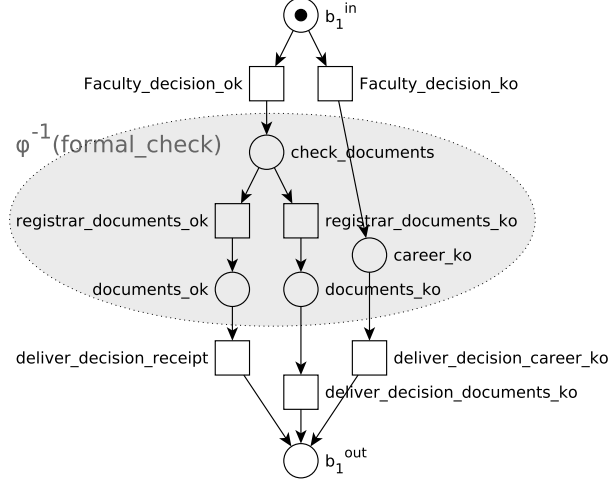


Fig. 6:  $S_1(\text{formal\_check})$  of Fig. 2.

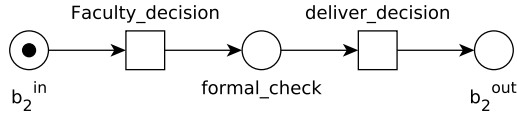


Fig. 7:  $S_2(\text{formal\_check})$  of Fig. 1.

**Definition 9.** Let  $\varphi : N_1 \rightarrow N_2$  be an  $\alpha$ -morphism and  $b_2 \in B_2$ .

Construct the SMD-EN Systems,  $S_1(b_2) = (B_{S_1}, E_{S_1}, F_{S_1}, m_0^{S_1})$  and  $S_2(b_2) = (B_{S_2}, E_{S_2}, F_{S_2}, m_0^{S_2})$  in this way:

$$B_{S_1} = \begin{cases} (\varphi^{-1}(b_2) \cap B_1) \cup \{b_1^{out}\} & \text{if } \bullet b_2 = \emptyset \\ (\varphi^{-1}(b_2) \cap B_1) \cup \{b_1^{in}\} & \text{if } b_2 \bullet = \emptyset \\ (\varphi^{-1}(b_2) \cap B_1) \cup \{b_1^{in}, b_1^{out}\} & \text{otherwise} \end{cases}$$

$$E_{S_1} = (\varphi^{-1}(b_2) \cap E_1) \cup \varphi^{-1}(\bullet b_2) \cup \varphi^{-1}(b_2 \bullet);$$

$$F_{S_1} = (F_1 \cap ((B_{S_1} \cup E_{S_1}) \times (E_{S_1} \cup B_{S_1}))) \cup F_{S_1}^{in} \cup F_{S_1}^{out}, \text{ where}$$

$$F_{S_1}^{in} = \{(b_1^{in}, e) : e \in \varphi^{-1}(\bullet b_2)\} \text{ and}$$

$$F_{S_1}^{out} = \{(e, b_1^{out}) : e \in \varphi^{-1}(b_2 \bullet)\};$$

$$m_0^{S_1} = \begin{cases} m_0^1 \cap \varphi^{-1}(b_2) & \text{if } \bullet b_2 = \emptyset \\ \{b_1^{in}\} & \text{otherwise} \end{cases}$$

$$\begin{aligned}
B_{S_2} &= \begin{cases} \{b_2, b_2^{out}\} & \text{if } \bullet b_2 = \emptyset \\ \{b_2, b_2^{in}\} & \text{if } b_2 \bullet = \emptyset \\ \{b_2, b_2^{in}, b_2^{out}\} & \text{otherwise} \end{cases} \\
E_{S_2} &= \bullet b_2 \bullet; \\
F_{S_2} &= (F_2 \cap ((B_{S_2} \cup E_{S_2}) \times (E_{S_2} \cup B_{S_2}))) \cup F_{S_2}^{in} \cup F_{S_2}^{out}, \text{ where} \\
F_{S_2}^{in} &= \{(b_2^{in}, e) : e \in \bullet b_2\} \text{ and } F_{S_2}^{out} = \{(e, b_2^{out}) : e \in b_2 \bullet\}; \\
m_0^{S_2} &= \begin{cases} m_0^2 \cap \{b_2\} & \text{if } \bullet b_2 = \emptyset \\ \{b_2^{in}\} & \text{otherwise} \end{cases}
\end{aligned}$$

Define  $\varphi^S$  as a map from  $S_1(b_2)$  to  $S_2(b_2)$ , which restricts  $\varphi$  to the elements of  $S_1(b_2)$ , and extends it with  $\varphi^S(b_1^{in}) = b_2^{in}$  and  $\varphi^S(b_1^{out}) = b_2^{out}$ .

Note that  $S_1(b_2)$  and  $S_2(b_2)$  are SMD-EN Systems and that  $\varphi^S$  is an  $\alpha$ -morphism.

Let  $\varphi : N_1 \rightarrow N_2$  be an  $\alpha$ -morphism and  $\varphi^S : S_1(b_2) \rightarrow S_2(b_2)$  as in Def. 9. By using  $\varphi^S$ , consider two labelling functions  $l_1$  and  $l_2$  such that the events in  $E_{S_2}$  are all observable, i.e.:  $l_2$  is the identity function, and the invisible events of  $S_1(b_2)$  are the ones mapped to conditions, i.e.:

$$\forall e \in E_{S_1} : l_1(e) = \begin{cases} \varphi^S(e) & \text{if } \varphi^S(e) \in E_{S_2} \\ \tau & \text{otherwise} \end{cases}$$

Let  $Unf(S_1(b_2))$  be the unfolding of  $S_1(b_2)$  with  $u : Unf(S_1(b_2)) \rightarrow S_1(b_2)$  folding function. The following lemma shows that, if the map,  $\varphi^S \circ u$ , obtained composing  $\varphi^S$  with  $u$  is an  $\alpha$ -morphism, then  $S_1(b_2)$  and  $S_2(b_2)$  are bisimilar.

**Lemma 3.** *Let  $\varphi : N_1 \rightarrow N_2$  be an  $\alpha$ -morphism, and  $\varphi^S$  as in Def. 9. Let  $Unf(S_1(b_2))$  be the unfolding of  $S_1(b_2)$  with  $u$  folding function. If  $\varphi^S \circ u$  is an  $\alpha$ -morphism from  $Unf(S_1(b_2))$  to  $S_2(b_2)$ , then  $r = \{(m_1, \varphi^S(m_1)) : m_1 \in [m_0^{S_1}]\}$  is a bisimulation, and  $(S_1(b_2), l_1)$  and  $(S_1(b_2), l_2)$  are bisimilar.*

In case the morphism corresponds to the refinement of a marked condition, we ask all the tokens of the corresponding bubble to be into in-conditions which are post-conditions of a pre-event, if it exists. System  $N_1$  is then called *well marked* with respect to  $\varphi$ .

**Definition 10.** *Let  $\varphi : N_1 \rightarrow N_2$  be an  $\alpha$ -morphism. System  $N_1$  is well marked with respect to  $\varphi$  if for each  $b_2 \in B_2$  one of the following conditions hold:*

- $\varphi^{-1}(b_2) \cap m_0^1 = \emptyset$  or
- if  $\bullet b_2 \neq \emptyset$  then there is  $e_1 \in \varphi^{-1}(\bullet b_2)$  such that  $\varphi^{-1}(b_2) \cap m_0^1 = e_1 \bullet$  or
- if  $\bullet b_2 = \emptyset$  then  $\varphi^{-1}(b_2) \cap m_0^1 = \bigcirc \varphi^{-1}(b_2)$

The following proposition states a set of conditions under which reachable markings are reflected by  $\alpha$ -morphisms.

**Proposition 2.** *Let  $\varphi : N_1 \rightarrow N_2$  be an  $\alpha$ -morphism such that  $N_1$  is well marked w.r.t.  $\varphi$  and  $\varphi^S \circ u$  be an  $\alpha$ -morphism from  $Unf(S_1(b_2))$  to  $S_2(b_2)$  then, for all  $m_2 \in [m_0^2]$ , there is  $m_1 \in [m_0^1]$  such that  $\varphi(m_1) = m_2$ .*



*Proof.* We will actually show a slightly stronger property, namely that  $m_1$  can be chosen so that its intersection with the set of conditions in the bubble refining  $b_2$  only contains elements in  $(N_1(\varphi^{-1}(b_2)))^\circ$ . The proof is by induction on the length of a firing sequence  $\sigma$  from  $m_0^2$  to  $m_2$ .

Suppose  $|\sigma| = 0$ . Then  $m_2 = m_0^2$ . By definition,  $\varphi(m_0^1) = m_0^2$ . If  $b_2 \notin m_0^2$ , then  $m_0^1 \cap \varphi^{-1}(b_2) = \emptyset$ . If  $b_2 \in m_0^2$ , then we use Lemma 3 to reach in  $N_1$  a marking in the bubble of  $b_2$  that contains only out-conditions, and we are done.

Suppose now  $|\sigma| = n+1$ . Then we can write  $\sigma = \sigma_1 e_2$ , with  $m_0^2[\sigma_1]m_1^2[e_2]m_2$ . By the induction hypothesis, there is  $m_1^1 \in [m_0^1]$  such that  $\varphi(m_1^1) = m_1^2$  and  $m_1^1 \cap \varphi^{-1}(b_2) \subseteq (N_1(\varphi^{-1}(b_2)))^\circ$ .

Since  $\varphi$  is surjective, there is at least one event in  $E_1$  that  $\varphi$  maps on  $e_2$ . If  $b_2 \notin \bullet e_2$ , then there exists  $e_1 \in \varphi^{-1}(e_2)$  such that  $m_1^1[e_1]$ . If  $b_2 \in \bullet e_2$ , by Lemma 3 there exists  $e_1 \in \varphi^{-1}(e_2)$  such that  $m_1^1[e_1]$ .  $\square$

Let  $N_i = (B_i, E_i, F_i, m_0^i)$  be a SMD-EN System for  $i = 1, 2$  and let  $\varphi : N_1 \rightarrow N_2$  be an  $\alpha$ -morphism. By using  $\varphi$ , the labelling functions are defined such that  $E_2$  are all observable, i.e.:  $l_2$  is the identity function, and the invisible events of  $N_1$  are the ones mapped to conditions, i.e.:

$$\forall e \in E_1 : l_1(e) = \begin{cases} \varphi(e) & \text{if } \varphi(e) \in E_2 \\ \tau & \text{otherwise} \end{cases}$$

From Prop. 1 and Prop. 2, it then follows that  $N_1$  and  $N_2$  are bisimilar.

**Proposition 3.** *Let  $\varphi : N_1 \rightarrow N_2$  be an  $\alpha$ -morphism such that  $N_1$  is well marked and  $\varphi^S \circ u$  is an  $\alpha$ -morphism from  $Unf(S_1(b_2))$  to  $S_2(b_2)$  then,  $(N_1, l_1)$  and  $(N_2, l_2)$  are bisimilar  $(N_1, l_1) \approx (N_2, l_2)$ .*

Prop. 2 and Prop. 3 are stated in the case in which only one condition is refined, but they can be generalized to multiple refinements, provided that in the neighbourhood of each event of  $N_2$  there is, at most, one refined condition. The examples in Fig. 5 show why this constraint is required.

## 5 Relations with Winskel morphisms

Let us now study the relation between  $\omega$ -morphisms and Winskel morphisms, as defined in [13].

A Winskel morphism from  $N_1$  to  $N_2$  is a pair  $(\eta, \beta)$  with  $\eta : E_1 \rightarrow_* E_2$  partial function and  $\beta : B_1 \rightarrow B_2$  finitary multirelation such that  $\beta(m_0^1) = m_0^2$  and  $\forall e \in E : \bullet(\eta(e)) = \beta(\bullet e)$  and  $(\eta(e))^\bullet = \beta(e^\bullet)$ . Note that if  $\eta(e)$  is undefined,  $\beta(\bullet e)$  and  $\beta(e^\bullet)$  should be the empty set.

Given an  $\omega$ -morphism from  $N_1$  to  $N_2$  we associate to it a Winskel morphism. This is possible by adding or deleting conditions to  $N_1$ , if needed. These conditions are *representations* of the abstract conditions refined in  $N_1$ . The obtained net is *canonical* with respect to  $\varphi$  as in the following definition.

**Definition 11.** Let  $\varphi : X_1 \rightarrow X_2$  be an  $\omega$ -morphism from  $N_1$  to  $N_2$ .  $N_1$  is canonical with respect to  $\varphi$  if every bubble,  $\varphi^{-1}(b_2)$  with  $b_2 \in B_2$ , contains one condition,  $b_1 \in \varphi^{-1}(b_2) \cap B_1$ , that satisfies the following constraints:

- $b_1 \in m_0^1 \Leftrightarrow b_2 \in m_0^2$ ;
- $\bullet b_1 = \varphi^{-1}(\bullet b_2)$ ;
- $b_1 \bullet = \varphi^{-1}(b_2 \bullet)$ .

We call that condition  $b_1$  representation of  $b_2$ , denoted  $r_{N_1}(b_2)$ .

If  $N_1$  is not canonical, it is always possible to construct its unique canonical version,  $N_1^C$ , by adding the missing representations, and marking them as their images, or by deleting the multiple ones. The corresponding morphism,  $\varphi^C$ , coincides with  $\varphi$ , plus the mapping of the new conditions on the corresponding conditions of  $N_2$ . It is easy to verify that the canonical version of a system, with respect to an  $\omega$ -morphism to another SMD-EN Systems, is unique up to isomorphisms.

**Proposition 4.**  $\varphi^C$  is an  $\omega$ -morphism from  $N_1^C$  to  $N_2$ .

Take  $N_1^C$ ,  $N_2$  and  $\varphi^C$ . Now, restrict  $\varphi^C$  to all the nodes of  $N_1^C$  that are not in a bubble  $\varphi^{-1}(b_2)$ , but for  $r_{N_1}(b_2)$ , for some  $b_2 \in B_2$  and call it  $(\varphi^C)^{rep}$ .

**Proposition 5.**  $((\varphi^C)^{rep} \downarrow E_1^C, (\varphi^C)^{rep} \downarrow B_1^C)$  is a Winskel morphism.

Any  $\alpha$ -morphism is an  $\omega$ -morphism. Adding to  $N_1$  the representation of each condition does not modify the behaviour, because of the constraint on sequential components. Hence, the result stated here hold for  $\alpha$ -morphisms. In this sense, we consider them as a special case of Winskel morphisms.

## 6 Conclusions

We have presented a notion of morphism for a basic class of Petri nets with the aim of supporting refinement/abstraction of local states. The morphism, in fact, formalizes the relation between a refined net system and an abstract one, by replacing local states of the target net system with subnets. The main idea is that if one starts with an abstract model with some required behavioural properties, then, by refining local states with subnets respecting some constraints, the refined net system will maintain the required behavioural properties. Indeed, the main results concern behavioural properties preserved and reflected by the morphisms. In particular, reachable markings are preserved, and we have characterized some conditions under which reachable markings are reflected, and under which the morphisms induce a bisimulation between net systems. Since bisimulation preserves deadlock freeness, this implies for example that, starting from a deadlock-free abstract system it is possible to refine it obtaining a system which is still deadlock-free. The constraints in order to preserve/reflect behavioural properties are structural and behavioural, where the behavioural ones are only local. On this morphism in [2], we have defined a notion of composition based on interface in the line of [4]. For what concerns future work, we plan to study the constraints under which this morphism can be defined for P/T nets and Coloured nets.

**Acknowledgments** Work partially supported by MIUR.

## References

1. Marek A. Bednarczyk and Andrzej M. Borzyszkowski. On concurrent realization of reactive systems and their morphisms. In Hartmut Ehrig, Gabriel Juhás, Julia Padberg, and Grzegorz Rozenberg, editors, *Unifying Petri Nets*, volume 2128 of *Lecture Notes in Computer Science*, pages 346–379. Springer, 2001.
2. Luca Bernardinello, Elisabetta Mangioni, and Lucia Pomello. Composition of elementary net systems based on  $\alpha$ -morphisms. In Proc. Workshop Componet 2012, Hamburg 2012.
3. Luca Bernardinello, Elisabetta Mangioni, and Lucia Pomello. Local state refinement on elementary net systems: an approach based on morphisms. Internal report (2012), available at <http://www.mc3.disco.unimib.it/pub/bmp2012-def.pdf>.
4. Luca Bernardinello, Elena Monticelli, and Lucia Pomello. On preserving structural and behavioural properties by composing net systems on interfaces. *Fundam. Inform.*, 80(1-3):31–47, 2007.
5. Wilfried Brauer, Robert Gold, and Walter Vogler. A survey of behaviour and equivalence preserving refinements of Petri nets. *Advances in Petri Nets 1990*, pages 1–46, 1991.
6. 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.
7. Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
8. Mogens Nielsen, Grzegorz Rozenberg, and P. S. Thiagarajan. Elementary transition systems and refinement. *Acta Inf.*, 29(6/7):555–578, 1992.
9. Julia Padberg and Milan Urbásek. Rule-based refinement of Petri nets: A survey. In Hartmut Ehrig, Wolfgang Reisig, Grzegorz Rozenberg, and Herbert Weber, editors, *Petri Net Technology for Communication-Based Systems*, volume 2472 of *Lecture Notes in Computer Science*, pages 161–196. Springer, 2003.
10. Lucia Pomello, Grzegorz Rozenberg, and Carla Simone. A survey of equivalence notions for net based systems. In Grzegorz Rozenberg, editor, *Advances in Petri Nets: The DEMON Project*, volume 609 of *Lecture Notes in Computer Science*, pages 410–472. Springer, 1992.
11. Grzegorz Rozenberg and Joost Engelfriet. Elementary net systems. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*, pages 12–121. Springer, 1996.
12. Walter Vogler. Executions: A new partial-order semantics of Petri nets. *Theor. Comput. Sci.*, 91(2):205–238, 1991.
13. Glynn Winskel. Petri nets, algebras, morphisms, and compositionality. *Inf. Comput.*, 72(3):197–238, 1987.

# Context Petri Nets

## Enabling Consistent Composition of Context-Dependent Behavior\*

Nicolás Cardozo<sup>1,2</sup>, Jorge Vallejos<sup>2</sup>, Sebastián González<sup>1</sup>,  
Kim Mens<sup>1</sup>, and Theo D'Hondt<sup>2</sup>

<sup>1</sup> ICTEAM Institute, Université catholique de Louvain  
Place Sainte-Barbe 2, 1348 Louvain-la-Neuve, Belgium  
{nicolas.cardozo, s.gonzalez, kim.mens}@uclouvain.be

<sup>2</sup> Software Languages Lab, Vrije Universiteit Brussel  
Pleinlaan 2, 1050 Brussels, Belgium  
{jvallejo, tjdhondt}@vub.ac.be

**Abstract.** Ensuring the consistent composition of context-dependent behavior is a major challenge in context-aware systems. Developers have to manually identify and validate existing interactions between behavioral adaptations, which is far from trivial. This paper presents a run-time model for the consistency management of context-dependent behavior, called context Petri nets. Context Petri nets provide a concrete representation of the execution context of a system, in which it is possible to represent the interactions due to dynamic and concurrent context changes. In addition, our model allows the definition of dependency relations between contexts, which are internally managed to avoid inconsistencies. We have successfully integrated context Petri nets with Subjective-C, a context-oriented programming language. We show how our model can be cleanly combined with the abstractions of the language to define and manage context-dependent behavior.

## 1 Introduction

Current sensing technology allows computing devices to be highly aware of their execution environment. To leverage the full potential of these sensing capacities, software systems should properly represent the sensed context and dynamically adapt their behavior accordingly. To develop such systems, the Context-Oriented Programming (COP) paradigm has emerged [4], which enables the definition and composition of context-dependent behavioral adaptations. However, consistently composing behavioral adaptations is still challenging. Developers need to manually ensure that insertion and withdrawal of adaptations preserve the expected behavior of the system. Different approaches have been proposed to prevent inconsistencies by defining *dependency relations* between contexts and

---

\* This work has been supported by the ICT Impulse Programme of the Brussels Institute for Research and Innovation.

their associated behavioral adaptations [6,8]. These dependencies constrain context interaction by conditioning the deployment of behavioral adaptations at a high abstraction level, which is well-suited to developers. Nevertheless, developers still have to manually check consistency of such interactions, which is far from trivial.

We claim that inconsistencies in the composition of context-dependent behavioral adaptations arise mainly by the *multiple* and *dynamic* nature of the system's context (an heterogeneous collection of data which can vary dynamically over time, and from one location to another). Without the appropriate support to represent such context and to deal with their dependencies and dynamic changes, it is often difficult to ensure that the behavioral adaptations associated to them do not interfere with each other. We then propose a Petri net-based execution model for context-oriented programming, called context Petri nets (CoPN), which enable a consistent representation and management of the context of a system. In our model, context changes are modeled as dynamic context activations and deactivations. Dependency relations between contexts are expressed by connecting activation/deactivation actions of different contexts. In addition, context Petri nets provide a concrete view of the system's state at every point in time, easing consistency management. Every activation/deactivation that generates an inconsistent state is immediately retracted to the state before its execution.

The remainder of the paper is organized as follows. Section 2 gives a brief background on COP, putting forward the requirements to provide consistent composition of behavioral adaptations. Section 3 presents the foundations of our context Petri nets model, and Section 4 explains how this model fulfills the composition requirements. Section 5 and 6 assess the approach, its relation to existing work, and possibilities for future work. Section 7 concludes the paper.

## 2 Requirements for Consistent Composition of Context-Dependent Behavior

Context-Oriented Programming (COP) allows software systems to adapt their behavior dynamically according to changes detected in their execution environment [4]. The core characterization of COP systems from which we start comprises the following concepts:

- *Contexts* represent particular situations detected during the execution of an application, with respect to which application behavior can be adapted as deemed appropriate.
- *Context activation* takes place whenever the situation for which the context stands is detected in the execution environment; correspondingly, *context deactivation* takes place when the given situation no longer occurs in the execution environment.

COP allows systems to define behavioral adaptations which are associated to particular contexts. Therefore, the adaptations are dynamically composed

with the system's basic functionality whenever the contexts become active. As illustration of contexts and context activations, consider the case of a context-aware mobile phone with Internet connectivity. The phone can gain access to the Internet by means of three different technologies: **WiFi**, **3G** and **Edge**. These contexts are active whenever the respective protocol is available, and inactive otherwise. The fact that the phone has any connectivity at all is signaled by the activation of a **Connection** context. Such activation follows the activation of **WiFi**, **3G** or **Edge**. Besides Internet connectivity, the phone also supports video calls. When **Connection** is active, video calls become possible, a situation that is signaled by activating the **VideoCall** context. Video calls require that there is enough battery power left —that is, for the **HighBattery** context to be active.

Although simple, this scenario already shows two peculiarities of contexts and context activation:

**Dynamicity** The activation state of contexts changes unannounced over time as different situations are detected in the execution environment.

**Multiplicity** Multiple contexts can be active at the same time, including the case that a same context can be activated more than once.

As an example of dynamicity, the **WiFi** context can be active intermittently as the user roams around in the city and wireless networks are found and left behind. As for multiplicity, the **Connection** context can be activated as much as three times, depending on whether **WiFi**, **3G** or **Edge** are available.<sup>3</sup>

The dynamicity and multiplicity of context activation can compromise the behavioral consistency of COP systems. For instance, inconsistencies can arise if adapted behavior is withdrawn from the system while it is executing [10] —a consequence of dynamicity. Inconsistencies can also arise when the adaptations of an active context contradict the adaptations of another active context —a consequence of multiplicity. Hence, to ensure consistent composition of behavioral adaptations, a COP system should provide support to cope with dynamicity and multiplicity. This means that the following requirements should be fulfilled:

**R.1 Dynamic Context Activation and Deactivation** Provide a consistent representation of the system's context. This implies that dynamic context changes should be clearly reflected in the system as they are detected in the system's execution environment.

**R.2 Consistent Interaction Between Multiple Contexts** Ensure that programs are always in a consistent state, even after a context activation or deactivation. In case of multiple activations of different contexts, the model should prevent contexts that interfere with each other to be active at the same time.

**R.3 Multiple Activations of the Same Context** Allow that a context is activated as many times as different instances of the situation represented by the context actually occur in the execution environment.

At present, we observe that no single COP approach appropriately support these three requirements. Most COP languages [4,8,9,14] define dedicated constructs for context *activation* and *deactivation*. However, only few of them

<sup>3</sup> The concept of multiple context activation is analogous to that of multisets in mathematics, in which an element can appear more than once in the multiset.

provide means to specify constraints between contexts. Subjective-C [8] defines language abstractions to specify dependency relations which are internally represented and managed using a dependency graph. ContextL [6] and EventCJ [14] allow defining context interactions programmatically by means of transition functions. The main problem with these approaches is that they require that developers manually verify the consistency of the context dependencies. This means that they need to check every possible interaction between context (de)activations. Furthermore, these approaches do not provide a structured way to compose context dependencies. As a result of this, developers have to manually encode the composition which is cumbersome, error-prone and typically leads to programs that are difficult to understand and maintain.

Concerning the multiple context activations, ContextL, EventCJ, and other COP languages that follow similar design decisions allow contexts to be activated only once. Subjective-C and Ambience [9], on the other hand, allow contexts to be activated as many times as necessary.

We now proceed to explain our proposal to address these requirements using a formal tool from the realm of concurrent systems modeling.

### 3 Context Petri Nets

To ensure the consistent composition of behavioral adaptations, we introduce a run-time model for COP called *context Petri nets (CoPN)*.<sup>4</sup> CoPN (read *co-pen*) is a Petri net-based formalism based on three variants of Petri nets: reactive Petri nets [7], static priorities [1], and inhibitor arcs [2]. Petri nets have been used extensively to describe the information control flow of non-deterministic, concurrent systems. This makes such a formalism suitable to cope with the dynamicity and multiplicity of context in software systems. In this section, we explain how to use CoPN to model contexts, dependencies between contexts, and the composition between such dependencies. We then discuss how the execution semantics of our model ensures that contexts can be always consistently activated.

#### 3.1 Structure of CoPNs

The CoPN model follows the definition of reactive Petri nets with inhibitor arcs and static priorities shown in Table 1. The components of a CoPN are defined by the tuple  $\mathcal{P} = \langle P, T, f, f_o, \rho, m_0 \rangle$  (1), where  $P$  is a finite set of places,  $T$  is a finite set of transitions,  $f$  is the flow function defining regular *arcs* between places and transitions,  $f_o$  is the flow function defining *inhibitor arcs* between places and transitions,  $\rho$  is a function defining *priorities* of transitions, and  $m_0$  is the initial marking function assigning *tokens* to places. This description of CoPNs follows from their formal definition [3].

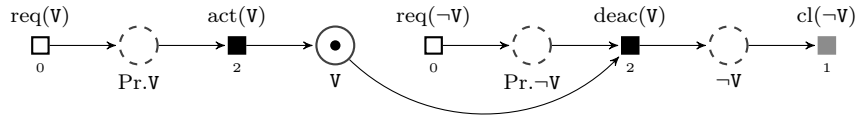
<sup>4</sup> CoPNs are fully implemented as a run-time model for the Subjective-C [8] language. The implementation is available for download at <http://released.info.ucl.ac.be/Tools/Context-PetriNets>.

(1) $\mathcal{P} = \langle P, T, f, f_o, \rho, m_0 \rangle$	(5) $f : (P \times T) \cup (T \times P) \rightarrow \mathbb{Z}^+$
(2) $P \cap T = \emptyset$	(6) $f_o : P \times T \rightarrow \{0, 1\}$
(3) $P = P_c \cup P_t$	(7) $\rho : T \rightarrow \mathbb{Z}^+$
(4) $T = T_e \cup T_i \cup T_c$	(8) $m_0 : P \rightarrow \mathbb{Z}^+$

Table 1: Context Petri nets components definition.

Places and transitions are disjoint sets (2). The set of places is divided into two disjoint sets:  $P_c$  of *context places*, and  $P_t$  of *temporary places* (3). The set of transitions is divided into three disjoint sets:  $T_e$  of *external transitions*,  $T_i$  of *internal transitions* and  $T_c$  of *internal cleaning transitions* (4). There cannot be arcs between two places or two transitions. Each arc defines how many tokens flow from, or to places (5). There can be maximum one inhibitor arc between a place and a transition (6). Transitions are given a firing order priority. Higher priority transitions fire before lower priority ones (7). Enabled transitions of the same priority fire randomly. Finally, tokens are assigned to places by means of the (initial) marking function (8).

An explanation of the mapping between Petri nets and COP concepts follows. As illustration, Fig. 1 shows how the `VideoCall` context from the example in Section 2 can be defined as a CoPN.

Fig. 1: CoPN representation of the `VideoCall` (V) context.

**Places** in CoPNs are used to capture the state of contexts. A context is defined in terms of four places defining the context's life cycle. A *context place*,  $P_c$ , (solid-line circle labeled `VideoCall` in Fig. 1) is used to represent the actual context and its activation state. The other three *temporary places*,  $P_t$ , (dashed circles in Fig. 1) are used to represent intermediate states of the context: preparing for activation (`Pr.VideoCall`), preparing for deactivation (`Pr.-VideoCall`), and flagged as already deactivated (`-VideoCall`).

Temporary places help to maintain consistency constraints when manipulating the activation state of contexts. Activation and deactivation of a context does not occur immediately, but needs to be requested first and processed carefully, since the request may be denied if it violates constraints imposed by other contexts. The *flag* temporary place (`-VideoCall` in Fig. 1) is used to ensure that a context is effectively deactivated once for every deactivation request (otherwise, the context would be emptied of all its tokens after just a single deactivation).

**Transitions** in CoPNs represent changes in the activation state of contexts. Transitions are divided in two categories: external and internal. *External transitions* (white squares in Fig. 1) are used to *request* a context activation or deactivation in response to a change detected in the execution environment. *Internal transitions* (black squares in Fig. 1) forward the requests to other dependent contexts, and trigger the actual activation or deactivation of contexts. Finally, a particular kind of internal transition *internal cleaning transitions* (gray square in Fig. 1) is used to *clean* the deactivation flag place.



Transition priorities are shown as small numbers under each transition in Fig. 1. External transitions are white transitions of priority 0. Internal transitions are black transitions of priority 2. Internal cleaning transitions are gray transitions of priority 1. Transition priorities are unequivocally identified by the transition color, hence priorities will be omitted in future.

**Tokens** represent the activation state of a context, depending on the place they occupy. In Fig. 1 the `VideoCall` context is active if its context place (labeled `VideoCall`) is marked, preparing for activation if place `Pr.VideoCall` is marked, preparing for deactivation if place `Pr.¬VideoCall` is marked, and already deactivated if place `¬VideoCall` is marked.

**Arcs** encode the possible ways in which tokens can flow from one place to another, mediated by transitions. Hence, arcs help encoding the way context activations and deactivations depend on each other. *Regular arcs*, noted as arrow-headed edges ( $\rightarrow$ ), permit to verify the presence of tokens in a place, thanks to the  $f$  flow function. *Inhibitor arcs*, depicted as circle-ended edges ( $\rightarrow\circ$ ), permit to verify the absence of tokens in a place, by means of the  $f_\circ$  flow function. Inhibitors are used for example to express that a context cannot be activated if another context is active.

### 3.2 Dynamics of CoPNs

CoPNs make it possible to represent and track the changes that occur in the system's execution environment. CoPNs can thereby be used as run-time representation of context. The following descriptions define the way context state is encoded in a CoPN, and how it evolves according to the constraints encoded in the structure of such CoPN.

- A transition  $t$  is *enabled* if its input places  $p_i$  from regular arcs contain at least  $f(p_i, t)$  tokens, its input places  $p_\circ$  from inhibitor arcs are empty, and no other transition  $t'$  with higher priority,  $\rho(t') > \rho(t)$ , is enabled.
- Transition *firing* modifies the state of the Petri net by removing as many as  $f(p_i, t)$  tokens from its input places  $p_i$ , and adding as many as  $f(t, p_{out})$  tokens to its output places  $p_{out}$ .
- External transitions are fired with the regular *may* fire semantics of Petri nets. That is, if a transition is enabled it may fire. In our model external transitions are fired as consequence of a change in the execution environment.
- Internal transitions are fired with a *must* fire semantics. That is, if an internal transition is enabled it must fire. Internal transitions are used to coordinate activation and deactivation among different contexts, according to the dependency relations established between them. Section 3.3 describes such dependencies.

CoPN model is used to ensure consistency of context activations, we define a CoPN to be in a *consistent state* if no temporary place is marked after *all* enabled internal transitions have fired.

### 3.3 Dependency Relations Between CoPNs

CoPN allows multiple activations of different contexts. To avoid conflicts in the adaptations of the different active contexts, our model enables the definition of dependency relations between contexts. We have taken as starting point the four dependency relations defined in Subjective-C [8], and modeled them in CoPN: *exclusion*, *weak inclusion*, *strong inclusion* and *requirement*. Each dependency relation defines how the activation state of a context influences that of another context. In CoPNs, this is achieved by connecting internal transitions of one context to the (temporary) places of another one, via an arc. Each arc expresses a *rule* describing the interaction between contexts. New dependency relations could be defined by describing such rules.

**Exclusion.** An exclusion dependency prevents two contexts from being active at the same time. However, both contexts may be simultaneously inactive. For example, the interaction between the *LowBattery* (L) and *HighBattery* (H) contexts of the mobile phone is defined by the CoPN shown in Fig. 2. These contexts clearly should not be active at the same time. If one of the contexts is active, the activation of the other is prevented by the corresponding inhibitor arc.

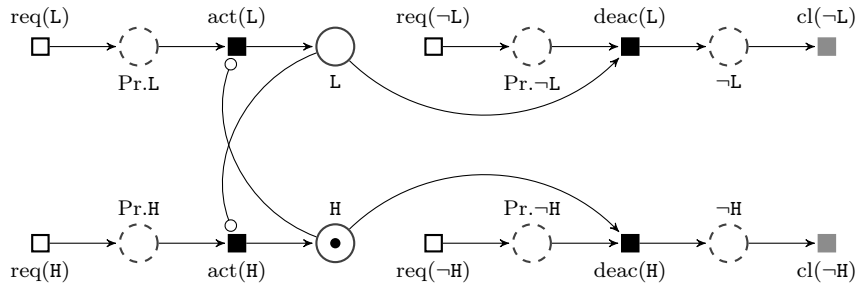


Fig. 2: Exclusion between Low Battery (L) and HighBattery (H).

Firing a request for activating the L context,  $req(L)$ , under the initial marking  $m_0(H)=1$  yields the marking  $m_1$ , where  $m_1(H)=1$  and  $m_1(Pr.L)=1$ . At this point, none of the internal transitions is enabled. In particular,  $act(L)$  is not enabled because of the inhibitor arc  $(H, act(L))$ . An inconsistent state has been reached since one of the temporary places,  $Pr.L$ , is marked. In this case, all of the the actions are reverted to the initial marking state. The request for the activation is denied, and the user is informed about the reason for the refusing the activation.

**Weak Inclusion** A weak inclusion represents a situation in which the activation (deactivation) of a context should automatically trigger the activation (deactivation) of another context. Note that the latter context can be activated or deactivated independently of (without effect on) the former. This interaction is shown in Fig. 3, using as example the case of the *Connectivity* and *VideoCall* contexts: activation of *Connectivity* automatically triggers the activation of *VideoCall*, meaning that video calls are normally available whenever the phone is connected to the Internet). The double arc in Fig. 3 is a visual shortcut that stands for two different arcs going in opposite directions.

