Editors: Daniel Moldt and
Heiko Rölke

Proceedings of the International Workshop on

P etri
N ets and
S oftware
E ngineering
PNSE'13

University of Hamburg
Department of Informatics

These proceedings are published online by the editors as Volume 989 at

CEUR Workshop Proceedings
ISSN 1613-0073
http://ceur-ws.org/Vol-989

Preface

These are the proceedings of the International Workshop on *Petri Nets and Software Engineering* (PNSE'13) in Milano, Italy, June 24–25, 2013. It is a co-located event of *Petri Nets 2013*, the 34th international conference on Applications and Theory of Petri Nets and Concurrency.

More information about the workshop can be found at

`http://www.informatik.uni-hamburg.de/TGI/events/pnse13/`

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, will be presented as well as their application and tools supporting the disciplines mentioned above.

The program committee consists of:

Hanna Klaudel (France)
Radek Kočí (Czech republic)
Fabrice Kordon (France)
Maciej Koutny (United Kingdom) Lars Kristensen (Norway)
Michael Köhler-Bußmeier (Germany)
Johan Lilius (Finland)
Robert Lorenz (Germany)
Daniel Moldt (Germany) (Chair)
Chun Ouyang (Australia)
Wojciech Penczek (Poland)
Laure Petrucci (France)
Lucia Pomello (Italy)
Heiko Rölke (Germany) (Chair)
Catherine Tessier (France)
H.M.W. (Eric) Verbeek (Netherlands)
Karsten Wolf (Germany)

We received 25 high-quality contributions. For each paper three to four reviews were made. The program committee has accepted six of them for full presentation. Furthermore the committee accepted six papers as short presentations and two short papers. Two more contributions were accepted as posters.

The international program committee was supported by the valued work of
Edmundo López Bóbeda,
Görkem Kılınç,
Reng Zeng,
Benoît Barbot,
Alexis Marechal and
Artur Meski
as additional reviewers. Their work is highly appreciated.

Furthermore, we would like to thank our colleagues in the local organization team at the University of Milano, Italy, 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!
Daniel Moldt and Heiko Rölke Hamburg, June 2013

PNSE'13 Proceedings

Part I

PNSE'13: Invited Talk

Coordination for Situated MAS: Towards an Event-driven Architecture

Andrea Omicini and Stefano Mariani

Dipartimento di Informatica–Scienza e Ingegneria (DISI)
Alma Mater Studiorum–Università di Bologna, Italy
{andrea.omicini, s.mariani}@unibo.it

Abstract Complex software systems modelled as multi-agent systems (MAS) are characterised by activities that are generated either by agents, or by the environment in its most general acceptation—that is, environmental resources and the spatio-temporal fabric. Modelling and engineering complex multi-agent systems (MAS) – such as pervasive, adaptive, and situated MAS – requires then to properly handle diverse classes of events: agent operations, resource events, spatio-temporal situation.
In the following, first we devise out the requirements for a software architecture for an agent-based middleware based on boundary artefacts, then we sketch a concrete architecture based on the TuCSoN middleware for MAS coordination.

1 Motivation

Today's complex computational systems more and more require strict coupling with the environment: pervasive, adaptive, self-organising systems need to work as situated systems, able to react to relevant changes in the environment, and to possibly act over it appropriately and timely. Interaction with the *environment* is then one of the main issue in complex computational systems nowadays [1].

On the other hand, agent-oriented abstractions and technologies provide a solid ground for complex system modelling and engineering: in particular, meta-models like A&A [2], middlewares like CArtAgO [3], JADE [4], TuCSoN [5], agent-oriented methodologies like Gaia [6], PASSI [7] and SODA [8] already proved their effectiveness in dealing with the engineering of complex software systems [9]. The *reactive* nature of situated systems, however, does not cope well with the *proactive* nature of agency, at least not with no compromise: in particular, the event-driven computational model pushed by system situation does not match straightforwardly the typical high-level programming model of agent-oriented languages—in particular those for intelligent agents.

While such issues are typically faced with more articulated agent languages and architectures – like hybrid agents architectures –, their increasing complexity (in particular in size and number of components and events) mandates for principled solutions, possibly at system level rather than at single-component level. Accordingly, in the following we sketch an *event-driven architecture* for agent middleware exploiting coordination abstractions for event handling, discuss its

abstract features, and describe a possible reification as a concrete architecture based on the TuCSoN middleware for multi-agent system (MAS) coordination.

2 MAS as Event-driven Systems

Situated systems have to deal with the environment as the main source of activity, as well as the foremost target for their own activity. Environment activity is typically modelled in terms of *events*, whose interaction with computational systems is articulated in a number of stages: at least, selection of potentially-relevant events, perception of selected events, delivering of perceived events to designed components, elaboration of events by components. Moreover, situatedness also means reactiveness to the spatio-temporal fabric: perceiving and reacting to events related to location and motion in space, and to the passage of time, are essential features of mobile and pervasive computing applications. In the overall, dealing with situatedness basically requires an *event-driven programming model*, along with a suitable choice of the representation language for environment events.

On the other hand, modelling a complex computational system as a MAS basically accounts to encapsulating system activities within agents. Whereas the notion of environment as a sort of external source of event is more or less easy to accept, the same does not hold for agents. However, agents in an open MAS are possibly not designed and controlled by the MAS designer: so, their activity should be in principle handled again as an unpredictable source of events: either for openness, or for the intrinsic complexity that an agent behaviour may in principle encapsulate. Accordingly, both organisation and security issues require modelling agents, too, as (possibly unpredictable) event sources within MAS, to be possibly handled via event-driven engineering techniques.

As a result, an event-driven view of MAS is possible, where agents and the environment are the sources of all activities, and the overall behaviour of the MAS is obtained by suitably modelling activities as events, and governing them through suitable event-driven models and technologies.

3 Boundary & Coordination Artefacts

Whereas agents and environment are the most suitable abstractions to handle activities in a MAS, artefacts – being reactive by definition – are the most suitable abstractions to encapsulate reactive behaviours—so, the most suitable way to handle events in a complex MAS, according to the A&A meta-model [2].

The first issue is to map activities of any sort – even possibly unpredictable ones – upon a set of *admissible events*—that is, those events that are accepted and handled by the MAS. Apart from an appropriate model, this requires suitably-defined architectural abstractions embedding such a mapping. This is in fact the role of *boundary artefacts*, which mediate between agents and the MAS, as well as between the MAS and its environment.

In particular, we envision a principled MAS architecture where each agent and each resource in the environment is associated to its own boundary artefact, working on the one hand as a proxy for the agent / resource within the MAS, on the other hand as a sort of interface for the agent / resource towards the MAS. Known examples of boundary artefacts are Agent Communication Contexts [10] and the abstractions of Law-Governed Linda [11].

However, once brought within a MAS by a boundary artefact, an admissible event has to be handled to possibly generate other events and / or computational activities, defining the overall behaviour of a MAS: for instance, to aggregate events from resources, like a bunch of sensors. This is the role of *coordination artefacts* [12], which capture admissible MAS events, and associate them to computational activities implementing coordination laws, possibly generating further events, and giving raise to *event chains*.

With respect to the classification of artefacts introduced by the A&A metamodel [13], *individual* and *resource artefacts* are basically represented here by boundary artefacts, whereas *social artefacts* play roughly the role of coordination artefacts, here. In principle, however, boundary artefacts have a much more limited function with respect to individual and resource artefacts, which are devoted also to contain the basic coordination policies related to individual agents and resources. Then, a more precise architectural mapping would require *individual coordination artefacts* to be associated to boundary artefacts in order to achieve the same sort of architectural functionality provided by A&A individual artefacts.

4 A Concrete Event-driven Architecture in TuCSoN

The abstract architecture sketched above essentially models complex MAS as composed of *proactive entities* (agents, environment resources, space-time fabric) and *reactive entities* (boundary and coordination artefacts), connected together by a net of co-ordinated events. Quite unsurprisingly, a possible reification of such an abstract architecture can be designed upon the TuCSoN middleware for MAS coordination [5].

First of all, it is quite easy to map coordination artefacts upon ReSpecT tuple centres [14], which are the coordination abstraction provided by TuCSoN. There, computational activities devoted to MAS coordination can be represented in terms of the ReSpecT logic-based specification language [15], allowing admissible events to be associated to *reactions*, possibly generating further events within a MAS.

Then, two middleware abstractions play the role of boundary artefacts in TuCSoN: *agent coordination contexts* (ACC) [16], for agents, and *transducers* [17], for resources. On the one hand, ACC play the role of security and organisation abstractions [18]: each agent has an associated ACC that mediates all the agent interactions with the TuCSoN system, working both as its representative within the TuCSoN-coordinated MAS, and as its interface towards the MAS itself, providing the agent with available operations. On the other hand, trans-

ducers [17] are in charge of representing individual resources, along with their own peculiar ways of interacting: each portion of the MAS environment represented by a resource is associated to its specific transducer, capable of two-way interaction to map meaningful resource events upon admissible MAS events.

Mapping our abstract event-driven architecture upon the TuCSoN middleware obviously mandates for a complete event driven model. In TuCSoN, this is achieved by *(i)* generalising the TuCSoN notion of admissible event, and *(ii)* extending ReSpecT as a full-fledge event-driven language, capable of dealing with general-purpose events, enabling ReSpecT tuple centres to work as event-driven abstractions for MAS coordination—as discussed in [19].

In order to test the effectiveness of the abstract architecture depicted above, as well as of the corresponding TuCSoN-based concrete architecture for event-driven engineering of complex MAS, experiments were conducted, by exploiting the TuCSoN technology in complex application scenarios. In particular, TuCSoN is currently adopted for the implementation of the Molecules of Knowledge (MoK for short) model for knowledge self-organisation [20], and for the testing of the SAPERE middleware for pervasive adaptive services [21].

The TuCSoN middleware is available as an open source project [22], and in its current stage of development features ACC in its main distribution. The most general notion of transducers (with transducer managers for middleware lifecycle) and the complete situated version of ReSpecT are instead currently under testing.

Acknowledgements

The authors would like to thank the organisers of PNSE'13 and ModBE'13 – and in particular Daniel Moldt – for inviting our contribution.

This work has been supported by the EU-FP7-FET Proactive project SAPERE – Self-Aware PERvasive service Ecosystems, under contract no. 256873.

References

1. Weyns, D., Omicini, A., Odell, J.J.: Environment as a first-class abstraction in multi-agent systems. Autonomous Agents and Multi-Agent Systems **14**(1) (February 2007) 5–30 Special Issue on Environments for Multi-agent Systems.
2. Omicini, A., Ricci, A., Viroli, M.: Artifacts in the A&A meta-model for multi-agent systems. Autonomous Agents and Multi-Agent Systems **17**(3) (December 2008) 432–456 Special Issue on Foundations, Advanced Topics and Industrial Perspectives of Multi-Agent Systems.
3. Ricci, A., Viroli, M., Omicini, A.: CArtAgO: A framework for prototyping artifact-based environments in MAS. In Weyns, D., Parunak, H.V.D., Michel, F., eds.: Environments for MultiAgent Systems III. Volume 4389 of LNAI. Springer (May 2007) 67–86 3rd International Workshop (E4MAS 2006), Hakodate, Japan, 8 May 2006. Selected Revised and Invited Papers.

4. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley (February 2007)
5. Omicini, A., Zambonelli, F.: Coordination for Internet application development. Autonomous Agents and Multi-Agent Systems **2**(3) (September 1999) 251–269 Special Issue: Coordination Mechanisms for Web Agents.
6. Zambonelli, F., Jennings, N.R., Wooldridge, M.J.: Developing multiagent systems: The Gaia methodology. ACM Transactions on Software Engeneering Methodologies **12**(3) (2003) 317–370
7. Cossentino, M., Gaglio, S., Sabatucci, L., Seidita, V.: The PASSI and agile PASSI MAS meta-models compared with a unifying proposal. In Pechoucek, M., Petta, P., Varga, L.Z., eds.: Multi-Agent Systems and Applications IV. Volume 3690 of LNCS., Springer (2005) 183–192 4th International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2005, Budapest, Hungary, September 15-17, 2005, Proceedings.
8. Molesini, A., Omicini, A., Denti, E., Ricci, A.: SODA: A roadmap to artefacts. In Dikenelli, O., Gleizes, M.P., Ricci, A., eds.: Engineering Societies in the Agents World VI. Volume 3963 of LNAI. Springer (June 2006) 49–62 6th International Workshop (ESAW 2005), Kuşadası, Aydın, Turkey, 26–28 October 2005. Revised, Selected & Invited Papers.
9. Zambonelli, F., Omicini, A.: Challenges and research directions in agent-oriented software engineering. Autonomous Agents and Multi-Agent Systems **9**(3) (November 2004) 253–283 Special Issue: Challenges for Agent-Based Computing.
10. Di Stefano, A., Santoro, C.: Modeling multi-agent communication contexts. In: 1st International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS 2002), Bologna, Italy, ACM Press (15–19 July 2002) 174–175 Proceedings.
11. Minsky, N.H., Leichter, J.: Law-Governed Linda as a coordination model. In Ciancarini, P., Nierstrasz, O., Yonezawa, A., eds.: Object-based Models and Languages for Concurrent Systems. Volume 924 of LNCS. Springer (1995) 125–146
12. Omicini, A., Ricci, A., Viroli, M., Castelfranchi, C., Tummolini, L.: Coordination artifacts: Environment-based coordination for intelligent agents. In Jennings, N.R., Sierra, C., Sonenberg, L., Tambe, M., eds.: 3rd international Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004). Volume 1., New York, USA, ACM (19–23 July 2004) 286–293
13. Omicini, A., Ricci, A., Viroli, M.: *Agens Faber*: Toward a theory of artefacts for MAS. Electronic Notes in Theoretical Computer Science **150**(3) (29 May 2006) 21–36 1st International Workshop "Coordination and Organization" (CoOrg 2005), COORDINATION 2005, Namur, Belgium, 22 April 2005. Proceedings.
14. Omicini, A., Denti, E.: From tuple spaces to tuple centres. Science of Computer Programming **41**(3) (November 2001) 277–294
15. Omicini, A.: Formal ReSpecT in the A&A perspective. Electronic Notes in Theoretical Computer Science **175**(2) (June 2007) 97–117 5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06), CONCUR'06, Bonn, Germany, 31 August 2006. Post-proceedings.
16. Omicini, A.: Towards a notion of agent coordination context. In Marinescu, D.C., Lee, C., eds.: Process Coordination and Ubiquitous Computing. CRC Press, Boca Raton, FL, USA (October 2002) 187–200
17. Casadei, M., Omicini, A.: Situated tuple centres in ReSpecT. In Shin, S.Y., Ossowski, S., Menezes, R., Viroli, M., eds.: 24th Annual ACM Symposium on Applied Computing (SAC 2009). Volume III., Honolulu, Hawai'i, USA, ACM (8–12 March 2009) 1361–1368