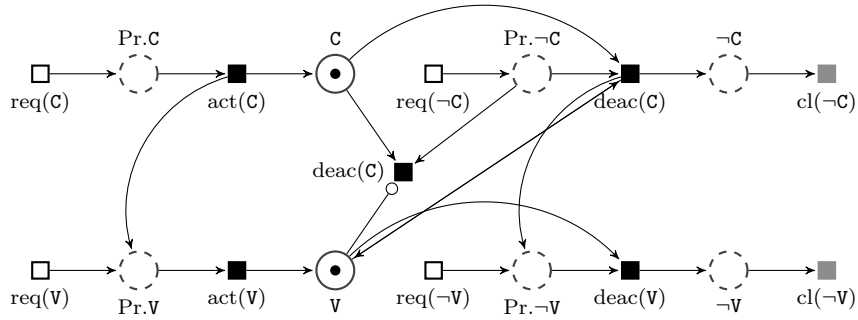


Fig. 3: Weak inclusion from Connectivity (C) to VideoCall (V).

**Strong Inclusion** A strong inclusion represents a dependency in which, similarly to a weak inclusion, activation or deactivation of a context triggers that of the related context. Additionally, deactivation of the latter context triggers back the deactivation of the former. These interactions are encoded by the CoPN shown in Fig. 4; as in weak inclusion, the double arc stands for two different arcs going in opposite directions. The CoPN encodes an interaction such that activation of WiFi results in the activation of Connectivity; reciprocally, if for some reason Connectivity is deactivated, then WiFi will also be deactivated.

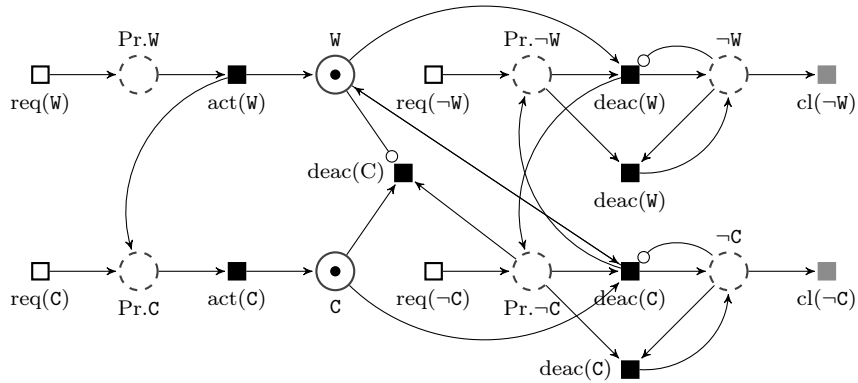


Fig. 4: Strong inclusion from WiFi (W) to Connectivity (C).

**Requirement** A requirement represents the situation in which activation of a context is possible only if another context is already active. This restriction implies that when the latter context is no longer active the former context must be deactivated. The CoPN corresponding to this interaction is shown in Fig. 5: VideoCall can be activated only if HighBattery is already active.

Thus far, contexts and dependency relations have been discussed as isolated CoPNs in the system. We now explain how different CoPNs can be composed to form a unified CoPN that the system can use as run-time model of the execution environment as a whole.

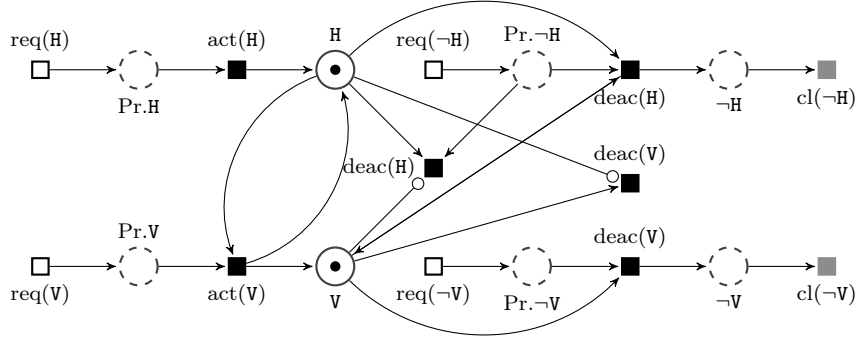


Fig. 5: Requirement of HighBattery (H) by VideoCall (V).

### 3.4 Composing Context Dependency Relations

This section provides an intuitive description of the steps needed to compose CoPNs.<sup>5</sup> A context only interacts with other contexts directly related to it. This provides modularity to the composition mechanism, because when composing, Petri net elements (arcs) are added only between the contexts being composed.

To preserve the semantics of dependency relations, CoPN composition extends the place combination technique of Petri nets [19]. As mentioned in the previous section, each dependency relation is comprised of a set of rules. Such rules must be verified to hold in the composed CoPN, after combining corresponding places and transitions. The verification process may add additional arcs when needed, to satisfy the rules.

Snippet 1 shows pseudo-code describing the composition of CoPNs. We explain the composition by means of an example of two dependency relations  $R_1(C_1, C)$  and  $R_2(C_2, C)$  between contexts  $C_1, C_2$  and  $C$ . For simplicity, we assume that the two relations are to be composed into an empty CoPN  $\mathcal{P}$ . The first step in the composition is to combine the  $C$  context common to both relations. This is done by taking the union of all corresponding elements associated to each context –that is, elements with the same label; inputs and outputs are collapsed into one (lines 3–6). Second, for all existing dependency relations in the CoPN each rule is checked to ensure that it is satisfied. Additional arcs might be added for transitions that match a rule but do not satisfy it (lines 7–10).

---

```

1 add  $C_1$  to  $\mathcal{P}$ 
2 add  $C_2$  to  $\mathcal{P}$ 
3 loop for  $e_1$  such that  $e_1 \in P_C \cup T_C$  in  $R_1$ 
4    $e_2$  such that  $e_2 \in P_C \cup T_C$  in  $R_2$ 
5     add  $e_1$  to  $\mathcal{P}$ 
6     if  $e_1 \neq e_2$  then add  $e_2$  to  $\mathcal{P}$ 
7 loop for  $R$  dependency relation in  $\mathcal{P}$ 
8    $c$  constrain rule in  $R$ 
9    $t$  transition in  $\mathcal{P}$ 
10  if  $t$  does not satisfy  $c$  then add new arc  $(t, c)$  to  $\mathcal{P}$ 

```

---

Snippet 1: CoPN composition algorithm.

<sup>5</sup> A full formal description of composition in CoPNs falls outside the scope of this paper, but it is available as technical report [3].

### 3.5 Programming Support for Context Petri Nets

The CoPN model can become complex as the system grows. However, developers interact with it through a language abstraction layer that hides such complexity. This section presents the context-oriented constructs of Subjective-C, and how these map to the underlying CoPN model.

```

Context declaration ::= @context( context-name [, bound] )
Context activation ::= @activate( context-name )
Context deactivation ::= @deactivate( context-name )
Dependency relations declaration ::=
  [ addExclusionBetween: context-name and: context-name ]
  [ addWeakInclusionFrom: context-name to: context-name ]
  [ addStrongInclusionFrom: context-name to: context-name ]
  [ addRequirementTo: context-name of: context-name ]

```

Fig. 6: Subjective-C method syntax to interact with CoPNs.

Fig. 6 shows the language constructs available in Subjective-C for the creation and manipulation of contexts, and hence CoPNs. A *context declaration* automatically generates a context structure as that of Fig. 1. The maximum number of times a context can be activated can be *bounded* by a positive integer. Context *activation* and *deactivation* fire the corresponding external transitions in the underlying CoPN, for example *req(VideoCall)* and *req(¬VideoCall)* in Fig. 1. Finally, a *dependency relation declaration* specifies the different dependency relations between two contexts, as described in Section 3.3.

For illustration, Snippet 2 shows definitions for `LowBattery` and `HighBattery` contexts. Lines 1 and 2 generate a CoPN as that of Fig. 1 for each context. The exclusion dependency defined between the two contexts in line 3 yields the CoPN shown in Fig. 2. Line 4 is the activation of the `LowBattery` context which (when successful) installs the behavior adaptations associated to it. Due to the `LowBattery` context being active, activation of the `HighBattery` context in Line 5 is denied and the cause of the denial is given to the user.

---

```

1 SContext *lb = @context(LowBattery);
2 SContext *hb = @context(HighBattery);
3 [addExclusionBetween: lb and: hb];
4 @activate(LowBattery);
5 @activate(HighBattery);

```

---

Snippet 2: Example of exclusion dependency declaration.

## 4 Consistent Composition of Context-Dependent Behavior in CoPN

Having explained the core of the CoPN model in Section 3, we now turn to the question of how the model satisfies the requirements for consistent composition of context-dependent behavior put forward in Section 2.

#### 4.1 Dynamic Context Activation and Deactivation (R.1)

CoPN provides a concrete representation of the system's context. The dynamic activation and deactivation of a context uses the definition of *consistent state* for a CoPN given in Section 3.2, which is ensured by the following process:

- Before external transitions are fired, the set of current active contexts is saved as the *current marking* of the system.
- If an inconsistency exists after firing all enabled internal transitions (that is, if a temporary place is still marked), *all* modifications made since the external transition firing are reverted. This is done by reinstating the previously saved current marking.
- In case an inconsistency exists, a message is prompt to the user with the reason preventing the activation or deactivation to take place.
- If the system reaches a consistent state, the current marking is updated to the marking found in the CoPN. A trace of all fired internal transitions is given to the user.

As an example, consider the discovery of a `Wifi` network connection in the mobile phone. The initial marking  $m_0$  of the CoPN representing the `Wifi` context is  $m_0(\text{Wifi})=0$  which is a consistent state. When a `Wifi` network connection is discovered, this generates an `@activate(Wifi)` message. The transition to request the context activation is fired,  $req(\text{Wifi})$ , adding a token to the temporary place `Pr.Wifi`. This changes the initial marking  $m_0$  to a new marking  $m_1$ , where  $m_1(\text{Pr.Wifi})=1$ . Such a marking enables the internal transition  $act(\text{Wifi})$  which now *must* fire according to the internal transition semantics described in Section 3.2. The firing moves the token from `Pr.Wifi` to `Wifi` yielding a marking  $m_2$  where  $m_2(\text{Wifi})=1$ . At this point none of the internal transitions is enabled, and none of the temporary places are marked. Therefore, the CoPN is in a *consistent state*. The case of context deactivation is similar to the context activation one.

#### 4.2 Consistent Interactions Between Multiple Contexts (R.2)

The CoPN model ensures the consistent state of a system in presence of multiple active contexts by means of dependency relations and dynamic context activations. As explained in Section 3.3, CoPN currently supports the 4 dependency relations defined in Subjective-C. Dependency relations encode interactions between contexts. Such interactions define sequences of activations and deactivations that leave the system in a consistent state.

The activation or deactivation of a context is constrained by the existing dependency relations. For example, in the case of the requirement dependency relation of Fig. 5, the `VideoCall` context is only activated when the `HighBattery` context is already active. Were this not be the case, the activation of `VideoCall` would leave the system in an inconsistent state, and is therefore retracted.

#### 4.3 Multiple Activations of the Same Context (R.3)

In CoPN, contexts can be activated multiple times. To support this behavior, CoPNs rely on the fact that a place can hold many tokens at once. Each token

represents an activation of the context. As such, the context (and thus the adaptations associated to this context) will remain available for as long as there are tokens in the context place. Fig. 7 shows an example where three internet protocols are available (e.g., Edge, Wifi, 3G) for the mobile phone. This condition is represented in the **Connectivity** context by three tokens.

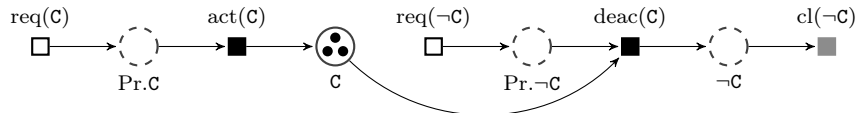


Fig. 7: Context **Connectivity** (C) is active three times.

Unlike existing COP approaches, CoPN allows developers to declare the activation of a context as *multiple* or *single*. For example, the generic **Connectivity** context can be activated multiple times, whereas a specific protocol like **3G** should be activated at most once. This extends existing COP approaches, which support either single-activation contexts (e.g. ContextL [4]), or multiple-activation contexts (e.g. Subjective-C), but not both.

## 5 Related Work

This section reviews related work by going through different context sensing approaches which provide a concrete representation of context, and by considering alternative modeling approaches that could be used for context-aware systems.

### 5.1 Context Representation

The Context Toolkit [20] and WildCat [5] frameworks provide abstractions for the representation of context information. Context information coming from sensors is represented by context objects. Gathered information can be contradictory or inconsistent. It is up to the system/developer to manually manage such inconsistencies.

CORTEX [21] is a middleware architecture that exploits the sentient object paradigm: so-called sentient objects receive events as input (from other sentient objects or sensors), process the events by means of an inference engine and generate further events as output. The sentient object model of CORTEX is intended for pro-active context-aware systems that autonomously invoke some action in response to relevant context changes. In contrast, our model deals with reactive systems. That is, upon a context change, the most appropriate context representation is activated.

These framework approaches provide useful modularization features to manage context information. However, they have little support for the dynamic activation of behavioral adaptations, and managing conflicts between them, making them ill-suited in face of the requirements presented in Section 2.

## 5.2 Alternative Approaches

CoPN serves both as a formal and run-time model of context. We now consider other approaches that could be used for the same purpose.

**State Diagrams** Automata [12] and statecharts [15] are used to describe system behavior based on its possible states, and the set of actions to be taken at each state. Automata and statecharts are normally used to verify system properties, such as program termination. Among the properties provided by these diagrams, composition is the most prominent. However, the system focuses only on one state at a time, this means that every state needs to associate all possible actions in the system. In the context of COP, where context activations represent the actions in the system, every state has to be connected to all such actions, making the model cluttered and complex. Additionally, both automata and statecharts formalisms would need to be extended to allow the interaction between contexts and multiple activations of a context.

**Process Algebra, Coalgebra and Modal Logics** Process algebra [11] is used to model concurrent processes, providing high-level abstractions for operations between processes such as parallel composition, communication, replication, and synchronization. Modal logics [17] have been used to represent necessity and possibility conditions about system properties. Modal logics are mostly used to express temporal conditions, but they also can be used to express conditions like program termination. Coalgebras [13] have been used to express dynamic behavior of systems. Typically, coalgebras specify state-based systems, where the state is considered as a black box and dynamic behavior is reasoned upon in terms of invariance and bisimilarity.

These formal methods could be used to model and reason about context-aware systems. However, concrete models based on these formalisms would need to be extended to match the requirements of Section 2, as we have done with the Petri net extensions used in CoPNs.

## 6 Future Work

Although the CoPN model can help in tackling some of the challenges for the consistent composition of behavioral adaptations, a number of challenging issues need to be further explored.

First, conflicts between external, internal and cleaning transitions are avoided by the separation of each class by their transition priorities. The question still remains, however, if within internal transitions conflicts exist. That is, if firing of a transition disables a previously enabled one, leading to different markings. Although, these type of conflicts are expected from the non-deterministic choice of transitions with the same priority, it should be proven that regardless of the firing order of transitions the same marking is always reached.

Second, CoPN provides consistency of dynamic behavior adaptations. However, the discussion presented in this work focuses on the management of interaction between contexts. How to identify such interactions, remains an open



question. Standard Petri net analysis techniques allow to reason about a system's behavior [16]. Such techniques could be used to identify interaction between contexts. The properties that could be used in the context of COP systems comprise (a) *reachability*, to identify if it is possible to have a particular configuration (i.e. marking) of active contexts, (b) *liveness*, to verify if a context can ever be activated or not, and (c) *persistence*, to spot isolated contexts in the Petri net. This analysis techniques can give upfront information about errors and redundancies in the system. Currently the CoPN model contains inhibitor arcs and is (in principle) unbounded, which makes these properties undecidable. However, the addition of bounds to contexts, and removal of inhibitor arcs when possible [18], could enable the analysis of such properties. We are currently studying which properties can be successfully verified for CoPN.

## 7 Conclusions

Ensuring consistent behavior adaptation of software systems is a challenging task. Inconsistencies in the composition of context-dependent behavior rise from interactions when such behavior is incompatible or contradictory. We identify three main requirements to support behavioral adaptations: dynamic context activation and deactivation, consistent interaction between multiple contexts, and multiple activations of the same context. As a way to address these requirements, this paper presents the context Petri nets (CoPN) model which builds on the dynamic activation and deactivation of contexts provided by context-oriented programming (COP) languages. CoPN uses different Petri net extensions to provide a precise and live representation of context, dependency relations between contexts, and their composition. The CoPN model makes explicit the different states in the activation life cycle of a context to cope with the reactive nature of COP systems, and to ensure that activations and deactivations are consistent. Consistent activation and deactivation of contexts is ensured by dynamically checking context dependency relations. If an inconsistency is encountered, CoPNs allows to rollback the faulty operation to the last registered consistent state. Afterwards, the user is informed of the cause of the error.

For the advantages provided in the management and assurance of consistent dynamic adaptations, context Petri nets are a convenient run-time representation of contexts, their activation and interaction in COP systems.

## References

1. Bause, F.: On the analysis of petri nets with static priorities. In: Acta Informatica. vol. 33, pp. 669 – 685 (1996)
2. Best, E., Koutny, M.: Petri net semantics of priority systems. Theoretical Computer Science 96, 175–215 (April 1992)
3. Cardozo, N., González, S., Mens, K., D'Hondt, T.: Context petri nets: Definition and manipulation. Tech. rep., Université catholique de Louvain and Vrije Universiteit Brussel (April 2012), <http://soft.vub.ac.be/Publications/2012/vub-soft-tr-12-07.pdf>

4. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: an overview of ContextL. In: Proceedings of the Dynamic Languages Symposium. pp. 1–10. ACM Press (Oct 2005), co-located with OOPSLA'05
5. David, P.C., Ledoux, T.: Wildcat: a generic framework for context-aware applications. In: Terzis, S., Donsez, D. (eds.) MPAC. ACM International Conference Proceeding Series, vol. 115, pp. 1–7. ACM (2005)
6. Desmet, B., Vanhaesebrouck, K., Vallejos, J., Costanza, P., Meuter, W.D.: The puzzle approach for designing context-enabled applications. In: XXVI International Conference of the Chilean Computer Science Society. pp. 23 – 29. IEEE (2007)
7. Eshuis, R., Dehnert, J.: Reactive petri nets for workflow modeling. In: Application and Theory of Petri Nets 2003. pp. 296–315. Springer (2003)
8. González, S., Cardozo, N., Mens, K., Cádiz, A., Libbrecht, J.C., Goffaux, J.: Subjective-C: Bringing context to mobile platform programming. In: Proceedings of the International Conference on Software Language Engineering. Lecture Notes in Computer Science, vol. 6563, pp. 246–265. Springer-Verlag (2011)
9. González, S.: Programming in Ambience: Gearing Up for Dynamic Adaptation to Context. Ph.D. thesis, Université catholique de Louvain (Oct 2008), <http://hdl.handle.net/2078.1/19684>, coll. EPL 211/2008. Promoted by Prof. Kim Mens
10. González, S., Mens, K., Heymans, P.: Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In: Proceedings of the Dynamic Languages Symposium. pp. 77–88. ACM Press, New York, NY, USA (Oct 2007), co-located with OOPSLA'07
11. Hennessy, M.: Algebraic Theory of Processes. MIT Press, Cambridge, Mass. (1988)
12. Hopcroft, J.E., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979)
13. Jacobs, B.: Exercises in coalgebraic specification. In: Algebraic and Coalgebraic Methods in the Mathematics of Program Construction. pp. 237–280 (2000)
14. Kamina, T., Aotani, T., Masuhara, H.: Eventcj: A context-oriented programming language with declarative event-based context transition. In: Proceedings of the International Conference on Aspect-Oriented Software Development. pp. 253–264. AOSD'11, ACM Press (Mar 2011)
15. Latella, D., Majzik, I., Massink, M.: Towards a formal operational semantics of uml statechart diagrams. In: Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS). pp. 465–482. Kluwer, B.V., Deventer, The Netherlands (1999)
16. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541 – 580 (April 1989)
17. P. Blalckburn, M. de Rijke, Y.V.: Modal Logic. Cambridge University Press (2001)
18. Reinhardt, K.: Reachability in petri nets with inhibitor arcs. Electronic Notes in Theoretical Computer Science 223, 239–264 (2008)
19. Reisig, W.: Simple composition of nets. In: Proceedings of the 30th International conference on Applications and Theory of Petri Nets. pp. 23 – 42. Springer-Verlag (June 2009)
20. Salber, D., Dey, A.K., Abowd, G.D.: The context toolkit: Aiding the development of context-enabled applications. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 434–441. ACM Press (1999)
21. Sørensen, C.F., Wu, M., Sivaharan, T., Blair, G.S., Okanda, P., Friday, A., Duran-Limon, H.: A context-aware middleware for applications in mobile ad hoc environments. In: Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing. pp. 107–110. MPAC '04, ACM, New York, NY, USA (2004)

# MuPSi - a multitouch Petri net simulator for transition steps

Thomas Irgang<sup>1</sup>, Andreas Harrer<sup>1</sup>, Robin Bergenthum<sup>2</sup>

<sup>1</sup>Lehrstuhl für Angewandte Informatik, Kath. Universitaet Eichstaett  
{thomas.irgang, andreas.harrer}@ku-eichstaett.de

<sup>2</sup>Lehrgebiet Softwaretechnik und Theorie der Programmierung, FernUni Hagen  
robin.bergenthum@fernuni-hagen.de

**Abstract.** Petri nets are very useful for modeling systems with concurrent behavior. There are many editors which support the creation of a Petri net and almost all editors offer the possibility of simulation, i.e. firing of transitions. Due to the limitation to possible user interaction using a conventional user interface having only one focus a concurrent firing of several transitions is not supported. In this paper a Petri net simulator (MuPSi) is presented, which supports the execution of transition steps, i.e. multisets of transitions, using a multitouch user interface. MuPSi enables concurrent execution of transitions in a single or multi-user environment. Simulation of transition steps helps to understand the concurrent behavior of a Petri net.

## 1 Introduction

Petri nets have a clear formal semantic and a simple graphical representation. They are a popular choice for integrated modeling of systems with concurrency in many application areas, such as software engineering, business process modeling, hardware design and controller synthesis. To support the creation of Petri nets there are many editors which differ depending on their field of application. VipTool [1,2] is specialized on partial order semantics of Petri nets. CPN Tools [3] considers an extension of Petri nets, so-called colored Petri nets. WoPeD [4] is developed for teaching purpose. Each of these editors offers the possibility to simulate Petri nets, i.e. they can simulate the firing of a sequence of transitions.

Usually the simulation consists of two steps. The Petri net editor calculates and marks all enabled transitions. Now the user may select and fire one of these transitions. Each firing changes the marking of the Petri net and the set of all enabled transitions is recalculated. This so-called token game is a very simple but also useful way to test and understand a given Petri net.

The token game allows to simulate the sequential behavior of a Petri net. The tool presented in this paper, the MuPSi (Multitouch Petrinetz-Simulator), removes this restriction by enabling the firing of transition steps. A transition step is enabled to fire if all transitions of the step can be executed simultaneously. If a step of transitions is enabled then also each linearization of the transition

step is enabled. For a lot of applications the difference between a step and all its linearizations does not matter. But there are applications where testing and understanding the concurrent behavior of a Petri net is of great value. A good example is in education, because understanding the true concurrent behavior of a Petri net is a reasonable learning objective. Therefore MuPSi allows to simulate and visualize concurrent firing of transitions in an intuitive way which is supported by various input mechanisms. MuPSi can be used as a desktop application as well as in a multitouch environment.

In a learning environment an elegant approach to concurrent firing of transition steps is using a multitouch device. Multitouch environments are becoming increasingly popular as a result of the development and propagation of smartphones and tablet computers. The ability to control multiple pointers in a program allows to select several transitions in a Petri net simultaneously and fire the selected transitions concurrently. For this purpose we have built a multitouch table as shown in Figure 1. With the help of this multitouch table and MuPSi multiple users can play the token game concurrently. If a multiset of transitions is selected to be fired concurrently and there are too little tokens MuPSi supports the users to solve this conflict by a reduction of the chosen transition step.



**Fig. 1.** MuPSi on a multitouch table

The next section discusses the representation of enabled transition steps in Petri nets. At first, in Section 2, different ways to indicate enabled transition steps are developed and evaluated. At second, in Section 3, different ways to allow users the creation of a transition step are discussed. In Section 4 we shortly

discuss how MuPSi supports the reduction of a disabled transition step to an enabled sub-step. Section 5 describes the implementation of MuPSi and Section 6 provides a short outlook to further research questions.

## 2 Representation of enabled transition steps in Petri nets

Petri nets have two different types of nodes: places and transitions. Places can be marked with tokens and a distribution of tokens over all places of the Petri net is called the marking of a Petri net. Transitions and places are connected by directed arcs. MuPSi considers a special class of Petri net so-called place/transition-nets (p/t-nets) [5]. A p/t-net considers arcs with weights and a transition is enabled to fire in a given marking if every place in the preset of the transition carries at least as many tokens as the weight of the arc from the place to the transition indicates. When firing a transition these tokens are consumed and the transition produces new tokens into places in its postset, again according to the weights of the outgoing arcs.

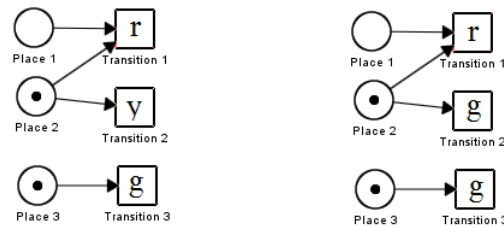
Petri net simulators support the user by highlighting enabled transitions, often by coloring them green. If you allow the firing of transition steps, i.e. a multiset of transitions, the number of tokens in the preset of all transitions given by the transition step must be big enough to enable all transitions concurrently. It happens that one transition is enabled to fire in some transition steps and is not enabled to fire in others. For this reason a strict partition into enabled and not enabled transitions is no longer possible using the step semantic. We extended the color scheme in MuPSi by the additional color yellow. Each yellow colored transition is enabled in the current marking, but there exist other transitions in the Petri net such that if both transitions occur together in a step the step is not enabled. All in all yellow transitions indicate possible conflicts. Green colored transitions symbolize a very low conflict potential, meaning it is possible to combine green transitions with other green transitions in a step of the Petri net. In the following section, we discuss several different approaches on dividing the set of all transitions of a given Petri net and its marking into sets of so-called green, yellow and red transitions. Each approach is implemented in MuPSi and can be useful in different scenarios. In a learning environment it can even be useful to switch between different approaches to get a better understanding of the nature of conflicts in a given Petri net and its concurrent behavior.

### Structural conflicts between transitions

Probably the simplest approach to divide the set of all enabled transitions of a Petri net into two sets of transitions, such that green transitions have low conflict potential and yellow a higher conflict potential, is a simple structural analysis of the Petri net. This approach is independent of the current marking of the Petri net. The idea is that an enabled transition can always occur once in a next step if there is no place in its preset which is also in the preset of another transition. If there exists a shared place the concurrent firing of both transitions can lead

to a conflict. This is a fairly simple method, since the partition of the set into yellow and green transitions is calculated only once. At runtime this solution requires no additional computational effort, since each transition, when enabled, is always marked with the same color, either yellow or green as computed in the beginning.

This approach can be refined by a small amount of additional computational effort. Assuming that the user cannot add a disabled transitions to a step, a disabled transition can never be in conflict. In the second approach a transition is marked green if it is enabled and there is no place in its preset which is shared with another enabled transition. Figure 2 provides a small example for the first two approaches. The left side of the figure shows the first while the right side of the figure shows the second approach. In this paper colors of the transitions can only be shown as labels. The labels are g(reen) ,y(ellow) or r(ed).



**Fig. 2.** Coloring of transitions using structural analysis. Left: first approach. Right: second approach.

The number of green colored transitions in the second approach is greater than or equal to the number of green colored transitions in the first approach, but the partition of green and yellow transitions has to be recalculated after each step, since it depends on the marking of the Petri net. Both approaches achieve that a set of green transitions together with only one yellow transition always is enabled to fire. If a step contains more than one yellow transition or some green colored transitions at least twice, it can happen that this step is not enabled.

### Conflicts between transitions

To further increase the number of green transitions, the third approach analyses the current marking in more detail. The idea is to color transitions green as long as the marking is sufficient to enable all of them concurrently. For this approach, first we try to fire the set of all enabled transitions in one transition step. If there is a place that causes a conflict each transition in its postset is colored yellow. The remaining enabled transitions are colored green.

Again, each set of green colored transitions together with one yellow colored transition is enabled to fire. The third approach is able to color even more



**Fig. 3.** Coloring of transitions using conflicts between enabled transitions. Left: second approach. Right: third approach.

transitions green. That means that a concurrent firing of more than one yellow colored transition leads to a conflict more probably.

**Conflicts between transitions allowing for limited autoconcurrency**

The fourth and fifth approach also consider autoconcurrency among transitions. Till now, we only took care of conflicts that arise in sets of transitions. MuPSi also supports a firing of multisets of transitions. In both approaches we have to fix an upper bound to autoconcurrency of transition occurrence. Defining such a limit is useful because only a transition with an empty preset can fire autoconcurrently without any limitation. Let  $n$  be such an upper bound for the autoconcurrency of all transitions.

In the fourth approach, we test the multiset of transitions which contains each enabled transition  $n$  times. As in approach three, transitions which contain places in its presets, which cause conflicts are colored yellow. All other enabled transitions are colored green. With this approach we can ensure that a step of green transitions is enabled if each transition is contained at most  $n$  times in the transition step. If we choose one as an upper bound, the fourth approach equals to the third approach. It is easy to see that the number of green colored transitions decreases with increasing  $n$ .

The fifth and most complex approach implemented in MuPSi uses the calculation of maximal steps of the Petri net in the given marking. A maximal step of a Petri net is an enabled step which is not included in any other enabled step. If a transition is at least  $n$  times contained in all maximal steps, the transition is enabled to fire at least  $n$  times independently from any other transition occurrence.



**Fig. 4.** Coloring of transitions considering autoconcurrency. Left: fourth approach (bound one and two). Right: approach five (bound one and two).

As a result of this approach each multiset of green colored transitions is enabled, if any transition is contained at most  $n$  times. As a second result we get that in any enabled multiset (consisting of yellow and green transitions) the number of each green transition can be increased to  $n$  without causing any conflicts.

### 3 Selection of a transition step

While playing the token game it is up to the user to select transition steps and fire them. Depending on the scenario different input methods fit better for the selection of the transition steps. One considered possible scenario is MuPSi running on a desktop computer, another scenario is MuPSi running on a multitouch device. MuPSi offers different input methods to fit both scenarios. Remark that while composing a step the coloring of the transitions is not updated, since we assume that each step is build concurrently by different users.

#### Manual trigger

The first and simplest way to select a transition step is to use an explicit manual trigger. The user adds transitions to a step by clicking enabled transitions. After building a transition step the user triggers the firing of the transition step by clicking a fire button. A disadvantage of this input method is that it is not well suited in a multi-user environment without choosing a user having a special role, i.e. a moderator, controlling the fire button. The great advantage of a manual trigger in a single user environment is that the user can build and fire transition steps without time pressure.

#### Fixed time intervals

In some scenarios the use of a fire button is not suitable. In a second approach MuPSi is able to use fixed time intervals. If an interval ends the selected step will be fired if possible. This means that for a fixed time, for example three seconds, transitions are collected and these transitions are fired concurrently at the end of the time interval. This input method is suitable for both: the simulation with a single user using a PC as well as for the collaborative use with a multi-touch device.

#### Sliding time intervals

A more flexible approach than using a fixed time interval is a so-called sliding time interval. If any user clicks a transition a short countdown starts. If a transition is pressed in this time period the transition is added to the transition step and the countdown restarts. This will continue until the countdown expires. This input method is suitable for a multi-touch table as well as for the usage with a PC. Typically the time interval for a single user should be larger than in



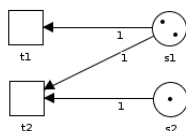
multi-user mode. Transition steps might become very large and more likely not executable if several active users continuously add transitions to a transition step and thereby restart the countdown. This way all transitions which are clicked within a short period of time are never split into two different transition steps. Such a splitting could be caused by using a fixed time interval.

### Time intervals by overlapping of actions

In the last, most elegant approach MuPSi distinguishes between pressing and releasing a transition. As long as any transition is pressed, transitions are added to a transition step. If no transition is pressed the collected step is fired. This input method is of course only usable with a multitouch device [6] and was the real impulse to develop MuPSi. In a single user environment the user may hold one transition down and then collect other transitions with his second hand to build a transition step. If a user wants to fire a transition autoconcurrently in one transition step he may tap one transition while holding the same transition pressed with his other hand. This approach leads to the feeling of real concurrent live simulation of a Petri net.

## 4 Conflict solving

In this section we describe the current conflict solving approach implemented in MuPSi. Since MuPSi is intended as a tool for teaching one of its essential features is to support the users if a disabled transition step is selected. Our approach for conflict solving is to offer the users sub-steps of the conflicting step.



**Fig. 5.** An example for conflict solving

To calculate possible sub-steps we model the problem as a linear integer optimization problem. A linear optimization problem consists of a target function and side conditions which restrict the set of solutions. For our model the interesting side conditions are defined through the net structure and the conflicting step. If we try to fire the step  $\{t_1, t_1, t_2\}$  in the net shown in Figure 5 this leads to a conflict in place  $s_1$ . To define the suitable target function we need to know the users intention, in case we have no further information we can only feedback different solutions using different target functions. Currently MuPSi implements four different target functions: maximization of the consumed tokens, maximization of the produced tokens, the maximal number of transition occurrence or a maximal number of occurrence of different transitions.

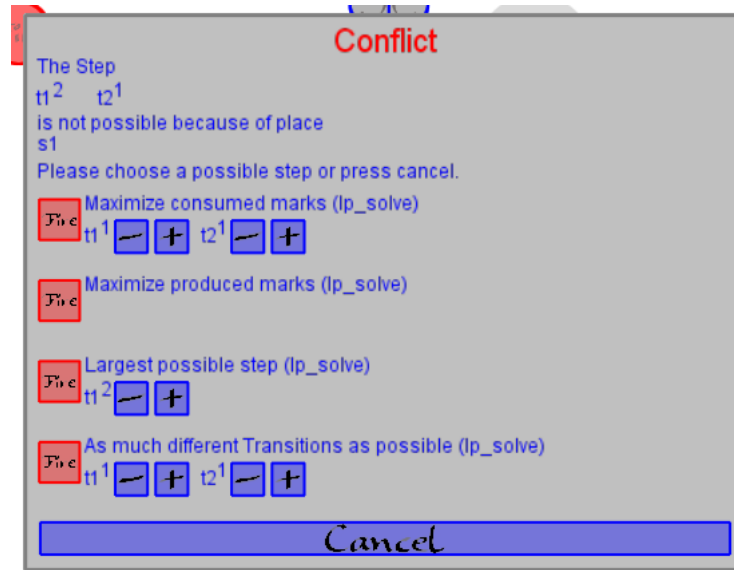


Fig. 6. The conflict solving dialog.

The solutions of the optimization problem are currently calculated with the help of the LPSolve library. LPSolve is a library which is released under LGPL (GNU Lesser General Public License) and solves different linear problems with a specialization to linear optimization problems. The calculated sub-steps are presented to the users through a dialog, see Figure 6, which allows the user to readjust the transition step. This is not yet a satisfying solution because it is a break in the multi-user operation since a dialog can only be handled by one user.

## 5 Implementation

MuPSi is currently implemented as part of a thesis of the first author of this paper. It will be available as a plug-in for VipTool and can be used as a stand-alone version at the moment. MuPSi is optimized for the use on a multitouch device and is implemented in Java. MuPSi is available for download on the MuPSi homepage ([www.fernuni-hagen.de/sttp/forschung/mupsi.shtml](http://www.fernuni-hagen.de/sttp/forschung/mupsi.shtml)). This section describes the implementation and functions of MuPSi in more detail.

There are currently three popular frameworks which provide multi-pointer input. The oldest of these frameworks is the multimouse driver which allows to use multiple mice to control multiple mousepointers. This is not a true multi-touch setting but it has the advantage that multi-touch input can be simulated without special hardware. In the consumer sector the Windows 7 multitouch framework has a very wide dissemination because it is part of the Windows 7 operating system. It supports multi-touch touchpads and inexpensive consumer

multi-touch screens. The disadvantage of these monitors is the limited support of only two input points. In the professional area the TUIO protocol is very common. TUIO is a network protocol which supports pointers and tagged objects. Tagged objects are physical objects with a unique ID. If an application wants to professionally use multi-touch tables it has to support this protocol. More information about the TUIO protocol and the usual hardware setup can be found at [www.tuio.org](http://www.tuio.org).

We decided to use the MT4J (multi-touch for Java) library, because it supports all of these input frameworks. The big advantage of this library is that it abstracts the input method and thus MuPSi can support all these input methods without software changes. The MT4J Library is released under GPL (GNU General Public License).

Our homemade multi-touch table, shown in Figure 1, uses a projector for rear projection onto a display output and an infrared camera to detect finger inputs. The table uses a basic FTIR surface as infrared light source. The usage of infrared light for finger detection is necessary to distinguish the input from the output image. For the calculation of the input data from the IR image, we use the Community Core Vision software, briefly CCV, from the NUI Group. CCV is an open source software which is released under LGPL and has several customizable filters in order to ensure a good detection. The detected inputs are sent to a given IP address using the TUIO protocol. The NUI Group is an open source community with the aim to provide natural user interfaces.

For the user interface we also used the MT4J Library. MT4J contains the Java OpenGL library, JOGL, and offers a set of multi-touch optimized GUI elements. The elements are grouped as a 3D scene graph. We use self defined user elements for MuPSi which are built from polygons. The usage of a 3D scene instead of a default Java GUI is useful because of the convenient implementation of scrolling, zooming and rotation which are common operations for multi-touch surfaces. MT4J allows to process the input with the aid of default and self defined gesture processors. MuPSi uses this for example to provide a two-finger-zoom gesture. The usage of hardware OpenGL support is preferred but it is also possible to use software rendering.

Figure 7 shows a screen shot of MuPSi. The manual fire button is labeled with 1. This button is only visible if the manual trigger is selected as fire mode. The undo and redo buttons are labeled with 2. MuPSi supports to undo all fired steps to make simulation more comfortable. All undone steps can be redone as long as no new step was fired. The tools button is labeled with 3. This button allows to move and zoom the loaded net. By clicking the button it is minimized to save screen space.

All of the coloring and step building methods presented in the previous sections are implemented in MuPSi and can be used by selecting them in the configuration file before starting the program. The coloring method can also be changed during the simulation by pressing the key *c*. The configuration file also allows the change of the resolution, the rendering method and all visual param-

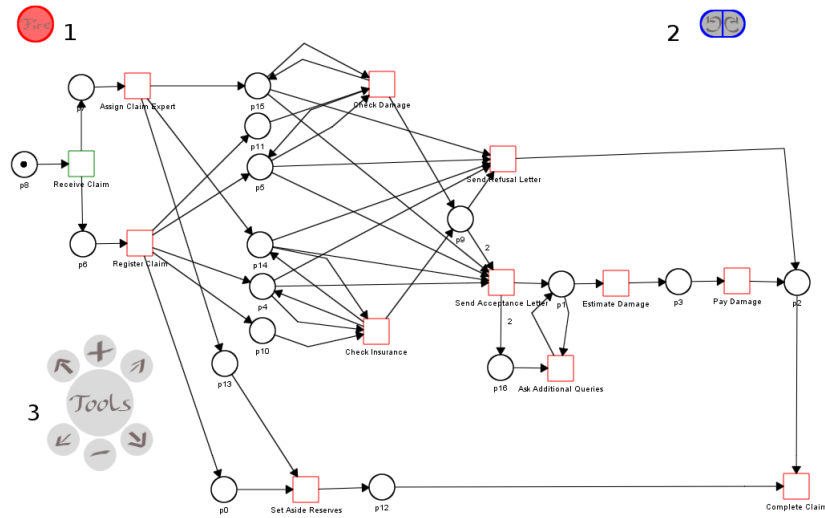


Fig. 7. A screenshot of MuPSi.

eters of the net elements. If visual parameters for the net elements are defined in the config file, the information in the PNML file is replaced with these values.

MuPSi is designed in a modular way to make it easy to implement new firing modes, coloring modes and conflict solvers but we have decided to offer no plugin infrastructure because MuPSi is already designed as a VipTool plugin.

Since MuPSi is not yet available as a VipTool plugin but as a standalone application, it is useful to give a brief description how to use it with various operating systems. MuPSi is, as a Java software, platform independent but some of the used libraries are platform dependent such as the LPSolve library and the OpenGL library contained in MT4J. The available MuPSi zip package contains different startup scripts for Windows x86 and x64 and for Linux x86 and x64. The startup scripts define different platform-dependent Java classpath variables. To support scenario dependent startup scripts a set of command line parameters is defined which sets the display resolution, the used PNML file and a few more settings. A full list of command line parameters is printed with the parameter `-h` or `-help`.

## 6 Conclusion and outlook

In this paper the tool MuPSi was presented. It is a Petri net simulator designed for multi-touch devices which supports simulation of transition steps. This enables a simple concurrent firing of transitions of a Petri net. We discussed several options to visualize enabled transition steps and to discover conflicts in a current marking of a Petri net. We discussed possible input modes for the creation of

transition steps. In MuPSi a single user or multiple users may select the most appropriate input mode for each scenario.

After a full implementation of MuPSi we will focus on the aspects of conflict resolution and presentation of these solutions. Another point we miss at the moment is a smooth VipTool integration.

## References

1. Desel, J., Juhás, G., Lorenz, R., Neumair, C.: Modelling and Validation with Vip-tool. In W.M.P. van der Aalst; A.H.M. ter Hofstede; M. Weske, ed.: Business Process Management. Volume 2678 of Lecture Notes in Computer Science., Springer (2003) 380–389
2. Bergenthum, R.: Algorithmen zur Verifikation von halbgeordneten Petrinetz-Abläufen: Implementierung und Anwendungen. Master's thesis, Katholische Universität Eichstätt-Ingolstadt (2006)
3. Jensen, K., Kristensen, L., Wells, L.: 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) (2007) 213–254
4. Freytag, T.: WoPeD – Workflow Petri Net Designer. In: ATPN. (2005)
5. Desel, J., W.Reisig: Place/Transition Petri Nets. In W. Reisig; G. Rozenberg, ed.: Petri Nets. Volume 1491 of Lecture Notes in Computer Science., Springer (1998) 123–174
6. Burmester, M., Koller, F., Höflacher, C.: Touch it, move it, scale it - multitouch. Technical report, HDM Stuttgart (2009)

# PetriPad – A Collaborative Petri Net Editor

Julian Burkhart, Michael Haustermann

University of Hamburg  
Faculty of Mathematics, Informatics and Natural Sciences  
Department of Informatics  
{4burkhar, 6hauster} (at) informatik.uni-hamburg.de

**Abstract.** Collaboration is one of the key aspects of software engineering and commonly includes working in spatially separated teams. Many tools exist to support such a workflow and are used extensively, especially for real-time communication, e.g. instant messaging systems and voice chats. In contrast, programming environments and editors used in general mostly lack synchronous real-time collaboration functionality.

In this work we present an informal specification of such a system in the context of a groupware Petri net editor and the implementation of our model as a proof of concept for the RENEW tool. To do this we revisit work on this subject done more than 10 years ago and update the proposed models to the current state of software engineering. As a result we are able to simplify the specification.

**Keywords:** collaborative editing, Petri nets, RENEW, MULAN/CAPA, multi-agent systems

## 1 Introduction

Distributed development of a common code base in a collaborative manner has become one of the key aspects of development in computer science. Many tools exist to support this kind of distributed workflow in different styles. The most common of them are source code management systems (SCM) that enable distributed, concurrent editing of shared documents.

However, SCMs are tailored for sequentially structured textual documents and perform poorly on graphically oriented data files. When using a graphical editor the user usually has only very limited control over the serialized file formats. Even when using a text-based format, slight changes in the editor can result in vast differences in the exported file. This makes it very hard for a SCM system to distinguish the changed parts from the (semantically) identical ones.

Also SCMs only cover scenarios where developers are working independently in the sense that while working at the same project in general the work of others does not immediately impact their own. This is fine for day-to-day development, but for real-time collaborative development it is not feasible that way. Possible applications for real-time collaboration are brainstorming ideas or teaching interactive courses over the web.

For synchronous editing of text documents there are web-based solutions, like Etherpad<sup>1</sup> and desktop applications, like ACE<sup>2</sup> or Gobby<sup>3</sup>. There are plugins for the popular development environment Eclipse<sup>4</sup>.

In this work we present a model for a collaborative Petri net editor. It is based on prior research described in Section 2, but varies in that it does not need any locking mechanisms of the graphical components manipulated by the users. The resulting model therefore is much simpler and is presented in Section 3. As a proof of concept we present the implementation of our model for the Petri net editor RENEW [17]. It is based on the multi-agent system (MAS) framework MULAN/CAPA [10]. The implementation is described in Section 4 and discussed in Section 5. At the end we summarize our work and give an overview of potential extensions of our model and of the possibilities for further research.

## 2 Related Work

The subject of collaborative Petri net editing is discussed in [1], which serves as a canonical case study for authors to present their approaches of combining Petri nets and object-oriented programming concepts. Some requirements for such an editor are outlined in [2]. The editor proposed is for hierarchically structured Petri nets and allows multiple users to work simultaneously on the same net from different terminals over the network. The users are organized in sessions and each user is able to work on multiple nets in different sessions at the same time.

Each user is capable of having a customized view on the net and a set of access privileges restricts the actions each user can make. For example a user might only be granted reading access to a net or parts of it. The concept of ownership of graphical elements is introduced to restrict the user's actions at a certain point in time. For each operation the appropriate ownership must be acquired beforehand. Some ownerships are exclusive (e.g. delete), others may be acquired from different users at the same time (e.g. modify). The request for some ownership may happen implicitly by selecting an item or explicitly by pressing a button. The case study describes different ownerships, how they could be requested and which restrictions apply if some user holds a certain ownership. We use the term *ownership* in the following section in this sense only unless otherwise specified.

The requirement analysis in the case study is intentionally incomplete. Other authors are encouraged to extend the requirements appropriately to their own concept or to focus on specific points to emphasize certain properties of a chosen formalism.

In the following subsections we describe three different approaches from [1].

---

<sup>1</sup> <http://etherpad.com>

<sup>2</sup> <http://sourceforge.net/projects/ace>

<sup>3</sup> <http://gobby.0x539.de>

<sup>4</sup> <http://www.saros-project.org>

### 2.1 Biberstein, Buchs, Guelfi

The authors of [6] describe a centralized approach. They build upon an architecture specified in [5] that is modeled in a formalism they introduced in [7] called *Concurrent Object-Oriented Petri Nets (CO-OOPN/2)*.

Their architecture consists of two layers: a graphical interface layer (viewports) and a centralized synchronization layer (server). Documents are stored on the server and cannot be manipulated directly by users but only by the server on request. The viewport's task is to display the current net and send user input to the server. Furthermore updates made by other users received from the server have to be displayed. The server takes requests from users, updates the net and informs all users about that update. Moreover it ensures the consistency of the net and guarantees the compliance with the user rights and ownerships as described in [2].

The procedure is as follows: a user changes an element in a net in his viewport. This change is transported from the viewport to the server. The server examines if the changes are compatible with the users rights and ownerships and either includes the change into the net or rejects it. A message is sent to all viewports, if the net changed. The net itself is represented as a tree. Components that may have subcomponents are called hierarchical components and stored as nodes of the tree. Components that may not have subcomponents, called atomic components, are stored as leaves of the tree. A net in this model is itself a hierarchical component.

### 2.2 Bastide, Palanque

The authors of [3] focus on locking mechanisms for objects to ensure consistency. The majority of the requirements in [2] are not taken into account. The implementation of the locking mechanism is described in great detail and down to a very technical level. The general concept is the locking of graphical elements that are selected in an editor for all other users. [3] describes how the synchronization between graphical elements and graphical editor works. They use the *cooperative objects* formalism to present their approach.

We do not go into further details, since we do not use any locking in our model.

### 2.3 Guerrero, Figueiredo, Perkusich

Guerrero, Figueiredo, Perkusich describe a decentralized multi-agent architecture [14] for the collaborative editor. They distinguish between two kinds of agents: user agents and manager agents. User agents may work on nets according to their access privileges and join or leave editing sessions. Manager agents are capable of performing administrative tasks for user and session management. They can also grant and revoke ownerships of graphical elements.

Each agent consists of three layers. The lowest layer is the communication layer. It provides message transportation services. The middle layer differs between user agents and manager agents. It is called control layer and management



layer respectively. At the top lies the application layer and represents the interface presented to the users of the system.

For a user agent the application layer is a graphical editor that allows the user to view and manipulate Petri nets. User inputs are first passed on to the control layer. The control layer verifies that the net manipulations are compliant with the ownerships the user holds. It may try to request additional ownerships, if needed for the desired action. If the control layer fails to acquire all necessary ownerships, the change is rejected.

The communication layer is used to exchange messages with other agents. Especially to request ownership and to inform other agents about acceptable changes. Incoming changes are passed up through the layers and displayed in the user interface (UI).

The manager agent's application layer is a system console, which enables the execution of administrative commands. These are performed by the manager agent's management layer.

### 3 Informal Model Specification

In this section we present our own approach. Since it differs from the aforementioned work, we first discuss the main principles behind it. Then we describe the resulting architecture in detail. Therefore we describe the requirements on the user interface, discuss different communication languages and explain how consistency can be guaranteed in our model.

#### 3.1 General Architecture

One of the goals of this work is to update the previous models to the current practice of UI design, especially the design of the collaborative text-editor Etherpad. Comparing different websites offering etherpad-based services<sup>5</sup>, the editor offers basic formatting tools only. There are no restrictions in terms of what parts of a document a user may edit. Session management is confined to merely distinguishing sessions, while access privileges are left out completely. The name that uniquely identifies an Etherpad session is incorporated into the URL leading to the edited document and subsequently it can be accessed by that URL without restrictions.

Despite of the absence of administrative capabilities, the result is well fitting to the task of collaborative writing. The UI is highly intuitive and access privileges are non-essential to enable collaborative work (though possibly helpful in some use cases). All conflicts arising from editing the same passage can be discussed in the integrated chat window.

From this brief analysis of Etherpad we draw three main principles for our proposed model.

<sup>5</sup> An overview of some of the available websites can be found on <http://etherpad.org/public-sites/>

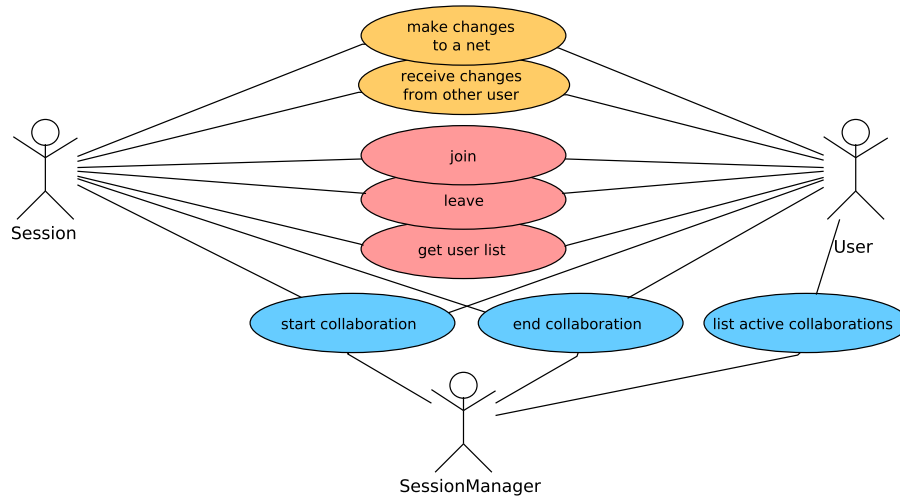


Fig. 1: The typical use cases of a collaborative editing system, i.e. manipulating shared Petri nets (the upper two use cases) and session handling (the rest).

1. All users have the same access privileges, i.e. no access privileges are present at all.<sup>6</sup>
2. The user is enabled to make any change he wants, i.e. no change is rejected afterwards or prohibited in the first place. Furthermore every change is implemented immediately in the users view.
3. Any arising conflicts are dealt with automatically and in a sensible way.

From the second point it is obvious that we had to omit locking mechanisms as described in [3] and [14] (cf. Section 2). This is also the primary source for simplification for our model.

We adopt a number of requirements from the original case study. The session management enables users to participate in multiple collaboration sessions and collaborate on multiple nets at the same time. A user's view on the shared nets is independent of the other users.

Additional requirement on the system is the separation of communication infrastructure and editor, so that the graphical user interface (GUI) could be exchanged. For example it should be possible for one of two collaborating users to work in a full-blown desktop client, while the other works in a web-based editor in an internet browser.

<sup>6</sup> It should be mentioned that we are not opposed to access privileges in general. A number of use cases can be thought of that require such a mechanism, e.g. tutoring or presentation purposes, but these are not the use cases we considered for this work.

The system is modeled as a multi-agent application similar to [14], but with some important differences. We identify three kinds of agents in the system (cf. Figure 1).

**User agents** represent the users of the system. Since we want the UI to be interchangeable, a user agent needs to be modular by design. It consists of a communication module with a well-defined interface to which an arbitrary editor can be connected.

**Session management agents** offer an entry point to user agents. They are autonomous and can start or terminate sessions on request of user agents or be queried for a list of existing collaboration sessions.

**Session agents** represent individual collaboration sessions and keep track of the edited nets and the participating users. It also functions as a central message relay between the participating user agents and resolve conflicts arising from concurrency. It also stores the current state of each net.

### 3.2 Requirements on User Interface

The UI connected to a user agent’s communication module has two essential tasks. First of all, it has to observe the actions the user takes. This follows from the manner changes are not requested at a central authority as in [6], but simply made and then passed on to others.

The UI should also recognize what actions result in meaningful changes. In this case meaningful refers to the completion of an action. While the user drags an element from one point to another, it might not be wise to flood the network with updates for every single pixel that the element is moved. Especially, because in the context of nets, dragging one element usually impacts the position of other elements that are connected to it as well.

To further reduce the number of messages, actions may be grouped together to batches and sent in one message to the session agent. The most straight forward way for any receiving agent to deal with batches is to first execute all operations that add new elements, then perform all operations that change existing elements and lastly all remove operations.

The second job of the UI is to integrate incoming changes from the session agent. A crucial requirement is for the integration to be done atomically and between user manipulations, so that it does not interfere with changes the user makes.

### 3.3 Communication Language

We consider compliance with the standards of the Foundation for Intelligent Physical Agents<sup>7</sup> (FIPA) [20] a baseline for our model. These include defining the message format (Agent Communication Language – ACL [11]) for all inter-agent communication. Choice exists however on the part of the content languages

<sup>7</sup> <http://www.fipa.org/>

for the actual message payload. Two possibilities will be discussed, namely the Petri Net Markup Language [24] (PNML) and using ontologies.

PNML is a ISO/IEC standard for higher order Petri nets that is still in development. Many Petri net tools have adopted it [8] and type definitions for various different flavors of Petri nets have been created. The latter is what makes PNML especially attractive for our work. The set goal of building a universal platform for collaborative editing agnostic to the actual editor in use would greatly benefit from a widely adopted standard. RENEW as our main aim for application of our work already has support for importing and exporting PNML.

The main downside of applying PNML to this work is that it only represents the net itself and not manipulation operations on it. Describing the actions performed directly, e.g. a `moveElement` or `addMarking` operation, is out of the question. PNML can only be used to describe the set of elements that changed and their relevant properties. That is however a viable solution and receiving agents can simply overwrite the received elements in their copies of the net.

Another possibility is to use an ontology to model Petri nets and the operations. Ontologies becoming increasingly popular in software engineering. They can be used as a glossary throughout all development phases and as a meta-language for specification. The de facto standard ontology modeling tool Protégé<sup>8</sup> has a built-in code generator, which can generate Java classes directly from an ontology. Protege is based on the Web Ontology Language (OWL) developed for the Semantic Web<sup>9</sup>. It has a variety of different syntaxes to choose from [19,15]. They can be machine-readable like the RDF-based XML syntax or better suited to be edited by humans like the Manchester Syntax, thus lowering the bar for adoption of OWL 2 considerably over its predecessor OWL 1.

In multi-agent systems ontologies form the basis for communication [13]. Without a common ontology to give meaning to objects and statements, there can be no exchange of knowledge, both between agent or humans. And specifically in the context of our implementation detailed in Section 4, the MULAN/CAPA framework relies heavily on ontologies. Not only for communication, but for modeling purposes as well.

The two possibilities, using PNML or ontologies, are not necessarily mutually exclusive either. Attempts have been made to define ontologies for higher order Petri nets, that are compliant with PNML [12,23]. Since we are aiming at interoperability to some extent in our model, we suggest [23] to be used to embed PNML into use in our multi-agent system.

### 3.4 Consistency Guarantee and Conflict Treatment

Guaranteeing consistency needs special care in our approach since users have their own synchronized copies of the shared nets. To ensure that identifiers for net elements are globally unique, we let the central session agent determine them. So when a user adds an element locally, a temporary identifier is assigned

<sup>8</sup> <http://protege.stanford.edu/>

<sup>9</sup> <http://www.w3.org/standards/semanticweb/>

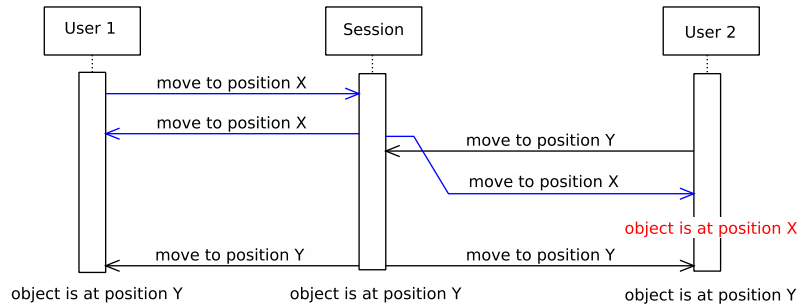


Fig. 2: Possible inconsistencies from varying message transit times.

to that element. The final identifier is received as a result message, when he informs the session agent about the new element. All changes to objects with temporary identifiers are buffered in the user clients until the final identifier has been determined.

To address the problem of messages overtaking one another, we add sequence numbers to each message. A global order of all messages is determined by the session agent, who orders them per user and integrates them into a global order by time of arrival. The session agent accordingly sets new sequence numbers to all messages before distributing them.

Our approach enables users to modify the same object concurrently. Thus we allow conflicting modifications that have to be dealt with. For conflicting changes to the same property of an element, the change processed last by the session agent wins. To enable a user to determine which change won in such a case, the session agent distributes change operations (but not additions or deletions) to all users including the sender (cf. Figure 2).

A minor inconvenience of the scenario in Figure 2 is that User 2 would implement the change of User 1 for a short period of time after he made his own change which would be undone shortly after that, when his own update is sent back to him. A possible remedy for this is that if a user made a change to property  $p$  of element  $a$ , all incoming updates to  $p$  of  $a$  can be ignored until his own change appears in the stream of updates.

Lastly we have to deal with incoming changes to elements that were already deleted. This situation occurs at the session agent, when a user makes a change to an element before receiving the delete message. It can also occur at the user agent when an element was deleted locally, but the delete message was not yet distributed. In either case the change can be dropped safely. The user from whom the change originated will eventually receive the delete message and subsequently delete the element himself. That way a consistent state is reached at quiescence, i.e. when all messages have been distributed and processed at each site.

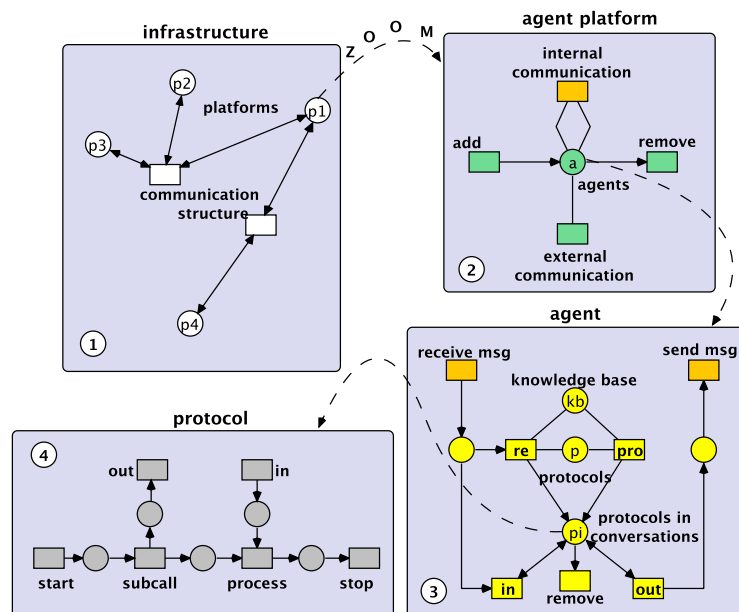


Fig. 3: The four levels of the MULAN model.

## 4 Implementation for Renew

In this section we give an overview of our proof-of-concept implementation. We will shortly introduce the RENEW Petri net editor and the MULAN/CAPA agent framework and then describe the implementation and the Ontology we use for communication.

### 4.1 Context of Implementation

**RENEW.** The Reference Net Workshop (RENEW) [17] is an editor for Petri net formalisms developed by the theoretical foundations of computer science group at the department of computer science at the University of Hamburg. It is highly modular and includes plugins for different Petri net formalisms. The integrated simulator can execute basic P/T-nets as well as higher order nets, e.g. colored Petri nets and reference nets, which are object-oriented Petri nets with synchronous channels that allow for tokens to be nets themselves [16,22]. Transitions of reference nets can also be inscribed with Java code, that is executed during simulation. This mechanism provides a seamless integration of object-oriented programming with specification and simulation of Petri nets.

**MULAN/CAPA.** In order to facilitate the implementation of our model, we built upon a framework for multi-agent applications called MULAN/CAPA.

MULAN (**M**ulti-**a**gent **n**ets) [21] is a reference architecture for multi-agent systems. It is modeled completely in reference nets and consists of four different levels as seen in Figure 3.

The highest level (level 1) is the infrastructure. The infrastructure connects multiple agent platforms to a network. A platform (level 2) provides the environment for agents. Apart from starting and terminating agents, it provides means for communication between agents. If sending and receiving agent are identical the communication is considered to be an internal communication, e.g. between different active protocols of the same agent. The next level are the actual agents (level 3). The agents can send and receive messages to and from other agents and the platform. This is the only externally observable behavior of an agent. Each communication between agents and agents or agents and a platform is described by protocols (level 4). It models the behavior of an agent or how agents communicate with each other. A protocol is developed in complementary parts for each participating agent and orders the flow of information and the messages sent.

CAPA (**C**oncurrent **A**gent **P**latform **A**rchitecture) [10] is a FIPA-compliant implementation of the MULAN model on top of RENEW and Java. CAPA facilitates building multi-agent applications based on the MULAN model. Heterogeneous systems with platforms implemented in other frameworks are possible due to the FIPA-compliance. The infrastructure level of the MULAN model is not implemented in CAPA. It emerges when combining an arbitrary number of instances of CAPA platforms that can be interconnected over a network.

The combination of model and implementation, plus the development environment, monitoring and debugging tools comprise the MULAN/CAPA framework. It provides a high level of concurrency since it is developed with reference nets that are by design concurrent.

#### 4.2 Coarse Design of the PetriPad Plugin

PetriPad consists of two parts. The multi-agent model is implemented in the MULAN/CAPA framework and we extend RENEW with a plugin, which connects it to the MAS. In terms of our informal model communication module is the agent connected to the editor.

To exchange data between the editor and the agent we use the WebGateway plugin for MULAN/CAPA. It implements a gateway architecture [4], which facilitates connecting HTML5-based web services to the MAS. The WebGateway acts as a bridge using the WebSocket protocol and although web applications are its original aim, it can hook up arbitrary software systems.

Our utilization of WebGateway can be seen in Figure 4. The RENEW Petri net editor serves as UI for an agent running in a remote MULAN/CAPA platform. We call that agent the *modeler agent*. The RENEW plugin tracks the changes made by the user and passes them through a WebSocket channel to the WebGateway, which relays them to the modeler agent in the PetriPad MAS.

The primary motivation for this architecture is that users do not need to be running a instance of the MULAN/CAPA platform locally. Instead only one

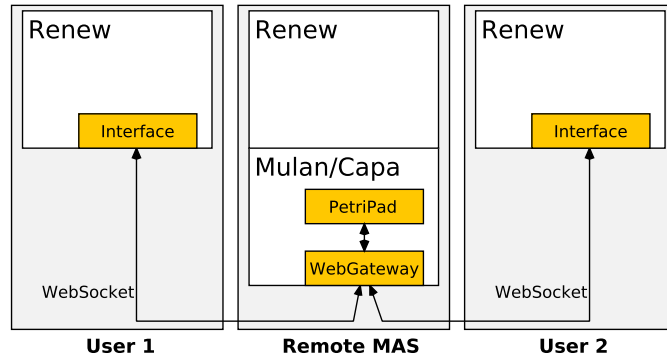


Fig. 4: Our proposed architecture of the communication infrastructure.

platform is needed to which an arbitrary number of editors can be connected and it is possible to use editors other than RENEW.

Our use case diagram (Figure 1) showed three agent types and a number of interactions between them. These translate directly to levels 3 and 4 of the MULAN model. Each interaction is implemented as a protocol for the participating agents.

### 4.3 Ontology

Although we argued in favor of using PNML compliant ontologies, we had to deviate from it to some extent. This is mostly due to the limited expressive power of the MULAN/CAPA default modeling formalism for ontologies. It is called *concept diagrams* [9] and can define a taxonomy of concepts in a UML-based graphical notation. Each concept can be described by a set of key-value-tuples and the values' respective domains. A domain may be a Java data type, another concept in the ontology or a list of either of them. Semantically *concept diagrams* coincide with the idea of frames [18] limited to defining concepts only and subsumption as the only relation.

Figure 5 shows a subset of our ontology dealing with reference nets and possible operations on them. The semantics are as follows. The nodes in the graph represent the defined concepts and the arcs define the subsumption hierarchy. Every concept has a name (bold) and attributes of the form  $k: t$ , where  $k$  is the name of the attribute and  $t$  its domain. A star ('\*') at the end of the domain definition denotes that an instance of the concept can have multiple values for that attribute, whereas the other attributes must be single-valued.

Our ontology distinguishes between the structural elements (**transition**, **place**, **arc**, **virtual-place**<sup>10</sup>) and the textual elements. Arcs in our model have type (e.g.

<sup>10</sup> A virtual place is a reference to a place. A place and its virtual copies are semantically identical.



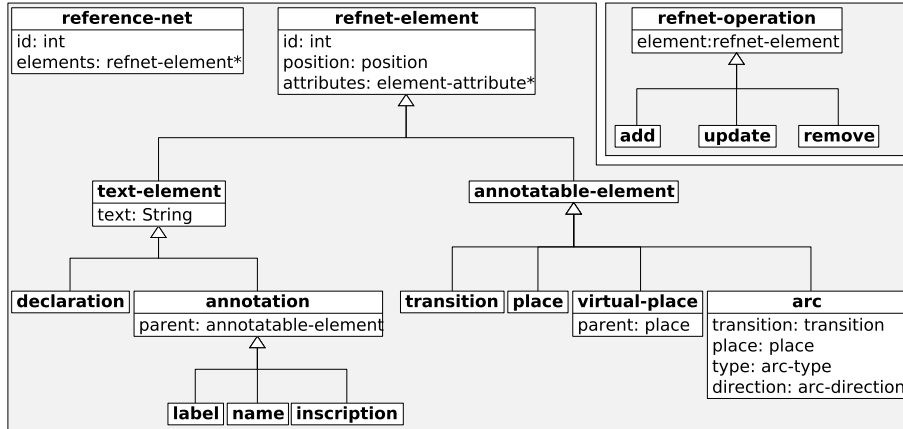


Fig. 5: A part of the ontology defining reference nets. As much as it is practical it is modeled with their formal properties in mind rather than their representation in RENEW.

normal, inhibitory) and direction attributes that define the way they interact with the associated place and transition.

The leaves of the text-element subtree model the four types of textual elements that are used in RENEW. A structural element can be named by a name annotation and inscriptions cover all semantically relevant annotations, e.g. markings, guards, weights, etc. All other annotations are labels and are ignored during simulation of the net. Declarations are free-standing textual elements that contain all declarations of Java objects used in the net and all imports for Java packages that are needed.

The top concept for elements (refnet-element) has an attribute attributes that may contain the graphical information of the element. The concept hierarchy for its type element-attributes is not depicted here, but contains e.g. color and size.

An operation on an element is modeled as refnet-operation. We distinguish three different kinds of operations: add, update and remove. In case of an update the element attribute contains the entire new state of the element that was updated.

Note that Figure 5 describes only part of our reference net ontology.

## 5 Discussion

Our decision to abandon locking mechanisms resulted in a huge simplification of the overall design of the collaborative Petri net editor. On the other hand abandonment of locking mechanisms entails certain problems with concurrent editing of the same elements. It can be confusing for users for example to have

their changes overwritten immediately by other users. One such case has been discussed in Section 3.4.

In this case the developers have to communicate to clear up the confusion and possibly revert some changes manually. We argue that in the targeted use case of constructive collaborative work this is a corner case that seldom arises and thus can be neglected.

The choice of using WebSocket as communication channel and a well-defined ontology for message contents leads to loose coupling of the infrastructure and the editor. This has two main benefits. Firstly different editors can be connected to the MAS and secondly the MAS can be offered as a service, which compatible editors can connect to from anywhere over the net.

Not mentioned before, the WebGateway can convert ontology communication into JSON or XML. These languages are common in web applications and thus facilitate connecting web-based editors in particular.

Modeling the system as a MAS resulted in a very modular architecture and naturally, it inherits certain properties from the MULAN/CAPA framework. On the one hand, implementing large parts of the system in Petri nets is a huge gain in terms of concurrency. On the other hand the message transport through the various layers of the MAS is far slower than direct peer-to-peer communication.

The ontology we designed for inter-agent communication is not yet compliant with the PNML standard, but in general the differences are minor. Using an ontology has the additional benefit of serving as a glossary in the development phase. It also provides a more abstract means of modeling in conjunction with the code generator of the MULAN/CAPA framework.

## 6 Summary

In the preceding chapters we gave a short overview of the prior work on collaborative Petri net editors. We argued that the specifications are incomplete in the sense that they only consider parts of the overall system.

The new model proposed in this work differs from all prior approaches in that it completely foregoes locking of elements in the edited nets. The goal is not to restrict the user in what he can manipulate and give immediate feedback to all inputs. We achieve this by using a multi-agent approach with a central message relay. By serializing all communication at the central relay we were able to implement user input immediately in the user's view and only subsequently send it to the relay.

Session management is incorporated in our model as well. A user can participate in multiple sessions simultaneously and each session can hold an arbitrary number of documents available to all users in that session.

As a proof of concept we implemented our model as a plugin for the Petri net editor RENEW and a multi-agent application based on the MULAN/CAPA framework.

## 7 Outlook

Since our approach uses WebSocket for communication it allows for using a web application as Petri net editor. With the canvas element and the WebSocket channel HTML5 offers all the required components to build such an editor.

Another topic of great significance for developing Petri nets is the synchronized simulation. The simulation of the models may reveal modeling errors. In a collaborative development process, on-line debugging a net has the benefit that every participant can observe the exact firing sequence in the net. Simulating the nets individually produces various different firing sequences for each participant due to the innate concurrency of Petri nets and is therefore of little use to collaborative debugging.

Looking a little farther behind the horizon, the MAS specified in our model is by itself agnostic to the subject of the collaborative work. We defined very basic manipulation operations that can be applied to a number of collaborative editing scenarios. The MAS could thus be developed into a service platform providing an infrastructure for collaborative editing. In order to build such an infrastructure a meta ontology for collaborative editing needs to be developed. It will facilitate building subject ontologies that describe particular subjects of collaborative work and the permissible manipulations and their effects. Compliant subject ontologies can then be used in conjunction with the collaboration infrastructure.