18. Omicini, A., Ricci, A., Viroli, M.: Agent Coordination Contexts for the formal specification and enactment of coordination and security policies. Science of Computer Programming **63**(1) (November 2006) 88–107 Special Issue on Security Issues in Coordination Models, Languages, and Systems.
19. Mariani, S., Omicini, A.: Event-driven programming for situated MAS with ReSpecT tuple centres. In: Workshop on Agent Based Computing: From Model to Implementation X (ABC:MI 2013), Koblenz, Germany (September 2013) Proceedings.
20. Mariani, S., Omicini, A.: Molecules of Knowledge: Self-organisation in knowledge-intensive environments. In Fortino, G., Bădică, C., Malgeri, M., Unland, R., eds.: Intelligent Distributed Computing VI. Volume 446 of Studies in Computational Intelligence., Springer (2013) 17–22 6th International Symposium on Intelligent Distributed Computing (IDC 2012), Calabria, Italy, 24-26 September 2012. Proceedings.
21. Zambonelli, F., Castelli, G., Ferrari, L., Mamei, M., Rosi, A., Di Marzo Serugendo, G., Risoldi, M., Tchao, A.E., Dobson, S., Stevenson, G., Ye, Y., Nardini, E., Omicini, A., Montagna, S., Viroli, M., Ferscha, A., Maschek, S., Wally, B.: Self-aware pervasive service ecosystems. Procedia Computer Science **7** (December 2011) 197–199 Proceedings of the 2nd European Future Technologies Conference and Exhibition 2011 (FET 11).
22. TuCSoN: Home page. `http://tucson.apice.unibo.it`

Part II

PNSE'13: Long Presentations

A Canonical Contraction for Safe Petri Nets*

Thomas Chatain and Stefan Haar

INRIA & LSV (CNRS & ENS Cachan)
61, avenue du Président Wilson
94235 CACHAN Cedex, France
{chatain, haar}@lsv.ens-cachan.fr

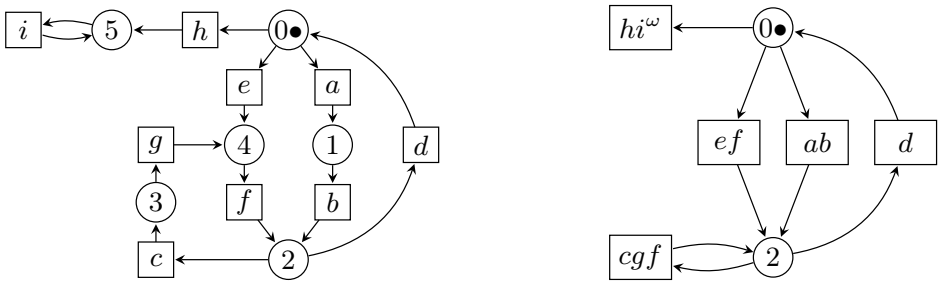
Abstract. Under maximal semantics, the occurrence of an event a in a concurrent run of an occurrence net may imply the occurrence of other events, not causally related to a, in the same run. In recent works, we have formalized this phenomenon as the *reveals* relation, and used it to obtain a contraction of sets of events called *facets* in the context of occurrence nets. Here, we extend this idea to propose a canonical contraction of general safe Petri nets into pieces of partial-order behaviour which can be seen as "macro-transitions" since all their events must occur together in maximal semantics. On occurrence nets, our construction coincides with the facets abstraction. Our contraction preserves the maximal semantics in the sense that the maximal processes of the contracted net are in bijection with those of the original net.

1 Introduction and Motivation

The properties of the long-run, *maximal* behaviour of discrete event systems contains also correlations between occurrences, i.e. relations of the type "if a fires, then b will fire sooner or later – unless it already has". This could be exploited in *predicting* (in the sense e.g. of failure prognosis, see [8]) events that inevitably will occur: Consider the sequential system shown in Figure 1(a). It is given here as a Petri net for convenience, but easily translated into an equivalent finite automaton of six states, eight transitions and initial state 0. When in state 0, the system can perform either a, e, or h. Whatever the choice of the first transition, however, in each case the *second* choice is imposed: after a no other transition than b is possible, after e only f, and after h only i.

It is known that structural transformations can facilitate verification of some system properties, as witnessed by e.g. Berthelot [3], Desel and Merceron [5], and other works. Here, we focus on other properties, those that depend only on the language of the *maximal* runs of the system, such as liveness properties, or particular other properties such as *diagnosability* or *predictability*, see [9,10]. In such a perspective, the system can be thought of as *contracted*: any stretch of consecutive transitions that occur always together in a maximal behavior provided that any *one* of them occurs, is fused into a single *macro-transition* that inherits pre- and post-places from the first and (if it exists) last transitions. In Figure 1(b): each of the new transitions is labeled with the transition chain that

* This work is supported by the French ANR project ImpRo (ANR-2010-BLAN-0317).



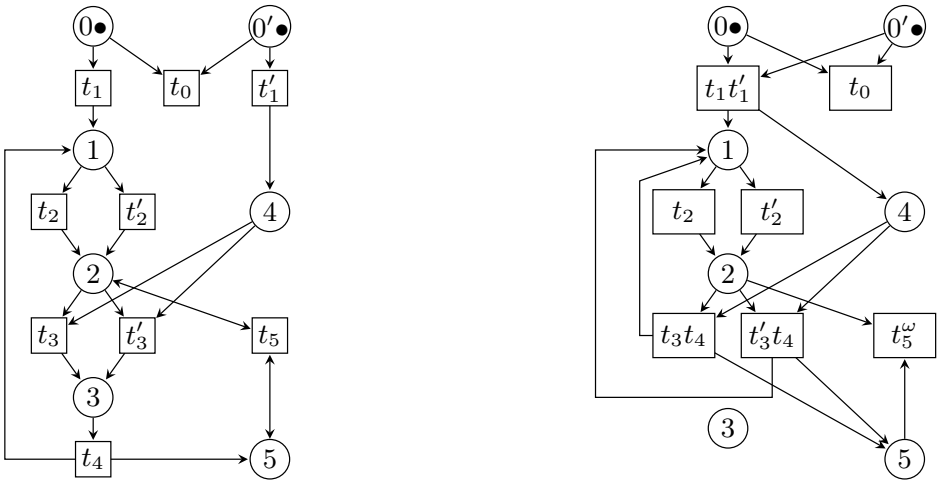
(a) (A Petri net representation of) an automaton (b) Its contraction

Fig. 1. Contracting automata by removing non-branching states (here 1, 3, 4 and 5)

it represents. Note that the infinite word hi^ω is obtained via a single macro-transition without post-place, since the word has no last transition. Of course, not all *temporal* properties of the system are preserved, since not all *finite* words survive the contraction: $abcg$ is a word produced by a run in Figure 1(a), but not in Figure 1(b) which has no intermediate word between (ab) and $(ab)(cgf)$. However, one sees quickly that the *maximal* words – which coincide with the *infinite* words – of the original system of Figure 1(a) are in bijection with the infinite words of the contracted system in Figure 1(b). This contraction represents a reduction of the original system onto its essential behavior.

When *concurrent* behavior in partial order semantics is considered, the language of words is replaced by a collection of partial orders representing the non-sequential runs. Best and Randell [4] considered atomicity of subnets in occurrence graphs, focusing on non-interference in the temporal behavior and identifying atomic and hence contractable blocks of behavior. The structures obtained can be embedded into non-branching occurrence nets, allowing the approach to be compared with ours. However, while the construction of facets appears geometrically similar, the approach of [6,7,1,2] focuses on the question of *logical occurrence* regardless of the order in which events occur. The theory of the reveals relation and of reduced occurrence nets is given in [6,7,1,2]. Figure 3(a) (whose formal discussion is postponed to Section 2) illustrates the facets of an occurrence net; the contraction of its facets yields the reduced occurrence net in Figure 3(b). The present work is based on a combination of the ideas shown, on the one hand, in the automata contraction such as in the example of Figure 1, and on the other hand of the facet contraction in the context of occurrence nets. We will identify *macro-transitions* in safe Petri nets that allow contraction with preservation of *maximal* semantics, and thus to give a contracted normal form for any given Petri net. If the definition is applied to occurrence nets, we obtain exactly the facets according to [6,7,1,2]. At the same time, the reduced net has never more, and generally much fewer, transitions than the original net.

The paper is organized as follows: We begin by recalling the basic definitions on unfoldings, and results from [6,7,1,2] concerning facets in occurrence nets, based on the *reveals*-relation, in Section 2. Section 3 contains the core of the present work, with the study of *macro-transitions* that generalize facets from



(a) A safe Petri net

(b) Its canonical contraction. (Place 3 is not used any more and could be forgotten.)

Fig. 2. Overview of the canonical contraction of a safe Petri net

occurrence nets to safe Petri nets. In Section 4, we identify the canonical reduced version for a given safe net. The relation between the operations of reduction and of unfolding is studied in Section 5. Finally, Section 6 concludes.

2 Reveals Relation and Facets in Occurrence Nets

Petri Nets, Occurrence Nets and Unfoldings. This part collects several basic definitions used below. In this paper, only safe Petri nets are considered.

Definition 1 (Petri Net). *A* Petri net (PN)*, or simply* net*, is a tuple* (P, T, F, M^0) *where* P *and* T *are sets of* places *and* transitions *respectively,* $F \subseteq (P \times T) \cup (T \times P)$ *is a* flow relation*, and* $M^0 \subseteq P$ *is an* initial marking*.*

For any node $x \in P \cup T$, we call *pre-set* of x the set ${}^\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$ and *post-set* of x the set $x^\bullet = \{y \in P \cup T \mid (x, y) \in F\}$. A *marking* of a net is a subset M of P. A transition t is *enabled* at M iff ${}^\bullet t \subseteq M$. Then t can *fire*, leading to $M' = (M \setminus {}^\bullet t) \cup t^\bullet$. In that case, we write $M \xrightarrow{t} M'$. A marking M is *reachable* if $M^0 \longrightarrow^* M$, where $\longrightarrow \stackrel{def}{=} \bigcup_{t \in T} \xrightarrow{t}$. A PN is *safe* iff for each reachable marking M, for each transition t enabled at M, $(t^\bullet \cap M) \subseteq {}^\bullet t$. As usual, in figures, transitions are represented as rectangles and places as circles. If $p \in M$, a black token is drawn in p (see Figure 2(a)).

Partial-order Semantics. *Occurrence nets* are used to represent the partial-order behaviour of Petri nets. We need a few definitions to introduce them. Denote by $\lessdot$ the *direct causality* relation defined as: for any transitions s and t, $s \lessdot t \stackrel{def}{\Leftrightarrow} s^\bullet \cap {}^\bullet t \neq \emptyset$. We write $<$ for its transitive closure and $\leq$ for its reflexive

transitive closure, called *causality*. For any transition t, the set $\lceil t \rceil \stackrel{def}{=} \{s \mid s \leq t\}$ is the *causal past* of t, and for $T' \subseteq T$, the causal past of T' is defined as $\lceil T' \rceil \stackrel{def}{=} \bigcup_{t \in T'} \lceil t \rceil$. Two distinct transitions s and t are in *direct conflict*, denoted by $s \#_d t$, iff ${}^\bullet s \cap {}^\bullet t \neq \emptyset$. Two transitions s and t are in *conflict*, denoted by $s \# t$, iff $\exists s' \in \lceil s \rceil, t' \in \lceil t \rceil : s' \#_d t'$, and the *conflict set* of t is defined as $\#[t] \stackrel{def}{=} \{s \mid s \# t\}$. Finally, two transitions s and t are *concurrent*, denoted by $s \; co \; t$, iff $\neg(s \# t) \wedge \neg(s \leq t) \wedge \neg(t \leq s)$.

Definition 2 (Occurrence net). *An* occurrence net *(ON) is a Petri net* (B, E, F, C^0) *where elements of* B *and* E *are called* conditions *and* events*, respectively, and such that:*

1. $\forall b \in C^0 \quad {}^\bullet b = \emptyset$,
2. $\forall b \in B \setminus C^0 \quad |{}^\bullet b| = 1$ *(no backward branching),*
3. $\forall e \in E \quad \neg(e < e)$ *($\leq$ is a partial order),*
4. $\forall e \in E \quad \neg(e \# e)$ *(no self-conflict),*
5. $\forall e \in E \quad |\lceil e \rceil| < \infty$ *(finite cones).*

Figure 3(a) gives an example of ON.

Occurrence nets are branching structures which have several possible executions in general. Each execution appears under the form of a *configuration*.

Definition 3 (Configurations and Maximal Configurations). *A* configuration *of an ON is a conflict-free and causally closed set of events, i.e.* $\omega \subseteq E$ *is a configuration iff* $\forall e \in \omega, (\#[e] \cap \omega = \emptyset) \wedge (\lceil e \rceil \subseteq \omega)$*. A configuration is* maximal *iff it is maximal w.r.t.* $\subseteq$*. We write* Ω_{gen} *for the set of all configurations and* Ω_{max} *for the set of maximal configurations.*

Executions of safe Petri nets will be represented as *non-branching processes*, using occurrence nets related to the original Petri net by a *net homomorphism*.

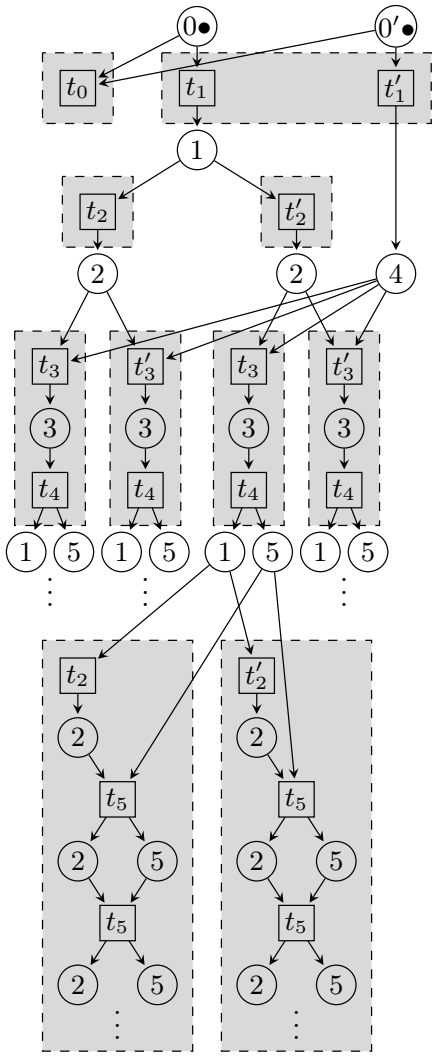
Definition 4 (Net homomorphism). *A* net homomorphism *from* $N = (P, T, F, M^0)$ *to* $N' = (P', T', F', M'^0)$ *is a pair of maps* $\pi = (\pi_P, \pi_T)$*, where* $\pi_P : P \to P'$ *and* $\pi_T : T \to T'$*, such that:*

- *for all* $t \in T$*,* $\pi_{P|{}^\bullet t}$ *(the restriction of* π_P *to* ${}^\bullet t$*) is a bijection between* ${}^\bullet t$ *and* ${}^\bullet \pi_T(t)$*, and* $\pi_{P|t^\bullet}$ *is a bijection between* $t^\bullet$ *and* $\pi_T(t)^\bullet$*;*
- *and* $\pi_{P|M^0}$ *maps injectively* M^0 *to (a subset of)* M'^0*.*

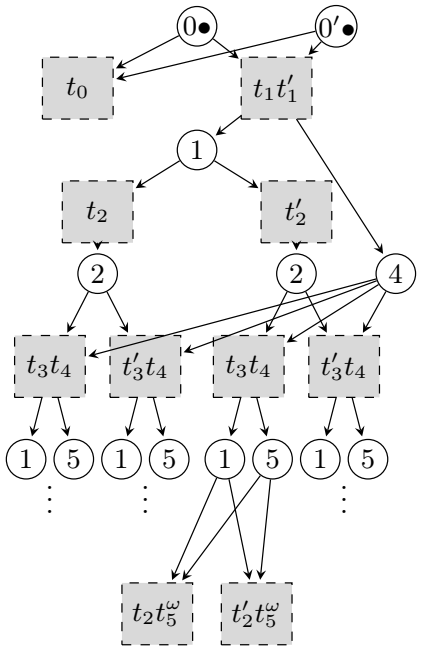
We will often write simply π *instead of* π_P *or* π_T*.*

Net homomorphisms preserve the semantics of nets in the sense that they map every firing sequence of N to a firing sequence of N', and $\pi_{P|M^0}$ needs not be a bijection for that. If a place p' of N' is not the image of any place of N, it simply means that the images in N' of the firing sequences of N do not use the token initially in p'. We need this subtlety to define macro-transitions later.