## References

1. Gul Agha, Fiorella De Cindio, and Grzegorz Rozenberg, editors. *Advances in Petri Nets: Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
2. Rémi Bastide, Charles Lakos, and Philippe A. Palanque. A Cooperative Petri Net Editor. In Agha et al. [1], pages 534–535.
3. Rémi Bastide and Philippe A. Palanque. Modeling a groupware editing tool with cooperative objects. In Agha et al. [1], pages 305–318.
4. Tobias Betz, Lawrence Cabac, and Matthias Wester-Ebbinghaus. Gateway architecture for Web-based agent services. In Franziska Kügl and Sascha Ossowski, editors, *Multiagent System Technologies*, volume 6973 of *Lecture Notes in Computer Science*, pages 165–172. Springer Berlin / Heidelberg, 2011.
5. O. Biberstein, D. Buchs, and N. Guel. CO-OPN/2 applied to the modeling of cooperative structured editors. In *Tech. Report 96/184, Swiss Federal Institute of Technology (EPFL), Software Engineering Laboratory*. Citeseer, 1996.
6. O. Biberstein, Didier Buchs, and Nicolas Guelfi. Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism. In Agha et al. [1], pages 73–130.
7. Olivier Biberstein and Didier Buchs. Structured Algebraic Nets with Object-Orientation. In G. Agha and F. de Cindio, editors, *Workshop on Object-Oriented Programming and Models of Concurrency'95*, pages 131–145, 1995. Turin.
8. Jonathan Billington, Søren Christensen, Kees van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The petri net markup language: Concepts, technology, and tools. In *Applications and Theory*

- of *Petri Nets 2003: 24th International Conference*, pages 1023–1024, Eindhoven, The Netherlands, June 2003.
9. Lawrence Cabac. *Modeling Petri Net-Based Multi-Agent Applications*. Dissertation, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, April 2010. <http://www.sub.uni-hamburg.de/opus/volltexte/2010/4666/>.
  10. Michael Duvigneau. *Bereitstellung einer Agentenplattform für petrinetzbasierte Agenten*. Diploma thesis, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg, December 2002.
  11. Foundation for Intelligent Physical Agents. FIPA ACL message structure specification. <http://fipa.org/specs/fipa00061/>.
  12. Dragan Gašević and Vladan Devedžić. Petri net ontology. *Knowledge-Based Systems*, 19(4):220 – 234, 2006.
  13. Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199 – 220, 1993.
  14. Dalton Serey Guerrero, Jorge C. A. de Figueiredo, and Angelo Perkusich. An object-based modular cpn approach: Its application to the specification of a cooperative editing environment. In Agha et al. [1], pages 338–354.
  15. Matthew Horridge and Peter F. Patel-Schneider. OWL 2 web ontology language manchester syntax. <http://www.w3.org/TR/owl2-manchester-syntax/>.
  16. Olaf Kummer. Introduction to Petri nets and reference nets. *Sozionik Aktuell*, 1:1–9, 2001. ISSN 1617-2477.
  17. 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.
  18. Marvin Minsky. A framework for representing knowledge. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.
  19. Peter F. Patel-Schneider and Boris Motik. OWL 2 web ontology language mapping to RDF graphs. <http://www.w3.org/TR/owl2-mapping-to-rdf/>.
  20. S. Poslad and P. Charlton. Standardizing agent interoperability: The FIPA approach. *Multi-Agent Systems and Applications*, pages 98–117, 2006.
  21. Heiko Rölke. *Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen*, volume 2 of *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2004.
  22. Rüdiger Valk. Petri nets as token objects - an introduction to elementary object nets. In Jörg Desel and Manuel Silva, editors, *19th International Conference on Application and Theory of Petri nets, Lisbon, Portugal*, number 1420 in Lecture Notes in Computer Science, pages 1–25, Berlin Heidelberg New York, 1998. Springer-Verlag.
  23. J.C. Vidal, M. Lama, and A. Bugarín. A high-level petri net ontology compatible with PNML. *Petri Net Newsletter*, 17:11 – 23, 2006.
  24. Michael Weber and Ekkart Kindler. The petri net markup language. In Hartmut Ehrig, Wolfgang Reisig, Grzegorz Rozenberg, and Herbert Weber, editors, *Petri Net Technology for Communication-Based Systems*, volume 2472 of *Lecture Notes in Computer Science*, pages 124–144. Springer Berlin / Heidelberg, 2003. 10.1007/978-3-540-40022-6\_7.

Poster Abstracts



# Agentworkflows for Flexible Workflow Execution

Thomas Wagner

University of Hamburg, Faculty of Mathematics, Informatics and Natural Sciences,  
Department of Informatics, Theoretical Foundations of Computer Science Group  
<http://www.informatik.uni-hamburg.de/TGI/>

**Abstract.** Dynamic aspects of workflow execution require flexible solutions. Especially in an interorganisational context many variable factors can only be determined during the actual execution of a workflow. These factors may require contextual, local changes within a process in order to adequately support and represent the real-world scenario. This paper describes the *agentworkflow* approach, which uses a combination of the agent and workflow concepts to address a number of challenges of workflow execution. Both agents and workflows are provided as high-level Petri nets.

The focus of this paper is the flexibility aspect of this approach. Agentworkflows allow the dynamic reconfiguration of the workflow specification. The key to this is the exchange of subworkflows depending on the changing circumstances of the workflow execution. This allows the workflow system to adapt to these circumstances and support users adequately.

**Keywords:** Workflows, Agents, Combination, Flexibility

## 1 Introduction

Business processes (BP) are becoming ever more complex, especially in large organisations. Their correct execution is crucial to the successful operation of a company. Any incident occurring due to errors in a BP needs to be avoided by all means. This is why BP are commonly facilitated with the help of workflow software systems. Workflow management systems (WFMS) map real-life BP to computerised representations and manage and control their execution, while automating the processes as far as possible and supporting human users during task execution.

Classically, WFMS are aimed at supporting static processes, that rarely change. Dynamic changes, which might become necessary on a case-by-case basis, are difficult to handle and often require different solutions for every situation. This is, however, a static approach to a dynamic problem and rather inefficient, especially if the required changes are only minor and affect very small parts of the workflow. Unforeseen changes that develop out of unique circumstances cannot easily be handled by a static WFMS. These changes cannot be handled on-the-fly and require a workflow modeller to implement a new version of the

overall workflow, especially dealing with the specific problem. The new workflow then has to be re-initiated and an equivalent to the previous workflows' state has to be established. This requires time and effort, which, in real-life situations, may be in short supply.

Because of this, another fundamental approach to workflow management can be helpful or even crucial. Instead of only supporting static workflows, a degree of flexibility should be added to the workflow management. It should be possible to exchange certain parts of a workflow during execution, depending on the current state of the system and workflow. If unforeseen changes occur, eligible workflow administrators and modellers need to be able to simply (re-)design the relevant parts and infuse them into the system in order to make further execution of the workflow possible. Standardised exception handling or skipping patterns can be used to automatically handle occurring issues and simplify the work of administrators.

Another important aspect relates to interorganisational workflows, i.e. workflows executed cooperatively between different organisational entities. In this context flexibility becomes especially important if partners in a workflow often change, which would require differing workflows each time in classical approaches. Interorganisational aspects will feature heavily in the discussion later on in this paper.

We propose an approach to flexibility in workflow execution that strongly relies on and profits from the agent-oriented paradigm. By using software agents we can exploit the naturally distributed nature of these software entities to profit in various ways. The approach, called *agentworkflows*, uses one agent for each workflow instance. This agent is responsible for the execution, handling and distribution of this workflow. The workflows themselves feature a hierarchical structure, utilising subworkflows nested in an overall workflow. These subworkflows are essential to the support of flexibility and will receive special attention in this paper.

This paper is structured as follows. In Section 2 we will discuss related work. Section 3 highlights the modelling background for our research. Section 4 then presents the agentworkflow concept and implementation, while Section 5 discusses the flexibility aspects of agentworkflows. Both agentworkflow concept and how it can be used for flexibility are illustrated in a simple example in Section 6, followed by the conclusion of the paper in Section 7.

## 2 Related Work

In this section we will discuss other approaches to flexibility in workflow management. The ADEPT (Application Development based on Encapsulated pre-modeled Process Templates) project described, for example, in [4] deals with flexible and robust workflow management. Using an advanced meta-model, which defines all possible, allowed process structures, the system developed in this project allows users to change single process instances or whole process templates. Most aspects of a process or template can be changed during run-time, as long as the



changes are allowed by the meta-model. Updating a process template updates all of its currently running instances, as long as the changes do not produce inconsistencies. The key to the flexibility in ADEPT is the so-called delta-layer. This layer is situated between process template data and process instance data and enables changes made to single instances. If the template is changed, the information in the delta-layer and process instance data are checked against the updated template and the decision whether to migrate or not is made. Compared to our approach, the agentworkflows, ADEPT offers a much higher degree of flexibility, but is very focussed on the processes. Agentworkflows add qualities to the workflow execution, which are usually only associated with agents. These go beyond the flexibility, which is the focus of *this* paper.

Another approach is presented in [6]. Similar to our approach the JBees WFMS is implemented using agents and Petri nets. Based on the agent-platform Opal and the Petri net tool JFern, the JBees system consists of several types of agents. Interactions between these agents enable the execution of workflows. Of special interest is the process agent, which similarly to our approach, encapsulates a process instance and is responsible for its execution. Workflow flexibility is also available in JBees using different algorithms to determine safe transfers between process instances. While JBees uses agents and Petri nets to implement workflow management it does not seem to offer the integration between agents and workflows we strive to achieve with our overall approach. The agentworkflows partially fulfil our goals and can be compared in scope to a WFMS like JBees.

[1] proposes a smart WFMS. During execution the WFMS possesses context awareness for a process, so that it can adapt to the current circumstances. The conceptual framework adds two key components to an otherwise regular WFMS: smart workflow descriptions and a context reasoner. The smart workflow descriptions contain generic activities that are only linked to particular tasks during runtime, similar to our approach. The context reasoner is responsible for evaluating current circumstances and linking the generic activities to tasks.

[3, 2] propose recursive ECATNets (Extended Concurrent Algebraic Term Nets) to model hierarchical and flexible workflows. Similar to our approach they differentiate between elementary tasks, which are directly executed by resources, and abstract tasks, which correspond to subworkflows.

Both the smart WFMS and the recursive ECATNets share similarities to how flexibility is handled (e.g. late coupling of tasks, representing subworkflows as tasks in overall workflows). However both approaches do not utilise agents. As mentioned before agentworkflows can benefit in more ways than flexibility from the employed agents.

All approaches, which have been discussed here, deal with flexibility in workflow execution in different and effective ways. Our approach exhibits similarities to certain aspects of these approaches, like the use of Petri nets and agent-orientation. In its current stage our approach may not offer the distinguished and elaborated possibilities offered by the other approaches. However, it is just one of the stepping stones toward a full integration of agents and workflows as

proposed in [11]. As such, the possibilities, w.r.t flexibility and other properties, in later stages will be significantly improved yet again, though this is outside the scope of this paper (see [17] for more information).

### 3 Modelling Background

The modelling background for our approach contains two major areas: agents and workflows. Agents are provided through the MULAN and CAPA agent architectures ([12, 5]). MULAN is a conceptual agent framework/architecture based entirely on reference nets, a high level Petri net formalism introduced in [8]. Every aspect of a multi-agent system in MULAN, from agent protocols to the overall systems, is modelled in reference nets. As the reference net formalism follows the nets-within-nets principle [13], each layer is nested within its upper layer creating a four-level hierarchy. CAPA is an extension to MULAN introducing full FIPA compliance to MULAN and replacing the upper levels of the MULAN hierarchy. This provides the functionality to allow distributed execution.

Workflows on the other hand are provided through workflow (Petri) nets. In our implementation workflow nets are specialised reference nets using a special transition to model the tasks of the process. These workflow nets were introduced in [7]. They follow the basic principles of (coloured) workflow Petri nets described in [14].

Since both agents and workflows have a common technological base, the reference net formalism, an integration of both concepts is not only possible in our approach, it is also quite natural. Both concepts can profit from one another, though the focus of this particular paper is on workflows benefiting from agent technology. The overall aspects of the work on integrating agents and workflows has been the subject of, for example, [11, 16, 10, 17].

MULAN/CAPA and workflow nets have previously been used to implement WFMS functionality. In [15] an agent-based WFMS (AgWFMS) was presented, which provides full workflow functionality using the above mentioned technologies. The functionality required is naturally divided between several types of agents. Users can log into the system remotely, but workflows are executed centralised. The AgWFMS managed to capture and support many interesting aspects, like interoperability due to the FIPA compliance of CAPA. Some aspects, however, such as distribution and flexibility of workflows, could not be adequately supported in the classic AgWFMS. This was one of the motivations to extend the system with our new approach.

The development and runtime environment for our systems is the RENEW (**R**eference **N**et **W**orkshop) editor. The editor was developed alongside the reference net formalism and is described, for example, in [9]. It supports the execution of all aspects described in this paper.

## 4 Agentworkflows

Our approach to flexible workflow management is called *agentworkflows*. This name reflects the combination of the agent and workflow concepts in this approach. The approach was originally described in [16] and general properties were discussed in [10, 17]. It is part of a larger, ongoing effort, also described in the previously mentioned works, but originating in [11]. This effort aims to combine agents and workflows into a new concept that exhibits the advantages and characteristics of both classical concepts. The overall goal in this is to address some of the shortcomings each classical concept (can) exhibit. The agentworkflow approach represents one of the later steps in the overall effort, that integrates both agents and workflows conceptually in the background, but still exhibits classical workflow behaviour to its environment.

The approach is based on the classic, regular AgWFMS, mentioned above. The agentworkflow approach replaces part of the workflow handling process in the AgWFMS and can thus be seen as a natural extension. The extended AgWFMS containing the enhanced functionality of and for agentworkflows is called AgWFMS\*.

The basic principle behind our agentworkflow approach is that of hierarchical nested subworkflows. In short a workflow basically consists of a number of subworkflows, which are orchestrated in an overall workflow, called the *structure-workflow*. The relation between structure-workflow and subworkflows is handled through the tasks of the structure-workflow. Tasks in the structure-workflow correspond to subworkflows. Subworkflows can, conceptually, be other structure-workflows also containing further subworkflows. However, for readability and simplicity we will restrict ourselves to a two-level hierarchy in this paper. This means that subworkflows consist only of tasks being executed by workflow resources (human or automated). In other words, the structure-workflow defines the basic outline and connection (the structure) between the different subworkflows. One key aspect of the development of this approach was to allow distributed workflow execution. For the agentworkflow approach in particular this manifests itself in the subworkflows. Each subworkflow is independent from the others and can be executed on a different registered system in the network, depending on the requirements of the particular subworkflow. Since the data- and control-flow of the workflow is handled within the structure-workflow the different subworkflows can be executed independently and only need to be coordinated at their beginning and end.

One designated agent, the so-called *structure-agent*, is responsible for the execution of the structure-workflow and the aforementioned distribution aspect. For each new workflow instance a new structure-agent is instantiated. This structure-agent is only responsible for his own agentworkflow instance. The structure-workflow is part of the structure agent. In fact, the agent is not only responsible for the execution of the structure-workflow, it even handles most of it itself. For this (and further autonomy reasons) it contains and replicates some parts of the AgWFMS functionality.

The system functions as follows: When the structure-agent is started it receives the structure-workflow definition from the AgWFMS\*. It instantiates and stores the workflow net internally and registers as a listener for this net (and this net only). This way, when tasks become activated the structure-agent can automatically react. When that happens it reads the description of the subworkflow directly from the task of the structure-workflow. From this description the structure-agent determines the circumstances for the subworkflow execution. It determines what kind of or what particular system this subworkflow needs to be executed on and then inquires which of these eligible systems are currently online and registered. It then proceeds to choose one of the (possibly) multiple systems for the actual execution of the subworkflow and contacts the interface agent of the chosen AgWFMS\*. After authentication the interface agent can decide to accept or reject the subworkflow. If it rejects the subworkflow the structure-agent chooses another system and tries instantiation of the subworkflow again, unless no suitable systems are found, in which case error handling must occur<sup>1</sup>. If it accepts the subworkflow, (optional) input data is sent from the structure-agent and the subworkflow is instantiated locally at that AgWFMS\*. This point is crucial to the flexibility aspect, since only the designation of the local subworkflow needs to be known to the structure-agent. The actual implementation of the subworkflow is, except for input and output data, independent from the structure-workflow. This will be discussed in the next section. The subworkflow is executed by the local<sup>2</sup> resources logged into that AgWFMS\*. When the subworkflow has finished its execution the (optional) output data is transferred, along with the confirmation of the subworkflows success, from the AgWFMS\* to the original structure-agent. The structure-agent then takes this information for the structure-workflow, which completes its own task-transition. This process is repeated for all subworkflows that become active during the execution of the structure-workflow. Once the structure-workflow reaches the final transition the workflow is finished and the structure-agent can persistently store the data and then terminate.

Figure 1 shows a snapshot of an execution of an exemplary agentworkflow. The structure-workflow is being executed by the structure-agent on AgWFMS\*1. It contains two active tasks/subworkflows that are currently being executed on two AgWFMS\*. AgWFMS\*1 is home to the structure-agent and is executing subworkflow A, while AgWFMS\*2 is executing subworkflow B. The figure clearly shows the relations between the agents and the workflows. The structure-workflow is only executed by the structure-agent. The structure-agent is in communication with the AgWFMS\* agents to oversee subworkflow execution. The subworkflows are in no way executed by the structure-agent, but only correspond to tasks in the structure-workflow. The only agents involved in the actual execution of the subworkflows are the ones of the local AgWFMS\* systems.

<sup>1</sup> This could, for example, include the search for alternate systems or require manual input from a user.

<sup>2</sup> *Local* in this context refers to the resources logged into this particular AgWFMS\* and does not exclude resources connected to the AgWFMS\* through a network.

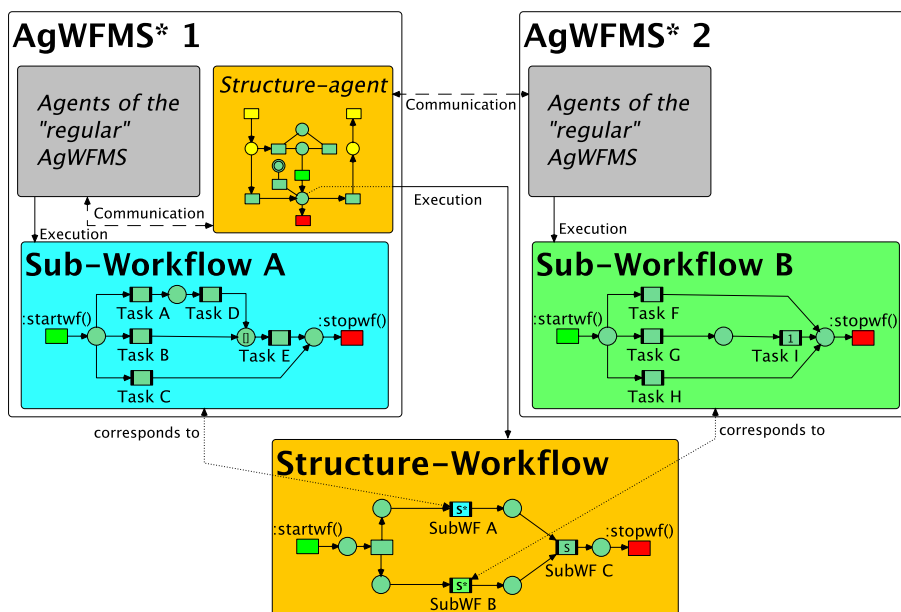


Fig. 1. Principle agentworkflow approach (from [17])

To conclude this description of the agentworkflow principle and AgWFMS\* system we will now shortly discuss its general attributes. This will not include the flexibility aspect, which will be discussed in the next section especially dedicated to this aspect. One of the most prominent features of the agentworkflow approach is the clear encapsulation of a workflow instance through a software agent. It provides the workflow instances with a very clear identity during its execution. This can be advantageous for monitoring and maintenance of the system. A disadvantageous aspect of the encapsulation is, that it increases the number of agents active within the system which can lead to performance issues. Furthermore if the structure-agent or its agent platform is terminated erroneously, the entire agentworkflow is lost. This is a problem we aim to fix in the future.

Additionally the encapsulation opens up many of the possibilities of software agents for workflow instances. Since, logically, the structure-agent and the workflow can be seen as equivalent many attributes usually associated with agents can be related to the workflow. This is further supported by the fact, that both the workflows and agents rely on the same technical background: reference nets. Both the approach and the technological implementation add to the integration of agent and workflow principles as proposed by the overall effort in [11].

The most obvious possibilities opened up by the integration are the distribution of workflow execution and interoperability. Since now all parts, including workflow instances, are implemented as agents, they can be distributed almost arbitrarily on the network. Also, since CAPA adheres to the FIPA standards,

interoperability with other agent systems is guaranteed, as long as the interfaces match up. In the future, further agent attributes can be added to the agentworkflow approach. Mobility for one matter can remedy some issues arising from the fact that the structure-agent serves as a central point of execution. In the original agentworkflow approach a AgWFMS\* platform would have to stay online for as long as the structure-agent was active. If the structure-agent was able to migrate to other platforms, the AgWFMS\* platform could shut down while the structure-agent continued its work on another AgWFMS\* in the network. This could also be used in error handling situations.

Furthermore, intelligence and autonomy could be added to the structure-agent. These two attributes could be used in various ways, for example resource access or control. But in combination with the mobility and distribution aspects this becomes even more interesting, since it is possible to create an intelligent, adaptive, migrating workflow instance. This is, however, outside of the scope of this paper.

## 5 Flexibility in Agentworkflows

In this section we will discuss the agentworkflows with regards to their flexibility aspects. As other attributes, such as mobility and intelligence, have already shortly been discussed before, we will not address these here.

The most crucial aspect of the agentworkflow approach with regards to flexibility is obviously the exchange of subworkflows and the loose coupling between structure-workflows and their related subworkflows. In the local view of the structure-agent (and as such, the structure-workflow) each subworkflow possesses only a name, a place to be executed at and a (possibly empty) input and output. The internal workings of a subworkflow are completely transparent to the structure-agent. In fact, they are completely irrelevant to the structure-workflow, as long as the subworkflow is completely executed and the correct output, w.r.t. types, is produced. As a simple example imagine a subworkflow dealing with checking the validity of a document nested within a structure-workflow for a bank. As input, this subworkflow would possess the document, the output could be a report on the document. How and in which order the different attributes of the document are checked is of no concern to the structure-workflow<sup>3</sup>, which just needs the report to continue its execution.

This key characteristic of agentworkflows can positively influence flexibility in workflow execution in a number of ways. First and foremost, it can enable the dynamic reconfiguration of the workflow specification. The simplest way to support this is to use a variable for the name of the subworkflow in the structure-workflow, instead of a constant identifier. This variable can depend on various factors, like results of previous subworkflows, and implicitly<sup>4</sup> represents the current circumstances of the workflow execution. When the task/subworkflow be-

<sup>3</sup> Of course it is of concern for the real-world application, but for this generalisation we can abstract from this.

<sup>4</sup> The variable has to be an identifier for the subworkflow.

comes activated now it is instantiated with an identifier depending on the current circumstances of the workflow execution. In this way, the structure-workflow can dynamically adapt to external factors of the execution. From a technical point of view, this is relatively easy to realise, since the unification algorithm of RENEW allows for such substitutions. The idea can even be extended to the target AgWFMS\* of the subworkflow. Depending on certain factors it might be necessary to not simply execute another subworkflow, but to also change the location. This can be achieved, pretty much in the same way it was done for the subworkflow identifier, although the computations might be a bit more complex. One might also consider extending this principle to input and output. This is however, more difficult, since input and output are usually quite closely tied to the subworkflow. Changing them in accordance to changing the subworkflow is possible (input and output are just variables anyway), but changing them independently would require the target platform to provide subworkflows with equal names and differing parameters. This would make the system more difficult to maintain, administrate and monitor.

It should be noted though, that the degree of flexibility added by this is not as high as could be desired. The subworkflows have to be known prior to workflow execution, so that they are available and compiled during runtime. This implies that unforeseen situations cannot be handled by our current approach. However, the particular flexibility added by the agentworkflows still enhances the system to deal with predictable, dynamic situations. This, in combinations with standardised error-handling mechanisms (e.g. subworkflows that involve administrators), can handle many practical scenarios. Furthermore, it is already quite easy to exchange or add subworkflows in the system. If unforeseen situations arise, adapted subworkflows could be modelled by a workflow modeller and introduced into the system. Making this functionality available more “on-the-fly” and integrating it more directly into the approach could remedy the current shortcomings in flexibility.

Though not implied by the “regular” agentworkflow approach it is also possible to modify the AgWFMS\* in order to support the flexibility mechanic from its side. In principle, this is mostly equivalent to the variable identifier version described above, only that the variables depends on factors of the target AgWFMS\* **and** the structure-agent. In this modification, the AgWFMS\* can autonomously decide which subworkflow to instantiate. It can take the *proposed* subworkflow, the input and the output, as well as its own state into consideration in this decision. The effect would generally be the same, only that the ultimate control would lie with the target AgWFMS\*. The combination of both approaches would yield an even higher degree of flexibility, since all factors local to the structure-agent *and* AgWFMS\* would be taken into consideration during subworkflow instantiation.

There are some further aspects the subworkflow characteristic adds to workflow execution with regards to flexibility, though these do not impact as much as the ones described above and also focus on other variations of flexibility. Since the structure-agent is ultimately responsible for choosing which AgWFMS\* should

execute a subworkflow, an adapted structure-agent with reasoning functionality and additional information about the different AgWFMS\* available could enhance flexibility in regards to load balancing or other similar efficiency factors. The agent could, for example choose (among the set of suitable AgWFMS\*) the system with the lowest workload or the one with the best network connection. The counterpart to this mechanic is to include the reasoning within the AgWFMS\*, which yields similar results. If the AgWFMS\* determines that its workload is too high, or that too few resources are available, it can reject the subworkflow, which would force the structure-agent to choose another AgWFMS\*.

The subworkflow hierarchy also contributes to flexibility. The two-level hierarchy discussed in this paper is only the simplest and easiest to describe version of the agentworkflow approach. When considering workflow hierarchies of more than two levels, each additional level offers a more fine-grained degree of flexibility. This can be especially useful in an interorganisational context, since the additional levels of organisation can profit there.

Interorganisational workflow execution deals with workflows that are executed cooperatively by different organisations. In this context each organisation is responsible for its own part of the workflow. Even though the organisations are working together within the process, each organisation is, of course, interested in keeping the confidential details from the other partners. Furthermore, each organisation will prefer to use their own WFMS, instead of relying on a central one which is used by all partners. For these reasons the agentworkflow approach is well suited for the interorganisational context, since the encapsulation ensures security and the interoperability and distribution aspects support the second requirement.

Nevertheless the flexibility aspects discussed above are also quite effective in this context. First and foremost, the ability to exchange subworkflows without influencing the overall structure-workflow is even more useful. In an interorganisational setting, factors that influence workflow execution can be even more varied, dynamic and demanding, since the collaboration introduces a lot of complexity in the real-world scenario. Being able to handle these factors by providing different subworkflows for different examples reduces the complexity again.

As mentioned before, adding additional levels to the hierarchy can also be used as an improvement in this context. By allowing subworkflows to be structure-workflows again, the flexibility aspects available on the interorganisational level (original structure-workflow) become available to the individual organisation as well. If the individual organisation uses this hierarchy-level to distribute work between different departments, additional levels would again open up these possibilities to the departments and so forth.

The structure-agents ability to choose a WFMS for subworkflow execution could take on a fully different role in the interorganisational context. By using negotiation concepts, different WFMS of different organisations could offer to take over subworkflows, which in the real-world scenario would translate to organisations competing for work assignments/contracts. This is, however, outside of the scope of the current agentworkflow approach.



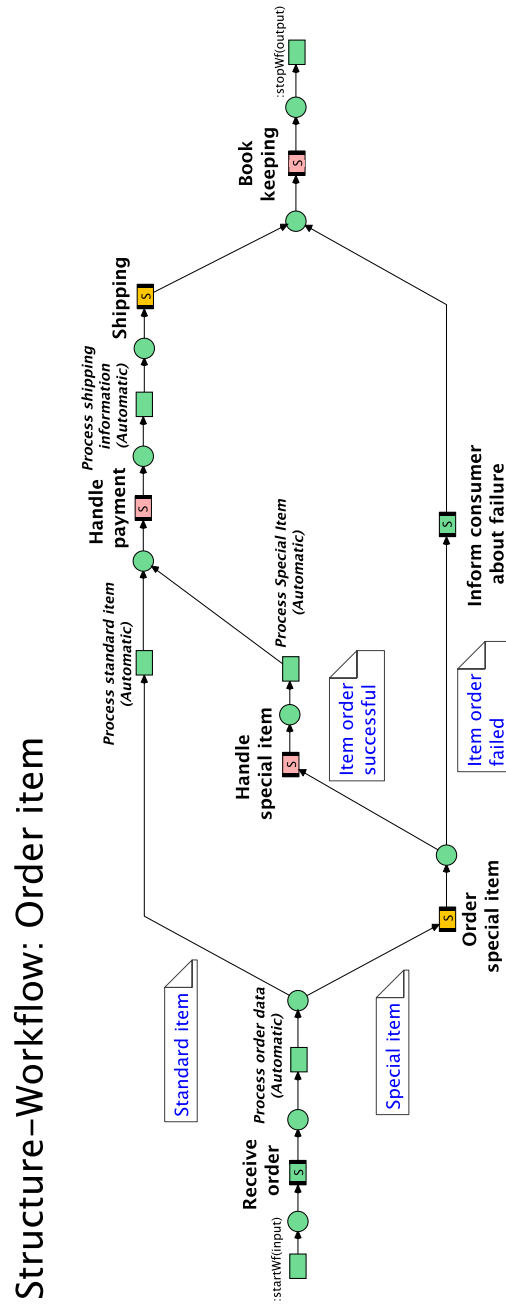


Fig. 2. Example Structure-Workflow

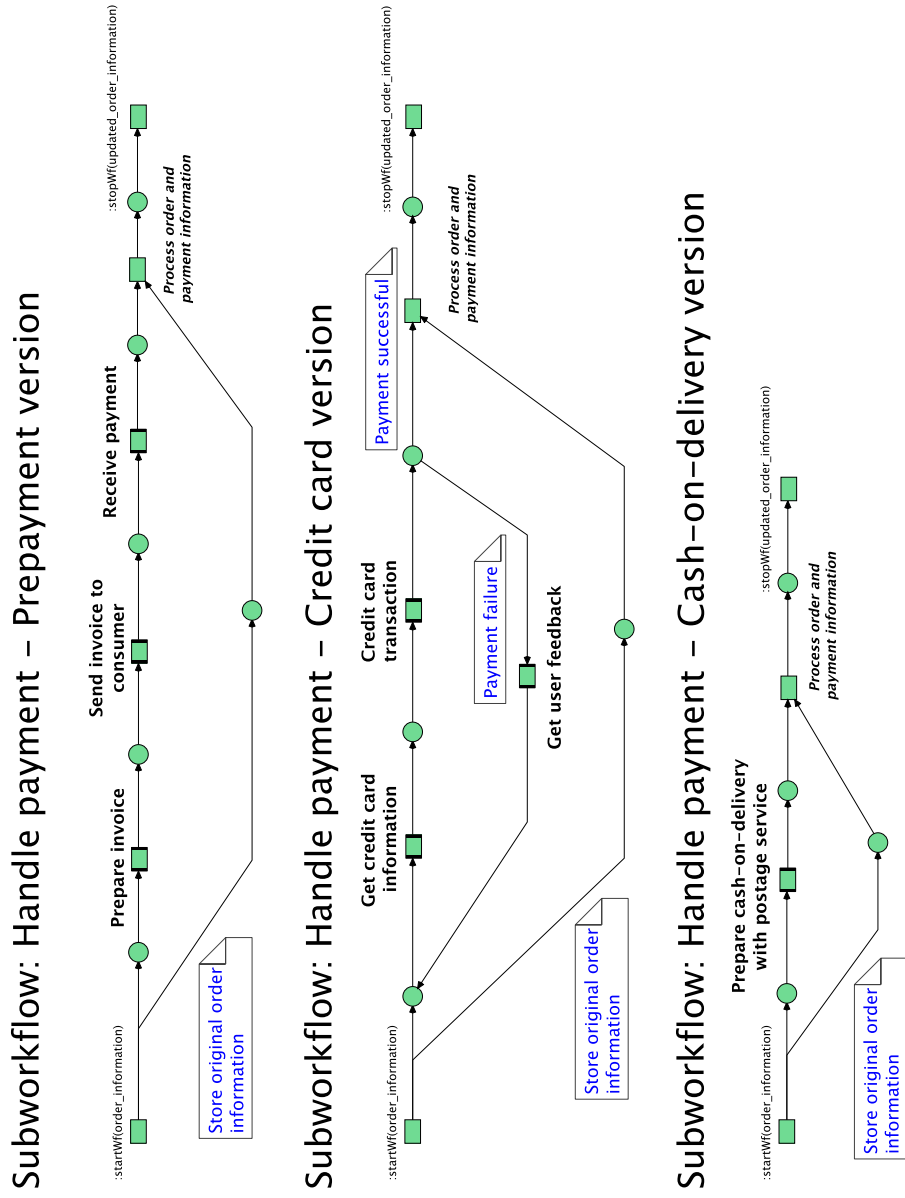


Fig. 3. Example Subworkflows

## 6 Example

In this section we will discuss a simple example of a flexible agentworkflow in an interorganisational setting. The structure-workflow can be seen in Figure 2. Workflow nets like these consist of regular Petri net places and transitions, as well as a few special elements. A workflow net starts with a transition connected to a synchronous channel labeled  $:startWf(input)$  over which it can receive parameters, and ends with a transition connected to synchronous channel labeled  $:stopWf(output)$  over which an optional result can be send. Tasks are represented as transitions with thick vertical bars, called task-transitions. Tasks in a structure-workflow representing subworkflows are also drawn as transitions with thick vertical bars and the letter  $S$  marked in the centre. The “regular” Petri net transitions within the figure represent (abstract) operations on the variables and data within the workflow. Usually they would/could feature more complex net structures as well as synchronous channels to receive outside information. However, to retain readability of the net for this paper we have chosen to abstract from a detailed and technical view in favour of a simplified version. For the same reason we omitted the exact, complex inscriptions on the example workflow nets. It should also be noted that, in order to keep the net size and complexity manageable error handling and aborting the workflow due to failures in subworkflows have largely been omitted.

Figure 2 represents a workflow for processing and handling an incoming order in a generic company offering many items. The company offers standard items, which are in stock and don’t need special treatment, and special items, which have to be ordered from a third-party provider and handled differently (e.g. large items or large quantities of items). The workflow encompasses the different steps from processing the incoming order, handling the standard or special item, handling payment, shipping via another third-party and finally book keeping. Each of these complex tasks is modelled as a subworkflow. It should be noted that we consider the main company to be in charge of this workflow, so that only the two subworkflows for the third-party providers feature distributed execution. We can observe three different types of subworkflows in this example.

The first type are regular subworkflows, which do not exhibit or require flexibility. In this example these are the processing of the original order (*Receive order*) and informing the consumer about problems in ordering a special item (*Inform consumer about failure*). These subworkflows do not need to be flexible, since, given the scenario, one version for each subworkflow can handle the possible circumstances. However, changing these subworkflows to be flexible would only require adding some variable processing ahead of them, changing the inscriptions to support the variables and providing the different subworkflows on the systems they would need to be executed on.

The second type of subworkflows are the flexible, interorganisational ones. These are ordering a special item (*Order special item*) and handling the shipping (*Shipping*). These are flexible since, depending on the item, they might have to be taken care of by different companies and through different subworkflows. For example ordering a bulky, large household item like a refrigerator would

require a different company and subworkflow than ordering a smartphone. These subworkflows illustrate how agentworkflows in general and their flexibility in particular can support interorganisational settings.

The third and final type of subworkflows are flexible, but local ones. These are handling a special item which has been ordered and received and now needs to be temporarily stored (*Handle special item*), the handling of the payment for a consumer (*Handle payment*) and book keeping and accounting (*Book keeping*). These are flexible since special items may have special requirements in storage or handling and the company might offer different ways of paying for an item.

Three different versions of the *Handle payment* subworkflow are shown in Figure 3. The three versions all represent the process involved in handling and receiving payment from the user. All three have in common that the original order information is stored for later processing with the last task before the subworkflow is finished (the lower branch in all three versions).

The topmost version represents prepayment by the user. The company prepares the invoice, sends it out and, at some later point, receives payment before the item is shipped. The middle version supports payment by credit card. The company receives credit card information from the consumer and executes the transaction. At this point the transaction was either successful or failed. If it failed user interaction (e.g. re-entering the credit card information) is required. If the transaction succeeds the item can be processed and shipped. The lowest version models payment by cash-on-delivery. In this case the cash-on-delivery only needs to be prepared with the postage service before the item can be shipped. In this case processing payment would be included in later stages of the overall workflow (e.g. a corresponding version of the *Book keeping* subworkflow).

This example illustrates the kind of scenario for which agentworkflow flexibility is especially suited. The different possibilities are known beforehand (e.g. the different payment options offered by the organisation) and each can be modelled accordingly. During execution the correct subworkflow can be instantiated and the requirements given by the variable factors of the workflow (in this simple example the choice of payment) can be fulfilled. While the different versions of the *Handle payment* subworkflow only differ in small parts, other scenarios could require more substantial changes in subworkflows. This could also easily be handled by the agentworkflow approach.

Though the workflow of Figure 2 is a simple example, it serves to illustrate the agentworkflow approach quite well. Subworkflows can feature distribution and flexibility, and it is also conceivable to mix subworkflows and regular tasks to loosen the hierarchy. If further aspects of agentworkflows, like intelligence and mobility, are considered, it becomes clear that even these already versatile ways only scratch the surface of the overall approach and its possibilities.

## 7 Conclusion

In this paper we have presented an approach to workflow management, called agentworkflows. It incorporates elements of both agent orientation and classic

workflow execution to combine strengths of both fields. The agentworkflow approach exhibits many interesting attributes, like encapsulation, intelligence and distribution. The focus of this paper though, was on the flexibility introduced by it. The flexibility of the agentworkflows relies on using a hierarchy of workflows and subworkflows and on allowing the dynamic exchange of subworkflows dependent on variable factors. This enables the dynamic reconfiguration of workflow instances at runtime. We discussed this aspect of the approach in detail and finally gave an example of how it could be used in an interorganisational context. The example illustrated the versatile ways, in which agentworkflows and their subworkflows could be deployed.

The flexibility introduced by the agentworkflow approach makes it more suitable for real-world scenarios than a classically rigid approach. However, there are currently some shortcomings to our approach, since subworkflows need to be known and compiled before execution of the overall structure-workflow. This limits the possibilities of the approach, since on-the-fly changes become difficult. These limitations, however, do not relate to the general approach and need to be fixed on a technological level, rather than a conceptual one. Addressing them is one of our goals for future work. Furthermore we also aim to address the other flexibility aspects discussed in this paper, as well as generally extend the agentworkflow approach with more concepts from both agents and workflows. Enhanced agent mobility, intelligence and distribution can greatly improve workflow execution and the process view given by workflows can enhance the agent-side. We hope to combine the two paradigms to profit from one another and further our overall goal to provide the desired complete integration of both. Ultimately, the intent is to develop a novel, general unit concept, which can serve as agent, workflow or both, depending on the dynamic requirements at runtime. Agentworkflows are one of the later steps towards that goal.

In conclusion, the agentworkflow approach possesses many qualities beneficial to flexible workflow execution. It serves as an important basis for future work regarding the integration of the agent and workflow concepts. By itself, the approach offers a simple, yet elegant way of supporting flexibility in workflow management.

## References

1. Abu Zafar Abbasi and Zubair A. Shaikh. A conceptual framework for smart workflow management. In *Information Management and Engineering, ICIME '09*, 2009.
2. Kamel Barkaoui, Hanifa Boucheneb, and Awatef Hicheur. Modelling and analysis of time-constrained flexible workflows with time recursive ecantnets. In Roberto Bruni and Karsten Wolf, editors, *WS-FM*, volume 5387 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 2008.
3. Kamel Barkaoui and Awatef Hicheur. Towards analysis of flexible and collaborative workflow using recursive ecantnets. In Arthur H. M. ter Hofstede, Boualem Benatallah, and Hye-Young Paik, editors, *Business Process Management Workshops*, volume 4928 of *Lecture Notes in Computer Science*, pages 232–244. Springer, 2007.

4. Peter Dadam, Manfred Reichert, Stefanie Rinderle-Ma, Kevin Göser, Ulrich Kreher, and Martin Jurisch. Von ADEPT zur AristaFlow BPM Suite - Eine Vision wird Realität: "Correctness by Construction" und flexible, robuste Ausführung von Unternehmensprozessen. *EMISA Forum*, 29(1):9–28, 2009.
5. Michael Duvigneau. Bereitstellung einer Agentenplattform für petrinetzbasierte Agenten. Diploma thesis, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg, December 2002.
6. Lars Ehrler, Martin Fleurke, Maryam Purvis, and Bastin Tony Roy Savarimuthu. Agent-based workflow management systems (WfMSs) - JBees: a distributed and adaptive WfMS with monitoring and controlling capabilities. *Information Systems and E-Business Management*, 4, Number 1 / January, 2006:5–23, 2005.
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. Olaf Kummer. *Referenznetze*. Logos Verlag, Berlin, 2002.
9. 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.
10. Daniel Moldt, José Quenum, Christine Reese, and Thomas Wagner. Improving a workflow management system with an agent flavour. 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 55–70, Vogt-Kölln Str. 30, D-22527 Hamburg, June 2010. University of Hamburg, Department of Informatics.
11. Christine Reese. *Prozess-Infrastruktur für Agentenanwendungen*. Dissertation, Vogt-Kölln Str. 30, D-22527 Hamburg, 2009.
12. Heiko Rölke. *Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen*, volume 2 of *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2004.
13. Rüdiger Valk. Concurrency in Communicating Object Petri Nets. In *Advances in Petri Nets: Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*, pages 164–195. Springer-Verlag, Berlin Heidelberg New York, 2001.
14. Wil M.P. van der Aalst. Verification of Workflow Nets. In *ICATPN '97: Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, volume 1248, pages 407–426, Berlin Heidelberg New York, 1997. Springer-Verlag.
15. Thomas Wagner. A Centralized Petri Net- and Agent-based Workflow Management System. Number FBI-HH-B-290/09 in Bericht, pages 29–44. University of Hamburg, September 2009.
16. 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.
17. Thomas Wagner, Jose Quenum, Daniel Moldt, and Christine Reese. Providing an agent flavored integration for workflow management. *LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, LNCS 6900, 2012.

# Cloud Transition: Integrating Cloud Calls into Workflow Petri Nets

Sofiane Bendoukha and Thomas Wagner

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

**Keywords:** Cloud computing, Workflow Management Systems, Workflow Petri Nets, Reference Nets, Cloud Workflow Transition

## Extended Abstract

In this paper, we present the Cloud Workflow Transition. An extension of Petri nets formalisms to adopt Cloud interactions. This allows workflows to request compute or storage services from the cloud. Such refinements permit to codify operational procedures into Petri net models and reduce user implication during the specification of their workflows. The main purpose behind our proposed refinements is to allow users to automatically execute workflows on distributed infrastructures (Cloud, SOA, grid, cluster).

Through the Cloud transition users can specify their requests formulated as tasks and parameters (see Figure 1). These requests will be treated in a transparent way i.e. that technical information is hidden from the user. The WFMS will then either accept the request and make the connection to the specified Cloud services according to user inputs or will reject it. The input places of the Cloud transition model the pre-conditions of an event, the input data for the computational task. The output places of the transition model the post-conditions associated with an event, the results of the computational task.

Our approach uses workflow Petri nets [1]. More specifically we use the reference net formalism [2] extended with a specialized workflow task transition [3]. RENEW, the **R**eference **N**et **W**orkshop, is our chosen tool for modeling with reference nets. A very interesting and useful property of reference nets in RENEW is their use of the so-called shadow layer. It hides the technical details from the user, who can concentrate on simply the nets.

The technical integration of the Cloud transition into our workflow nets and workflow management system is carried out in three main steps: The integration into the existing workflow net formalism for RENEW [3], the integration into the current WFMS in RENEW [4] and finally the integration into the user interface. Due to the dynamic aspect of the cloud computing, further integration issues are investigated such as including Quality-of-Service (QoS) requirements (time and expenditure limit). The WFMS should be able to identify and handle failures and support reliable execution in the presence of concurrency to guarantee a high level of performance and availability of services.

We plan to define the operational semantics of the Cloud transition using RENEW as well as a working use case. As an example, we intend to include the Cloud transition to model and enact a storage workflow using existing Cloud storage services within a Petri net-based multi-agent system.

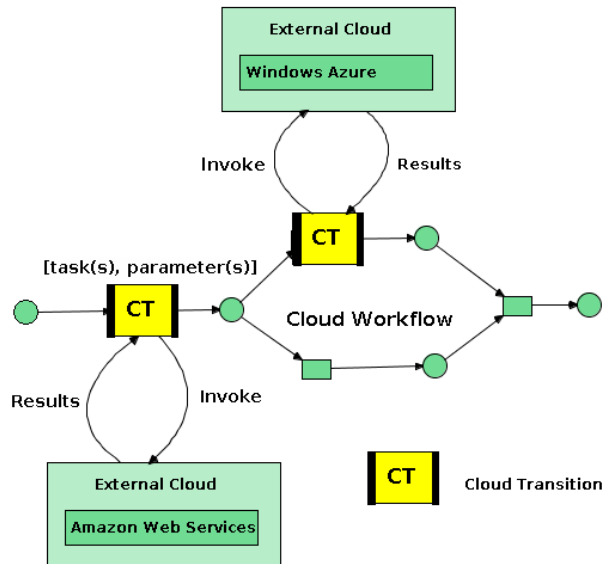


Fig. 1. The cloud transition

## References

1. Aalst, W.: Verification of workflow nets. In Azéma, P., Balbo, G., eds.: Application and Theory of Petri Nets 1997. Volume 1248 of Lecture Notes in Computer Science., Springer-Verlag, Berlin (1997) 407–426
2. Kummer, O.: Referenznetze. Logos Verlag, Berlin (2002)
3. Jacob, T., Kummer, O., Moldt, D., Ultes-Nitsche, U.: Implementation of workflow systems using reference nets – security and operability aspects. In Jensen, K., ed.: Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, Ny Munkegade, Bldg. 540, DK-8000 Aarhus C, Denmark, University of Aarhus, Department of Computer Science (August 2002) DAIMI PB: Aarhus, Denmark, August 28–30, number 560.
4. Wagner, T.: A centralized Petri net- and agent-based workflow management system. In Duvigneau, M., Moldt, D., eds.: Proceedings of the Fifth International Workshop on Modeling of Objects, Components and Agents, MOCA'09, Hamburg. Number FBI-HH-B-290/09 in Report of the Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, University of Hamburg, Department of Informatics (September 2009) 29–44



# A Concurrent Simulator for Petri Nets Based on the Paradigm of Actors of Hewitt

Luca Bernardinello and Francesco Adalberto Bianchi

Dipartimento di Informatica sistemistica e comunicazione  
Università degli studi di Milano-Bicocca  
Viale Sarca 336 U14, Milano (Italy)  
`luca.bernardinello@unimib.it`

**Abstract.** In this paper we propose a concurrent simulator for Petri nets based on the model of Actors of Hewitt. The classes of Petri nets that are supported for the simulation are Place-Transition Nets and Elementary Nets. The simulator is written in Scala, a programming language with a library implementing the Actors model.

**Keywords:** Petri Nets, Simulation, Actor Model, Scala programming language

## 1 Introduction

This paper deals with the design and implementation of a concurrent simulator for Petri nets based on Hewitt's Actor model. Different versions of the simulator have been built: two versions for Marked Graphs, one version for Free Choice Nets and finally the simulator for general Petri Nets. All the versions support both the firing rule of Elementary Nets and the firing rule of Place-transitions Nets. In this paper we will only describe the case of Place-Transition nets.

The simulator is based on the algorithm described by Dirk Taubner in [4], and is written in Scala, an object-oriented language built over Java. We chose to use the Actor model for the implementation of the concurrent simulator to make a comparison between the level of concurrency offered by this paradigm and that offered by Java threads. The algorithm is explained in Section 2.

The paradigm of Actors was introduced in 1973 by Carl Hewitt in [1]. An Actor is an independent entity which, concurrently to other Actors, can receive messages from other Actors, change its internal state, send new messages and create new Actors. Each Actor is identified by a mailing address; an Actor needs to know the mailing address of the Actors with which it wants to communicate. Each message contains the address of the sender.

Section 3 describes how Scala implements the Actor model. Several experiments have been made whose results are discussed in Section 5.

## 2 The Simulation Algorithm

Starting from Taubner's analysis ([4]) we have built a simulator such that two concurrent firings in the model correspond to (potentially) concurrent activi-

ties in the program. The strategy applied in our simulator provides a process for each place and for each transition of the net. Each transition communicates with its input places to check the firing condition. If enabled, the transition asks its neighbouring places either to decrement or to increment their number of tokens. The processes associated to places manage their number of tokens and the requests made by transitions. The communication between place and transition must follow a specific protocol because of conflicts. In a conflict situation a place must manage requests from different transitions; for this reason, a token reservation strategy is needed. Conflicts can also generate deadlocks in a simulator; if two transitions have made some of their reservations but they need a further reservation that they can't make because of the reservations of the other, a deadlock situation is generated. To avoid it, a transition must be able to cancel previous reservations.

The strategies just described are applied in the *Polling of places in a fixed order algorithm* (also called PTO). Each transition sequentially polls the processes corresponding to its input places in a fixed order and waits for the answer. At the first refusal, the transition cancels reservations made earlier and restarts polling from the first place. If the transition receives affirmative answers from all the places, it informs each place to increment or decrement its number of tokens. On the other hand, a place manages the reservation requests: if it can satisfy a request, it reserves the necessary token to the transition and it sends an affirmative message to the transition; if it can not satisfy a request, it sends a negative message to the transition.

### 3 Scala and Actor Model

Several programming languages realize concurrency through the paradigm of Actors. Scala has been designed in 2001 by Martin Odersky and it was released for the first time in 2004 on Java platform. Scala runs on the Java Virtual Machine, providing a high compatibility with existing Java code. Scala is a pure object-oriented language, supporting also functional programming. A library is provided implementing the Actor model. Each Actor in Scala has a mailbox in which it stores the received messages. The `react()` and `receive()` methods pick a message from the mailbox and check if it matches one of the patterns specified in the Actor. Scala Actors using `react()` are lightweight in comparison to Java threads. In particular, if an Actor implements the `receive()` method, Scala makes a one to one mapping with Java Virtual Machine threads; therefore each Scala Actor is implemented as a Java thread. For this reason, a program implemented through Scala Actors using `receive()` method has similar performances of a program based on Java threads. An Actor using the `react()` method instead, when it is started its behavior is captured in a closure and its stack is discarded. At this point, the Actor is suspended and the associated thread is free to execute other tasks. When the Actor receives a message, the corresponding closure is executed by a thread. Using this strategy, a thread is able to execute more than one Actor at the same time.

## 4 Implementation of the Simulator

The simulator is composed first of all by the `Net`, `Place` and `Transition` classes that define the structure of the net. When the user asks for the simulation of the specified Petri net, the class `Net` provides for the creation of the Actors needed. Two kinds of Actors are needed: one to represent places and one to represent transitions. For this reason, the classes `PlaceActor` and `TransitionActor` have been implemented. Each implements an Actor whose behavior reflects the task described in Taubner's algorithm.

In the first version for Marked Graphs, in which conflicts are not allowed, the reservations and cancellation strategies are not needed. In a second version of the simulator of Marked Graphs each place is seen as just a communication channel between two transitions, and not as an Actor.

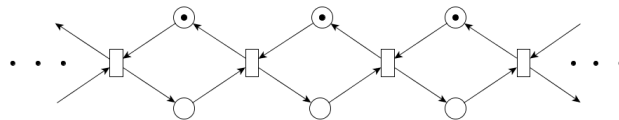
Another version simulates Free Choice Nets, in which if two or more transitions are in conflict, then they have a single input place (that is the same for all the transitions in conflict). In this case the reservation strategy is needed, but there is no need for cancelling reservations.

Finally, the simulator for general Petri nets has been implemented. This version reflects Taubner's algorithm, implementing both reservation and cancellation strategies.

## 5 Experimental Results

A series of experiments has been made in order to compare the degree of concurrency offered by the simulator based on Actors and the simulator based on Java threads: by degree of concurrency we mean the number of processes that the simulator can start during a simulation. We recall that when we speak about simulator based on Java threads we mean a simulator implemented through Scala Actors using `receive()` method. Moreover we have studied the ratio between the number of transition firings and the number of cancellations of reservations.

For the first experiments we used a net with a regular structure, whose size is simply parameterizable. In particular, we considered the following net, where the basic module can be replicated:

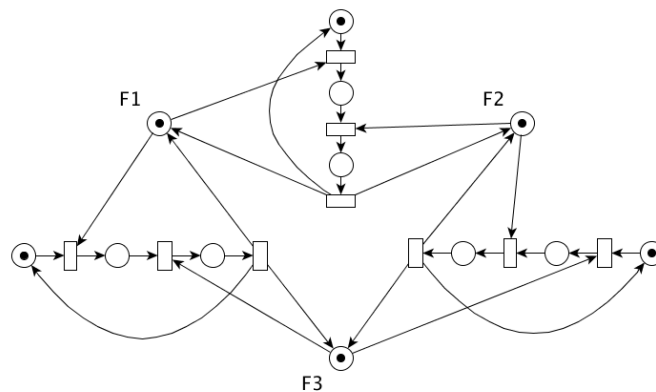


The experiments have been conducted on two different computers: the first has a dual-core processor with 2 GB of memory, the second has a quad-core processor with 4 GB of memory. The results of experiments made on first computer are the following:

	Processes	Core 1	Core 2	Memory used
Actors	18 000	30%	30%	2000/2000 Mb
Threads	200	100%	100%	1200/2000 Mb

From the table it can be seen that the number of processes started in the case of Actors is much greater. In the case of Actors the number of started processes is limited by memory, which is totally occupied; in case of threads, the limit comes from the processors usage. The experiments made on the second configuration have yielded results consistent with those just described.

The aim of the second type of experiments was to record some significant parameters to test the efficiency of the implementation. The net used in this case models the dining philosophers (see Fig 1). Specifically, we consider a variant of the model in which some philosophers take first their left fork, while the others take first the right fork.



**Fig. 1.** The net of three dining philosophers.

The first kind of data we wanted to record was the number of transitions that fire and the number of negative answers that the transitions receive.

An interesting property shown by data concerns the ratio between the number of transitions that fire and the number of refusal received by transitions (see table below); this ratio is constant and independent from time and number of transitions. Its value is close to 0.1, which means that each transition, to obtain the permission to fire, receives on average ten refusals. This result is justified by the fact that each transition continually tries to fire, this suggests to explore changes to the algorithm in order to reduce the number of refusals.

Time (sec)	Philosophers	Transitions	Negative answers	Ratio
60	50	2344	22564	0.1038
	250	2357	22678	0.1039
	1000	2355	22421	0.1051
600	500	23620	227666	0.1037

The second kind of data that we have recorded is the number of transitions that fire and the number of messages sent by transitions to cancel a reservation (see the table below). We recall that the number of negative answers (recorded in the first experiments) and the number of cancellations of reservations are not the same. In fact if a transition receives a negative answer from the first place, it does not need to cancel any reservation. In this case the ratio between the number of transitions and the number of cancellations of reservations is approximately 1, which means that each transition sends on average a single message to cancel reservations before firing.

Time (sec)	Philosophers	Transitions	Delete reservation	Ratio
60	50	2332	2359	0.9885
60	250	2344	2252	1.0408
60	1000	2320	1868	1.2419
600	500	23635	24139	0.9791

## 6 Conclusion

Taubner's algorithm allowed us to implement a concurrent simulator for Petri nets. Furthermore Scala has proved to be a very simple and elegant programming language, which, thanks to Actors model, offers a level of concurrency significantly higher than threads Java. Planned future developments are the implementation of a GUI for the simulator and other tools for analyzing the results of simulations. We will also explore alternative algorithms which exploit structural properties of the net to be simulated. In particular, we will consider nets that can be decomposed in several State Machine. An Actor will be associated to each sequential component.

## References

1. Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.
2. Martin Odersky, Lex Spoon, Bill Venners. *Programming in Scala*. Artima, First edition, 2008.
3. Philipp Haller, Frank Sommers. *Actors in Scala*. Artima, 2012.
4. Dirk Taubner. On the implementation of Petri nets. In *European Workshop on Applications and Theory of Petri Nets*, pages 418–434, 1987.

# A Petri Net Approach to Synthesize Intelligent State Machine Models from Choreography<sup>\*</sup>

Toshiyuki Miyamoto<sup>1</sup> and Yasuwo Hasegawa<sup>1</sup>

Graduate School of Engineering, Osaka University,  
Suita, Osaka 565-0871, Japan  
miyamoto@eei.eng.osaka-u.ac.jp

**Abstract.** Application of service-oriented architecture, which builds the entire system by a combination of independent software components, to a wide variety of computer systems is expected. The problem to synthesize state machine models of the services from a communication diagram representing the overall specifications of service interaction is known as the choreography realization problem. It should be minded on automatic synthesis that software models should be simple to be understood easily by software engineers. In this paper, we propose a method to synthesize hierarchical state machine models for the choreography realization problem. The proposed method is evaluated using a metric for intelligibility.

## 1 Introduction

In recent years, the internationalization of activities and information technology in the enterprise has intensified competition between companies. Companies are under pressure to respond quickly to business needs, and the period for making changes to existing business and launching new businesses has been shortened. For this reason, the need to change or build quickly information systems has been increasing.

Under such circumstances, service-oriented architecture (SOA)[12] has been attracting attention as the architecture of information systems in the enterprise. In SOA, an information system is built by composing independent software units called services.

In this paper, we consider the problem of synthesizing a concrete model from an abstract specification. It is not easy for the designers to design a concrete model directly from requirements since there exists huge gaps. But, defining an abstract specification is relatively simple. Therefore, if we can automatically synthesize a concrete model from abstract high-quality specification, it is expected that designer's workload is greatly reduced and product quality is improved.

In the field of software engineering, there exist several investigations that synthesize the concrete model from the abstract specification. Harel et al. proposes a methodology for synthesizing statechart models from scenario-based requirements[5]. Whittle et al. proposes a methodology for synthesizing hierarchical

---

<sup>\*</sup> This work was supported by KAKENHI (23500045).

state machine models from expressive scenario descriptions[13]. Liang et al. defines a set of comparison criteria, and surveys 21 different synthesis approaches presented in literature based on the criteria[7]. In addition, the theory of regions has been attracting attention as a method to synthesize nets[3].

In SOA, the problem to synthesize the concrete model from an abstract specification is known as the choreography realization problem[11]. In which the abstract specification, called *choreography*, is defined as a set of interactions among services, which are given in a dependency relation of messages sent and received; the concrete model is called the *service implementation* which defines the behavior of the service. This paper utilizes the communication diagram and the state machine of UML 2.x[10] to describe the choreography and the service implementation, respectively.

Bultan and Fu formally introduced the choreography realization problem in [2]. They used collaboration diagrams of UML1.x and showed some conditions for a given choreography to be realizable. In addition, they showed a method to represent the service implementation as the state space in which a state was defined as a set of unsent messages, and they also showed a method to map to a set of finite state machines. However, it is not intelligible because the number of states increases exponentially as the number of messages increases. Furthermore, they have adopted the semantics that message send and receive events for a synchronous call occur simultaneously. Under this semantics, the UML specification that “the execution of the call operation action waits until the execution of the invoked behavior completes and a reply transmission is returned to the caller”[10] can not be represented.

Miyamoto et al. have proposed a method to synthesize hierarchical state machines from the choreography given in communication diagrams[8]. In the method, dependency constraints between message send and receive events are represented by Petri nets[9]; the state machine is synthesized from its reachability space. A method to extract the hierarchical structure by analyzing the reachability space is given, but the technique can only be applied to simple cases.

This paper proposes a method of converting a Petri net into the state machine directly. It is shown that there is a relation between the possibility of direct conversion and structural properties of Petri nets. At first the proposed method converts the Petri net so as to satisfy the structural properties, then it converts Petri nets into hierarchical state machines without generating their state spaces.

This paper is organized as follows. Section 2 introduces an UML subset, called *subset of UML for formally describing choreography and behavioral feature* (cbUML), to discuss the choreography realization problem, and an extended Petri net, called message mark graph (MMG). The proposed method, called Construct State-machine Cutting Bridges (CSCB) method, is evaluated in terms of the intelligibility in Sect. 3. However, in this paper it is assumed that the choreography is given in a single communication diagram as the first stage of the study. Section 4 is the conclusion.

## 2 Preliminaries

### 2.1 cbUML

Let us introduce a subset of UML, called cbUML, for the discussion in this paper.

**Definition 1 (cbUML).** *cbUML is a tuple  $(\mathcal{C}, \mathcal{M}, \mathcal{A}, \mathcal{CD}, \mathcal{SM})$ , where  $\mathcal{C}$  is the set of classes,  $\mathcal{M}$  is the set of messages,  $\mathcal{A}$  is the set of attributes,  $\mathcal{CD}$  is the set of communication diagrams, and  $\mathcal{SM}$  is the set of state machines. Each of messages and attributes is owned by a class, and behavior of a class is defined by a state machine.*

**Messages** The set  $\mathcal{M}$  of messages are partitioned with respect to the sort of messages:  $\mathcal{M} = \mathcal{M}_{sop} \cup \mathcal{M}_{aop} \cup \mathcal{M}_{rep}$ , where  $\mathcal{M}_{sop}$  is the set of *synchronous messages* generated by synchronous calls,  $\mathcal{M}_{aop}$  is the set of *asynchronous messages* generated by asynchronous calls, and  $\mathcal{M}_{rep}$  is the set of *reply messages* to synchronous messages. Let  $\mathcal{M}_s = \mathcal{M}_{sop}$  and  $\mathcal{M}_a = \mathcal{M}_{aop} \cup \mathcal{M}_{rep}$ . Correspondence between the synchronous call and its response is given by the function  $refer : \mathcal{M} \mapsto \mathcal{M} \cup \{\text{nil}\}$ , such that  $\forall m \in \mathcal{M}_{sop} : refer(m) \in \mathcal{M}_{rep}$ ,  $\forall m \in \mathcal{M}_{rep} : refer(m) \in \mathcal{M}_{sop}$ , and  $\forall m \in \mathcal{M}_{aop} : refer(m) = \text{nil}$ . Note that  $\forall m \in \mathcal{M}_{sop} \cup \mathcal{M}_{rep} : refer(refer(m)) = m$ .

There is a difference in behavior during interactions due to differences in the sort of message as follows: In a synchronous call, caller's execution is stopped until it receives a reply from the callee. On the other hand, in the asynchronous call, the caller is possible to continue to operate, regardless of the behavior of the callee side.

In UML, each message has two events: a *send event* and a *receive event*. For a synchronous message, it is considered that they occur simultaneously. However, for the later discussion we need two events that occur separately. Therefore we define that each synchronous message has two events: a *preparation event* for message sending and a *send-receive event*, where the preparation event is a caller's event and the send-receive event is a callee's event. A preparation event and a send-receive event for a synchronous message  $m \in \mathcal{M}_s$  are denoted by  $\$m$  and  $!m$ , respectively. For an asynchronous message  $m \in \mathcal{M}_a$ , its send event and receive event are denoted by  $!m$  and  $?m$ , respectively. Hereafter a send event is a send-receive event for a synchronous message or a send event for an asynchronous message. The set  $\Sigma$  of message events and the set  $\Delta$  of send events are defined as follows:

$$\begin{aligned} \Sigma &= \{\$m, !m \mid m \in \mathcal{M}_s\} \cup \{!m, ?m \mid m \in \mathcal{M}_a\}, \text{ and} \\ \Delta &= \{!m \mid m \in \mathcal{M}\}. \end{aligned}$$

**Communication Diagrams** Communication diagrams show interactions where the arcs between the communicating lifelines are decorated with description of the passed messages and their sequencing.



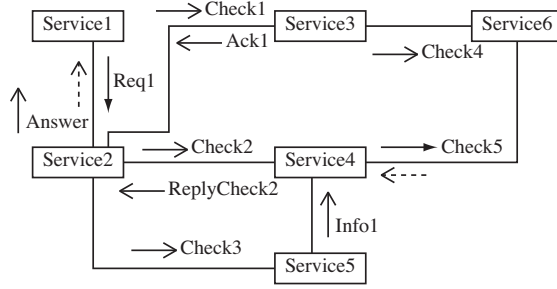


Fig. 1. A communication diagram

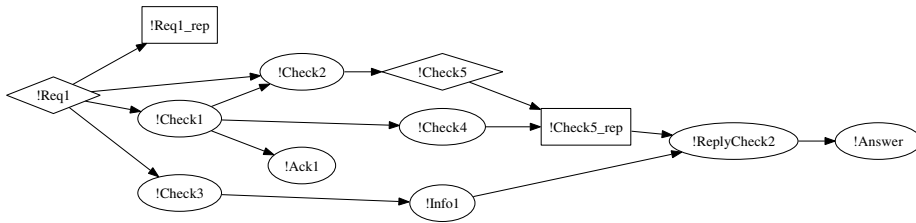


Fig. 2.  $D_{cd}$

**Definition 2 (Communication Diagram).** A communication diagram  $cd \in \mathcal{CD}$  is a tuple  $cd = (\mathcal{C}_{cd}, \mathcal{M}_{cd}, Conn_{cd}, line_{cd}, D_{cd})$ , where  $\mathcal{C}_{cd} \subseteq \mathcal{C}$  is the set of instances of classes (called lifelines or objects) in  $cd$ ,  $\mathcal{M}_{cd} \subseteq \mathcal{M}$  is the set of messages in  $cd$ ,  $Conn_{cd} \subseteq \mathcal{C}_{cd} \times \mathcal{C}_{cd}$  is the set of connectors, which is given as a symmetric relation on  $\mathcal{C}_{cd}$ ,  $line_{cd} : \mathcal{M}_{cd} \mapsto Conn_{cd}$  assigns a connector for each message, and  $D_{cd} \subseteq \Delta_{cd} \times \Delta_{cd}$  is a dependency relation among send events. Note that the reflexive and transitive closure of  $D_{cd}$  is a partial order.

A conversation is the sequence of messages exchanged among the objects. The set of conversations defined by a communication diagram  $cd$  is denoted by  $\mathfrak{C}(cd) \subseteq 2^{\mathcal{M}^*}$ , where  $\mathcal{M}^*$  is the set of all sequences of messages.

**Definition 3.** A conversation  $\sigma = m_1 m_2 \dots m_n$  is in  $\mathfrak{C}(cd)$  if and only if  $\sigma \in \mathcal{M}^*$  and the corresponding sequence  $\gamma = !m_1 !m_2 \dots !m_n$  of send events satisfies  $\forall i, j \in [1..n] : (!m_i, !m_j) \in D_{cd} \Rightarrow i < j$ .

Figure 1 shows a communication diagram. In this example, messages Req1 and Check5 are synchronous messages, and dashed lines with open arrow head are their reply messages. Suppose that the dependency relation among send events is given as shown in Fig. 2, where rhombuses, rectangles, and ellipses indicate synchronous calls, their reply, and asynchronous calls, respectively. The following sequence is a conversation of the example.

$$\sigma = \text{Req1 Check1 Req1\_rep Check2 Check3}$$

**State Machines** The behavior of each object is described by a state machine.

**Definition 4 (State Machine).** A state machine is a tuple  $sm = (V_{sm}, R_{sm}, top_{sm}, cont_{sm}, TR_{sm}, E_{sm}, Const_{sm}, Beh_{sm})$ , where  $V_{sm} = SS_{sm} \cup CS_{sm} \cup FS_{sm} \cup IS_{sm}$  is the set of vertices<sup>1</sup>,  $R_{sm}$  is the set of regions,  $top \in R_{sm}$  is the top region,  $cont_{sm} : (V_{sm} \cup R_{sm}) \setminus \{top_{sm}\} \mapsto (CS_{sm} \cup R_{sm})$  is an ownership relation between vertices and regions,  $TR_{sm}$  is the set of transition relations,  $E_{sm}$  is the set of events,  $Const_{sm}$  is the set of constraints, and  $Beh_{sm}$  is the set of behaviors.

In UML state machines, although there are various kinds of states and pseudo-states, only simple states, composite states, final states, and initial pseudo-states are used in this paper. A composite state is able to own one or more regions, and a region is able to own vertices. The function  $cont_{sm}$  represents the ownership of vertices and regions, and  $cont_{sm}(x_1) = x_2$  means that  $x_1$  is owned by  $x_2$ . For a  $x \in V_{sm} \cup R_{sm}$ , let  $des(x) = \{x' \mid \exists i > 0 : cont_{sm}^i(x') = x\}$  be the set of descendants of  $x$ , where  $cont_{sm}^1(\cdot) = cont_{sm}(\cdot)$  and  $cont_{sm}^i(\cdot) = cont_{sm}(cont_{sm}^{i-1}(\cdot))$  ( $i > 1$ ).

**Definition 5 (Orthogonal State).** If there exist vertices  $v_1, v_2 \in V_{sm}$  and different regions  $r_1, r_2 \in R_{sm}$ ,  $r_1 \neq r_2$  such that  $cont_{sm}(r_1) = cont_{sm}(r_2)$ ,  $v_1 \in des(r_1)$ , and  $v_2 \in des(r_2)$ , two vertices  $v_1, v_2$  are called orthogonal, and denoted by  $v_1 \perp v_2$ .

**Definition 6.** A set  $\hat{V}_{sm} \subset V_{sm}$  of vertices is called consistent if and only if for each pair  $v_1, v_2 \in \hat{V}_{sm}$  of vertices  $v_1 \perp v_2$ ,  $v_1 \in des(v_2)$ , or  $v_2 \in des(v_1)$ .

The set  $E_{sm}$  of events is given as  $E_{sm} = \Sigma_{sm} \cup \{\tau\}$ , where  $\Sigma_{sm}$  is the set of message events in the state machine and  $\tau$  is the *completion event* that occurs when a transition with no trigger event fires.

A transition relation  $etr \in TR_{sm}$  is a tuple  $etr = (src, trig, grd, eff, tgt)$ , where  $src \in V_{sm}$  is the originating vertex of the transition, a trigger  $trig \in E_{sm}$  is the event that makes the transition fire, a guard  $grd \in Const_{sm}$  is a constraint, an effect  $eff \in Beh_{sm}$  is an optional behavior to be performed when the transition fires, and  $tgt \in V_{sm}$  is the target vertex. Note that  $\{src, tgt\}$  must not be consistent. According to the UML specification[10], triggers, guards, and effects are denoted like “ $trig[grd]/eff$ ” in diagrams. It is supposed that  $\Sigma_{sm} \subseteq Beh_{sm}$ , and a caller’s event of message sending becomes an effect and a callee’s event becomes a trigger.

Due to space limitations, the details of operational semantics of state machines are omitted, and the steps of doing synchronous calls and asynchronous calls are explained by examples.