Definition 5 (Branching process). *Let* $N = (P, T, F, M^0)$ *be a PN. A* branching process *of* N *is a pair* (O, π)*, where* $O = (B, E, F', C^0)$ *is an ON and* π *is a homomorphism from* (B, E, F', C^0) *to* (P, T, F, M^0) *such that for all* $t, t' \in E$*,* $\left({}^\bullet t = {}^\bullet t' \wedge \pi(t) = \pi(t')\right) \Rightarrow t = t'$*.*



(a) A prefix of the unfolding of the Petri net of Figure 2(a). Dashed boxes indicate facets.



(b) The corresponding reduced ON. The condensed labels of facets indicate the events that they contain; e.g. the facet labeled t_1t_1' is the one depicted in Figure 4(b).

Fig. 3. An ON and its reduction through the facet abstraction.

Definition 6 (Run). *A* run *of a safe Petri net $N = (P, T, F, M^0)$ is a branching process (O, π) of N with $O = (B, E, F', C^0)$ such that E is a configuration and $\pi(C^0) = M^0$.*

Definition 7 (Prefix). *For Π_1, Π_2 two branching processes, Π_1 is a* prefix *of Π_2, written $\Pi_1 \sqsubseteq \Pi_2$, if there exists an injective homomorphism h from ON_1 into ON_2, such that the composition $\pi_2 \circ h$ coincides with π_1.*

Definition 8 (Maximal run). *A run ρ is* maximal *if it is not a proper prefix of any run, i.e. for every run ρ', if ρ is a prefix of ρ', then ρ and ρ' are isomorphic.*

We define a function μ which allows us to construct the run $\mu(\omega)$ corresponding to a configuration ω of an ON.

Definition 9 (μ). *Let $O = (B, E, F, C^0)$ be an occurrence net. Every conflict-free set of events $E' \subseteq E$ defines a run $\mu(E')$ of the Petri net $(B, E, F, {}^\bullet E' \setminus E'^\bullet)$[1]. The occurrence net $\mu(E')$ has E' as events, their pre- and post-sets as conditions, and ${}^\bullet E' \setminus E'^\bullet$ as initial conditions. The arcs are the restriction of F to these events and conditions, and the folding homomorphism π is the identity.*

Definition 10 (Unfolding). *Let N be a PN. By Theorem 23 of [11], there exists a unique (up to an isomorphism) $\sqsubseteq$-maximal branching process, called the* unfolding *of N and denoted $\mathcal{U}(N)$; by abuse of language, we will also call unfolding of N the ON obtained by the unfolding.*

Remark. Occurrence nets are linked to safe Petri nets in the sense that the partial order unfolding semantics of such Petri nets yields occurrence nets, as defined above. The converse is true for occurrence nets corresponding to regular trace languages: Following Zielonka [12], any regular trace language $\mathcal{L}$ is accepted by an asynchronous automaton $A_\mathcal{L}$; moreover, $A_\mathcal{L}$ can be synthesized directly from $\mathcal{L}$. As there are natural translations from asynchronous automata into safe Petri nets, the approach extends immediately into a procedure that takes as input an occurrence net ON and synthesizes a safe Petri net N whose unfolding semantics yields again ON (up to isomorphism). The present paper aims *not* at mimicking this synthesis but rather provides a contraction on the generating safe Petri net itself; the relation between unfolding and reduction will be clarified below, in particular Theorems 4 and 5, as well as Figure 6.

Reveals Relation and Facets Abstraction. The structure of an ON defines three relations over its events: *causality*, *conflict* and *concurrency*. But these structural relations do not express all logical dependencies between the occurrence of events in maximal configurations. A central fact is that concurrency is not always a logical independency: it is possible that the occurrence of an event implies, under the perspective of *maximal* runs, the occurrence of another one, which is structurally concurrent. This happens with events t_1 and t'_1 in Figure 3(a): we observe that t_1 is in conflict with t_0 and that any maximal configuration contains either t_0 or t'_1. Therefore, if t_1 occurs in a maximal configuration, then t_0 does not occur and eventually t'_1 necessarily occurs. Yet t_1 and t'_1 are concurrent.

Another case is illustrated by events labeled t_3 and t_4 on the left of the same figure: because t_3 is a causal predecessor of t_4, the occurrence of t_4 implies the occurrence of t_3; but in any maximal configuration, the occurrence of t_3 also implies the occurrence of t_4, because t_4 is the only possible continuation to t_3 and nothing can prevent it. Then t_3 and t_4 are actually made logically equivalent by the maximal progress assumption.

[1] Notice that $(B, E, F, {}^\bullet E' \setminus E'^\bullet)$ is not an occurrence net in general: it satisfies items 3, 4 and 5 of Definition 2, but items 1 and 2 may not hold.

Definition 11 (Reveals relation [6,7,1,2]). *We say that event* e reveals *event* f*, and write* $e \triangleright f$*, iff* $\forall \omega \in \Omega_{max}$*,* $(e \in \omega \Rightarrow f \in \omega)$*.*

Definition 12 (Facets Abstraction in Occurrence Nets[6]). *Let* $\sim$ *be the equivalence relation defined by* $\forall\ e, f \in E:\ e \sim f \overset{def}{\Leftrightarrow} (e \triangleright f) \wedge (f \triangleright e)$*. Then a* facet *of an ON is an equivalence class of* $\sim$*.*

In Figure 3(a), the facets are highlighted in grey. If ψ is a facet, then for any maximal configuration $\omega \in \Omega_{max}$ and for any event e such that $e \in \psi$, $e \in \omega$ iff $\psi \subseteq \omega$. In this sense, facets can be seen as atomic sets of events (under the maximal semantics). Denote the set of O's facets as $\Psi(O)$.

For any facet and for any configuration, either *all* events in the facet are in the configuration or *no* event in the facet is in the configuration. Therefore, facets can be seen as events.

Definition 13 (Reduced occurrence net). *A* reduced ON *is an ON* (B, E, F, C^0) *such that* $\forall e_1, e_2 \in e, \quad e_1 \sim e_2 \iff e_1 = e_2$*.*

As shown in [6,1], every occurrence net $O = (B, E, F, C^0)$ has a uniquely defined *reduction* ON $\overline{O}$ whose events are the facets of O and whose conditions those from B that are post-conditions of a maximal event of some facet:

Definition 14 (Reduction of an occurrence net). *The* reduction *of occurrence net* $O = (B, E, F, C^0)$ *is the occurrence net* $\overline{O} = (\overline{B}, \Psi(O), \overline{F}, C^0)$*, where*

$$\overline{B} = C^0 \cup \{b \in B:\ \exists\ \psi \in \Psi(O),\ e \in \psi:\ (e, b) \in F \wedge b^\bullet \cap \psi = \emptyset\} \tag{1}$$

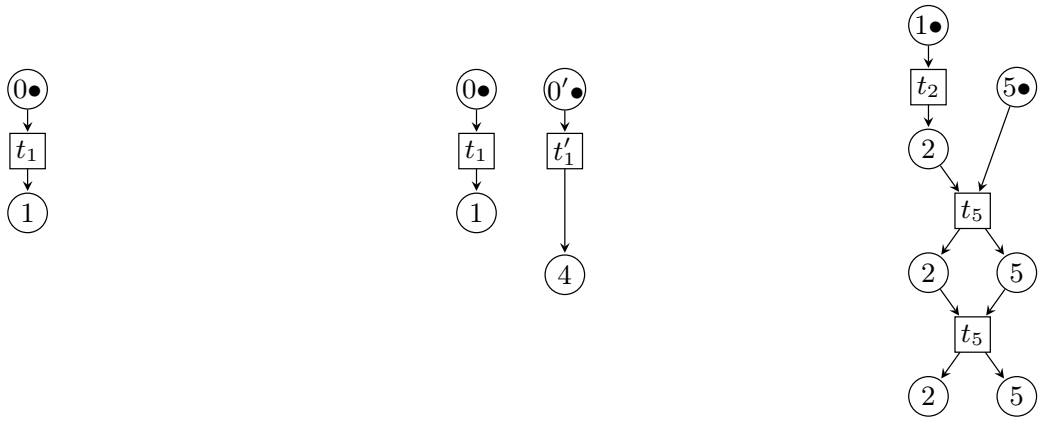
$$\begin{aligned} \overline{F} = \quad & \big\{(b, \psi):\ b \in \overline{B} \wedge \exists\ e \in \psi:\ (b, e) \in F\big\} \\ \cup\ & \big\{(\psi, b):\ b \in \overline{B} \wedge \exists\ e \in \psi:\ (e, b) \in F\big\} \end{aligned} \tag{2}$$

Figure 3 shows the facets of an occurrence net and its reduction.

3 Generalizing Facets to Safe Petri Nets

Preliminaries. We propose to identify pieces of partial-order behaviour of a safe Petri net, under the form of *macro-transitions* which group events that always occur together when at least one of them occur in any maximal run of the original net. There will be a fundamental difference in the approach here with respect to the work in [6,7,1,2]: there, the set of events to be contracted (the *facets*) were obtained as the strongly connected components of a transitive binary *reveals*-relation, where a reveals b iff any run containing a also contains b. Here , such a relation is not available on the level of transitions. Our approach is thus to identify directly sets of transitions such that, if any one of them fires, all others fire sooner or later.

Definition 15 (Macro-transition). *Let* $N = (P, T, F, M^0)$ *be a PN. A* macro-transition *of* N *is a run* $\phi = (O, \pi)$ *of* $(P, T, F, \pi(C^0))$ *(the net* N *initialized with the image of the initial conditions* C^0 *of* O*) such that for any reachable marking* M *of* N *with* $\pi(C^0) \subseteq M$ *and for any maximal run* ρ *of* (P, T, F, M) *(the net* N *starting at* M*), if there exists a nonempty prefix* ϕ' *of* ϕ *which is also a prefix of* ρ*, then the* entire ϕ *is a prefix of* ρ*.*



(a) ϕ_1 (b) ϕ_2 (c) ϕ_3

Fig. 4. Examples of macro-transitions of the Petri net of Figure 2(a)

Figures 4 and 5 show examples and counter-examples of macro-transitions of the Petri net of Figure 2(a).

- ϕ_1 is trivially a macro-transition.
- In ϕ_2 we have two events: an occurrence of t_1 and one of t'_1. The initial conditions of ϕ_2 are mapped to places 0 and $0'$ of N. The only reachable marking of N which contains $\{0, 0'\}$ is $\{0, 0'\}$ itself; in $\{0, 0'\}$, if one of the two transitions fire, the other one will necessarily fire in any maximal run.
- Consider now ϕ_3: again the only reachable marking of N which contains $\{1, 5\}$ is $\{1, 5\}$ itself. From it, if t_2 fires, it is necessarily followed by an infinite sequence of firings of t_5. ϕ_3 is exactly a prefix of it.

We also find counter-examples here:

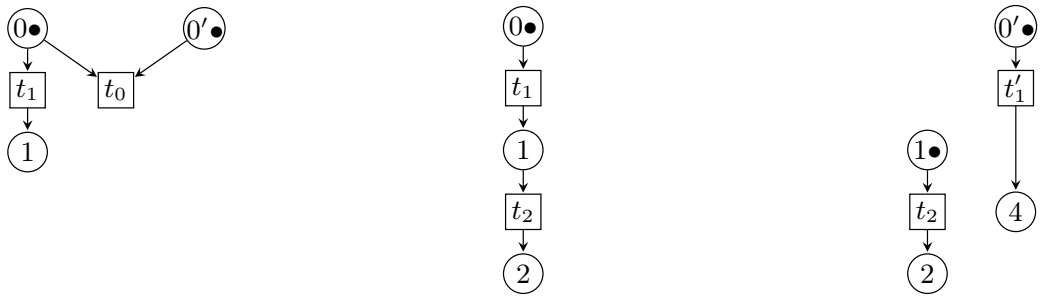
- ϕ_4 is not a macro-transition as it is not a run: t_0 and t_1 are in conflict.
- ϕ_5 is not a macro-transition because an occurrence of t_1 is not necessarily followed by an occurrence of t_2.
- Concerning ϕ_6, it is exactly a prefix of every maximal run from $\{1, 0'\}$ starting by an occurrence of t_2, but not of every run starting by an occurrence of t'_1 (because t'_2 can fire instead of t_2).

The two following properties are immediate consequences of the definition.

Property 1. Any single transition $t \in T$ induces a macro-transition defined as the (unique, up to isomorphism) non-branching process which contains a single event mapped to t and whose initial conditions are mapped to ${}^\bullet t$. For example, the facet induced by t_1 in the net of Figure 2(a) is the one depicted in Figure 4(a).

Property 2. Let ϕ be a macro-transition of a Petri net N. Then any prefix of ϕ with the same initial conditions as ϕ is also a macro-transition of N.

Definition 16 (Φ-contracted net). *Given a set Φ of macro-transitions of a Petri net $N = (P, T, F, M^0)$, we construct the* Φ-contracted net $N_{/\Phi}$ *by replacing*



(a) ϕ_4 (b) ϕ_5 (c) ϕ_6

Fig. 5. Counter-examples of macro-transitions of the Petri net of Figure 2(a)

the transitions of N by new transitions which summarize the macro-transitions. The contracted net is formally defined as the net $N_{/\Phi} = (P, \Phi, F_\Phi, M^0)$ where the macro-transitions are interpreted as transitions and with the flow relation F_Φ defined such that, for every $\phi = (O, \pi) \in \Phi$, ${}^\bullet\phi$ is the image by π of the initial conditions of O, and $\phi^\bullet$ is the image by π of the conditions of O that are not consumed by any event of O.

To express the soundness of this contraction, we define a function χ which maps any branching process (O, π) of the contracted net $N_{/\Phi}$ to a branching process of N. Intuitively, χ simply expands every event e of O into a set of events corresponding to the content of the macro-transition $\pi(e)$. For example, the reduced unfolding of Figure 3(b), viewed as a branching process of the contraction of the unfolding U of Figure 3(a), is mapped by χ to U.

Definition 17 (χ). *Let $N = (P, T, F, M^0)$ be a Petri net, Φ a set of macro-transitions of N and $\rho = (O, \pi)$ a branching process of the contracted net $N_{/\Phi}$, with $O = (B, E, F, C^0)$. We define the branching process $\chi(\rho)$ of N as $\chi(\rho) = (O', \pi')$ with $O' = (C^0 \cup \chi_{cond}(E), \chi_{events}(E), \chi_{cond}(E), \chi_{arcs}(E), C^0)$ where χ_{events}, χ_{cond} and χ_{arcs} associate to every event $e \in E$ a set of events $\chi_{events}(e)$, a set of conditions $\chi_{cond}(e)$ and a set of arcs $\chi_{arcs}(e)$, all specified below. Remember that e is an occurrence of transition $\pi(e)$ of $N_{/\Phi}$, which is also a macro-transition of N and thus has the form (O_e, π_e) with O_e an occurrence net and π_e a net homomorphism from O_e to $(P, T, F, \pi(C_e^0))$, where C_e^0 are the initial conditions of O_e.*

The set $\chi_{events}(e)$ is defined as the set of pairs (e, f) with f an event of O_e; it represents an occurrence of each of the events that were grouped inside the macro-transition $\pi(e)$ of the contracted net $N_{/\Phi}$.