Figure 3 shows the execution of the asynchronous call. Gray states are active. When state machine sm1 transitions from state s11 to state s12 by the occurrence of the completion event, an asynchronous call is executed. At this time the send

<sup>1</sup>  $SS_{sm}$  is the set of *simple states*,  $CS_{sm}$  is the set of *composite states*,  $FS_{sm}$  is the set of *final states*, and  $IS_{sm}$  is the set of *initial pseudo states*.

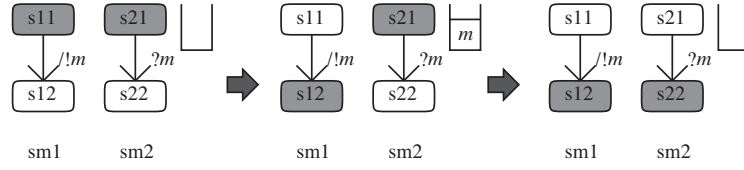


Fig. 3. Steps for an asynchronous call

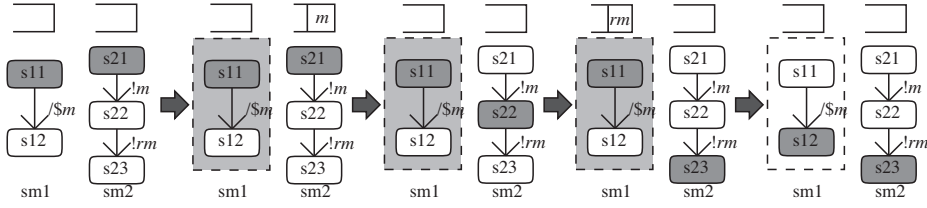


Fig. 4. Steps for a synchronous call

event  $!m$  occurs, and the message  $m$  will be appended at the end of the queue of  $sm2$ . The state machine  $sm2$  transitions from state  $s21$  to state  $s22$  consuming the message  $m$  by the occurrence of the receive event  $?m$ .

Figure 4 shows the execution of the synchronous call. A synchronous call is executed in  $sm1$ . At this time, the preparation event  $!m$  occurs in  $sm1$ , and the region that contains the transition is suspended. In addition, the message  $m$  is appended at the end of the queue of  $sm2$ . The state machine  $sm2$  transitions from state  $s21$  to  $s22$  consuming the message  $m$  by the occurrence of the send-receive event  $!m$ . The  $sm2$  sends a reply message  $rm$  to  $sm1$  on transitioning from  $s22$  to  $s23$ . At this time the send event  $!rm$  occurs, and the message  $rm$  is appended at the end of the queue of  $sm1$ . The  $sm1$  releases the suspended region, and transitions from state  $s11$  to state  $s12$  consuming the reply message  $rm$  by the occurrence of the receive event  $?rm$ . Note that the receive event  $?rm$  does not appear in the state machine, because we are using the region suspend mechanism.

The set of all conversations obtained by the execution of a set  $\mathcal{SM}$  of state machines is denoted by  $\mathfrak{C}(\mathcal{SM})$ .

### 2.2 Petri nets

The proposed method represents the dependency relation between message send and receive events by using Petri nets[9]. Since this paper assumes that the choreography is given by a communication diagram, Petri nets that appear in this paper are marked graphs.

A Petri net  $N = (P, T, F, W)$  is called *ordinary* when  $\forall (x, y) \in F : W(x, y) = 1$ . Then the weight function  $W$  is omitted. For  $x \in P \cup T$ , the set  $\{y \in P \cup T \mid (y, x) \in F\}$  is called the preset of  $x$ , and denoted by  $\bullet x$ . In the same way, the set  $\{y \mid (x, y) \in F\}$  is called the postset of  $x$ , and denoted by  $x\bullet$ .

A place  $p \in P$  is called a *source* place and a *sink* place when  $\bullet p = \emptyset$  and  $p\bullet = \emptyset$ , respectively. In the same way a transition  $t \in T$  is called a source transition and a sink transition when  $\bullet t = \emptyset$  and  $t\bullet = \emptyset$ , respectively. A transition  $t$  is called a *fork* transition and a *join* transition when  $|t\bullet| \geq 1$  and  $|\bullet t| \geq 1$ , respectively. The sets of join transitions and fork transitions are denoted by  $T_{join}$  and  $T_{fork}$ , respectively. Under the standard definition, if  $\forall p \in P : |\bullet p| = 1$  and  $|p\bullet| = 1$ , then the Petri net is called a *marked graph*. In this paper, we relax the condition as  $\forall p \in P : |\bullet p| \leq 1$  and  $|p\bullet| \leq 1$ .

**Definition 7 (Message Marked Graph).** A message marked graph (MMG) is a tuple  $N = (P, T, F, W, G, A)$ , where the underlying Petri net  $(P, T, F, W)$  satisfies the following conditions:

1.  $N$  is an ordinary and acyclic,
2. there exist only one source place  $p_s$  and only one sink place  $p_e$ ,
3. no source transitions and sink transitions exist, and
4.  $|p_s\bullet| = 1$ ,  $|\bullet p_e| = 1$ , and  $\forall p \in P \setminus \{p_s, p_e\} : [|\bullet p| = 1, |p\bullet| = 1]$ .

$G : T \mapsto 2^T$  is a firing constraint, and the partial function  $A : T \mapsto \Sigma$  assigns an event for the transition.

A state of MMG is expressed by a pair  $(M, X)$ , where  $M : P \mapsto \mathbb{Z}^+$  is a marking and  $X : T \mapsto \mathbb{B}$  is a firing configuration, where  $\mathbb{Z}^+$  is the set of non-negative integers and  $\mathbb{B} = \{\text{true}, \text{false}\}$ . The initial state  $(M_0, X_0)$  of MMG is given as follows:

$$M_0(p) = \begin{cases} 1 & \text{if } p = p_s \\ 0 & \text{otherwise, and} \end{cases}$$

$$X_0(t) = \text{false} \quad (\forall t \in T).$$

A transition  $t \in T$  is enabled if and only if  $\forall p \in \bullet t : M(p) \geq W(p, t)$  and  $\forall t' \in G(t) : X(t') = \text{true}$ . A new state  $(M', X')$  obtained by the firing of transition  $t$  is given as follows:

$$M'(p) = M(p) - W(p, t) + W(t, p), \text{ and}$$

$$X'(t') = \begin{cases} \text{true} & \text{if } t' = t \\ X(t') & \text{otherwise.} \end{cases}$$

**Handles and Bridges** Let  $N = (P, T, F)$ , and  $N_1 = (P_1, T_1, F_1)$  be a subnet of  $N$ . An elementary path  $H = (n_1, \dots, n_r), r \geq 2$  of  $N$  is a *handle* of  $N_1$  if and only if  $H \cap (P_1 \cup T_1) = \{n_1, n_r\}$ .

Let  $N = (P, T, F)$ , and  $N_1 = (P_1, T_1, F_1)$  and  $N_2 = (P_2, T_2, F_2)$  be subnets of  $N$ . An elementary path  $B = (n_1, \dots, n_r), r \geq 2$  is a *bridge* from  $N_1$  to  $N_2$  if and only if  $B \cap (P_1 \cup T_1) = \{n_1\}$  and  $B \cap (P_2 \cup T_2) = \{n_r\}$ .

For a transition  $t \in T$ ,  $FJ(t) \subseteq T$  is a set of terminal vertices of handles starting from  $t$ . Similarly,  $JF(t) \subseteq T$  is a set of starting vertices of handles terminating at  $t$ . Please refer to [4] for more detail about handles and bridges.

**Convertible MMG** Intuitively, if two states in a state machine are consistent, both states may be active concurrently. The UML specification prohibits drawing a transition between consistent states. An MMG is a representation of the relationship between the order of the messages in the marked graph; in general cases a state machine which satisfies the specification can not be directly, namely without generating its state space, derived from the MMG.

Let us introduce a subclass of MMG called the convertible MMG (CMMG), from which we can get a state machine satisfying the specification directly by using Algorithm 1 shown later.

**Definition 8.** A MMG is called a CMMG if the following conditions hold:

1.  $|T_{fork}| = |T_{join}|$
2.  $T_{fork} \cap T_{join} = \emptyset$
3. For any pair of handles  $H, H'$  in the MMG, only one of following conditions holds: (a)  $H$  and  $H'$  share the same starting vertex and the same terminal vertex, and (b) the starting vertices of  $H$  and  $H'$  are different and the terminal vertices of  $H$  and  $H'$  are different.
4. If  $A(t) = \$m$ , then  $t \bullet \bullet = \{t'\}$ ,  $A(t') = ?refer(m)$ .

From the definition of CMMG, for all  $t \in T_{fork}$  (resp.  $t \in T_{join}$ )  $|FJ(t)| = 1$  (resp.  $|JF(t)| = 1$ ). Moreover the following lemma holds.

**Lemma 1.** Let  $N$  be a MMG,  $N_1$  be a subnet of  $N$ , and  $H$  be a handle of  $N_1$ . If  $N$  is a CMMG, then there is no bridge from  $H$  to  $N_1$ .

Algorithm 1 shows how a CMMG is converted to a state machine. In the algorithm, if  $A(t) \in \{!m \mid m \in M_s\} \cup \{?m \mid m \in M_a\}$  then  $Event(t) = A(t)$ , and  $Constraint(t) = \bigwedge_{a \in G(t)} fired_a$ . The mapping  $Behavior(t)$  is given as follows:

$$Behavior(t) = \begin{cases} own(m).m(\dots) & \text{if } A(t) \in \{\$m \mid m \in \mathcal{M}_{sop}\} \\ \text{send } m(\dots) \text{ to } own(m) & \text{if } A(t) \in \{!m \mid m \in \mathcal{M}_{aop}\} \\ \text{reply to } m(\dots) & \text{if } A(t) \in \{!m \mid m \in \mathcal{M}_{rep}\} \end{cases}$$

where,  $own(m)$  is the owner object of message  $m$ , and in cbUML these expressions show a synchronous call, an asynchronous call, and a reply for a synchronous call. In addition, if  $fired_t \in A$ , then add an expression ' $fired_t = true$ ' to  $Behavior(t)$ . The 'new' expression shows a new element is generated.

**Lemma 2.** A CMMG is directly convertible to a state machine.

Figure 5 shows an example of CMMG, and Fig. 6 is the synthesized state machine by Algorithm 1. In Fig. 5, places on each edge,  $p_s$  and  $p_e$  are omitted.

## 3 Choreography Realization Problem

### 3.1 Choreography Realization Problem

By a single communication diagram, one scenario that is an interaction of objects in the system are described. All behavior of the system is given by a set of communication diagram; this is referred to as choreography.

**Algorithm 1:** Converting CMMG to a state machine

---

**Input:** CMMG  $(P, T, F, G, A)$   
**Output:** State machine  $(V, R, top, cont, TR, E, Const, Beh)$ , Attribute  $\mathcal{A}$

```

1 begin
2    $\mathcal{A} \leftarrow \{fired_t \mid t \in \bigcup_{t' \in T} G(t')\};$ 
3    $E \leftarrow \{Event(t) \mid t \in T\};$ 
4    $Const \leftarrow \{Constraint(t) \mid t \in T\};$ 
5    $Beh \leftarrow \{Behavior(t) \mid t \in T\};$ 
6    $V \leftarrow \emptyset;$ 
7    $R \leftarrow \emptyset;$ 
8    $t_{init} \leftarrow p_s \bullet;$ 
9    $t_{end} \leftarrow \bullet p_e;$ 
10   $top \leftarrow \text{new Region}();$ 
11   $RNG(t_{init}, top, t_{end});$ 
12   $RNG(t, r, t_e)$ 
13   $ip \leftarrow \text{new InitialPseudoState}(); cont(ip) \leftarrow r;$ 
14  if  $Event(t) = Constraint(t) = \varepsilon$  then
15     $s \leftarrow ip$ 
16  else
17     $s \leftarrow \text{new SimpleState}(); cont(s) \leftarrow r;$ 
18    new Transition  $(ip, \varepsilon, \varepsilon, s);$ 
19  while  $t \neq t_e$  do
20     $ev \leftarrow Event(t); const \leftarrow Constraint(t); beh \leftarrow Behavior(t);$ 
21    if  $ev = const = beh = \varepsilon \wedge |t \bullet| = 1$  then
22       $t \leftarrow t \bullet \bullet;$  continue;
23    if  $A(t) \in \{\$m \mid m \in \mathcal{M}_s\}$  then  $t \leftarrow t \bullet \bullet;$ 
24    if  $|t \bullet| \geq 2$  then
25       $s' \leftarrow \text{new CompositeState}(); cont(s') \leftarrow r;$ 
26      forall the  $p' \in t \bullet$  do
27         $r' \leftarrow \text{new Region}(); cont(r') \leftarrow s;$ 
28         $RNG(p' \bullet, r', FJ(t));$ 
29       $t \leftarrow FJ(t);$ 
30    else
31       $s' \leftarrow \text{new SimpleState}(); cont(s') \leftarrow r;$ 
32       $t \leftarrow t \bullet \bullet;$ 
33    new Transition  $(s, ev, const, beh, s');$ 
34     $s \leftarrow s';$ 
35   $fs \leftarrow \text{new FinalState}();$ 
36  new Transition  $(s, \varepsilon, \varepsilon, fs);$ 

```

---

Intuitively, the choreography realization problem is the problem to determine whether it is possible to synthesize a set of state machines which realize the choreography. In addition, it is desired to synthesize the state machines. The choreography realization problem is formally defined as follows.

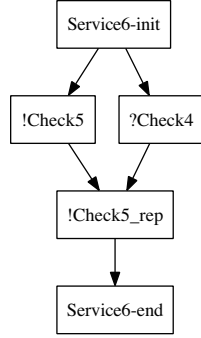


Fig. 5. CMMG

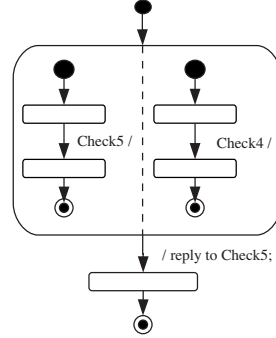


Fig. 6. Generated state machine

*Problem 1.* For a given set  $\mathcal{CD}$  of communication diagrams, is it possible to synthesize the set  $\mathcal{SM}$  of state machines which satisfy  $\mathfrak{C}(\mathcal{CD}) = \mathfrak{C}(\mathcal{SM})$ ? If possible, obtain the set of state machines.

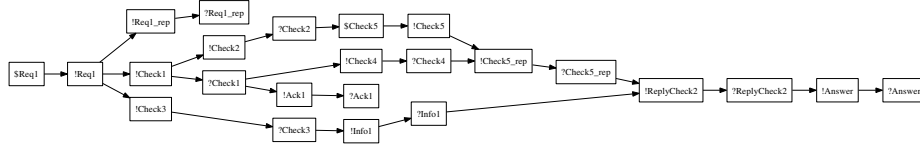
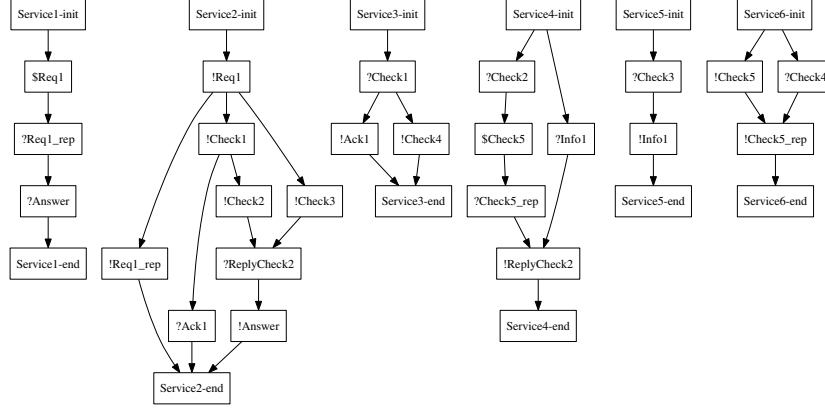
In the case of un-realizable choreography, it is desired to synthesize state machines which behave as close to the choreography as possible. It is called weakly realizable if there exist state machines which satisfy  $\mathfrak{C}(\mathcal{CD}) \supseteq \mathfrak{C}(\mathcal{SM})$ . For a weakly realizable choreography, obtain the set of state machines whose  $\mathfrak{C}(\mathcal{SM})$  is maximal.

In [2], sufficient realizability conditions for a class of collaboration diagrams have been shown. We suppose that given  $\mathcal{CD}$  is (weak) realizable hereafter and the set  $\mathcal{CD}$  contains only one communication diagram.

### 3.2 CSCB Method

The proposed CSCB method synthesizes state machines from a communication diagram as below. Due to space limitations the details of the algorithm are omitted.

1. Construct a dependency relation  $\Rightarrow_{cd}$  on the set of events.  
 For each object  $c$ , perform the following steps.
2. Derive a dependency relation  $\Rightarrow_{cd}^c$  from  $\Rightarrow_{cd}$ .
3. Construct an MMG from  $\Rightarrow_{cd}^c$ .
4. Cut T-T bridges from the MMG.
5. Separate fork and join transitions in the MMG.
6. Find one-to-one correspondence between  $T_{fork}$  and  $T_{join}$  in the MMG.
7. Perform Algorithm 1.


 Fig. 7.  $\Rightarrow_{cd}$ 

 Fig. 8.  $\Rightarrow_{cd}^c$  and MMGs for Service1, Service2, ... are shown from left to right.

**Construction of dependency relation  $\Rightarrow_{cd}$**  The dependency relation  $\Rightarrow_{cd} \subseteq \Sigma_{cd} \times \Sigma_{cd}$  on the set of events is given by the following expression:

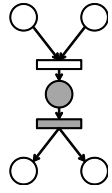
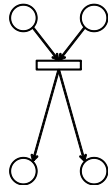
$$\begin{aligned} \Rightarrow_{cd} = & D_{cd} \cup \{(\$m, !m) \mid m \in \mathcal{M}_s^{cd}\} \cup \{(!m, ?m) \mid m \in \mathcal{M}_a^{cd}\} \\ & \cup \{(?m_1, !m_2) \mid m_1 \in \mathcal{M}_a^{cd}, m_2 \in \mathcal{M}_a^{cd}, (!m_1, !m_2) \in D_{cd}\} \\ & \cup \{(?m_1, \$m_2) \mid m_1 \in \mathcal{M}_a^{cd}, m_2 \in \mathcal{M}_s^{cd}, (!m_1, !m_2) \in D_{cd}\} \\ & \cup \{(!m_1, \$m_2) \mid m_1 \in \mathcal{M}_s^{cd}, m_2 \in \mathcal{M}_s^{cd}, (!m_1, !m_2) \in D_{cd}\} \end{aligned}$$

Figure 7 shows the dependency relation  $\Rightarrow_{cd}$  for the communication diagram shown in Fig. 1.

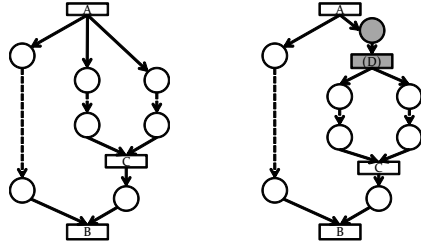
**Deriving  $\Rightarrow_{cd}^c$  and Construction of MMG** At first,  $\Rightarrow_{cd}$  is transitively reduced, then the dependency relation  $\Rightarrow_{cd}^c$  for each object  $c$  is derived. At this time, in order to satisfy the condition 4 of CMMG, for all synchronous message  $m \in \mathcal{M}_a$ , if there exists an event  $e \neq ?refer(m)$  such that  $(\$m, e) \in \Rightarrow_{cd}^c$ , then a relation  $(?refer(m), e)$  is added in  $\Rightarrow_{cd}^c$ .

The dependency relations  $\Rightarrow_{cd}^c$ , which are derived from the dependency relation  $\Rightarrow_{cd}$  shown in Fig. 7, are shown in Fig. 8. Here, since  $(\$Req1, ?Req1_{rep})$ ,





**Fig. 9.** Separating fork and join transition



**Fig. 10.** Finding one-to-one correspondence

$(\$Req1, ?Answer) \in \Rightarrow_{cd}^c$  for Service1, a relation  $(?Req1, ep, ?Answer)$  is added in  $\Rightarrow_{cd}^{Service1}$ .

MMGs are constructed by converting vertices into transitions, adding a place for each edge, and adding source and sink places in Fig. 8.

**Cutting T-T bridges** As shown in Lemma 1, since bridges are unnecessary in CMMGs, they are cut. In the example in Fig. 8,  $(!Check1 !Check2 ?ReplyCheck2)$  of Service2 is a bridge. After removing edges  $(!Check1, !Check2)$  and  $(!Check2, ?ReplyCheck2)$ , edge  $(Service2-init, !Check2)$  and  $(!Check2, Service2-end)$  are added. At that time, in order to avoid changing the behavior, the following firing conditions are added.

$$G(t) = \begin{cases} !Check1 & \text{if } A(t) = !Check2 \\ !Check2 & \text{if } A(t) = ?ReplyCheck2 \end{cases}$$

Cutting all bridges is not always necessary. Let  $U$  be a set of bridges and  $f : U \mapsto 2^U$  be a function such that  $f(u)$  is a set of bridges which will not be bridges by cutting bridge  $u$ . Then, the problem to finding the set of bridges results in the set cover problem[6].

**Separating fork and join transitions** If there exists fork and join transition, it is split into a fork transition and a join transition as shown in Fig. 9.

**Finding one-to-one correspondence** As shown in Fig. 10, dummy transition D is added in order to find one-to-one correspondence between fork and join transitions..

**Lemma 3.** *The MMG obtained by applying steps 1~6 of CSCB method is a CMMG.*

Figure 11 shows CMMGs obtained from MMG in Fig 8.

**Conversion into state machines** By performing Algorithm 1, state machines shown in Fig.s 12, 14, 13, 15, 16, and 6 are obtained.

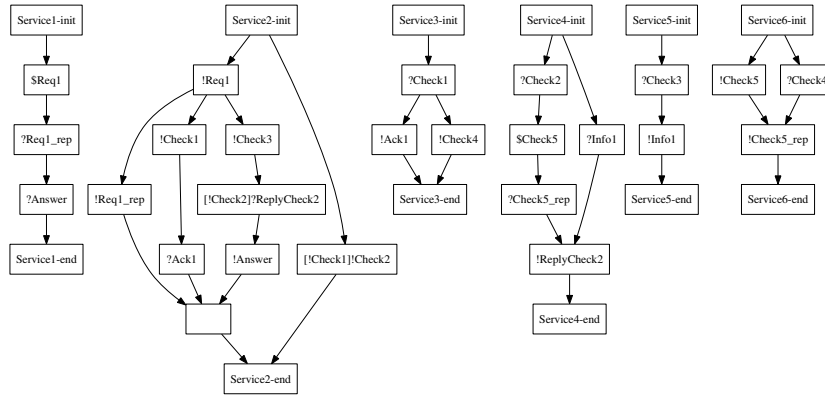


Fig. 11. CMMG of the example

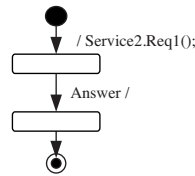


Fig. 12. State machine of Service1

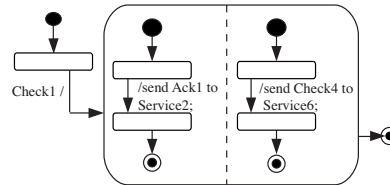


Fig. 13. State machine of Service3

### 3.3 Intelligibility Evaluation

Antonio et al. have experimentally evaluated the relationship between metrics and intelligibility of the state machines by measuring time to understand state machines[1]. According to the result, state machines are intelligible the smaller the following metrics: the number of simple states (NSS), the number of transitions (NT), and the number of guards (NG). In this section, the CSCB method

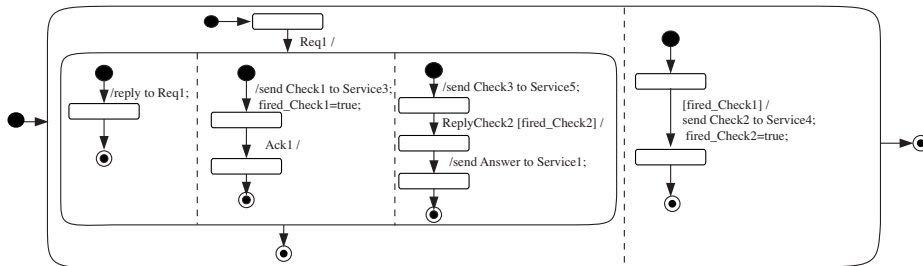


Fig. 14. State machine of Service2

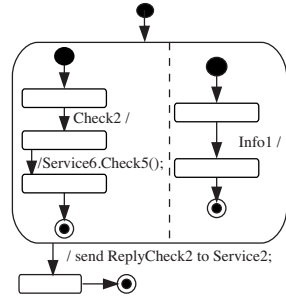


Fig. 15. State machine of Service4

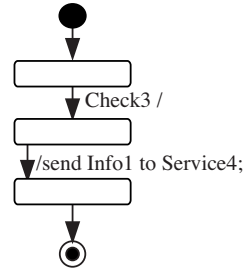


Fig. 16. State machine of Service5

Table 1. Evaluation result

	Method in [2]			Method in [8]			CSCB		
	NSS	NT	NG	NSS	NT	NG	NSS	NT	NG
Service1	5	7	0	5	9	0	2	3	0
Service2	31	59	0	31	59	0	9	17	2
Service3	5	7	0	5	9	0	5	9	0
Service4	8	11	0	7	11	0	6	10	0
Service5	3	4	0	3	4	0	3	4	0
Service6	5	7	0	5	9	0	5	9	0

is evaluated by comparing with Bultan’s method[2], the state space generation method[8] by using the above metric.

The Bultan’s method[2] synthesize flat state machines from the dependency relation. Suppose the number of events relating to object  $c$  to be  $|\Sigma^c|$ , then the number of states of the state machine becomes  $2^{|\Sigma^c|}$ . This method, however, generates plenty of unreachable state from the initial state. In this paper, state machines after removing these unreachable states are used.

The state space generation method[8] generates a state space for each MMG at first, and then converts the state spaces into state machines. The method, however, tries to find “independent sequences” in the state space, and tries to reduce the number of states by using composite states. Therefore, when no independent sequence is found, the same result with the Bultan’s method is obtained.

Table 1 shows values of the metrics of state machines which are obtained from the communication diagram in Fig. 1. Note that in the Bultan’s method and the state space generation method, the reply message to a synchronous call is considered as an asynchronous message which is independent with the synchronous call. On the other hand, in the proposed method, the state transition relating to a synchronous call terminates only when it receives the reply message. The proposed method adds relation at step2 so as each preparation event for message sending has only the receive event of the reply message as an immediate successor. Therefore, in the dependency relation for Service1, events ?Req1\_rep

and ?Answer are in concurrent for the Bultan's method and the state space generation method, but they are in sequential for the CSCB method.

As for Service2, since the state space generation method failed to find independent sequences, the state space are converted into a state machine as is. In contrast, the proposed method succeeds to significantly reduce the number of states by cutting bridges.

## 4 Conclusion

In this paper, we considered the approach to the choreography realization problem considering intelligibility of synthesized state machines. We proposed a method to synthesize state machines without generating state spaces from the choreography defined by single communication diagram. We evaluated the proposed method by using metrics about intelligibility of the generate state machines.

## References

1. Antonio Cruz-Lemus, J., Genero, M., Piattini, M.: Metrics for UML Statechart Diagrams. In: Genero, M., Piattini, M., Calero, C. (eds.) *Metrics for Software Conceptual Models*, pp. 237–272. Imperial College Press, London (2005)
2. Bultan, T., Fu, X.: Specification of realizable service conversations using collaboration diagrams. *Service Oriented Computing and Applications* 2(1), 27–39 (2008)
3. Desel, J., Yakovlev, A. (eds.): *Proceedings of 2nd Workshop on Application of Region Theory* (Jun 2011)
4. Esparza, J., Silva, M.: Circuits, Handles, Bridges and Nets. *Lecture Notes in Computer Science* 483, 209–242 (1991)
5. Harel, D., Kugler, H., Pnueli, A.: Synthesis revisited: generating statechart models from scenario-based requirements. In: Kreowski, H.J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) *Formal Methods in Software and Systems Modeling*, pp. 309–324. Springer (Jan 2005)
6. Jungnickel, D.: *Graphs, Networks and Algorithms*. Springer, 3rd edn. (2007)
7. Liang, H., Dingel, J., Diskin, Z.: A Comparative Survey of Scenario-based to State-based Model Synthesis Approaches. In: *2006 International workshop on Scenarios and state machines: models, algorithms, and tools*. pp. 5–11 (2006)
8. Miyamoto, T., Kurahata, H., Fujii, T., Hosokawa, R.: Synthesis of state machine diagrams from communication diagrams using petri nets. *Innovations in Systems and Software Engineering* 6, 39–46 (2010)
9. Murata, T.: Petri nets: Properties, analysis and applications. *Proc. IEEE* 77(4), 541–580 (Apr 1989)
10. OMG: Unified modeling language, <http://www.uml.org/>
11. Su, J., Bultan, T., Fu, X., Zhao, X.: Towards a theory of web service choreographies. In: *Proceedings of the 4th international conference on Web services and formal methods*. pp. 1–16 (2008)
12. Thomas, E.: *Service-Oriented Architecture*. Prentice Hall (2004)
13. Whittle, J., Jayaraman, P.K.: Synthesizing hierarchical state machines from expressive scenario descriptions. *ACM Transactions on Software Engineering and Methodology* 19(3), 1–45 (Jan 2010)

# SYNOPS - Generation of Partial Languages and Synthesis of Petri Nets <sup>\*</sup>

Robert Lorenz, Markus Huber, Christoph Etzel, and Dan Zecha

Department of Computer Science  
University of Augsburg, Germany  
`robert.lorenz@informatik.uni-augsburg.de`

**Abstract.** We present the command line tool SYNOPS. It allows the term-based construction of partial languages consisting of different kinds of causal structures representing runs of a concurrent system: labeled directed acyclic graphs (LDAGs), labeled partial orders (LPOs), labeled stratified directed acyclic graphs (LSDAGs) and labeled stratified order structures (LSOs). It implements region based algorithms for the synthesis of place/transition nets and general inhibitor nets from behavioural specifications given by such partial languages.

## 1 Introduction

Synthesis of Petri nets from behavioral descriptions has been a successful line of research since the 1990s. There is a rich body of nontrivial theoretical results and there are important applications in industry, in particular in hardware design [9,12], in control of manufacturing systems [25] and recently also in workflow design [23,22,1,10,4].

The synthesis problem is the problem to construct, for a given behavioral specification, a Petri net such that the behavior of this net coincides with the specified behavior (if such a net exists). There are many different methods which are presented in literature to solve this problem for different classes of Petri nets. They differ mainly in the Petri net class and the model for the behavioral specification considered. On the other hand, all these methods are based on one common theoretical concept, the notion of a *region* of the given behavioral specification.

In this paper, we present a new tool for the region based synthesis of Petri nets from behavioral specifications given by so called partial languages. A partial language is a set of finite causal structures, where a causal structure represents causal relationships between events of a finite run of a concurrent system. If the concurrent system is given by a Petri net, events represent transition occurrences. Expressible causal relationships are for example direct and indirect causal dependency, concurrency and synchronicity of events. The tool supports different kinds of causal structures, describing different semantics of different Petri net classes and having different expressiveness and interpretation:

---

<sup>\*</sup> Supported by the German Research Council, Project SYNOPS 2008 - 2012 [13]

- Labelled acyclic graphs (LDAG): LDAGs represent runs underlying process nets of place/transition-nets. They are used to specify all direct causal dependencies caused by token flow between transitions occurrences.
- Labelled partial orders (LPO): LPOs represent non-sequential runs of place/transition-nets. They are used to specify all "earlier than"-relations (which we call indirect causal dependencies) between transitions occurrences. Unrelated events are called concurrent. LPOs are transitively closed LDAGs.
- LDAGs extended by synchronicity (LSDAG): LSDAGs represent runs underlying process nets of general inhibitor nets according to the a-priori-semantics. They are DAGs extended by "not later than"-relations between events. A cycle of "not later than"-relations between events represents a synchronous step of events, i.e. it is possible to distinguish between concurrency and synchronicity.
- Labelled stratified order structures (LSO): LSOs represent non-sequential runs of general inhibitor nets according to the a-priori-semantics. LSOs are transitively closed LSDAGs.

This means, by a partial language the set of runs of a Petri net for different Petri net classes and different net semantics can be specified. It depends on the application area, which Petri net class and which kind of causal structures are appropriate or available for solving a concrete synthesis problem. In [10,4] case studies are presented illustrating the applicability and usefulness of synthesis from partial languages in practise.

Infinite behaviour can be represented by an infinite set of finite runs, i.e. an infinite partial language (where one finite run can be the prefix of another finite run).

The tool allows to construct finite partial languages (allowing to specify finite behaviour) of the mentioned types via command line using a term-based notation. This term based notation allows to compose runs from a set of basic runs by several composition operators (sequential and parallel composition and iteration). For the synthesis of nets the tool implements algorithms based on a technique using so called token flows developed in the project SYNOPS [13]. Up to now, only an algorithm for the synthesis of place/transition nets from finite sets of LPOs is supported.

The paper is organized as follows. In section 2 we briefly recall some basic mechanism of region-based synthesis. In section 3 we present some new technical developments for the synthesis of place/transition nets from finite sets of LPOs. In section 4 we describe the architecture and the components of the tool. In particular we describe how to specify finite sets of LPOs, LDAGs, LSDAGs and LSOs via command line. In section 5 we present some case studies involving the implemented algorithm for the synthesis of place/transition nets from finite sets of LPOs. In section 6 we briefly compare the tool to other synthesis tools. In section 7 we give a brief outlook onto current and further developments and in section 8 we give some hints for downloading and testing the tool.

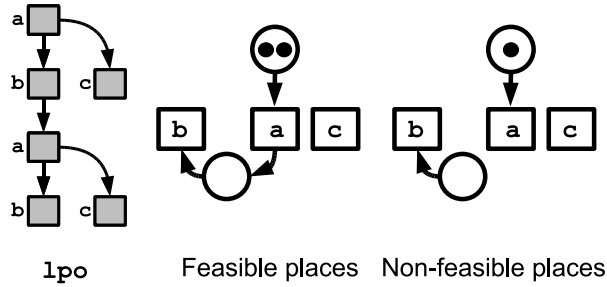


Fig. 1. An LPO (left part) and two feasible and two non-feasible places w.r.t this LPO.

## 2 Region based Synthesis

In this section, we denote the set of runs of a net  $N$  by  $L(N)$ .  $L(N)$  is called the language generated by  $N$ . We formally consider the following synthesis problem w.r.t. different Petri net classes and different types of partial languages:

**Given:** A prefix-closed partial language  $L$  over a finite alphabet of transition names  $T$ .

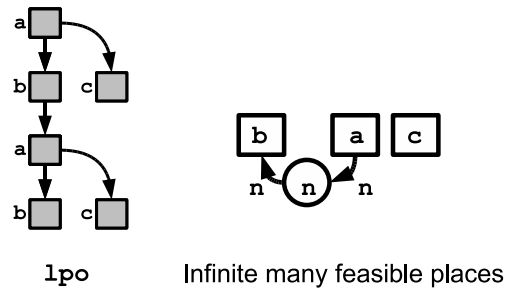
**Searched:** A Petri net  $N$  with set of transitions  $T$  and  $L(N) = L$ .

That means, we search for an exact solution of the problem. Such an exact solution may not exist, i.e. not each language  $L$  is a *net language*.

The classical idea of region-based synthesis is as follows: First consider the net  $N$  having an empty set of places and set of transitions  $T$ . This net generates each run in  $L$  (i.e.  $L \subseteq L(N)$ ), because there are no places restricting transition occurrences. But it generates much more runs. Since we are interested in an exact solution, we restrict  $L(N)$  by adding places.

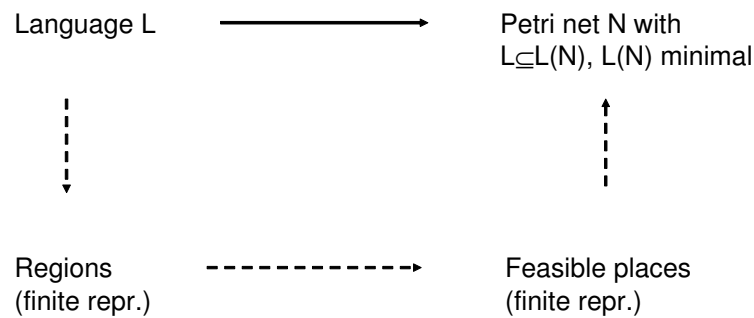
There are places  $p$ , which restrict the set of runs too much in the sense that  $L \setminus L(N) \neq \emptyset$ , if  $p$  is added to  $N$ . Such places are called *non-feasible (w.r.t.  $L$ )*. We only add so called *feasible* places  $p$  satisfying  $L \subseteq L(N)$ , if  $p$  is added to  $N$  (Figure 1). The idea of region-based synthesis is to add *all* feasible places to  $N$ . The resulting net  $N_{sat}$  is called the *saturated feasible net*. On the one hand,  $N_{sat}$  has by construction the following very nice property:  $L(N_{sat})$  is the smallest net language satisfying  $L \subseteq L(N_{sat})$ . This is clear, since  $L(N_{sat})$  could only be further restricted by adding non-feasible places. This property directly implies that there is an exact solution of the synthesis problem if and only if  $N_{sat}$  is such an exact solution. Moreover, if there is no exact solution,  $N_{sat}$  is the best approximation to such a solution "from above".

On the other hand, this result is only of theoretical value, since the set of feasible places is in general *infinite* (Figure 2). Therefore, for a practical solution, a finite subset of the set of all feasible places is defined, such that the net  $N_{fin}$  defined by this finite subset fulfills  $L(N_{fin}) = L(N_{sat})$ . Such a net  $N_{fin}$  is called



**Fig. 2.** The shown place is feasible w.r.t. the left LPO for each integer  $n \in \mathbb{N}$ .

*finite representation* of  $N_{sat}$ . In order to construct such a finite representation, in an intermediate step a feasible place is defined through a so called *region* of the given language  $L$ .

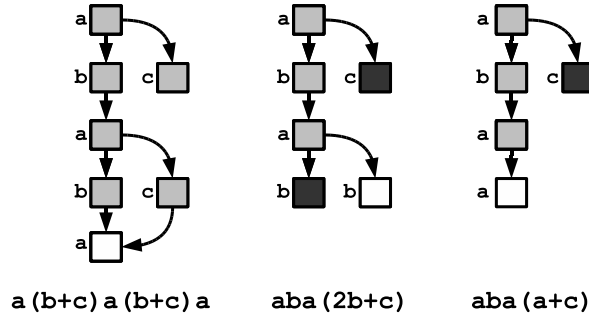


**Fig. 3.** The approach of region-based synthesis.

The described approach is common to all known region-based synthesis methods (see Figure 3) and can be applied to all kinds of partial languages. In particular, this approach can be applied to different notions of regions (of a partial language) and of finite representations  $N_{fin}$ . There are two types of definitions of regions and two types of definitions of finite representations, covering all known region-based synthesis methods [16].

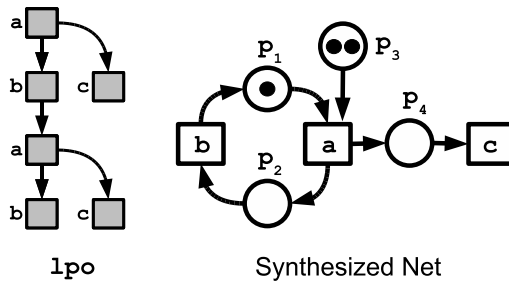
Experiments in the first phase of the project SYNOPS showed that the so called separation representation produces Petri nets which are simpler and more compact, especially having less places [2]. Moreover, it turned out that so called token flow regions can be computed more efficiently in the presence of much concurrency. Therefore, the first synthesis algorithm implemented in SYNOPS computes a place/transition net from a finite set of LPOs using the separation representation of the set of all token flow regions. Note that this variant is not yet implemented in other tools.





**Fig. 4.** Several wrong continuations of the LPO shown in the previous figures. A wrong continuation consists of a prefix (grey color) and a follower step (black) including an additional event (white) and represents one or more step sequences.

For computing the separation representation, first all so called wrong continuations of  $L$  are constructed. The set of wrong continuations represents the behaviour which is not specified. Briefly, a wrong continuation consists of a prefix of some specified run together with a follower step of transition occurrences extending a specified run by one additional event. Figure 4 shows examples of wrong continuations. For every wrong continuation, the synthesis algorithm tries to compute a place prohibiting the wrong continuation (for details on how to compute such a place we refer to [16]). The synthesized Petri net is an exact solution (does not have runs which are not specified) if and only if each wrong continuation can be prohibited by some place. Figure 5 shows the result of the synthesis algorithm. The wrong continuations shown in Figure 4 are forbidden by the places  $p_3$ ,  $p_2$  and  $p_1$  (from left to right).



**Fig. 5.** Synthesized net (right part) for the partial language only containing the LPO shown in the left part.

The synthesized Petri net depends on the considered order of wrong continuations, since places often prohibit more than one wrong continuation. It is advantageous to compute such places first, which prohibit much wrong contin-

uations. Therefore several new methods were implemented for constructing an appropriate order of wrong continuations. In the next Section 3 some technical details of the implemented synthesis algorithm are described.

### 3 Newly developed Techniques

In this section we briefly introduce wrong continuations formally and describe some newly developed ideas optimizing the synthesis procedure.

A *multiset* over a set  $T$  is a function  $m : T \rightarrow \mathbb{N}$ . A *step*  $T$  is a multiset over  $T$ . Addition  $+$  on multisets is defined by  $(m + m')(a) = m(a) + m'(a)$ . We write  $\sum_{a \in T} m(a)a$  to denote a multi-set  $m$ . The relation  $\leq$  between multiset is defined through  $m \leq m' \iff \forall a \in T(m(a) \leq m'(a))$ . An *LPO* over a set  $T$  is a tuple  $(V, <, l)$  where  $V$  is the finite set of events,  $< \subseteq V \times V$  is a partial order, and  $l : V \rightarrow T$  is a labelling function. For  $W \subseteq V$  we define the multiset  $l(W)(a) = |\{v \in W \mid l(v) = a\}|$ . An LPO  $(W, <, l)$  is a *prefix* of an LPO  $(V, <, l)$  if  $W \subseteq V$  and  $(w \in W) \wedge (v < w) \Rightarrow (v \in W)$ . A step sequence  $w = \alpha_1 \dots \alpha_n$  can be represented by an LPO, where each step  $\alpha_i$  corresponds to a set of pairwise unordered events and events from different steps are ordered according to the step sequence. A step sequence  $\sigma$  is a *step linearization* of an LPO  $(V, <, l)$ , if the partial order representing  $\sigma$  contains  $<$ . For example, the step sequences  $a(b+c)a(b+c)$ ,  $ab(a+c)(b+c)$  and  $aba(b+2c)$  are step linearizations of the LPO shown in Figure 5.

Throughout this section, let  $L$  be a prefix closed partial language of LPOs. We denote  $L^{step}$  the set of all *step-linearizations* of LPOs in  $L$ . Since  $lpo \in L$  is a run of a net  $N$  if and only if each step linearization of  $lpo$  is a step execution of  $N$ , wrong continuations are defined formally as step sequences which extend elements from  $L^{step}$  by one event as follows:

**Definition 1 (Wrong Continuation).** *Let  $\sigma = \alpha_1 \dots \alpha_{n-1} \alpha_n \in L^{step}$  and  $t \in T$  such that  $w_{\sigma,t} = \alpha_1 \dots \alpha_{n-1}(\alpha_n + t) \notin L^{step}$ , where  $\alpha_n$  is allowed to be the empty step. Then  $w_{\sigma,t}$  is called wrong continuation of  $L$ .*

*We call  $\alpha_1 \dots \alpha_{n-1}$  the prefix and  $\alpha_n + t$  the follower step of the wrong continuation.*

To prohibit a wrong continuation, one needs to find a feasible place  $p$  such that after occurrence of its prefix there are not enough tokens to fire its follower step. A prefix  $\alpha_1 \dots \alpha_{n-1}$  of a wrong continuation stepwise linearizes a prefix of an LPO in  $L$ . A follower step of such a LPO-prefix can be constructed by taking a subset of its direct successor in the LPO and add an event with a new label. This means, a wrong continuation can be represented on the level of LPOs, where wrong continuations having the same follower step and whose prefixes stepwise linearize the same LPO-prefix need not be distinguished. For example  $a(b+c)a(b+c)a$ ,  $aba(2b+c)$  and  $aba(a+c)$  are wrong continuations of the LPO shown in Figure 5. Their representations on the level of LPOs are shown in Figure 4 (from left to right).

Since the follower marking after the occurrence of a prefix of a wrong continuation only depends on the number of occurrences of each transition (but not on their ordering), the following statement holds:

**Proposition 1.** *Let  $w_{\sigma,t} = \alpha_1 \dots \alpha_{n-1}(\alpha_n + t)$  be a wrong continuation and  $\sigma' = \alpha'_1 \dots \alpha'_{m-1}\alpha_n \in L^{step}$  satisfying  $\alpha_1 + \dots + \alpha_{n-1} = \alpha'_1 + \dots + \alpha'_{m-1}$ . Then  $w_{\sigma,t}$  can be prohibited if and only if  $w_{\sigma',t}$  can be prohibited.*

That means in particular, for storing the set of all wrong continuations it is enough to construct all pairs  $(l(W), l(S))$ , where  $(W, <, l)$  is a prefix of some LPO in  $L$  and  $S$  is a subset of direct successors of  $(W, <, l)$  extended by an additional event. For example, the wrong continuation  $a(b+c)a(b+c)a$  is stored in the form  $(2a+2b+2c, a)$ . Moreover, the follower steps of wrong continuations with equivalent prefixes need to be merged.

We now define an order on the set of wrong continuations.

**Definition 2 (More restrictive wrong Continuation).** *A wrong continuation  $w_{\sigma,t}$  is more restrictive than a wrong continuation  $w_{\sigma',t'}$ , if the following holds: If  $w_{\sigma,t}$  is not a step execution of a place/transition net  $N$ , then  $w_{\sigma',t'}$  is not a step execution of  $N$ .*

If it is possible to forbid a wrong continuation, then automatically all less restrictive wrong continuations are forbidden, too. This means, if one considers more restrictive wrong continuations first, then less places are computed and runtime is faster.

If two wrong continuations have equivalent prefixes and the follower step of the first is included in the follower step of the second, then the first wrong continuation is more restrictive than the second one, since its follower step needs less tokens. For example  $a(b+c)a(2b)$  is more restrictive than  $a(b+c)a(2b+c)$  in this sense.

**Proposition 2.** *Let  $w_{\sigma,t} = \alpha_1 \dots \alpha_{n-1}(\alpha_n+t)$  and  $w_{\sigma',t'} = \alpha'_1 \dots \alpha'_{m-1}(\alpha'_m+t')$  be wrong continuations of  $L$  satisfying  $\alpha_1 + \dots + \alpha_{n-1} = \alpha'_1 + \dots + \alpha'_{m-1}$  and  $(\alpha_n + t) \leq (\alpha'_m + t')$ . Then  $w_{\sigma,t}$  is more restrictive than  $w_{\sigma',t'}$ .*

If the last step of a wrong continuation is sequentialized by several terminal steps of a second wrong continuation, then the second wrong continuation is more restrictive than the first one, since a step is not enabled, if a sequentialization of the step is not enabled in a marking. For example  $a(b+c)aa$  is more restrictive than  $a(b+c)(2a)$  in this sense.

**Proposition 3.** *Let  $w_{\sigma,t} = \alpha_1 \dots \alpha_{n-1}(\alpha_n+t)$  and  $w_{\sigma',t'} = \alpha'_1 \dots \alpha'_{m-1}(\alpha'_m+t')$  be wrong continuations of  $L$  satisfying  $\alpha_1 + \dots + \alpha_{n-1} + (\alpha_n + t) = \alpha'_1 + \dots + \alpha'_{m-1} + (\alpha'_m + t')$  and  $\alpha_1 + \dots + \alpha_{n-1} \geq \alpha'_1 + \dots + \alpha'_{m-1}$ . Then  $w_{\sigma,t}$  is more restrictive than  $w_{\sigma',t'}$ .*

According to these observations, wrong continuations are ordered in the following way: Wrong continuations with longer prefixes are considered first and if two wrong continuations have equal prefix, then the wrong continuation with the shorter follower step is considered first.

## 4 Architecture and Functionality

### 4.1 Overview

The SYNOPS tool is implemented strictly following advanced object oriented paradigms using a classical 3-tier-architecture:

- The client tier is realized as a command line interface (CLI). In the meanwhile we also provide a graphical user interface (GUI) which additionally visualizes Petri net synthesis results. The CLI (resp. GUI) and the middle tier are loosely coupled, such that an easy and fast change is possible.
- The middle tier (**SynCore**) encapsulates data types for the supported kinds of runs, sets of such runs and Petri nets and basic operations for creating, manipulating and destroying such objects. It can only be accessed via a facade (**SynShell**).
- Sets of runs and synthesized Petri nets are stored in text files. For Petri nets the PNML-standard is used, such that synthesis results can be visualized by many Petri net editors. For storing sets of runs we use a simple self-created text format which lists runs, events and edges.

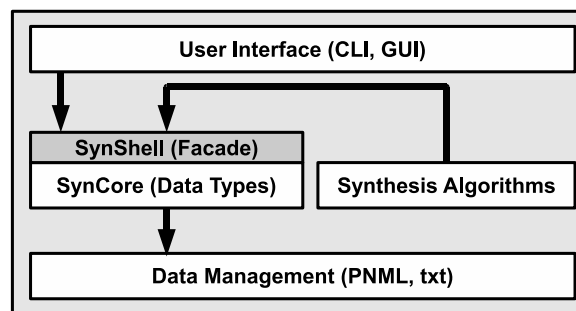


Fig. 6. Architecture of the system.

Figure 6 shows the architecture. Synthesis algorithms are connected with **SynCore** through a plug-in system, where each plug-in communicates with **SynCore** via the facade **SynShell**. **SynShell** implements interfaces supporting such a plug-in system.

### 4.2 The middle tier SynCore

A run consists of a finite set of events labelled by action names and a finite set of directed edges between events. A run can be represented through the four different causal structures previously mentioned.

The object of interest are sets of runs, since synthesis algorithms are operating on such sets. Every event has an ID which is unique within a run. Every run has

an ID which is unique within a set of runs. Sets have global unique IDs. This way, each object can be identified by a combination of IDs in the usual way. For example, the identifier `set1.lpo5.event3` represents the event with ID `event3` in the run with ID `lpo5` belonging to the set with ID `set1`.

There are several useful operations for manipulation of these data structures, for example operations testing consistency properties of runs specified by the user (such as cycle-freeness), operations computing the transitive closure of runs specified by the user, operations computing all prefixes of a run (based on a modified version of the algorithm of Warshall [24]) and operations computing the direct successors of a prefix of a run.

A Petri net consists of places, transitions and two kinds of edges between places and transitions (flow edges and inhibitor edges). Places have a unique ID, a name, a number of tokens and a maximum capacity of tokens (which can be infinity). Transitions have a unique ID and a name. Edges have a weight. This way several low level Petri net classes can be represented such as place/transition nets and inhibitor nets. Moreover, there are several restrictions available such as a bound of 1 for arc weights in order to represent elementary Petri nets. Such restrictions are realized by overwriting methods in specialized classes. This modular construction makes it easy to extend the framework by other net classes in future.

### 4.3 Synthesis algorithms

So far, there is only one synthesis algorithm implemented in the download version of the tool: The algorithm `syn-tf-sep` computes place/transition nets from finite sets of LPOs using the separation representation of the set of token flow regions.

### 4.4 Command line interface CLI

The CLI allows easy construction of long runs and sets of runs using a term-based notation. Currently, each command may only contain one operation. Complicated terms are constructed stepwise command by command.

A set (of runs) is opened by `'set ID'`. After opening a set, runs of the set can be specified. Runs can only be specified within a set. Finally, a set is closed by `'tes'`.

A run is opened by `'dag ID'` (for LDAGs), `'lpo ID'` (for LPOs), `'sdag ID'` (for LSDAGs) or `'lso ID'` (for LSOs). After opening a run, events and edges of the run can be specified. Events and edges can only be specified within a run. A run is closed by `'gad'`, `'opl'`, `'gads'` or `'osl'`. After closing a run, consistency of the user input is checked. Moreover, in case of LPOs and LSOs, the transitive closure is constructed (that means, it is not necessary to specify all transitive edges). Finally, all prefixes of the run are computed, preparing the synthesis computation.

An event is specified by `'event ID LABEL'`. An edge in a LDAG or an LPO between two events with IDs `e1` and `e2` is specified by `'et e1 e2'`. A "not later

than" edge is specified by 'nlt e1 e2'. Using these operations, simple runs can be constructed such as LPO lpo1 shown in Figure 7. Figure 8 shows the syntax for specifying lpo1.

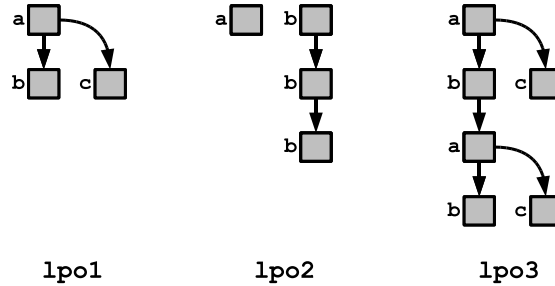


Fig. 7. Examples of LPOs.

---

```

1 set set1
2 lpo lpo1
3 event a a
4 event b b
5 event c c
6 et a b
7 et a c
8 opl
9 tes

```

---

Fig. 8. Syntax for specifying LPO lpo1.

There are several operations for combining existing runs:

- If **run1** and **run2** are runs, then by '**append ID run1 run2**' the sequential composition of **run1** and **run2** is stored in a run with ID **ID**. Sequential composition means, that from each event in **run1** to each event in **run2** an LPO-edge is drawn.
- If **run1** and **run2** are runs, then by '**compose ID run1 run2**' the parallel composition of **run1** and **run2** is stored in a run with ID **ID**. Parallel composition means, that between event in **run1** and **run2** there are no edges.
- If **run** is a run, then by '**iterate ID run N**' the **run** is **N** times sequentially composed with itself (iterated) and the result is stored in a run with ID **ID**.

Using these operations, longer runs can be constructed such as LPO lpo2 shown in Figure 7. Figure 9 shows the syntax for specifying lpo2.

It is also possible to apply sequential composition and iteration only partially w.r.t. a so called interface. An interface specifies explicitly, which events of the

---

```

1 set set1
2 lpo a
3 event a a
4 opl
5 lpo b
6 event b b
7 opl
8 iterate lpo1 b 3
9 compose lpo2 a lpo1
10 tes

```

---

**Fig. 9.** Syntax for specifying LPO lpo2.

previous run are in direct causal dependency with which events of the subsequent run. Only between such events an edge is drawn. An interface is specified as an option of the operations `append` and `iterate` of the form `'-interface EDGELIST'`, where an edge in `EDGELIST` between events with IDs `e1` and `e2` is specified by `'e1 < e2'` and edges are separated by a space. An interface can be used to specify LPO lpo3 shown in Figure 7. Figure 10 shows the syntax for specifying lpo3.

---

```

1 set set1
2 lpo a
3 event a a
4 opl
5 lpo b
6 event b b
7 opl
8 lpo c
9 event c c
10 opl
11 compose lpo1 b c
12 append lpo2 a lpo1
13 iterate lpo3 lpo2 2 -interface b<a
14 tes

```

---

**Fig. 10.** Syntax for specifying LPO lpo3.

It is possible to use a run specified in a certain set within another set by using its fully qualified ID. The same holds for events.

A run or a set of runs with ID `ID` can be stored by `'save ID FILE'` at location `FILE`. A run or set of runs stored at location `FILE` can be loaded by `'load FILE'`. A run can be loaded only within an opened set of runs.

At each stage of the input, by `'state all'` all objects constructed so far are printed in form of text. The notation used here is the same as in the case of saving objects. If the GUI is used, by `'plot ID'` the run with ID `ID` is visualized.

A synthesis algorithm `ALG` can be applied to a set of runs with ID `ID` by `'ALG ID [OPTIONS]'`. The synthesized Petri net is stored in PNML format, such that it can be visualized by Petri net editors. If the GUI is used instead of the CLI, the Petri net is also visualized. The user is noticed, if the synthesized net is an exact solution or not. If the algorithm uses the separation representation and the net is not an exact solution, all wrong continuations which could not be prohibited are returned as a tuple  $(prefix, step)$ , where *prefix* and *step* are given by their Parikh-vector (counting the number of transition occurrences in the prefix and in the follower step). As already mentioned, only the algorithm `syn-tf-sep` is available in the download version. This algorithm has no options, so far.

The program is exited by `'exit'`.

#### 4.5 Storing Petri nets and sets of runs

Synthesized Petri nets are stored in the Petri Net Markup Language (PNML) [20], version 2009. Runs are stored in a simple text format listing events and edges. As an example, Figure 11 shows the text file storing LPO `lpo3` from Figure 7.

---

```

1 lpo lpo3
2 event a a
3 event b b
4 event c c
5 event a_1 a
6 event b_1 b
7 event c_1 c
8 < a b
9 < a c
10 < a_1 b_1
11 < a_1 c_1
12 < b a_1
13 < a a_1
14 < a b_1
15 < a c_1
16 < b b_1
17 < b c_1
18 opl

```

---

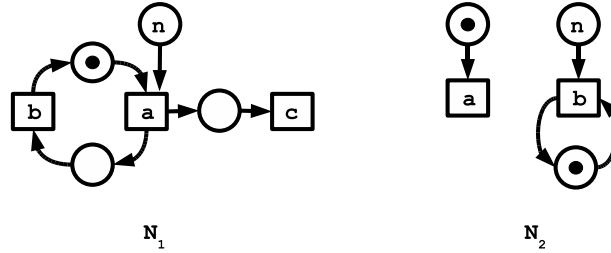
**Fig. 11.** Text format for storing runs.

## 5 Case Studies

We tested the algorithm `syn-tf-sep` w.r.t. two aspects: Performance, and compactness and simplicity of the synthesized net.



For testing compactness, we constructed several simple Petri nets with different initial markings having a finite set of runs, synthesized a net from this set of runs and compared the result with the initial net. Figure 12 shows two of the considered Petri nets with parametrized initial marking allowing different numbers of iterations. The complete set of considered sets of runs can be downloaded with the tool. In all cases the synthesized net and the initial net coincided.



**Fig. 12.** Petri nets having runs `lp01` ( $N_1$  with  $n = 1$ ), `lp02` ( $N_2$  with  $n = 3$ ) and `lp03` ( $N_1$  with  $n = 2$ ).

For testing performance we considered the following examples used in [2] for comparing performance and number of places of the synthesized net of two algorithms implemented in VIPTOOL (which also can be downloaded with the tool):

- LPOs for testing performance in presence of non-determinism (all LPOs are given in the form of step sequences):  $lp0_1 = b$ ,  $lp0_2 = a(a + b)$ ,  $lp0_3 = c(2a)$ ,  $lp0_4 = cb$  and  $lp0_5 = cc$ .
- LPOs for testing performance in presence of concurrency (the notion uses iteration of events of the form  $a^n$  and a parallel composition operator  $\parallel$ ):  $lp0_{6,n} = a^n \parallel b^n \parallel c^n$ .

Algorithm `basis` computes place/transition nets from finite sets of LPOs using the basis representation of the set of token flow regions. Algorithm `classic` computes place/transition nets from finite sets of LPOs using the separation representation of the set of transition regions of the step language corresponding to the set of LPOs. It turned out in [2] that algorithm `basis` performed much better in case of much concurrency and little nondeterminism (test series  $lp0_{6,n}$ ) and the other way round that algorithm `classic` performed much better in case of little concurrency and much nondeterminism (test series with combinations of  $lp0_1 - lp0_5$ ). Moreover, algorithm `classic` computed smaller nets.

Our experimental results show, that algorithm `syn-tf-sep` computes as small nets as algorithm `classic`, since it also uses the separation representation. Concerning performance on the other side, algorithm `syn-tf-sep` performs much better than algorithm `classic` and little worse than algorithm `basis` for the test series  $lp0_{6,n}$  (for example runtimes 13 ms for the LPO-set  $\{lp0_{6,2}\}$  and 132 ms for  $\{lp0_{6,3}\}$ ). Concerning the test series with combinations of  $lp0_1 - lp0_5$ ,

it performs as fast as algorithm `classic` (for example runtimes 5 ms for the LPO-set  $\{lpo_1, lpo_2\}$  and 6 ms for  $\{lpo_1, lpo_2, lpo_3, lpo_4, lpo_5\}$ ). Altogether, it is able to cope with nondeterminism and concurrency (since we ran the algorithms `basis` and `classic` several years ago on another system at another institut as `syn-tf-sep`, it does not make sense to compare absolute runtimes).

Currently we are working on a more efficient implementation concerning concurrency. In particular, it is possible to significantly reduce the number of prefixes, which need to be computed, by considering a more compact representation of iterations (which is currently implemented in the context of infinite iterations, see Section 7).

## 6 Comparison to other Tools

Up to our best knowledge, the only tool which also supports synthesis from partial languages is the graphical Petri net editor VIPTOOL [11]. In VIPTOOL many synthesis algorithms for languages of LPOs of the first phase 2008 - 2010 of the project SYNOPSIS are implemented [5,2,3,16,15]. VIPTOOL concentrates on business process modelling and has also verification and simulation capabilities. VIPTOOL currently is further developed and maintained at Distance University in Hagen (Germany), while the project and tool SYNOPSIS is developed at Augsburg University (Germany). In contrast to VIPTOOL, the SYNOPSIS tool supports more kinds of causal structures and Petri net classes and more general classes of infinite partial languages (see section 7). It only concentrates on synthesis capabilities and is text based. It mainly serves for rapid implementation and evaluation of newly developed term based representations of infinite partial languages and synthesis algorithms. For such term based representation and synthesis algorithms, which turn out to be stable, an integration into VIPTOOL is planned.

There is another tool-supported line of research considering transition systems instead of languages as behavioral specification. The tool [6] computes distributable bounded Petri nets from such specifications. In [8,7] tools are described which synthesize labelled Petri nets with non-unique transition names (here the techniques are different to the presented ones).

One application of synthesis algorithms is process mining. There is a big tool frame work called PROM [17] which integrates many different mining and analysis capabilities concerning process models and event logs. The mining tools are based on descriptions of the sequential behavior of systems (which cannot directly represent concurrency).

## 7 Outlook

Currently, an analogous algorithm is implemented for the synthesis of general inhibitor nets from finite sets of LSOs [21] based on results in [15]. For this, some new ideas concerning wrong continuation were developed (which are not presented here due to lack of space). Within another bachelor thesis, operations

for the specification of infinite sets of LPOs and a corresponding synthesis algorithm are implemented at the time of writing [18] based on results in [5,14]. In [19] a synthesis algorithm which computes place/transition nets from finite sets of LDAGs is described. This algorithm still needs some improvements which are currently implemented.

The presented set of operations is currently extended by the following operations allowing fast generation of large sets of runs: Alternative composition of runs, sequential composition, parallel composition and iteration of sets of runs, and standard operations on sets (of runs) like union, intersection, difference.

In order to increase usability, we plan to implement shortcuts for all operations (such a 'a<b' instead of 'et a b' or 'lpo1 a<b' instead of 'append lpo1 a b') and the possibility to use more than one operation in a command (such as 'lpo1 a<(b|c)' instead of the sequence 'compose lpo0 b c' and 'append lpo1 a lpo0').

Further steps are: Adapting the algorithms to restricted net classes such as elementary nets and workflow nets and to the use of additional information such as predefined places or undesired runs.

## 8 Download

The tool can be downloaded from the project webpage [13]. There are executable program files for 32 Bit and 64 Bit Windows systems, with and without GUI. On the webpage you also find the example sets of runs we used to evaluate the tool.

## References

1. R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Process Mining Based on Regions of Languages. In G. Alonso, P. Dadam, and M. Rosemann, editors, *BPM*, volume 4714 of *Lecture Notes in Computer Science*, pages 375–383. Springer, 2007.
2. R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Synthesis of Petri Nets from Finite Partial Languages. *Fundam. Inform.*, 88(4):437–468, 2008.
3. R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Synthesis of Petri Nets from Scenarios with Viptool. In K. M. van Hee and R. Valk, editors, *Petri Nets*, volume 5062 of *Lecture Notes in Computer Science*, pages 388–398. Springer, 2008.
4. R. Bergenthum, J. Desel, S. Mauser, and R. Lorenz. Construction of Process Models from Example Runs. *T. Petri Nets and Other Models of Concurrency*, 2:243–259, 2009.
5. R. Bergenthum, J. Desel, S. Mauser, and R. Lorenz. Synthesis of Petri Nets from Term Based Representations of Infinite Partial Languages. *Fundam. Inform.*, 95(1):187–217, 2009.
6. B. Caillaud. Synops-Homepage., 2002. <http://www.informatik.uni-augsburg.de/lehrstuehle/inf/projekte/synops/>.
7. J. Carmona, J. Cortadella, and M. Kishinevsky. Genet: A Tool for the Synthesis and Mining of Petri Nets. In *ACSD*, pages 181–185. IEEE Computer Society, 2009.

8. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrifly: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers. *IEICE Trans. of Informations and Systems*, E80-D(3):315–325, 1997.
9. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Hardware and Petri Nets: Application to Asynchronous Circuit Design. In *ICATPN 2000, LNCS 1825*, pages 1–15. Springer, 2000.
10. J. Desel. From Human Knowledge to Process Models. In R. Kaschek, C. Kop, C. Steinberger, and G. Fliedl, editors, *UNISCON*, volume 5 of *Lecture Notes in Business Information Processing*, pages 84–95. Springer, 2008.
11. J. Desel. VipTool-Homepage., 2010. <http://www.fernuni-hagen.de/se/viptool.html>.
12. M. B. Josephs and D. P. Furey. A Programming Approach to the Design of Asynchronous Logic Blocks. In *Concurrency and Hardware Design 2002, LNCS 2549*, pages 34–60. Springer, 2002.
13. R. Lorenz. Synops-Homepage., 2010. <http://www.informatik.uni-augsburg.de/lehrstuehle/inf/projekte/synops/>.
14. R. Lorenz, J. Desel, and G. Juhas. Models from scenarios. In *Proceedings of "Advanced Course on Petri Nets 2010", T. Petri Nets and Other Models of Concurrency*, Lecture Notes in Computer Science. Springer, to appear in 2012.
15. R. Lorenz, S. Mauser, and R. Bergenthum. Theory of Regions for the Synthesis of Inhibitor Nets from Scenarios. In J. Kleijn and A. Yakovlev, editors, *ICATPN*, volume 4546 of *Lecture Notes in Computer Science*, pages 342–361. Springer, 2007.
16. R. Lorenz, S. Mauser, and G. Juhás. How to synthesize Nets from Languages: a Survey. In S. G. Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, and R. R. Barton, editors, *Winter Simulation Conference*, pages 637–647. WSC, 2007.
17. Process Mining Group Eindhoven Technical University: ProM-Homepage. <http://www.promtools.org/prom5/>.
18. J. Robl. Synthese von Petrinetzen aus unendlichen Mengen beschrifteter partieller Ordnungen, to be finished july 2012. Bachelor thesis, Universität Augsburg.
19. K. Rüschenbaum. Synthese von Petrinetzen aus endlichen Mengen beschrifteter, gerichteter, azyklischer Graphen, 2011. Bachelor thesis, Universität Augsburg.
20. P. team. PNML.org: The Petri Net Markup Language home page, 8 2011. <http://www.pnml.org/>.
21. M. Urban. Synthese von Inhibitornetzen aus endlichen Mengen beschrifteter, geschichteter Ordnungen, to be finished march 2012. Bachelor thesis, Universität Augsburg.
22. W. M. P. van der Aalst and C. W. Günther. Finding Structure in Unstructured Processes: The Case for Process Mining. In *ACSD*, pages 3–12. IEEE Computer Society, 2007.
23. W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data Knowl. Eng.*, 47(2):237–267, 2003.
24. S. Warshall. A Theorem on Boolean Matrices. *Journal of the ACM* 9, (1):11–12, 1962.
25. M. Zhou and F. D. Cesare. *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems*. Kluwer, 1993.

# Modeling and Simulation-Based Design Using Object-Oriented Petri Nets: A Case Study

Radek Kočí and Vladimír Janoušek

Faculty of Information Technology, Brno University of Technology,  
Bozetechova 2, 612 66 Brno, Czech Republic  
{koci, janousek}@fit.vutbr.cz

**Abstract.** The aim of the paper is to show basic elements of a system design methodology which uses Object oriented Petri nets. The methodology features conformity with UML and uses simulation as a means to verify the models in all system development phases. Simulation also helps in making decisions about structural and behavioral specification of the system. The paper will demonstrate layered modeling technique based on Object oriented Petri nets.

## 1 Introduction

Modeling and Simulation-Based Design (MSBD) of systems denotes a set of techniques and tools intended for the software system development which is based on formal models, model continuity, and simulation techniques. Its goal is to increase efficiency and reliability of development processes including the software system deployment. The key activities in the system development are specification, testing, validation, and analysis (e.g., of performance, throughput, etc.). Most of the methodologies use models for system specification, i.e., for defining the structure and behavior of developed system. There are different kinds of models, from models of low-level formal basis to pure formal models. Each kind has its advantages and disadvantages. Less formal models (e.g., UML) allows to quickly describe basic system concepts, in the other hand, they do not allows to check the system correctness or validity by means of testing or formal methods—the system has to be implemented before its testing. The more advanced approaches (e.g., Executable UML and Model Driven Architecture [15]) allow to simulate models, i.e., to provide simulation testing. The pure formal models (e.g., Petri Nets, calculus, etc.) allows to use formal or simulation approaches to complete the testing and analysis activities.

The paper aims at system specification using a formalism of Object Oriented Petri Nets [2, 3] (OOPN). The idea of merging Petri nets and objects has been found and elaborated in the 1990's independently by several researchers. The approach closest to our work is the system Renew [11] and associated formalism of Nets-in-Nets [16, 14, 1]. Renew supports modeling of systems using layered Petri Nets and Java language. Similarly to the system Renew, the proposed approach fully supports an integration of formal objects described by Petri Nets

and other objects (e.g., it allows to reference and communicate with Smalltalk objects and Petri Net objects uniformly). This feature eases interfacing objects with surrounding world and consequently facilitates hardware-in-the-loop simulation. The idea of using models in all development stages, in conjunction with hardware-in-the-loop simulation, was working up in several projects and is supported by several tools, e.g., the MetaEdit System [13] or Simulink [12]. MetaEdit supports Domain Specific Modeling which allows to generate code from high-level models defined for the domain-specific language. Simulink is aimed at design control systems and hardware architectures. It allows for hardware-in-the-loop simulation for testing designed models in a real environment. Contrary to the works cited above, the proposed approach uses the same model in all development phases including deployment (or final implementation). The formalism of OOPN, in conjunction with the design and simulation framework PNtalk [5], can be directly interpreted and, consequently, integrated into the target system [6].

This paper summarizes methodical approach to system design using Object Oriented Petri Nets. It is a result of previous activities on the field of system design techniques [4, 7], modeling techniques [10], simulation testing and analysis [8], and combination of the OOPN formalism and the UML language [9]. The paper is organized as follows. First, we introduce the used formalism of Object Oriented Petri Nets in section two. The section three describes the basis of design methodology resulted from principles of Modeling and Simulation Based Design. The next three sections deal with particular parts of design methodology including the demonstration on the simple case study. We conclude by summarizing of results and definition of future works.