The set $\chi_{conds}(e)$ is defined as the set of pairs (e, b) with b a condition created by an event of O_e; it represents all conditions created by events in $\chi_{events}(e)$. The initial conditions of $\pi(e)$ are not reproduced since they will be merged with the final conditions of the occurrence of the macro-transition that created them.

Now the arcs in $\chi_{arcs}(e)$ connect naturally every event (e, f) to the conditions (e, b) with $b \in f^\bullet$, and every condition (e, b) with $b \in {}^\bullet f$ to the event (e, f).

It remains the case of the initial conditions of O_e: for every initial condition b of O_e, there exists a unique condition $b' \in {}^\bullet e$ such that $\pi(b') = \pi_e(b) \in P$. Either this b' is an initial condition of O or it is created by an event $e' \in E$. In the first case, b' is also an initial condition of O' and an arc is added in $\chi_{arcs}(e)$ to connect it to any event $(e, f) \in \chi_{events}(e)$ representing an event f of O_e which consumes b. In the second case b' comes from a final condition of $\pi(e')$, which appears in $\chi_{cond}(e')$ and serves as the origin of the arcs.

Finally, we define the homomorphism π' from O' to N. It maps simply every event (e, f) to the transition $\pi_e(f) \in T$, and every condition (e, b) to $\pi_e(b) \in P$. On the set C_0 of initial conditions, π' coincides with π : $\pi_{|C_0} \equiv \pi'_{|C_0}$.

Lemma 1 (Soundness). *Let N be a Petri net and Φ a set of macro-transitions of N. The function χ maps any branching process (O, π) of the contracted net $N_{/\Phi}$ to a branching process of N.*

Proof. By construction of χ. □

Definition 18 (Completeness). *A set Φ of macro-transitions of a Petri net $N = (P, T, F, M^0)$ is* complete *if for every reachable marking M of the contracted net $N_{/\Phi} = (P, \Phi, F', M^0)$ and every transition $t \in T$ firable from M, the run of (P, T, F, M) composed of all the events revealed by the initial occurrence of t in the unfolding of (P, T, F, M), is the image by χ of a run of (P, Φ, F', M).*

Lemma 2. *Let N be a Petri net and Φ a complete set of macro-transitions of N. Then every maximal run ρ of N is (isomorphic to) the image by χ of a maximal run ρ' of $N_{/\Phi}$.*

Proof. To construct the ρ', start from the process with no events and initial conditions corresponding to the initial marking of N (which is also the initial marking of $N_{/\Phi}$). Then, as long as there are events in ρ which are not in $\chi(\rho')$, take one which is minimal w.r.t. causality and call it e. (Among the possible choices, e should be of minimal depth[2] so that every event of ρ is eventually in $\chi(\rho')$.) The transition t of N which is the image of e by the homomorphism of ρ, can fire from the marking M reached after ρ' (which is also the marking reached after $\chi(\rho')$). By the completeness hypothesis, there exists a run of (P, Φ, F', M) whose image by χ yields all the events revealed by the firing of t from M. Then ρ' can be augmented by this run. Our e of ρ is now one of the new events in $\chi(\rho')$; and the other new events are also in ρ because they are revealed by the occurrence of t from M and ρ is maximal.

Notice that at each step, $\chi(\rho')$ is a prefix of ρ. The iteration may not terminate but, since ρ' always grows, we consider its limit (containing all the events that are eventually added). By construction this limit is the desired process. □

Definition 19 (Non-Redundancy). *A set Φ of macro-transitions of a Petri net $N = (P, T, F, M^0)$ is called* non-redundant *if for every transition $t \in T$, at most one macro-transition $\phi \in \Phi$ starts by[3] t.*

[2] The depth of an event e is the size of the longest path from an initial condition to e.

[3] By "ϕ starts by t", we mean that there exists an event in ϕ which is mapped to t and consumes only initial conditions of ϕ.

Theorem 1 (Facets as Macro-Transitions). *Let $O = (B, E, F, C^0)$ be an occurrence net and $\psi \subseteq E$ a facet of O. Then $\mu(\psi)$ is a macro-transition of O. Moreover the image by μ of all the facets of O is a complete non-redundant set of macro-transitions of O.*

Proof. Consider a reachable set of conditions $C \supseteq {}^\bullet\psi$, and let ω be a maximal run of (B, E, F, C) starting by a nonempty prefix of $\mu(\psi)$. Then ω starts by $\mu(\{e\})$ with e an initial event of ψ. By Definition 12, e reveals all the events in ψ. This implies that ω starts by the entire $\mu(\psi)$.

For completeness, remark that for every run ρ of the contracted ON, the events in $\chi(\rho)$ are a union of facets of O. After such a run, every maximal run is again a union of facets.

Non-redundancy holds because the facets are a partition of the events. □

4 Canonical Contraction

Before defining our canonical contraction, we study the markings that are reachable after a run of a contracted net.

For every configuration O, we call *cut* of O the set of conditions which are created and not consumed along O. When O is the support of a finite run (O, π) of a net N, the homomorphism π maps the cut of O to a reachable marking of N. And conversely every reachable marking of N is the image of the final conditions of a finite run.

But in this paper we focus on maximal runs, which are in general infinite. And the image of a cut of an infinite run may be only a *subset* of a reachable marking of N. An example is the maximal run of the net of Figure 1(a) containing an occurrence of h and an infinite chain of i's. All the conditions are consumed, and the cut is empty. Yet the empty marking is not reachable after any finite run.

Then we call *asymptotically reachable* (or *a-reachable* for short) in N any marking that is the image of the cut of a (possibly infinite) run of N.

Lemma 3 (A-Reachability in a Contracted Net). *Let N be a Petri net and Φ a set of macro-transitions of N. Any marking a-reachable in $N_{/\Phi}$ is also a-reachable in N.*

Proof. This is an immediate consequence of Lemma 1. □

Notice however that in general not every marking a-reachable in N is a-reachable in $N_{/\Phi}$. And this is actually what allows us to skip some intermediate markings and give a more compact representation of the behaviour of the net.

In this sense we can say that a complete contracted net $N_{/\Phi}$ is more compact than another $N_{/\Phi'}$ if all markings a-reachable in $N_{/\Phi}$ are also a-reachable in $N_{/\Phi'}$. We will show now that there exists a complete non-redundant contracted net which is optimal w.r.t. this criterion: i.e. all markings a-reachable in this contracted net are a-reachable in any complete non-redundant contracted net.

Definition 20 ($\mathcal{M}_N$ and $\mathcal{R}_N$). *We define inductively a set $\mathcal{M}_N$ of markings of M and a set $\mathcal{R}_N$ of runs as the smallest sets satisfying:*

- *$M^0 \in \mathcal{M}_N$;*
- *for every $M \in \mathcal{M}_N$, for every transition t firable from M, $\mu(E) \in \mathcal{R}_N$, where E is the set of events revealed by the initial occurrence of t in $U((P,T,F,M))$;*
- *for every $M \in \mathcal{M}_N$, for every $\rho \in \mathcal{R}_N$ such that ${}^\bullet\rho \subseteq M$, the marking $(M \setminus {}^\bullet\rho) \cup \rho^\bullet$ reached after firing ρ from M, belongs to $\mathcal{M}_N$;*
- *for every $\rho_1, \rho_2 \in \mathcal{R}_N$, the largest common prefix of ρ_1 and ρ_2 is in $\mathcal{R}_N$.*

Theorem 2. *Let $N = (P,T,F,M^0)$ be a Petri net and Φ a non-redundant complete set of macro-transitions. All markings of $\mathcal{M}_N$ are a-reachable in $N_{/\Phi}$.*

Proof. Let $N_\Phi = (P, \Phi, F', M^0)$. The theorem is a direct consequence of the following lemma: for every marking M a-reachable in N every run $\rho \in \mathcal{R}_N$ firable from M satisfies the property that ρ is the image by χ a run ρ' of (P, Φ, F', M). This lemma is proved by induction, following the construction of $\mathcal{R}_N$: at each step of the construction, we prove that if all the runs in the current $\mathcal{R}_N$ satisfy the property, then the new runs added to $\mathcal{R}_N$ also satisfy it. Initialization of the induction is trivial since $\mathcal{R}_N$ is initially empty.

By completeness of Φ, the property is satisfied by all the runs of the form $\mu(E)$ with E the set of events revealed by the initial occurrence of a transition t in $U((P,T,F,M))$. For every run ρ constructed as the largest common prefix of two runs ρ_1 and ρ_2 already in $\mathcal{R}_N$, assume that ρ_1 and ρ_2 satisfy our property and call ρ'_1 and ρ'_2 the corresponding runs of the contracted net. By non-redundancy of Φ, ρ'_1 and ρ'_2 must coincide on the largest common prefix ρ of ρ_1 and ρ_2. Then ρ is the image by χ of the largest common prefix of ρ'_1 and ρ'_2. □

Definition 21 (Canonical contraction $\overline{N}$). *We define the* canonical contraction *of a safe Petri net N as the contracted net $\overline{N} \stackrel{def}{=} N_{\Phi_N}$ where Φ_N is the set of nonempty runs of $\mathcal{R}_N$ which are minimal w.r.t. the prefix relation.*

Theorem 3. *For every safe Petri net N, the set Φ_N of macro-transitions in $\overline{N}$ is complete and non-redundant, and the set of states a-reachable in $\overline{N}$ is precisely $\mathcal{M}_N$. Moreover $|\Phi_N| \leq |T|$.*

Proof. Completeness is ensured by the insertion in $\mathcal{R}_N$ of all the runs of the form $\mu(E)$ with E the set of events revealed by the initial occurrence of a transition t in $U((P,T,F,M))$. For redundancy, assume two runs ρ_1 and ρ_2 of $\mathcal{R}_N$ both start by an occurrence of t. Then their common prefix ρ is nonempty and is in $\mathcal{R}_N$. Then ρ_1 and ρ_2 are not minimal in $\mathcal{R}_N$ w.r.t. the prefix relation, and they are not in Φ_N. By construction all the states a-reachable in $\overline{N}$ are in $\mathcal{M}_N$. Finally the inequality $|\Phi_N| \leq |T|$ is a direct consequence of the non-redundancy of Φ_N. □

Illustration. Let us construct the canonical contraction of the net N of Figure 2(a). $\mathcal{M}_N$ contains the initial marking $\{0, 0'\}$. From this marking t_0, t_1 and t'_1 are firable. Since t_1 and t'_1 reveal each other, $\mathcal{R}_N$ contains the runs t_0 and

$t_1t'_1$, and $\mathcal{M}_N$ contains the reached markings $\{\}$ and $\{1,4\}$. From $\{1,4\}$, t_2 and t'_2 can fire; they reveal nothing, so they are added as such to $\mathcal{R}_N$. The marking $\{2,4\}$ is now reachable; it is added to $\mathcal{M}_N$. From $\{2,4\}$, t_3 and t'_3 can fire, and in both cases an occurrence of t_4 necessarily follows. Hence t_3t_4 and t'_3t_4 are added to $\mathcal{R}_N$. We can now reach $\{1,5\}$ and fire t_2 or t'_2 again. But, from $\{1,5\}$ firing t_2 (or t'_2) reveals an infinite sequence of occurrences of t_5. For this $t_2t_5^\omega$ and $t'_2t_5^\omega$ are added to $\mathcal{R}_N$. But, since t_2 and t'_2 already appear "alone" – i.e. as singleton transitions – in $\mathcal{R}_N$, marking $\{2,5\}$ obtained after firing them from $\{1,5\}$ must also be added to $\mathcal{M}_N$. And from it, t_5^ω can fire and is added to $\mathcal{R}_N$. Now, Φ_N is constructed by extracting the runs of $\mathcal{R}_N$ that are minimal w.r.t. the prefix relation. Here we get all of them, except $t_2t_5^\omega$ and $t'_2t_5^\omega$. The resulting contracted net is shown in Figure 2(b).

Contraction and Automata. It is clear that applying our contraction to the Petri net representation N of an automaton (i.e. a Petri where every transition has exactly one input- and one output-place) removes the deterministic states (or places), i.e. those from which there is no choice. Concretely, these places will not appear in the set $\mathcal{M}_N$. The macro-transitions are the paths between non-deterministic states with only deterministic intermediate states.

5 Reductions and Unfoldings

When *concurrent* behavior in partial order semantics is considered, our contraction is related to the facets reduction [6].

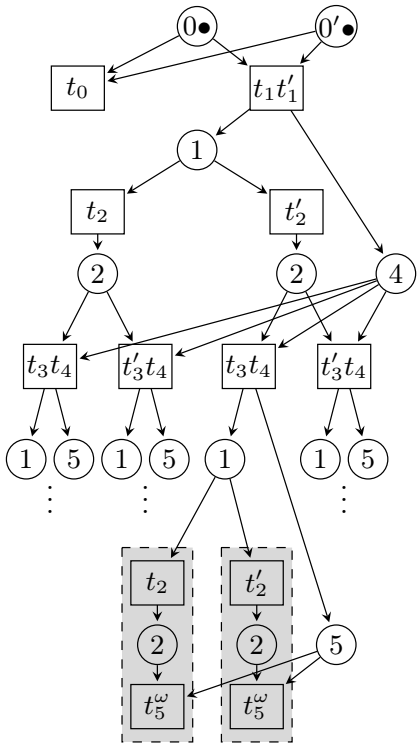
Theorem 4 (Reduction as contraction of ONs). *For every occurrence net O, the canonical contraction of O is isomorphic to its facet reduction.*

Proof. By Definition 20, all runs in $\mathcal{R}_O$ correspond to unions of facets of O. Now, let $\rho \in \mathcal{R}_O$ be a run containing more than one facet. By definition of facets, the reveals relation on facets is antisymmetric. Then one of O's initial facets, say ψ_1, does not reveal the other, say ψ_2. Take an initial event e of ψ_1 and a marking $M \in \mathcal{M}_O$ from which ρ can fire; e is firable from M in O. Therefore $\mathcal{R}_O$ contains the run ρ' containing the events revealed by e from M. This run contains ψ_1 but not ψ_2. By definition, $\mathcal{R}_O$ contains the largest common prefix of ρ and ρ'. Hence ρ is not minimal in $\mathcal{R}_O$ w.r.t. the prefix relation, and is not in Φ_O. □

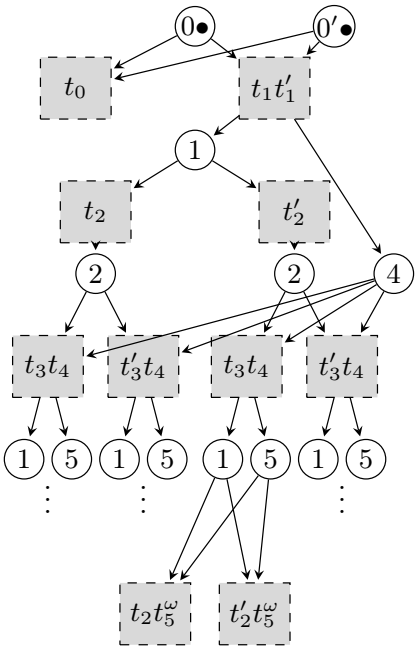
As illustrated in Figure 6, the operation of reduction does not entirely commute with unfolding. That is, in general, the unfolding $U(\overline{N})$ of reduced Petri net $\overline{N}$ is coarser, as an occurrence net, than the reduction $\overline{U(N)}$ of the original net N's unfolding. In the example of Figure 6, the facets labeled $t_2t_5^\omega$ and $t'_2t_5^\omega$ in $\overline{U(N)}$ are both split into two events of $U(\overline{N})$.

However, one retrieves the reduction of $U(N)$ from $U(\overline{N})$ as follows.

Theorem 5. *For every net N, applying the occurrence net facet reduction to $U(\overline{N})$ yields $\overline{U(N)}$ up to isomorphism.*



(a) The unfolding of the contracted Petri net of Figure 2(b). Remark that the unfolding is not reduced: the last occurrence of t_2 and the following t_5^ω are in the same facet (similarly for t_2' and the following t_5^ω).



(b) Its reduction (or contraction) is isomorphic to the reduction of the unfolding of N already represented in Figure 3(b).

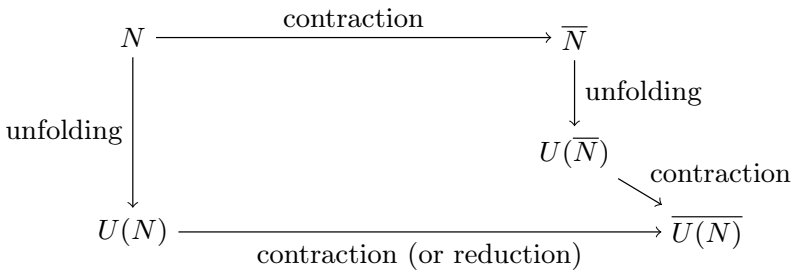


Fig. 6. Unfolding and contraction.

Proof. By definition of macro-transitions, for every event e of $U(\overline{N})$, all the events of $U(N)$ which are in $\chi_{events}(e)$, reveal each other. Then $\chi_{events}(e)$ is included in a facet ψ of $U(N)$. And for two events e_1 and e_2 of $U(\overline{N})$, an event in $\chi_{events}(e_1)$ reveals (in $U(N)$) an event in $\chi_{events}(e_2)$ iff e_1 reveals e_2 in $U(\overline{N})$. Therefore the facets reduction of $U(\overline{N})$ regroups e_1 and e_2 into the same facet iff the events in $\chi_{events}(e_1)$ and those in $\chi_{events}(e_2)$ are in the same facet. □

6 Conclusion

We have presented a method for identifying and contracting *macro-transitions* in safe Petri nets. The procedure includes and justifies our previous work in [6,7,1,2] focusing on *facets* in occurrence nets. The result is a unique contracted 1-safe Petri net with no more macro-transitions than transitions in the original net. The construction provides a unique *canonical* version for any given 1-safe Petri net, whose maximal behaviour offers a condensed view of the maximal behaviour of the original net. By computing offline the canonical version, verification procedures for any property that depends only on the maximal run behavior can be run on the smaller contracted net instead. Computing the contraction (with finite representations of the macro-transitions) is in general costly (computing the reveals relation on the unfolding of a finite Petri net is PSPACE-complete [7]), but in practice many syntactic sufficient conditions can be used to identify macro-transitions. Hence our contraction appears as an optimal, canonical contraction, to which other contractions based on macro-transitions can be compared.

References

1. S. Balaguer, T. Chatain, and S. Haar. Building tight occurrence nets from reveals relations. In *Proceedings of the 11th International Conference on Application of Concurrency to System Design*, pages 44–53. IEEE Computer Society Press, 2011.
2. S. Balaguer, T. Chatain, and S. Haar. Building occurrence nets from reveals relations. *Fundamenta Informaticae*, 123(3):245–272, 2013.
3. G. Berthelot. Checking properties of nets using transformation. In *Applications and Theory in Petri Nets*, volume 222 of *LNCS*, pages 19–40. Springer, 1985.
4. E. Best and B. Randell. A formal model of atomicity in asynchronous systems. *Acta Informatica*, 16(1):93–124, 1981.
5. J. Desel and A. Merceron. Vicinity respecting homomorphisms for abstracting system requirements. In *Proc. Int. Workshop on Abstractions for Petri Nets and Other Models of Concurrency (APNOC)*, 2009.
6. S. Haar. Types of asynchronous diagnosability and the *reveals*-relation in occurrence nets. *IEEE Transactions on Automatic Control*, 55(10):2310–2320, 2010.
7. S. Haar, C. Kern, and S. Schwoon. Computing the reveals relation in occurrence nets. In *Proceedings of GandALF'11*, volume 54 of *Electronic Proceedings in Theoretical Computer Science*, pages 31–44, 2011.
8. R. Kumar and S. Takai. Decentralized prognosis of failures in discrete event systems. *IEEE Transactions on Automatic Control*, 55(1):48–59, 2010.
9. A. Madalinski and V. Khomenko. Diagnosability verification with parallel LTL-X model checking based on Petri net unfoldings. In *Control and Fault-Tolerant Systems (SysTol'2010)*, pages 398–403. IEEE Computing Society Press, 2010.
10. A. Madalinski and V. Khomenko. Predictability verification with parallel LTL-X model checking based on Petri net unfoldings. In *Proc. of the 8th IFAC Symposium on fault detection, diagnosis and safety of technical processes (SAFEPROCESS'2012)*, pages 1232–1237, 2012.
11. M. Nielsen, G. D. Plotkin, and G. Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13:85–108, 1981.
12. W. Zielonka. Notes on finite asynchronous automata. *RAIRO, Theoretical Informatics and Applications*, 21:99–135, 1987.

Symbolic Verification of ECA Rules

Xiaoqing Jin, Yousra Lembachar, and Gianfranco Ciardo

Department of Computer Science and Engineering,
University of California, Riverside
{jinx,ylemb001,ciardo}@cs.ucr.edu

Abstract. Event-condition-action (ECA) rules specify a decision making process and are widely used in reactive systems and active database systems. Applying formal verification techniques to guarantee properties of the designed ECA rules is essential to help the error-prone procedure of collecting and translating expert knowledge. The nondeterministic and concurrent semantics of ECA rule execution enhance expressiveness but hinder analysis and verification. We then propose an approach to analyze the dynamic behavior of a set of ECA rules, by first translating them into an extended Petri net, then studying two fundamental correctness properties: *termination* and *confluence*. Our experimental results show that the symbolic algorithms we present greatly improve scalability.

Keywords: ECA rules, termination, confluence, verification

1 Introduction

Event-condition-action (ECA) [12] rules are expressive enough to describe complex events and reactions. Thus, this event-driven formalism is widely used to specify complex systems [1, 3], e.g., for industrial-scale management, and to improve efficiency when coupled with technologies such as embedded systems and sensor networks. Analogously, active DBMSs enhance security and semantic integrity of traditional DBMSs using ECA rules; these are now found in most enterprise DBMSs and academic prototypes thanks to the SQL3 standard [9]. ECA rules are used to specify a system's response to events, and are written in the format "on the occurrence of a set of events, if certain conditions hold, perform these actions". However, for systems with many components and complex behavior, it may be difficult to correctly specify these rules.

Termination, guaranteeing that the system does not remain "busy" internally forever without responding to external events, and *confluence*, ensuring that any possible interleaving of a set of triggered rules yields the same final result, are fundamental correctness properties. While termination has been studied extensively and many algorithms have been proposed to verify it, confluence is particularly challenging due to a potentially large number of rule interleavings [2].

Researchers began studying these properties for active databases in the early 90's [2, 4, 11, 14], by transforming ECA rules into some form of graph and applying various static analysis techniques on it to verify properties. These approaches

based on a static methodology worked well to detect redundancy, inconsistency, incompleteness, and circularity. However, since static approaches may not explore the whole state space, they could easily miss some errors. Also, they could find scenarios that did not actually result in errors due to the found error states may not be reachable. Moreover, they had poor support to provide concrete counterexamples and analyze ECA rules with priorities. [2] looked for cycles in the rule-triggering graph to disprove termination, but the cycle-triggering conditions may be unsatisfiable. [4] improved this work with an activation graph describing when rules are activated; while its analysis detects termination where previous work failed, it may still report false positives when rules have priorities. [5] proposed an algebraic approach emphasizing the condition portion of rules, but did not consider priorities. Other researchers [11, 14] chose to translate ECA rules into a Petri Net (PN), whose nondeterministic interleaving execution semantics naturally model unforeseen interactions between rule executions. However, as the analysis of the set of ECA rules was through structural PN techniques based on the incidence matrix of the net, false positives were again possible.

To overcome these limitations, dynamic analysis approaches using model checking tools such as SMV [15] and SPIN [6] have been proposed to verify termination. While closer to our work, these approaches require manually transforming ECA rules into an input script, assume a priori bounds for all variables, provide no support for priorities, and require the initial system state to be known; our approach does not have these limitations. [8] analyzes both termination and confluence by transforming ECA rules into Datalog rules through a "transformation diagram"; this supports rule priority and execution semantics, but requires the graph to be commutative and restricts event composition. However, most of these works show limited results, and none of them properly addresses confluence; we present detailed experimental results for both termination and confluence. UML Statecharts [17] provide visual diagrams to describe the dynamic behavior of reactive systems and can be used to verify these properties. However, event dispatching and execution semantics are not as flexible as for PNs [13].

Our approach transforms a set of ECA rules into a PN, then dynamically verifies termination and confluence and, if errors are found, provides concrete counterexamples to help debugging. It uses our tool SMART, which supports PNs with priorities to easily model ECA rules with priorities. Moreover, a single PN can naturally describe both the ECA rules as well as their nondeterministic concurrent environment and, while our MDD-based symbolic model-checking algorithms [18] require a finite state space, they do not require to know a priori the variable bounds (i.e., the maximum number of tokens each place may contain). Finally, our framework is not restricted to termination and confluence, but can be easily extended to verify a broader set of properties.

The rest of the paper is organized as follows: Sect. 2 recalls Petri nets; Sect. 3 introduces our syntax for ECA rules; Sect. 4 describes the transformation of ECA rules into a Petri net; Sect. 5 presents algorithms for termination and confluence; Sect. 6 shows experimental results; Sect. 7 concludes.

2 Petri Nets

As an intermediate step in our approach, we translate a set of ECA rules into a *self-modifying Petri net* [16] (PN) with *priorities* and *inhibitor arcs*, described by a tuple $(\mathcal{P}, \mathcal{T}, \pi, \mathbf{D}^-, \mathbf{D}^+, \mathbf{D}^\circ, \mathbf{x}_{init})$, where:

- $\mathcal{P}$ is a finite set of *places*, drawn as circles, and $\mathcal{T}$ is a finite set of *transitions*, drawn as rectangles, satisfying $\mathcal{P} \cap \mathcal{T} = \emptyset$ and $\mathcal{P} \cup \mathcal{T} \neq \emptyset$.
- $\pi : \mathcal{T} \to \mathbb{N}$ assigns a *priority* to each transition.
- $\mathbf{D}^- : \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{\mathcal{P}} \to \mathbb{N}$, $\mathbf{D}^+ : \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{\mathcal{P}} \to \mathbb{N}$, and $\mathbf{D}^\circ : \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{\mathcal{P}} \to \mathbb{N} \cup \{\infty\}$ are the *marking-dependent* cardinalities of the *input*, *output*, and *inhibitor* arcs.
- $\mathbf{x}_{init} \in \mathbb{N}^{\mathcal{P}}$ is the *initial marking*, the number of *tokens* initially in each place.

Transition t has *concession* in marking $\mathbf{m} \in \mathbb{N}^{\mathcal{P}}$ if, for each $p \in \mathcal{P}$, the input arc cardinality is satisfied, i.e., $\mathbf{m}_p \geq \mathbf{D}^-(p, t, \mathbf{m})$, and the inhibitor arc cardinality is not, i.e., $\mathbf{m}_p < \mathbf{D}^\circ(p, t, \mathbf{m})$. If t has concession in $\mathbf{m}$ and no other transition t' with priority $\pi(t') > \pi(t)$ has concession, then t is *enabled* in $\mathbf{m}$ and can *fire* and lead to marking $\mathbf{m}'$, where $\mathbf{m}'_p = \mathbf{m}_p - \mathbf{D}^-(p, t, \mathbf{m}) + \mathbf{D}^+(p, t, \mathbf{m})$, for all places p (arc cardinalities are evaluated in the current marking $\mathbf{m}$ to determine the enabling of t and the new marking $\mathbf{m}'$). In our figures, $tk(p)$ indicates the number of tokens in p for the current marking, a thick input arc from p to t signifies a cardinality $tk(p)$, i.e., a *reset* arc, and we omit arc cardinalities 1, input or output arcs with cardinality 0, and inhibitor arcs with cardinality ∞.

The PN defines a discrete-state model $(\mathcal{S}_{pot}, \mathcal{S}_{init}, \mathcal{A}, \{\mathcal{N}_t : t \in \mathcal{A}\})$. The *potential state space* is $\mathcal{S}_{pot} = \mathbb{N}^{\mathcal{P}}$ (in practice we assume that the reachable set of markings is finite or, equivalently, that there is a finite bound on the number of tokens in each place, but we do not require to know this bound a priori). $\mathcal{S}_{init} \subseteq \mathcal{S}_{pot}$ is the set of *initial states*, $\{\mathbf{x}_{init}\}$ in our case (assuming an arbitrary finite initial set of markings is not a problem). The set of (asynchronous) model *events* is $\mathcal{A} = \mathcal{T}$. The *next-state function* for transition t is $\mathcal{N}_t$, such that $\mathcal{N}_t(\mathbf{m}) = \{\mathbf{m}'\}$, where $\mathbf{m}'$ is as defined above if transition t is enabled in marking $\mathbf{m}$, and $\mathcal{N}_t(\mathbf{m}) = \emptyset$ otherwise. Thus, the next-state function for a particular PN transition is deterministic, although the overall behavior remains nondeterministic due to the choice of which transition should fire when multiple transitions are enabled.