## 2 Modeling Formalisms

### 2.1 Object Oriented Petri Nets

An *object-oriented Petri net* (OOPN) is a triple  $(\Sigma, c_0, oid_0)$  where  $\Sigma$  is a system of classes,  $c_0$  an initial class, and  $oid_0$  the name of an initial object from  $c_0$ . A *class* is specified by an object, a set of method nets, a set of synchronous ports and negative predicates. Object nets describe possible autonomous activities of objects, while method nets describe reactions of objects to messages sent to them from the outside. Each net is described by means of high-level Petri nets.

*Object nets* consist of places and transitions. Every place has its initial marking. Every transition has conditions (i.e., inscribed testing arcs), preconditions (i.e., inscribed input arcs), a guard, an action, and postconditions (i.e., inscribed output arcs). *Method nets* are similar to object nets but, in addition, each of them has a set of parameter places and a return place. Method nets can access places of the appropriate object nets in order to allow running methods to modify states of objects, which they are running in.

Object nets can also contain special kinds of transitions—synchronous ports and negative predicates. *Synchronous ports* are special transitions, which cannot

fire alone but only dynamically fused to some other transitions, which activate them from their guards via message sending. Every synchronous port embodies a set of conditions, preconditions, and postconditions over places of the appropriate object net, and further a guard, and a set of parameters. Thus, synchronous ports combine concepts of *transitions* (they have to satisfy preconditions and guards; if the synchronous port is fired, the postconditions are performed) and *method nets* (they have to be called from a guard of another transition).

A synchronous port can be activated via a message sent from a guard of some transition. During transition fireability testing, the searching for suitable variables binding uses backtracking mechanism which takes in account also guard expressions which can consequently test synchronous ports fireability. A synchronous port can be activated with either bound, or unbound formal parameters. In the second case, activation of the synchronous port can bind the formal parameter to some value which can be further used by the calling transition.

*Negative predicates* are special variants of synchronous ports. Its semantics is inverted—the calling transition is fireable if the negative predicate is not fireable.

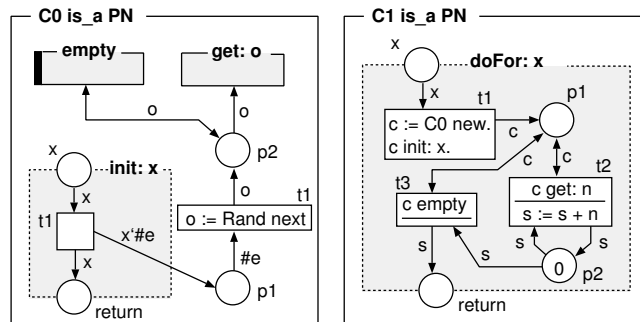


Fig. 1. An OOPN example.

An example illustrating the important elements of the OOPN formalism is shown in Figure 1. There are depicted two classes C0 and C1. The object net of the class C0 consists of places p1 and p2 and one transition t1. The object net of the class C1 is empty. The class C0 has a method `init:`, a synchronous port `get:`, and a negative predicate `empty`. The class C1 has a method `doFor:`. An invocation of the method `doFor:` leads to random generation of x numbers and a return of their sum.

Let us investigate what happens if we call the method `doFor:` with a value 3 on the instance of a class C1 (the instance will be denoted by `obj1`). First, the transition `t1` is fired with following actions: the instance of the class C0 is created (the instance will be denoted by `obj0`) and the symbol `#e` is put to the place `p1` of the object net `obj0` three times (see the method net `init:`). Now, the object net of `obj0` generates three random numbers (the transition `t1`) and puts

them into the place `p2`. Second, the transition `t2` of the object net `obj1` tests if there is any random number in the object net `obj0`—then the synchronous port `get:` is firable. If the transition `t2` fires, the synchronous port `get:` fires too. Since the variable `n` is unbound, the calling binds any random number from the place `p2` of the object net `obj0` to the variable `n`. The transition `t2` of the object net `obj1` then adds this value to the sum (the variable `s`). Third, the transition `t3` of the object net `obj1` tests if there is no random number in the object net `obj0`—then the negative predicate `empty` is firable. If the transition `t3` fires, it places the sum (the variable `s`) to the return place as a method result.

### 3 Modeling and Simulation Based Design Technique

Modeling and Simulation Based Design (MSBD) is a technique of system design where the system is specified in a form of an executable model which can be verified using simulation experiments. During the development, the model is incrementally refined and each development step is tested and verified. There are two phases which rotate until the system development is finished: Modeling phase and simulation phase. We will especially take into account the techniques of the modeling. In the following sections, we will demonstrate their basic concepts in a small case study.

#### 3.1 Modeling Technique

The modeling technique is focused on the technique of system description, i.e., how the models are created. The technique stems from the classic approach of class identification and definition and extends it to the new features. The design process comprises:

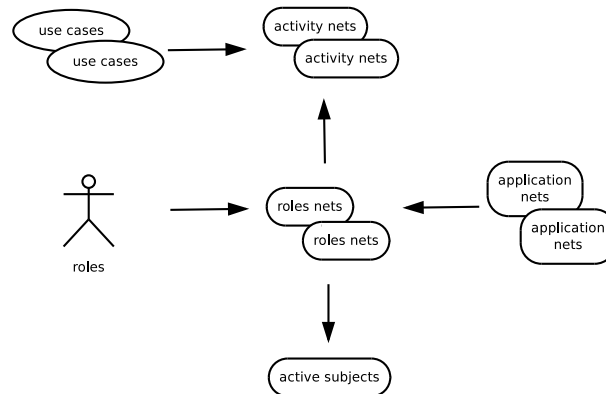
- the identification of use cases of the system,
- the specification of roles and active subjects,
- the specification of activity nets—it is similar to workflow modeling,
- the specification of application nets.

The models are layered hierarchically as shown in Figure 2. Each arrow shows what layers encapsulated another ones. There is a special relationship between use cases in UML and activity nets, and roles in UML and roles nets. These mentioned nets represent appropriate use cases and roles in the system (see the sections 4 and 5). Each role net encapsulate one active subject (see the section 4.1). Each role encapsulate activity nets (see the section 5.3). Moreover, each role can encapsulate another role, and the active subject can also encapsulate another active subject. It allows to get a new view to the role (or active subject) based on the existing one.

The way of system usage is defined by application nets which encapsulate roles nets. Each role has its own set of allowed activities which offers to application nets. The application net can then instantiate and use this activity (see



the section 6). The execution of layered nets are synchronized by means of synchronous ports. The nested nets define synchronous port for synchronization of executions and the net at higher layer is controlled by calling these ports. This principle will be demonstrated at the appropriate places in following parts.



**Fig. 2.** The layered architecture: an overview.

The design activities are performed in a sequence *use cases–roles and subjects–activity nets–application nets*. But, the system is developed incrementally, in each step, we model selected parts, make decision what part is to be modeled at what layer, and, if necessary, we decide what nets should be changed. Thus, the developer has to go back to designed layer and modify them. The decisions are supported by simulation techniques, i.e., designed models are simulated, the statistic data can be collected, the condition testing can be performed, etc.

### 3.2 Case Study

We will demonstrate the key features of presented modeling technique on the simple case study. It concerns the reservation system whereas the structure and workflow is only taken into account. The real data associated with the real system will not be modeled.

As we mentioned in the brief description of design process, the process starts with use cases identification. The use case diagram is one of the key diagrams in system specification defined by the Unified Modeling Language (UML). The use case defines a functionality of the system, it is usually complex set of functions to achieve a particular behavior. There is a set of actors who can interact with the use case. The use cases of our case study is shown in Figure 3—there is one actor and two use cases. It represents a system allowing users (an actor *User*) to sign up to the system (a use case *Login*) and to edit its reservations (a use case *Edit Reservation*). In UML, the use case is supplemented with its specification

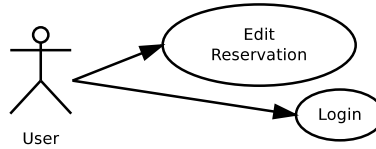


Fig. 3. The use case diagram.

(a text description or other models from UML). Presented technique supposes that the use case is specified by OOPN as will be demonstrated in the section 5.

## 4 Roles and Active Subjects

### 4.1 Specification of Active Subjects

Each actor from use case diagrams has its own subset of use cases it can participate with. However, different actors usually have the same basis, e.g., the user represented by its name can act as different actors (administrator, customer, manager, etc.) having different set of behavior (use cases). So we can identify *active subjects* (e.g., person) and their *roles* (e.g., customer, manager). Although we call it an active subject, it does not perform any autonomous action. It only models current state and possible actions shared by all its roles.

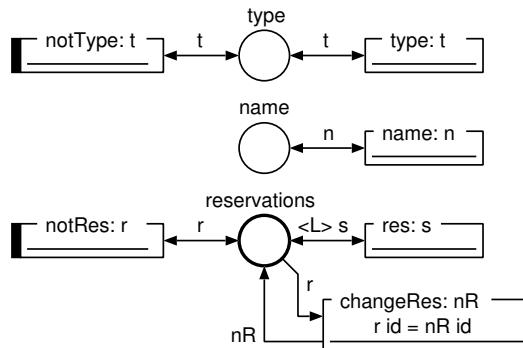


Fig. 4. The active subject Member (modeled as OOPN class).

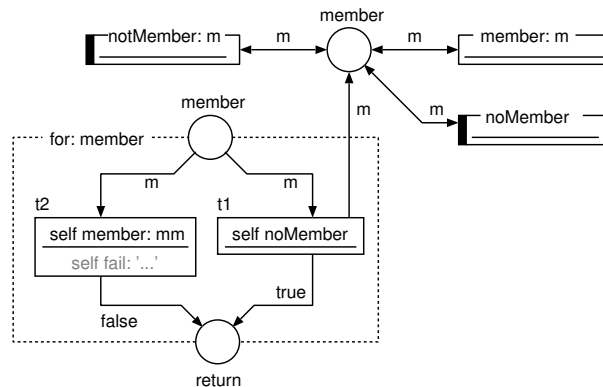
The example of an active subject is shown in Figure 4. It depicts the object net of OOPN class `Member` storing information about person's *name*, *type*, and set of *reservations*. The *type* is a set of role identifications the member can act as. Attributes are stored in places (*name*, *type*, and *reservations*) and are

accessible by means of synchronous ports. Ports including negative predicates can also serve for testing whether some value of the attribute is set or not.

In real system, the reservations should be stored in some database system, but it is possible to use places for the same purpose for a relative small number of records. But, in this case, there is a problem how to model the iteration of the place content effectively, e.g., if we want to show list of records. The arc between the place `reservations` and the synchronous port `res:` is denoted by a symbol `<L>`. It means that the variable `s` is bound to whole content of the place and this content is accessible as a list (the list is assigned to the variable `s`). Then it is possible to use conventional approach to this list. To change a reservation, the synchronous port `changeRes:` is defined. Each reservation has its unique identification accessible via a call `id`. The event is fireable, if there is the reservation (`r`) in the place `reservations` with the same identification as the given one (`nR`). Then the old reservation is replaced by its new variant.

### 4.2 Specification of Roles

The member can act in different roles—for our needs we will describe only one role called *User*. The role is modeled as an object net of OOPN class `User`. The example is shown in Figure 5. The role should know about an active subject this role is intended for. In our example, this information is stored in the place `member`.



**Fig. 5.** The role `User` (modeled as OOPN class).

The net `User` defines two testing negative predicates `notMember:` (it is true if the role does not represent given member `m`) and `noMember` (it is true if there is no represented member) and one synchronous port `member:.`. The synchronous port can server for testing (if the role represents given member `m`) or for attribute collection (the example will be shown in chapter 5).

The attributes should be initialized by method nets. Figure 5 shows one net as an example. The method net `for:` initializes the attribute `member` and tests if the role was not initialized yet. If the role is not initialized, the transition `t1` is fireable (the negative predicates `noMember` is true). If the role is already initialized, the transition `t2` is fireable (the synchronous port `member:` is true for a member `m`). In this case, the method net returns `false` and can generate an exception (the calling of `self fail: '...'`); it is useful for testing.

### 4.3 System as a Special Role

The system usually needs means for persistence, accessing shared objects etc. For this purposes, we introduce a special role net called `Application` (see Figure 6). It allows for getting members (the external event `member:`), storing logged users (the external event `newUser:`), etc.

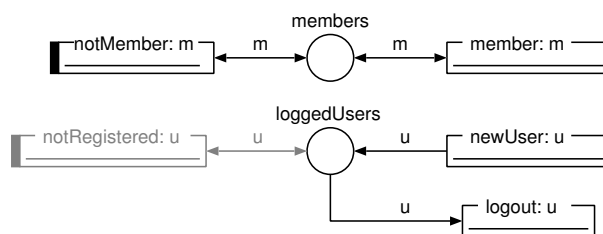


Fig. 6. The role `Application` (modeled as OOPN class).

## 5 Activity nets

### 5.1 Specification of Activity nets

An activity net describes a use case of the system. Each use case is modeled as an OOPN class. Its object net contains transitions, synchronous ports, and places. Since a transition is conditioned only by its input places, it models *internal event* in the activity. On the other hand, a synchronous port is intended for activation from the outside of the activity object. Therefore the synchronous port represent *external event*. Each place in the activity net represents the state of the activity. The state can be tested by means of synchronous ports or negative predicates.

Let us demonstrate this principles in our small example. The example defines `Member`'s role `User` who participates in the use case `editReservation`. The activity net which corresponds to the use case is modeled by the object net of OOPN class `EditReservation` which is shown in Figure 7. The activity net has to know about the role which is associated with the activity. The role is stored in the

place `user`. This attribute should be initialized by a method net—because the concept is similar to the net intended for the same purpose (e.g., the net `for:` in the role `User`), the implementation is not shown here.

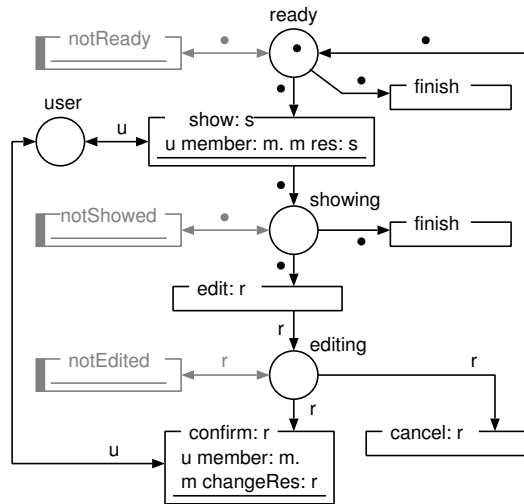


Fig. 7. The activity `EditReservation` (modeled as OOPN class).

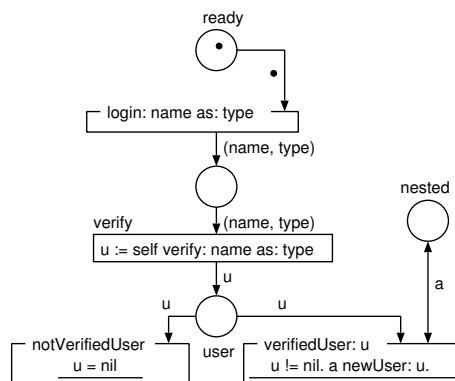
The basic workflow consists of following events: list of reservations representation (see the external event `show:`), one reservation editing (the external event `edit:`), and confirmation (the external event `confirm:`) or to cancellation (the external event `cancel:`) of changes. The net defines three places representing three states of the activity (`ready`, `showing`, and `editing`). The activity can be finished from states `ready` and `showing` by external event `finish`.

It is possible to add synchronous ports for testing activity states. There are defined events for testing whether the activity is not in the defined state, modeled as negative predicates `notReady`, `notShowned`, and `notEdited`. This predicates are true (fireable) only if the state is not satisfied.

## 5.2 Specification of Actions

The activity usually needs to define some actions. Actions are associated either with internal events or with external events. In the case of internal events, actions are defined inside the transitions (in the action part) or in subnets—then the net is modeled as a method of the appropriate OOPN class and is called from the internal event. In the case of external event, actions are modeled by calling another external events (synchronous ports) from the event’s guard.

Figure 7 shows the second approach. For example, the external event `show:` detects the member which is represented by the role `User` (calling the synchronous port `u member: m`—because the variable `m` is unbound, the object net `Member` stored in the place `member` in the object net `User` is bound to the variable `m`). Then the synchronous port `m res: s` is called and the variable `s` then the bound to a list of reservations. The transition, which calls this event, can then use this list (it is shown in the chapter 6).



**Fig. 8.** The activity Login.

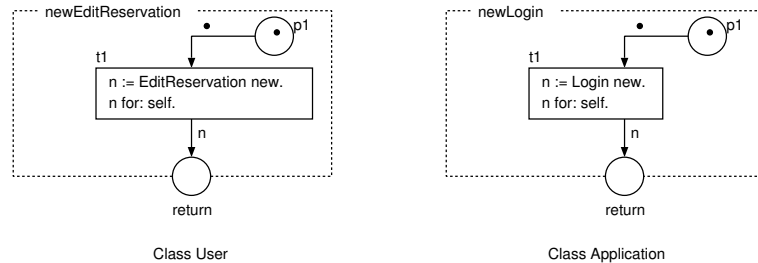
Figure 8 shows the activity net *Login* corresponding with the use case *Login*. The basic workflow consists of following events: user’s data getting (the external event `login:as:`), the data verification (the internal event `verify`), and testing if the desired role has been created (the external event `verifiedUser:`) or not (the external event `notVerifiedUser`). The action of user verification is associated with the internal event modeled as the sub-net `verify:as:` called from the internal event `verify`. The sub-net is not shown because its implementation is not important for this paper.

There are also actions associated with the external event. If the user is verified and its role is detected (the external event `verifiedUser:`), one action is performed—adding the role into the special role `Application` (`a newUser: u`); see the section 4.3.

### 5.3 Instantiation of Activity Nets

The activity nets have to be instantiated to serve for their purpose. Because each activity is usually allowed for only specified roles, it is just a role which can instantiate an activity. The example for the role `User` is shown in Figure 9 on the left. The role allows for reservation editing so that the role defines the method net `newEditReservation` which creates a new activity net for reservation edit.

Each activity has to know for which role it is created—it is set by the message `for::`.



**Fig. 9.** The methods creating instances of activities.

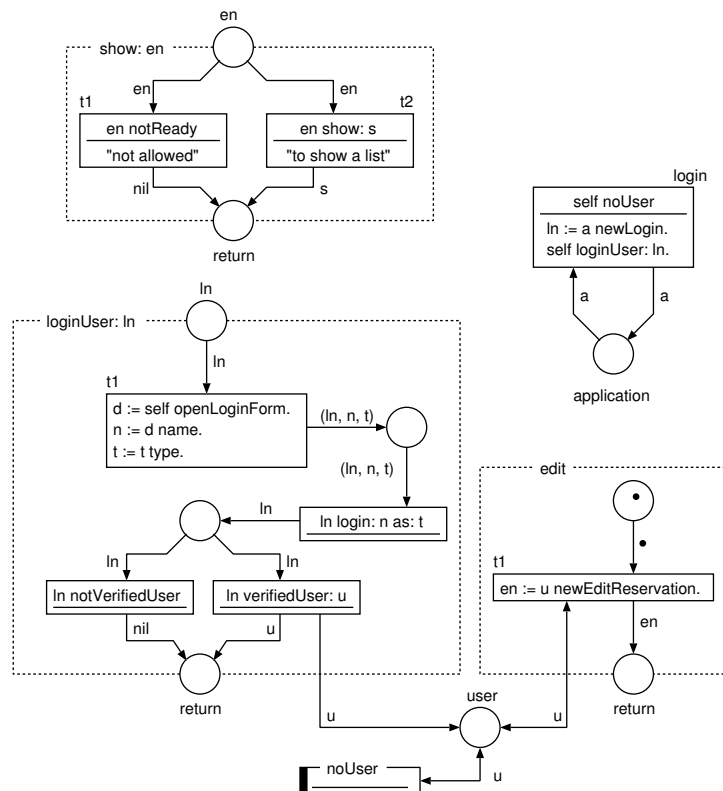
There is a special kind of activity nets associated with the system instead of the role. These activities are instantiated from special roles as they were introduced in the section 4.3. As an example we can take an activity of *user login* associated with the role **Application**. Figure 9 on the right shows the instantiation of the activity net **Login** from the role **Application**; the principle is same as in ordinary roles.

## 6 Application Nets

Application nets model a presentation layer of the system. We can also imagine it as an interface to the system. The application net uses roles to generate a list of permitted activities and to instantiate the concrete activity net. The particular behavior of application nets is then controlled by the appropriate activity net. The application nets execution is synchronized by means of external events (synchronous ports) of control (activity) nets.

The presented case study needs some kind of user interface—the part of application net **UserInterface** is shown in Figure 10. The application net **UserInterface** represents an interface to behavior of one role of **User**—how the user can work with the system. The net knows the user (i.e., the role **User**) and the special role **Application**.

Let us investigate the modeled functionality of the application net. First, the transition **login** tests if the user is logged to the system (see the negative predicate **noUser**). If there is no logged user (no role **User** is stored in the place **user**, the activity net **Login** is created (via the special role **Application**) and then the method net **loginUser::** is called. Its execution is controlled by the activity net **Login** using synchronous ports and negative predicates. If the login action is successful, the created role **User** is placed into place **user**. If the login action failed, the procedure is repeated, i.e., the transition **login** is fired again.



**Fig. 10.** UserInterface. The topmost level of the application modeled as OON class.

The procedure can be performed at most one at the same time. It is assured by the precondition of the transition `login` to the place `application`.

Second, if the logged user perform an action *edit reservation*, the activity net `EditReservation` is created (see the transition `t1` in the method `net edit`). The activity net then serves as a control mechanism watching if some operation is allowed or not. For example, the method `net show`: makes decision if it is possible to show the reservation list (the activity net is in appropriate state) or not. The method net has `show`: one parameter `en` of the activity net which is used for control.

The model of application net uses rather asynchronous processing of events. For instance, the methods `edit` and `show` represent fragments of a behavior controlled by the activity net `EditReservation`. Each fragment is called as a reaction to the event. For example, there can be generated a graphic user interface offering a button to start reservation editing—if the actor press this button, it generates an event and the method net `edit` is called and the instance of activity net `EditReservation` is created. Then, the first event of activity is to



show a list of reservation—as a reaction, the method `net show:` is called. Its execution checks the activity state and collects a list of reservations (calling the synchronous port `en show: s` in the transition `t2`). Then the list can be displayed. The details of graphics user interface and its cooperation with application nets are not described here.

## 7 Conclusion

The paper dealt with Modeling and Simulation Based Design using the formalism of Object Oriented Petri Nets. It presented the key ideas of a modeling technique which is based on layered approach. The presented approach is a part of the development methodology, which allows to use formal models in all phases of system development including as basic design, analysis and also programming means with a vision to allow to combine simulated and real components and to deploy models as the target system with no code generation.

So far, the models are interpreted in deployed application and are connected to other software components. The advantage is a possibility to monitor, to profile, to test, and to debug application at the model level with no needs to model transformations. The disadvantage is a possible inefficiency of the model interpretation. Therefore we plan to investigate an approach allowing to transform models into low-level models. Nevertheless, this transformation have to fulfil a condition of having a view to the system at the model level. It means, that it has to be possible to get a state from low-level models and to impose a new state to the low-level model.

**Acknowledgment.** This work has been partially supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), by BUT FIT grant FIT-11-1, and by the Ministry of Education, Youth and Sports under the contract MSM 0021630528.

## References

1. L. Cabac, M. Duvigneau, D. Moldt, and H. Rölke. Modeling dynamic architectures using nets-within-nets. In *Applications and Theory of Petri Nets 2005. 26th International Conference, ICATPN 2005, Miami, USA*, pages 148–167, 2005.
2. M. Češka, V. Janoušek, and T. Vojnar. *PNTalk — a Computerized Tool for Object Oriented Petri Nets Modelling*, volume 1333 of *Lecture Notes in Computer Science*, pages 591–610. Springer Verlag, 1997.
3. V. Janoušek and R. Kočí. PNTalk Project: Current Research Direction. In *Simulation Almanac 2005*. FEL ČVUT, Praha, CZ, 2005.
4. R. Kočí and V. Janoušek. System Design with Object Oriented Petri Nets Formalism. In *The Third International Conference on Software Engineering Advances Proceedings ICSEA 2008*, pages 421–426. IEEE Computer Society, 2008.
5. R. Kočí and V. Janoušek. On the Dynamic Features of PNTalk. In *International Workshop on Petri Nets and Software Engineering 2009*, pages 189–206. University of Pierre and Marie Curie, 2009.

6. R. Kočí and V. Janoušek. *Simulation Based Design of Control Systems Using DEVS and Petri Nets*, volume 5717 of *Lecture Notes in Computer Science*, pages 849–856. Springer Verlag, 2009.
7. R. Kočí and V. Janoušek. Towards Simulation-Based Design of the Software Systems. In *The Fourth International Conference on Software Engineering Advances – ICSEA'09*, pages 452–457. IEEE Computer Society, 2009.
8. R. Kočí and V. Janoušek. OOPN and DEVS Formalisms for System Specification and Analysis. In *The Fifth International Conference on Software Engineering Advances*, pages 305–310. IEEE Computer Society, 2010.
9. R. Kočí and V. Janoušek. Towards Design Method Based on Formalisms of Petri Nets, DEVS, and UML. In *ICSEA 2011, The Sixth International Conference on Software Engineering Advances*, pages 299–304, 2011.
10. R. Kočí, V. Janoušek, and F. Zbořil. Object Oriented Petri Nets - Modelling Techniques Case Study. *International Journal of Simulation Systems, Science & Technology*, 10(3):32–44, 2010.
11. O. Kummer, F. Wienberg, and et al. *An Extensible Editor and Simulation Engine for Petri Nets: Renew*, volume 3099 of *Lecture Notes in Computer Science*, pages 484–493. Springer Verlag, 2004.
12. MathWorks. Simulink – Simulation and Model-Based Design. <http://www.mathworks.com>, February 2010.
13. MetaCase. Domain-Specific Modeling with MetaEdit+. <http://www.metacase.com>, February 2010.
14. D. Moldt. OOA and Petri Nets for System Specification. In *Object-Oriented Programming and Models of Concurrency*. Italy, 1995.
15. C. Raistrick, P. Francis, J. Wright, C Carter, and I. Wilkie. *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004.
16. R. Valk. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In *Jorg Desel, Manuel Silva (eds.): Application and Theory of Petri Nets; Lecture Notes in Computer Science*, volume 120. Springer-Verlag, 1998.

# Porting the Renew Petri Net Simulator to the Operating System Android

Dominic Dibbern

University Hamburg  
Faculty of Mathematics, Informatics und Natural Sciences  
Department Informatics

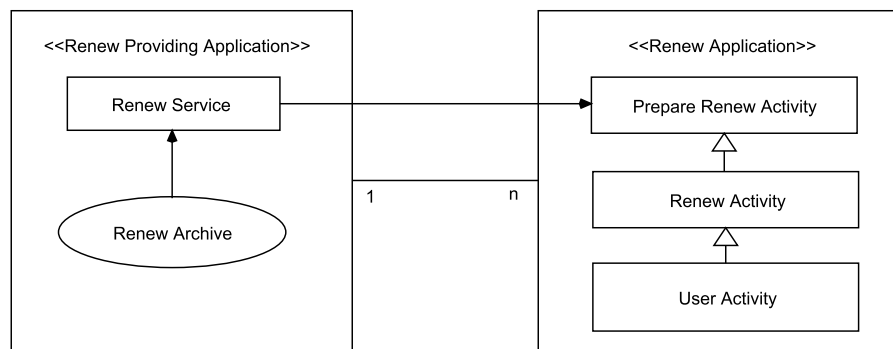
**Abstract.** This article describes the steps to identify, extract and adapt the core parts of the RENEW simulator in order to make them usable on an Android-powered device. As RENEW is build in Java and has a plugin-architecture, a few plugins have to be adapted to get a working Petri net simulator for the operating system Android, which applications are widely written in Java. The result of this work is a framework for simulating and using Petri nets in an Android application.

To extend the Petri net-based Agent-Oriented Software Engineering (PAOSE, see [1]) approach, to use Petri net-based agents on mobile devices like smartphones and tablet computers, the first step is to develop a Petri net simulator for this type of devices. As RENEW is used to create and simulate the agents, its simulation engine has been ported to the operating system Android. RENEW (see [2]) is written in Java and Android applications are usually also written in Java (see [3]), so the expenditure to port the application is not large. Furthermore, RENEW has a plugin-architecture, which allows to extract the needed core plugins and embed them into an Android application.

As a matter of fact, there are only three plugins needed for the simulation engine plus a plugin management system. As Android misses some libraries from a standard Java version, there are some plugins that have to be adapted. The missing libraries are essentially the graphical user interface and the remote method invocation capabilities. Fortunately, the graphical user interface is disconnected from and not referenced in the simulator core plugins. The remote method invocation is encapsulated and can easily be extracted to a new plugin, without a loss of functionality. Some additional minor changes are not further discussed here. With these changes RENEW's simulation engine is able to run on an Android device, using the same codebase as the full RENEW. This is an advantage, by having an up-to-date version on the mobile devices.

Although the RENEW simulation engine can now be used on Android, the start-up of application differs from the way of normal Java applications. This plus a relative huge amount of needed space for the plugins, on a mobile device with limited space for applications, implies several difficulties to create a practical system architecture for Android. Figure 1 shows the envisioned system-architecture. There is one application providing RENEW, which contains and distributes the

RENEW plugin archives to other application, that want to make use of the simulator. To allow application developers an easy use of RENEW, three components have been build. The RENEW Service as the server and the Prepare RENEW Activities as clients communicate at application start-up and automatically start the RENEW simulation engine. The RENEW Activity is a convenient extension of the Prepare RENEW Activity for easy control of the simulation engine. To develop a new application, using the simulation engine, the developer has to extend the RENEW activity and make use of its control functions. To reduce the use of disk space and to ensure an easy update there is one application providing RENEW for a couple of applications, which use the RENEW simulator.



**Fig. 1.** System architecture showing an Android application, which is able to integrate Petri nets into application development

As a result the simulation engine of RENEW is used to create a framework for Android applications, allowing them to take advantage of developing Petri nets instead of normal code, if it is useful. This prototypical implementation is the first step to extend the PAOSE approach into the mobile world of smartphones.

## References

1. Cabac, L.: Modeling Petri Net-Based Multi-Agent Applications. Dissertation, Universität Hamburg, Department Informatik, Vogt-Kölln Str. 30, D-22527 Hamburg (Apr 2010), <http://www.sub.uni-hamburg.de/opus/volltexte/2010/4666/>
2. Kummer, O., Wienberg, F., Duvigneau, M., Cabac, L.: Renew – the Reference Net Workshop (Aug 2009), <http://www.renew.de/>, release 2.2
3. Open Handset Alliance: Android developers. Website (2012), <http://developer.android.com>

# SonarEditor: A Tool for Multi-Agent-Organizations Modelling

Jan Bolte

Department of Informatics, TGI, University of Hamburg  
8bolte@informatik.uni-hamburg.de

**Abstract.** This paper presents the SonarEditor, which supports the creation of SONAR models. It provides pre-build net components, a well-formedness check and a wizard that aims at the support for possible enhancements of the model.

SonarEditor [1] is a prototypical implementation that enables multi-agent system developers to model organizational models following the SONAR formalism. It is implemented as plugin for RENEW [3]. The SonarEditor consists of three parts: pre-build net components [2], which help to create models, a well-formedness check that can check the well-formedness of the model and a wizard that can be used to enhance the model. Well-formedness of a model is defined by the definition of an organization and that the model is acyclic [4, Section 3.1].

SONAR is an approach to model organizations based on Petri nets. Such a model is composed of a delegation net and a set of distributed workflow nets (DWFs). The DWFs model the real workflows and the delegation net all possible courses of actions in an organization. The SonarEditor focuses on the delegation net, which is a Petri net  $(P, T, F)$  with  $P$  a set of Tasks,  $T$  a set of Implementations and  $F$  a set of arcs between  $P$  and  $T$ . Every Task and Implementation has to be assigned to a Position. This Position models a position in the organization. A Task models a task, which has to be executed. The Implementations define how every Position can implement each task. There are four different types of Implementations allowed by formal definition of the delegation net. The four types are named *execute*, *delegate*, *split*, *refine* and the pre-build net components for them are presented in the bottom of Figure 1. The top of Figure 1 displays the toolbar of the SONAR net components. There are from left to right *Position*, *Initial Task*, *Task*, *execute*, *delegate*, *split*, *refine*, *refine+split*. These are the net components for the delegation net. *refine+split* is the combination of the Implementations *refine* and *split*. The next components *Declaration Node*, *Role* and *DWFAction* are the components for creating DWFs. The last three buttons trigger the well-formedness check and the wizard: the first two trigger the well-formedness check with and without DWFs, the last triggers the wizard.

The SonarEditor shows an error frame if errors occur while checking the well-formedness. This frame contains a list of these errors and three buttons: *Select*, *Select All* and *Cancel*. If an error item is selected in the error frame the button *Select* can be pushed to select the corresponding element in the delegation net. With the button *Select All* all elements causing errors will be selected.

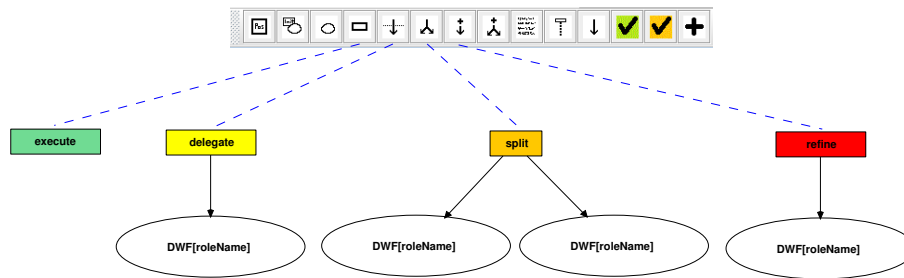


Fig. 1. Toolbar and SONAR net components

The last button of the toolbar starts the wizard, which supports the user by proposing possible enhancements based on the existing DWFs and delegation net. The wizard pages hold their previous and their following page and manage the remaining GUI of the wizard. There are five different wizard pages in this implementation:

**ErrorPage** is the first page, if a Task is selected, which causes an error.

**InitOrPosPage** is the first page, if nothing is selected and you can create a Position or start the creation of an initial Task with this page.

**TypePage** is the first page, if the selected Task does not cause an error. On this page you can define the type of the new Implementation. In the cases of *execute* and *split* this page is the last page.

**PosPage** is the page, where one can select the Position of the initial Task or the output Task of a new *delegate* Implementation.

**DWFPage** is the page, where you can select the DWF of the initial Task or the refining DWF in a *refine* or *refine+split* Implementation.

The SonarEditor supports the user with net components, a well-formedness check and a wizard. By these means, it supports the creation of well-formed SONAR models. A possible extension to this tool would be the consideration of the DWF well-formedness in the well-formedness check.

## References

1. Jan Bolte. Werkzeug-Unterstützung für organisationsorientierte Modellierung in SONAR, 2012. Bachelor thesis.
2. Lawrence Cabac. Net components: Concepts, tool, praxis. In Daniel Moldt, editor, *Petri Nets and Software Engineering, International Workshop, PNSE'09. Proceedings*, Technical Reports Université Paris 13, pages 17–33, 99, avenue Jean-Baptiste Clément, 93 430 Villetaneuse, June 2009. Université Paris 13.
3. Olaf Kummer, Frank Wienberg, Michael Duvigneau, and Lawrence Cabac. Renew – the Reference Net Workshop. <http://www.renew.de/>, 2012. Release 2.3.
4. Michael Köhler-Bußmeier and Matthias Wester-Ebbinghaus. Analysing Model Transformations in SONAR. 2012.