3 ECA syntax and semantics

ECA rules have the format "**on** *events* **if** *condition* **do** *actions*". If the *events* are activated and the boolean *condition* is satisfied, the rule is triggered and its *actions* will be performed. In active DBMSs, events are normally produced by explicit database operations such as *insert* and *delete* [1] while, in reactive systems, they are produced by sensors monitoring environment variables [3], e.g., temperature. Many current ECA languages can model the environment and distinguish between **environmental** and **local** variables [2, 4, 6, 11, 14, 15]. Thus, we designed a language to address these issues, able to handle more general cases and allow different semantics for **environmental** and **local** variables (Fig. 1).

$$
\begin{aligned}
\mathit{env\_vars} &:= \textbf{environmental } \mathit{env\_var} && \text{READ-ONLY BOUNDED NATURAL}\\
\mathit{loc\_vars} &:= \textbf{local } \mathit{loc\_var} && \text{READ-AND-WRITE BOUNDED NATURAL}\\
\mathit{factor} &:= \mathit{loc\_var} \mid \mathit{env\_var} \mid (\ \mathit{exp}\ ) \mid \mathit{number}\\
\mathit{term} &:= \mathit{factor} \mid \mathit{term} * \mathit{term} \mid \mathit{term} \,/\, \mathit{term} && \text{“/” IS INTEGER DIVISION}\\
\mathit{exp} &:= \mathit{exp} - \mathit{exp} \mid \mathit{exp} + \mathit{exp} \mid \mathit{term} && \text{“}\mathit{number}\text{” IS A CONSTANT} \in \mathbb{N}\\
\mathit{rel\_op} &:= \ \geq \mid \leq \mid =\\
\mathit{assignment} &:= \mathit{env\_var} \textbf{ into } \mathit{loc\_var}\ [, \mathit{assignment}]\\
\mathit{ext\_ev\_decl} &:= \textbf{external } \mathit{ext\_ev}\ [\ \textbf{activated when } \mathit{env\_var}\ \mathit{rel\_op}\ \mathit{number}\ ]\\
&\quad [\ \textbf{read } (\mathit{assignment})\ ]\\
\mathit{int\_ev\_decl} &:= \textbf{internal } \mathit{int\_ev}\\
\mathit{ext\_evs} &:= \mathit{ext\_ev} \mid (\mathit{ext\_evs} \textbf{ or } \mathit{ext\_evs}) \mid (\mathit{ext\_evs} \textbf{ and } \mathit{ext\_evs})\\
\mathit{int\_evs} &:= \mathit{int\_ev} \mid (\mathit{int\_evs} \textbf{ or } \mathit{int\_evs}) \mid (\mathit{int\_evs} \textbf{ and } \mathit{int\_evs})\\
\mathit{condition} &:= (\mathit{condition} \textbf{ or } \mathit{condition}) \mid (\mathit{condition} \textbf{ and } \mathit{condition}) \mid\\
&\quad \textbf{not } \mathit{condition} \mid \mathit{exp}\ \mathit{rel\_op}\ \mathit{exp}\\
\mathit{action} &:= \textbf{increase } (\mathit{loc\_var},\ \mathit{exp}) \mid \textbf{decrease } (\mathit{loc\_var},\ \mathit{exp}) \mid\\
&\quad \textbf{set } (\mathit{loc\_var},\ \mathit{exp}) \mid \textbf{activate } (\mathit{int\_ev})\\
\mathit{actions} &:= \mathit{action} \mid (\mathit{actions} \textbf{ seq } \mathit{actions}) \mid (\mathit{actions} \textbf{ par } \mathit{actions})\\
\mathit{ext\_rule} &:= \textbf{on } \mathit{ext\_evs}\ [\textbf{if } \mathit{condition}]\ \textbf{do } \mathit{actions}\\
\mathit{int\_rule} &:= \textbf{on } \mathit{int\_evs}\ [\textbf{if } \mathit{condition}]\ \textbf{do } \mathit{actions}\ [\textbf{with priority } \mathit{number}]\\
\mathit{system} &:= [\mathit{env\_vars}]^{+}[\mathit{loc\_vars}]^{*}[\mathit{ext\_ev\_decl}]^{+}[\mathit{int\_ev\_decl}]^{*}\\
&\quad [\mathit{ext\_rule}]^{+}[\mathit{int\_rule}]^{*}
\end{aligned}
$$

Fig. 1. The syntax of ECA rules.

Environmental variables are used to represent environment states that can only be measured by sensors but not *directly* modified by the system. For instance, if we want to increase the temperature in a room, the system may choose to turn on a heater, eventually achieving the desired effect, but it cannot directly change the value of the temperature variable. Thus, environmental variables capture the nondeterminism introduced by the environment, beyond the control of the system. Instead, local variables can be both read and written by the system. These may be associated with an actuator, a record value, or an intermediate value describing part of the system state; we provide operations to set (absolute change) their value to an expression, or increase or decrease (relative change) it by an expression; these expressions may depend on environmental variables.

Events can be combinations of atomic events activated by environmental or internal changes. We classify them using the keywords **external** and **internal**. An external event can be **activated when** the value of an environmental variable crosses a threshold; at that time, it may take a snapshot of some environmental variables and **read** them into local variables to record their current values. Only the action of an ECA rule can instead **activate** internal events. Internal events are useful to express internal changes or required actions within the system.

These two types of events cannot be mixed within a single ECA rule. Thus, rules are *external* or *internal*, respectively. Then, we say that a state is *stable* if only external events can occur in it, *unstable* if actions of external or internal rules are being performed (including the activation of internal events, which may then trigger internal rules). The system is initially stable and, after some external events trigger one or more external rules, it transitions to unstable states where internal events may be activated, triggering further internal rules. When all actions complete, the system is again in a stable state, waiting for environmental changes that will eventually trigger external events.

The condition portion of an ECA rule is a boolean expression on the value of environmental and local variables; it can be omitted if it is the constant true.

The last portion of a rule specifies which actions must be performed, and in which order. Most actions are operations on local variables which do not directly affect environmental variables, but may cause some changes that will conceivably be reflected in their future values. Thus, all environmental variables are read-only from the perspective of an action. Actions can also **activate** internal events. Moreover, to handle complex action operations, the execution semantics can be specified as any partial order described by a series-parallel graph; this is obtained through an appropriate nesting of **seq** operators, to force a sequential execution, and **par** operators, to allow an arbitrary concurrency. The keyword **with priority** enforces a priority for internal rules. If no priority is specified, the default priority of an internal rule is 1, the same as that of external rules.

We now discuss the choices of execution semantics for our language, to support the modeling of reactive systems. The first choice is how to couple the checking of events and conditions for our ECA rules. There are (at least) two options: *immediate* and *deferred*. The event-condition checking is *immediate* if the corresponding condition is immediately evaluated when the events occur, it is *deferred* if the condition is evaluated at the end of a cycle with a predefined frequency. One critical requirement for the design of reactive systems is that the system should respond to external events from the environment [10] as soon as possible. Thus, we choose immediate event-condition checking: when events occur, the corresponding condition is immediately evaluated to determine whether to trigger the rule. We stress that deferred checking can still be modeled using immediate checking, for example by adding an extra variable for the system clock and changing priorities related to rule evaluation to synchronize rule evaluations. However, the drawback of deferred checking is that the design must tolerate false ECA rule triggering or non-triggering scenarios. Since there is a time gap between event activation and condition evaluation, the environmental conditions that trigger an event might change during this period of time, causing rules supposed to be triggered at the time of event activation to fail because the "current " condition evaluation are now inconsistent.

Another important choice is how to handle and model the concurrent and nondeterministic nature of reactive systems. We introduce the concept of *batch* for external events, similar to the concept of transaction in DBMSs. Formally, the boundary of a batch of external events is defined as the end of the execution

of all triggered rules. Then, the system starts to receive external events and immediately evaluates the corresponding conditions. The occurrence of an external event closes a batch if it triggers one or more ECA rules; otherwise, the event is added to the current batch. Once the batch closes and the rules to be triggered have been determined, the events in the current batch are cleaned-up, to prevent multiple (and erroneous) triggerings of rules. For example, consider ECA rules r_a: "**on** a **do** $\cdots$" and r_{ac}: "**on** (a **and** c) **do** $\cdots$", and assume that the system finishes processing the last batch of events and is ready to receive external events for the next batch. If external events occur in the sequence "$c, a, \ldots$", event c alone cannot trigger any rule so it begins, but does not complete, the current batch. Then, event a triggers both rule r_a and r_{ac}, and thus, closes the current batch. Both rules are triggered and will be executed concurrently. This example shows how, when the system is in a stable state, the occurrence of a single external event may trigger one or more ECA rules, since there is no "contention" within a batch on "using" an external event: rule r_a and r_{ac} share event a and both rules are triggered and executed. If instead the sequence of events is "$a, c, \ldots$", event a by itself constitutes a batch, as it triggers rule r_a. This event is then discarded by the clean-up so, after executing r_a and any internal rule (recursively) triggered by it, the system returns to a stable state and the subsequent events "$c, \ldots$" begin the next batch. Under this semantic, all external events in one *batch* are processed concurrently. Thus, unless there is a termination error, the system will process all triggered rules, including those triggered by the activation of internal events during the current batch, before considering new external events. This batch definition provides maximum nondeterminism on event order, which is useful to discover design errors in a set of ECA rules.

We also stress that, in our semantics, the system is frozen during rule execution and does not respond to external events. Thus, rule execution is instantaneous, while in reality it obviously requires some time. However, from a verification perspective, environmental changes and external event occurrences are nondeterministic and asynchronous, thus our semantic allows the verification process to explore all possible combinations without missing errors due to the order in which events occur and environmental variables change.

3.1 Running example

We now illustrate the expressiveness of our ECA rules on a running example: a light control subsystem in a smart home for senior housing. Fig. 2 lists the requirements in plain English (R_1 to R_5). Using motion and pressure sensors, the system attempts to reduce energy consumption by turning off the lights in unoccupied rooms or if the occupant is asleep. Passive sensors emit signals when an environmental variable value crosses a significant threshold. The motion sensor measure is expressed by the boolean environmental variable *Mtn*. The system also provides automatic adjustment for indoor light intensity based on an outdoor light sensor, whose measure is expressed by the environmental variable $ExtLgt \in \{0, ..., 10\}$. A pressure sensor detects whether the person is asleep and is expressed by the boolean environmental variable *Slp*.

environmental	*Mtn, ExtLgt, Slp*
local	*lMtn, lExtLgt, lSlp, lgtsTmr, intLgts*
external	*SecElp* **read** (*Mtn* **into** *lMtn*, *ExtLgt* **into** *lExtLgt*, *Slp* **into** *lSlp*) *MtnOn* **activated when** $Mtn = 1$ *MtnOff* **activated when** $Mtn = 0$ *ExtLgtLow* **activated when** $ExtLgt \leq 5$
internal	*LgtsOff, LgtsOn, ChkExtLgt, ChkMtn, ChkSlp*
(R_1) When the room is unoccupied for 6 minutes, turn off lights if they are on.	
r_1	**on** *MtnOff* **if** ($intLgts > 0$ **and** $lgtsTmr = 0$) **do set** (*lgtsTmr*, 1)
r_2	**on** *SecElp* **if** ($lgtsTmr \geq 1$ **and** $lMtn = 0$) **do increase** (*lgtsTmr*, 1)
r_3	**on** *SecElp* **if** ($lgtsTmr = 360$ **and** $lMtn = 0$) **do** (**set** (*lgtsTmr*, 0) **par activate** (*LgtsOff*))
r_4	**on** *LgtsOff* **do** (**set** (*intLgts*, 0) **par activate** (*ChkExtLgt*))
(R_2) When lights are off, if external light intensity is below 5, turn on lights.	
r_5	**on** *ChkExtLgt* **if** ($intLgts = 0$ **and** $lExtLgt \leq 5$) **do activate** (*LgtsOn*)
(R_3) When lights are on, if the room is empty or a person is asleep, turn off lights.	
r_6	**on** *LgtsOn* **do** (**set** (*intLgts*, 6) **seq activate** (*ChkMtn*))
r_7	**on** *ChkMtn* **if** ($lSlp = 1$ **or** ($lMtn = 0$ **and** $intLgts \geq 1$)) **do activate** (*LgtsOff*)
(R_4) If the external light intensity drops below 5, check if the person is asleep and set the lights intensity to 6. If the person is asleep, turn off the lights.	
r_8	**on** *ExtLgtLow* **do** (**set** (*intLgts*, 6) **par activate** (*ChkSlp*))
r_9	**on** *ChkSlp* **if** ($lSlp = 1$) **do set** (*intLgts*, 0)
(R_5) If the room is occupied, set the lights intensity to 4.	
r_{10}	**on** *MtnOn* **do** (**set** (*intLgts*, 4) **par set** (*lgtsTmr*, 0))

Fig. 2. ECA rules for the light control subsystem of a smart home.

MtnOn, *MtnOff*, and *ExtLgtLow* are external events activated by the environmental variables discussed above. *MtnOn* and *MtnOff* occur when *Mtn* changes from 0 to 1 or from 1 to 0, respectively. *ExtLgtLow* occurs when *ExtLgt* drops below 6. External event *SecElp* models the system clock, occurs every second, and takes a snapshot of the environmental variables into local variables *lMtn*, *lExtLgt*, and *lSlp*, respectively. Additional local variables *lgtsTmr* and *intLgts* are used. Variable *lgtsTmr* is a timer for R_1, to convert the continuous condition "the room is unoccupied for 6 minutes" into 360 discretized *SecElps* events. Rule r_1 initializes *lgtsTmr* to 1 whenever the motion sensor detects no motion and the lights are on. The timer then increases as second elapses, provided that no motion is detected (rule r_2). If the timer reaches 360, internal event *LgtsOff* is activated to turn off the lights and to reset *lgtsTmr* to 0 (rule r_3). Variable *intLgts* acts as an actuator control to adjust the internal light intensity.

Our ECA rules contain internal events to model internal system actions or checks not observable from the outside. *LgtsOff*, activated by rule r_3 or r_7, turns the lights off and activates another check on outdoor light intensity through internal event *ChkExtLgt* (rule r_4). *ChkExtLgt* activates *LgtsOn* if $lExtLgt \leq 5$

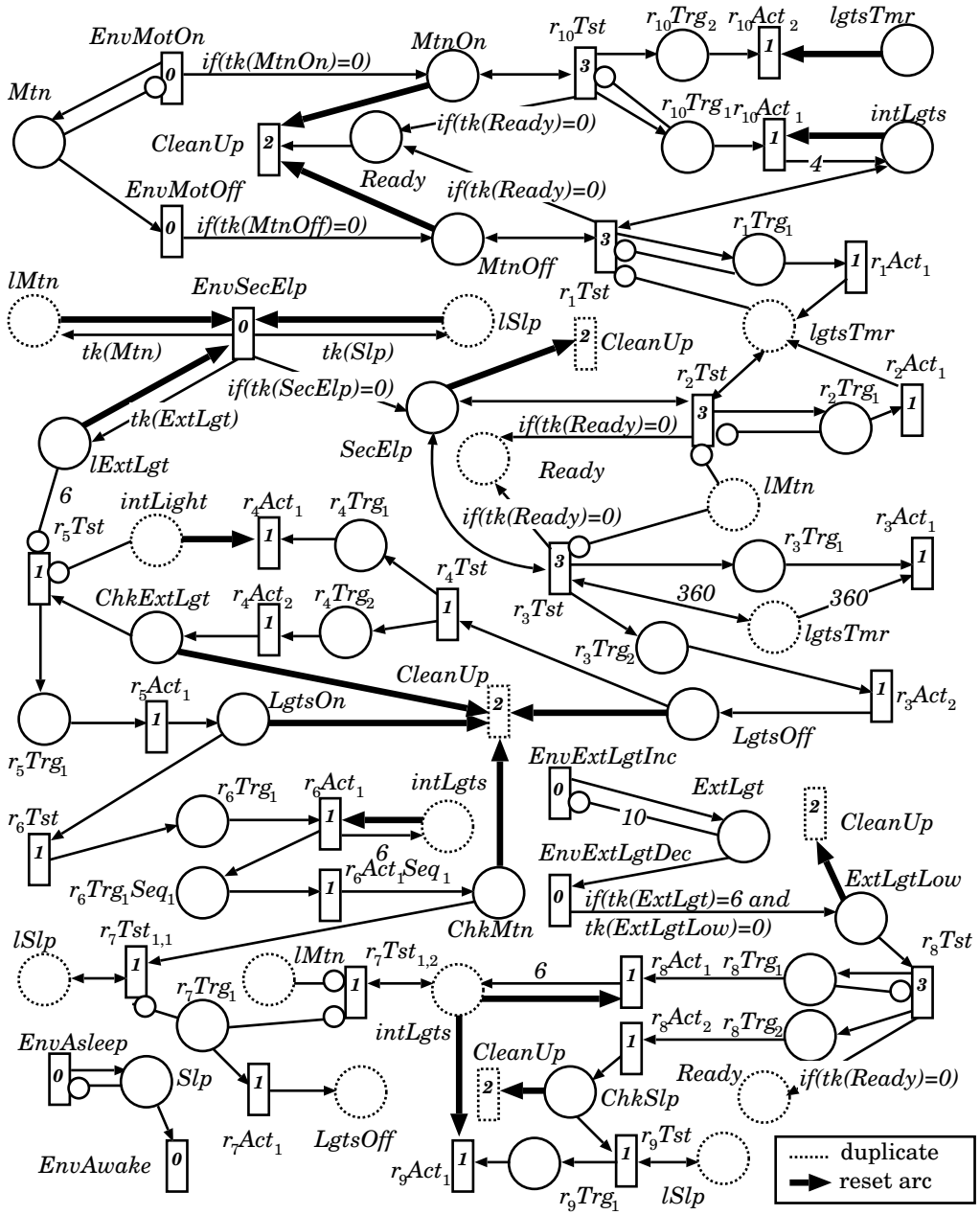


Fig. 3. The PN for ECA rules in Fig. 2.

(rule r_5). *ChkSlp* is activated by rule r_8 to check whether a person is asleep. If true, the event triggers an action that turns the lights off (rule r_9). Internal event *ChkMtn*, activated by rule r_6, activates *LgtsOff* if the room is unoccupied and all lights are on, or if the room is occupied but the occupant is asleep (rule r_7).

4 Transforming a set of ECA rules into a PN

We now explain the procedure to transform a set of ECA rules into a PN. First, we put each ECA rule into a *regular* form where both *events* and *condition* are

disjunctions of conjunctions of events and relational expressions, respectively. All rules in Fig. 2 are in this form. While this transformation may in principle cause the expressions for events and conditions to grow exponentially large, each ECA rule usually contains a small number of events and conditions, hence this is not a problem in practice. Based on the immediate event-condition checking assumption, a rule is triggered iff "$trigger \equiv events \wedge condition$" holds.

Next, we map variables and events into places, and use PN transitions to model event testing, condition evaluation, and action execution. Any change of variable values is achieved through input and output arcs with appropriate cardinalities. Additional control places and transitions allow the PN behavior to be organized into "phases", as shown next. ECA rules r_1 through r_{10} of Fig. 2 are transformed into the PN of Fig. 3 (dotted transitions and places are duplicated and arcs labeled "$if(cond)$" are present only if $cond$ holds).

4.1 Occurring phase

This phase models the occurrence of external events, due to environment changes over which the system has no control. The PN firing semantics perfectly matches the nondeterministic asynchronous nature of these changes. For example, in Fig. 3, transitions *EnvMotOn* and *EnvMotOff* can add or remove the token in place *Mtn*, to nondeterministically model the presence or absence of people in the room (the inhibitor arc from place *Mtn* back to transition *EnvMotOn* ensures that at most one token resides in *Mtn*). Firing these environmental transitions might nondeterministically enable the corresponding external events. Here, firing *EnvMotOn* generates the external event *MtnOn* by placing a token in the place of the same name, while firing *EnvMotOff* generates event *MtnOff*, consistent with the change in *Mtn*. To ensure that environmental transitions only fire if the system is in a stable state (when no rule is being processed) we assign the lowest priority, 0, to these transitions. As the system does not directly affect environmental variables, rule execution does not modify them. However, we can take snapshots of these variables by copying the current number of tokens into their corresponding local variables using marking-dependent arcs. For example, transition *EnvSecElp* has an output arc to generate event *SecElp*, and arcs connected to local variables to perform the snapshots, e.g., all tokens in *lMtn* are removed by a reset arc (an input arc that removes all tokens from its place), while the output arc with cardinality $tk(Mtn)$ copies the value of *Mtn* into *lMtn*.

4.2 Triggering phase

This phase starts when $trigger \equiv events \wedge condition$ holds for at least one external ECA rule. If, for rule r_k, *events* and *condition* consist of n_d and n_c disjuncts, respectively, we define $n_d \cdot n_c$ test transitions $r_k Tst_{i,j}$ with priority $P+2$, where i and j are the index of a conjunct in *events* and one in *condition*, while $P \geq 1$ is the highest priority used for internal rules (in our example, all internal rules have default priority $P = 1$). Then, to trigger rule r_k, only one of these transitions, e.g., $r_7 Tst_{1,1}$ or $r_7 Tst_{1,2}$, needs to be fired (we omit i and j if $n_d = n_c = 1$). Firing

a test transition means that the corresponding events and conditions are satisfied and results in placing a token in each of the *triggered* places $r_k\,Trg_1, \ldots, r_k\,Trg_N$, to indicate that Rule r_k is triggered, where N is the number of outermost parallel actions (recall that **par** and **seq** model parallel and sequential actions). Thus, $N = 1$ if r_k contains only one action, or an outermost sequential series of actions. Inhibitor arcs from $r_k\,Trg_1$ to test transitions $r_k Tst_{i,j}$ ensure that, even if multiple conjuncts are satisfied, only one test transition fires. The firing of test transitions does not "consume" external events, thus we use double-headed arrows between them. This allows one batch to trigger multiple rules, conceptually "at the same time". After all enabled test transitions for external rules have fired, place *Ready* contains one token, indicating that the current batch of external events can be cleared: transition *CleanUp*, with priority $P+1$, fires and removes all tokens from external and internal event places using reset arcs, since all the rules that can be triggered have been marked. This ends the triggering phase and closes the current batch of events.

4.3 Performing phase

This phase executes all actions of external rules marked in the previous phase. It may further result in triggering and executing internal rules. Transitions in this phase correspond to the *actions* of rules with priority in $[1, P]$, the same as that of the corresponding rule. An action activates an internal event by adding a token to its place. This token is consumed as soon as a test transition of any internal rule related to this event fires. This is different from the way external rules "use" external events. Internal events not consumed in this phase are cleared when transition *CleanUp* fires in the next batch. When all enabled transitions of the performing phase have fired, the system is in a stable state where environmental changes (transitions with priority 0) can again happen and the next batch starts.

4.4 ECA rules to PN translation

The algorithm in Fig. 4 takes external and internal ECA rules $\mathbf{R}_{ext}$, $\mathbf{R}_{int}$, with priorities in $[1, P]$, environmental and local variables $\mathbf{V}_{env}$, $\mathbf{V}_{loc}$, and external and internal events $\mathbf{E}_{ext}$, $\mathbf{E}_{int}$, and generates a PN. After normalizing the rules and setting P to the highest priority among the rule priorities in $\mathbf{R}_{int}$, it maps environmental variables $\mathbf{V}_{env}$, local variables $\mathbf{V}_{loc}$, external events $\mathbf{E}_{ext}$, and internal events $\mathbf{E}_{int}$, into the corresponding places (Lines 5, 6, and 8). Then, it creates phase control place *Ready*, transition *CleanUp*, and reset arcs for *CleanUp* (Lines 4-5). We use arcs with marking-dependent cardinalities to model expressions. For example, together with inhibitor arcs, these arcs ensure that each variable $v \in \mathbf{V}_{env}$ remains in its range $[v_{min}, v_{max}]$ (Lines 8-10). These arcs also model the **activated when** portion of external events (Line 17), rule conditions (Line 33), and assignments of environmental variables to local variables (Lines 19-20 and lines 14-15). The algorithm models external events and environmental changes (Lines 11-24); it connects environmental transitions such as t_{vInc} and t_{vDec} to their corresponding external event places, if any, with an arc

TransformECAintoPN $(\mathbf{R}_{ext}, \mathbf{R}_{int}, \mathbf{V}_{env}, \mathbf{V}_{loc}, \mathbf{E}_{ext}, \mathbf{E}_{int})$

1 normalize $\mathbf{R}_{ext}$ and $\mathbf{R}_{int}$ into regular form and set P to the highest rule priority
2 create a place $Ready$ • *to control "phases" of the net*
3 create transition $CleanUp$ with priority $P+1$ and $Ready$ $-[1]\rightarrow$ $CleanUp$
4 **for each** event $e \in \mathbf{E}_{ext} \cup \mathbf{E}_{int}$ **do**
5 create place p_e and p_e $-[tk(p_e)]\rightarrow$ $CleanUp$
6 create place p_v, for each variable $v \in \mathbf{V}_{loc}$
7 **for each** variable $v \in \mathbf{V}_{env}$ with range $[v_{min}, v_{max}]$ **do**
8 create place p_v and transitions t_{vInc} and t_{vDec} with priority 0
9 create t_{vDec} $-[\text{if}(tk(p_v) > v_{min})1 \text{ else } 0]\rightarrow$ p_v
10 create t_{vInc} $-[1]\rightarrow$ p_v and p_v $-[v_{max}]\multimap$ t_{vInc}
11 **for each** event $e \in \mathbf{E}_{ext}$ **activated when** v op val, for $op \in \{\geq \mid =\}$ **do**
12 create t_{vInc} $-[\text{if}(tk(p_v)$ op $val)1 \text{ else } 0]\rightarrow$ p_e
13 **if** e reads $v \in \mathbf{V}_{env}$ into $v' \in \mathbf{V}_{loc}$ **then**
14 create $p_{v'}$ $-[\text{if}(tk(p_v)$ op $val)tk(p_{v'}) \text{ else } 0]\rightarrow$ t_{vInc}
15 create t_{vInc} $-[\text{if}(tk(p_v)$ op $val)tk(p_v) \text{ else } 0]\rightarrow$ $p_{v'}$
16 **for each** event $e \in \mathbf{E}_{ext}$ **activated when** v op val, for $op \in \{\leq \mid =\}$ **do**
17 create t_{vDec} $-[\text{if}(tk(p_v)$ op $val)1 \text{ else } 0]\rightarrow$ p_e
18 **if** e reads $v \in \mathbf{V}_{env}$ into $v' \in \mathbf{V}_{loc}$ **then**
19 create $p_{v'}$ $-[\text{if}(tk(p_v)$ op $val)tk(p_{v'}) \text{ else } 0]\rightarrow$ t_{vDec}
20 create t_{vDec} $-[\text{if}(tk(p_v)$ op $val)tk(p_v) \text{ else } 0]\rightarrow$ $p_{v'}$
21 **for each** event $e \in \mathbf{E}_{ext}$ without an **activated when** portion **do**
22 create t_e and t_e $-[\text{if}(tk(p_e) = 0)1 \text{ else } 0]\rightarrow$ p_e
23 **if** e reads $v \in \mathbf{V}_{env}$ into $v' \in \mathbf{V}_{loc}$ **then**
24 create $p_{v'}$ $-[tk(p_{v'})]\rightarrow$ t_e and t_e $-[tk(p_v)]\rightarrow$ $p_{v'}$
25 **for each** rule $r_k \in \mathbf{R}_{ext} \cup \mathbf{R}_{int}$ with n_d event disjuncts, n_c condition disjuncts, actions A, and priority $p \in [1, P]$ **do**
26 create trans. $r_k\, Tst_{i,j}, i \in [1, n_d], j \in [1, n_c]$,w/priority $P+2$ if $r_k \in \mathbf{R}_{ext}$, else p
27 **for each** event e in disjunct i **do**
28 create p_e $-[1]\rightarrow$ $r_k\, Tst_{i,j}$
29 create $r_k\, Tst_{i,j}$ $-[1]\rightarrow$ p_e, if $e \in \mathbf{E}_{ext}$
30 **for each** conjunct $v \leq val$ or $v = val$ in disjunct j **do**
31 create p_v $-[val + 1]\multimap$ $r_k\, Tst_{i,j}$
32 **for each** conjunct $v \geq val$ or $v = val$ in disjunct j **do**
33 create p_v $-[val]\rightarrow$ $r_k\, Tst_{i,j}$ and $r_k\, Tst_{i,j}$ $-[val]\rightarrow$ p_v
34 **if** actions A is "$(A_1$ *par* $A_2)$" **then** $n_a = 2$ **else** $n_a = 1$, $A_1 = A$
35 **for each** $l \in [1, n_a]$ **do**
36 create places $r_k\, Trg_l$ and transitions $r_k\, Act_l$ with priority p
37 create $r_k\, Trg_l$ $-[1]\rightarrow$ $r_k\, Act_l$ and $r_k\, Tst_{i,j}$ $-[1]\rightarrow$ $r_k\, Trg_l$
38 $SeqSubGraph(A_l, \text{"}r_k\, Acl_l\text{"}, l, p)$
39 **for each** $r_k \in \mathbf{R}_{ext}, i \in [1, n_d], j \in [1, n_c]$ **do**
40 create $r_k\, Tst_{i,j}$ $-[\text{if}(tk(Ready) = 0)1 \text{ else } 0]\rightarrow$ $Ready$ and $r_k\, Trg_1$ $-[1]\multimap$ $r_k\, Tst_{i,j}$

Fig. 4. Transforming ECA rules into a PN: a $-[k]\rightarrow$ b means "an arc from a to b with cardinality k; a $-[k]\multimap$ b means "an inhibitor arc from a to b with cardinality k.

whose cardinality evaluates to 1 if the corresponding condition becomes true upon the firing of the transition and the event place does not contain a token already, 0 otherwise (e.g., the arcs from *EnvExtLigDec* to *ExtLgtLow*).

Next, rules are considered (Lines 25-40). A rule with n_d event disjuncts and n_c condition disjuncts generates $n_d \cdot n_c$ testing transitions. To model the parallel-sequential action graph of a rule, we use mutually recursive procedures (Fig. 5 and Fig. 6). Procedure *SeqSubGraph* first tests all atomic actions, such as "set",

```
ParSubGraph(Pars, Pre, p)                    • Pars: parallel actions, Pre: prefix
 1  for each l ∈ {1, 2} do                   • according to the syntax Pars_2 has two components
 2    create place PreAct_l Trg_l Seq_l and transition PreAct_l w/priority p
 3    create Pre –[1]→ PreAct_l and PreAct_l –[1]→ PreAct_l Trg_l Seq_l
 4    SeqSubGraph(Pars_l, "PreAct_l Trg_l Seq_l", l, p);
```

Fig. 5. Processing **par**.

```
SeqSubGraph(Seqs, Pre, i, p)                 • Seqs: sequential actions, Pre: prefix
 1  if Seqs sets variable v to val then
 2    create p_v –[tk(p_v)]→ Pre and Pre –[val]→ p_v
 3  else if Seqs increases variable v by val then
 4    create Pre –[val]→ p_v
 5  else if Seqs decreases variable v by val then
 6    create p_v –[val]→ Pre
 7  else if Seqs activates an internal event e then
 8    create Pre –[1]→ p_e
 9  else if the outermost operator of Seqs is par then
10    ParSubGraph(Seqs, "Pre", p)                    • Recursion on parallel part
11  else if the outermost operator of Seqs is seq then
12    SeqSubGraph(Seqs_1, "Pre", 1, p)               • Seqs_1 is the first part of Seq
13    create place PreTrg_i Seq_1 and transition PreAct_i Seq_1
14    create Pre –[1]→ PreTrg_i Seq_1 and PreTrg_i Seq_1 –[1]→ PreAct_i Seq_1
15    SeqSubGraph(Seqs_2, "PreTrg_i Seq_1", 2, p)    • Seqs_2 is the second part of Seq
```

Fig. 6. Processing **seq**.

"increase", "decrease", and "activate". Then, it recursively calls *ParSubGraph* at Line 10 if it encounters parallel actions. Otherwise, it calls itself to unwind another layer of sequential actions at Line 12 and Line 15 for the two portions of the sequence. Procedure *ParSubGraph* creates control places and transitions for the two branches of a parallel action and calls *SeqSubGraph* at Line 4.

5 Verifying properties

The first step towards verifying correctness properties is to define $\mathcal{S}_{init}$, the set of initial states, corresponding to all the possible initial combinations of system variables (e.g., *ExtLgt* can initially have any value in $[0, 10]$). One could consider all these possible values by enumerating all legal stable states corresponding to possible initial combinations of the environmental variables, then start the analysis from each of these states, one at a time. However, in addition to requiring the user to explicitly provide the set of initial states, this approach may require enormous runtime, also because many computations are repeated in different runs. Our approach instead computes the initial states symbolically, thanks to the nondeterministic semantics of Petri nets, so that the analysis is performed once starting from a single, but very large, set $\mathcal{S}_{init}$.

To this end, we add an initialization phase that puts a nondeterministically chosen legal number of tokens in each place corresponding to an environmental variable. This phase is described by a subnet consisting of a transition *InitEnd*

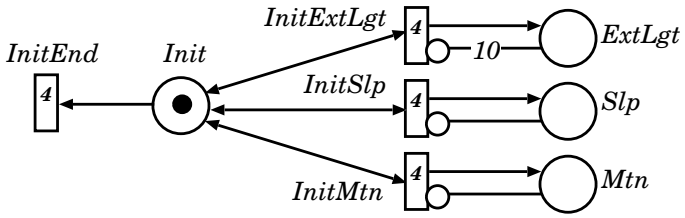


Fig. 7. The initialization phase for the smart home example.

with priority $P+3$, a place *Init* with one initial token, and an initializing transition with priority $P+3$ for every environmental variable, to initialize the number of tokens in the corresponding place. Fig. 7 shows this subnet for our running example. We initialize the Petri net by assigning the minimum number of tokens to every environmental variable place and leaving all other places empty, then we let the initializing transitions nondeterministically add a token at a time, possibly up to the maximum legal number of tokens in each corresponding place. When *InitEnd* fires, it disables the initializing transitions, freezes the nondeterministic choices, and starts the system's normal execution.

This builds the set of initial states, ensuring that the PN will explore all possible initial states, and avoids the overhead of manually starting the PN from one legal initial marking at a time. Even though the overall state space might be larger (it equals the union of all the state spaces that would be built starting from each individual marking), this is normally not the case, and, anyway, having to perform just one state space generation is obviously enormously better.

After the initialization step, we proceed with verifying termination and confluence using our tool SMART, which provides symbolic reachability analysis and CTL model checking with counterexample generation [7].

5.1 Termination

Reactive systems constantly respond to external events. However, if the system has a *livelock*, a finite number of external events can trigger an infinite number of rule executions (i.e, activate a cycle of internal events), causing the system to remain "busy" internally, a fatal design error. When generating the state space, all legal batches of events are considered, due to the PN execution semantics, again avoiding the need for an explicit enumeration, this time, of event batches.

> A set $\mathcal{G}$ of ECA rules satisfies *termination* if no infinite sequence of internal events can be triggered in any possible execution of $\mathcal{G}$. This can be expressed in CTL as $\neg\mathsf{EF}(\mathsf{EG}(\mathit{unstable}))$, stating that there is no cycle of unstable states reachable from an initial, thus stable, state.

Both traditional breadth-first-search (BFS) and saturation-based [19] algorithms are suitable to compute the EG operator. Fig. 8 uses saturation, which tends to perform much better in both time and memory consumption when analyzing large asynchronous systems. We encode transitions related to external

```
bool Term(mdd S_init, mdd2 N_int)
 1  mdd S_rch ← StateSpaceGen(S_init, N_ext ∪ N_int);
 2  mdd S_unst ← Intersection(S_rch, ExtractUnprimed(N_int));
 3  mdd S_p ← EF(EG(S_unst));
 4  if S_p ≠ ∅ then return false                          • provide error trace
 5  else return true;

mdd ExtractUnprimed(mdd2 p)                              • p unprimed
 1  if p = 1 then return 1;
 2  if CacheLookUp(EXTRACTUNPRIMED, p, r) return r;
 3  foreach i ∈ V_p.v do
 4    mdd r_i ← 0;
 5    if p[i] ≠ 0 then                  • p[i] is the node pointed edge i of node p
 6      foreach j ∈ V_p.v s.t. p[i][j] ≠ 0 do
 7        r_i ← Union(r_i, ExtractUnprimed(p[i][j]))
 8  mdd r ← UniqueTableInsert({r_i : i ∈ V_p.v});
 9  CacheInsert(EXTRACTUNPRIMED, p, r);
10  return r;
```

Fig. 8. Algorithms to verify the termination property.

events and environmental variable changes into $\mathcal{N}_{ext}$. Thus, the internal transitions are $\mathcal{N}_{int} = \mathcal{N} \backslash \mathcal{N}_{ext}$. After generating the state space $\mathcal{S}_{rch}$ using constrained saturation [18], we build the set of states $\mathcal{S}_{unst}$ by symbolically intersecting $\mathcal{S}_{rch}$ with the unprimed, or "from", states extracted from $\mathcal{N}_{int}$. Then, we use the CTL operators EG and EF to identify any nonterminating path (i.e., cycle).

5.2 Confluence

Confluence is another desirable property to ensure consistency in systems exhibiting highly concurrent behavior.

> A set $\mathcal{G}$ of ECA rules satisfying termination also satisfies confluence if, for any *legal* batch b of external events and starting from any particular stable state s, the system eventually reaches a *unique* stable state.

We stress that what constitutes a legal batch b of events depends on state s, since the condition portion of one or more rules might affect whether b (or a subset of b) can trigger a rule (thus close a batch). Given a legal batch b occurring in stable state s, the system satisfies *confluence* if it progresses from s by traversing some (nondeterministically chosen) sequence of unstable states, eventually reaching a stable state uniquely determined by b and s. Checking confluence is therefore expensive [2], as it requires verifying the combinations of all stable states reachable from $\mathcal{S}_{init}$ with all legal batches of external events when the system is in that stable state. A straightforward approach enumerates all legal batches of events for each stable state, runs the model, and checks that the set of reachable stable states has cardinality one. We instead only check that, from each reachable unstable state, exactly one stable state is reachable; this avoids enumerating all legal batches of events for each stable state. Since

```
bool ConfExplicit(mdd S_st, mdd S_unst, mdd2 N_int)
 1  foreach i ∈ S_unst
 2    mdd S_i ← StateSpaceGen(i, N_int);
 3    if Cardinality(Intersection(S_i, S_st)) > 1 then return false; • provide error trace
 4  return true;
```

```
bool ConfExplicitImproved(mdd S_st, mdd S_unst, mdd2 N_int, mdd2 N)
 1  mdd S_frontier ← Intersection(RelProd(S_st, N), S_unst);
 2  while S_frontier ≠ ∅ do                  • if S_frontier is empty, it explores all S_unst
 3    pick i ∈ S_frontier;
 4    mdd S_i ← StateSpaceGen(i, N_int);
 5    if Cardinality(Intersection(S_i, S_st)) > 1 then return false; • provide error trace
 6    else S_frontier ← S_frontier \ Intersection(S_i, S_unst);   • exclude all unstable
      states reached by i
 7  return true;
```

Fig. 9. Explicit algorithms to verify the confluence property.

```
bool ConfSymbolic(mdd S_st, mdd S_unst, mdd2 N_int)
 1  mdd2 TC ← ConstraintedTransitiveClosure(N_int, S_unst);
 2  mdd2 TC_u2s ← FilterPrimed(TC, S_st);
 3  return CheckConf(TC_u2s);
```

```
bool CheckConf(mdd2 p)
 1  if p = 1 then return true
 2  if CacheLookUp(CHECKCONF, p, r) return r;
 3  foreach i ∈ V_p.v, s.t. exist j, j' ∈ V_p.v, j ≠ j', p[i][j] ≠ 0, p[i[j'] ≠ 0 do
 4    foreach j, j' ∈ V_p.v, j ≠ j' s.t. p[i][j] ≠ 0, p[i[j'] ≠ 0 do
 5      if p[i][j] = p[i][j'] return false;          • Confluence does not hold
 6      mdd f_j ← ExtractUnprimed(p[i][j]);            • Result will be cached
 7      mdd f_j' ← ExtractUnprimed(p[i][j']);        • No duplicate computation
 8      if Intersection(f_i, f_j') ≠ 0 then return false;
 9  foreach i, j ∈ V_p.v s.t. p[i][j] ≠ 0 do
10    if CheckConf(p[i][j]) = false return false;
11  CacheInsert(CHECKCONF, p, true);
12  return true;
```

Fig. 10. Fully symbolic algorithm to verify the confluence property.

nondeterministic execution in performing phase is the main reason to violate confluence and the system is in unstable states in our definition, checking the evolution starting from unstable states will fulfill the purpose.

The brute force algorithm *ConfExplicit* in Fig. 9 enumerates unstable states and generates reachable states only from unstable states using constrained saturation [18]. Then, it counts the stable states in the obtained set. We observe that, starting from an unstable stable u, the system may traverse a large set of unstable states before reaching a stable state. If unstable state u is reachable, so are the unstable states reachable from it. Thus, the improved version *ConfExplicitImproved* first picks an unstable state $\mathbf{i}$ and, after generating the states reachable from $\mathbf{i}$ and verifying that they include only one stable state, it excludes all visited unstable states (Line 6). Furthermore, it starts only from states $\mathbf{i}$ in the *frontier*, i.e., unstable states reachable in one step from stable

Termination (time: sec, memory: MB)					
Model	$\lvert\mathcal{S}_{rch}\rvert$	T_t	T_c	M_p	M_f
PN_t	$2.66{\cdot}10^6$	0.009	9.665	358.68	88.36
PN_c	$2.61{\cdot}10^6$	0.005	9.497	344.42	87.88
PN_1	$8.99{\cdot}10^6$	0.010	11.559	391.52	89.78
PN_2	$1.78{\cdot}10^7$	0.010	24.477	673.33	158.66
PN_3	$2.61{\cdot}10^7$	0.010	85.171	1686.46	559.52
PN_4	$5.02{\cdot}10^7$	0.010	14.541	491.80	105.90

Confluence (time: min, memory: GB, –: out of memory)							
Model	$\lvert\mathcal{S}_{rch}\rvert$	Best Explicit			Symbolic		
		T_{be}	M_p	M_f	T_s	M_p	M_f
PN_c	$2.38{\cdot}10^6$	4.51	4.24	4.10	5.11	2.02	0.22
PN_1	$8.12{\cdot}10^6$	40.25	14.53	14.33	6.40	2.31	0.27
PN_2	$1.61{\cdot}10^7$	34.40	0.85	0.08	10.11	2.59	0.25
PN_3	$2.33{\cdot}10^7$	>120.00	–	–	60.09	2.59	0.25
PN_4	$4.55{\cdot}10^7$	>120.00	–	–	23.33	4.66	0.52

Table 1. Results to verify the ECA rules for a smart home.

states (all other unstable reachable states are by definition reachable from this frontier). However, we stress that these, as most symbolic algorithms, are heuristics, thus they are not guaranteed to work better than the simpler approaches.

Next, we introduce a fully symbolic algorithm to check confluence in Fig. 10. It first generates the transition transitive closure (TC) set from $\mathcal{N}_{int}$ using constrained saturation [19], where the "from"' states of the closure are in $\mathcal{S}_{unst}$ (Line 1). The resulting set encodes the reachability relation from any reachable unstable state without going through any stable state. Then, it filters this relation to obtain the relation from reachable unstable states to stable states by constraining the "to" states to set $\mathcal{S}_{st}$. Thus, checking confluence reduces to verifying whether there exist two different pairs $(\mathbf{i}, \mathbf{j})$ and $(\mathbf{i}, \mathbf{j}')$ in the relation: Procedure *CheckConf* implements this check symbolically. While computing TC is a an expensive operation [19], this approach avoids separate searches from distinct unstable states, thus is particularly appropriate when $\mathcal{S}_{unst}$ is huge.

6 Experimental results

Table 1 reports results for a set of models run on an Intel Xeon 2.53GHz workstation with 36GB RAM under Linux. For each model, it shows the state space size ($\lvert\mathcal{S}_{rch}\rvert$), the peak memory ($M_p$), and the final memory (M_f). For termination, it shows the time used to verify the property (T_t) and to find the *shortest*

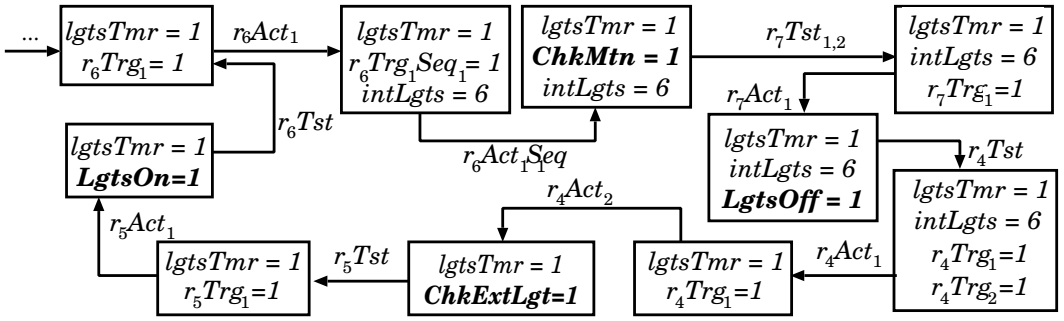


Fig. 11. A termination counterexample (related to rules r_4 to r_7).

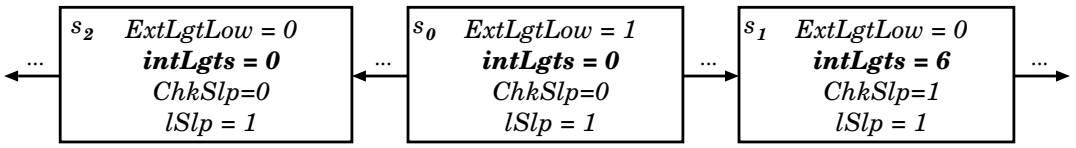


Fig. 12. A confluence counterexample (related to rules r_8 and r_9).

counterexample (T_c). For confluence, it reports the best runtime between our two explicit algorithms (T_{be}) and for our symbolic algorithm (T_s). Memory consumption accounts for both decision diagrams and operation caches.

Net PN_t is the model corresponding to our running example, and fails the termination check. Even though the state space is not very large, counterexample generation is computationally expensive [20] and consumes most of the runtime. The shortest counterexample generated by SMART has a long tail consisting of 1885 states and leads to the 10-state cycle of Fig. 11 (only the non-empty places are listed for each state, and edges are labeled with the corresponding PN transition). Analyzing the trace, we can clearly see (in bold) that, when lights are about to be turned off due to the timeout, $lMtn = 0$, and the external light is low, $ExtLgt \leq 5$, the infinite sequence of internal events $(LgtsOff, ChkExtLgt, LgtsOn, ChkMtn)^\omega$ prevents the system from terminating. Thus, rules r_4, r_5, r_6, and r_7 need to be investigated to fix the error. Among the possible modifications, we choose to replace rule r_5 with r'_5 : **on** *ChkExtLgt* **if** ((*intLgts*= 0 **and** *lExtLgt* $\leq$ 5) **and** *lMtn*= 1) **do activate** (*LgtsOn*), resulting in the addition of an input arc from *lMtn* to r_5 *Tst*. The new corrected model is called PN_c in Table 1, and SMART verifies it holds the termination property.

We then run SMART on PN_c to verify confluence, and found 72,644 bad states. Fig. 12 shows one of these unstable states, s_0, reaching two stables states, s_1 and s_2. External event *ExtLgtLow* closes the batch in s_0 and triggers rule r_8, which sets *intLgt* to 6 and activates internal event *ChkSlp*, which in turn sets *intLgt* to 0 (we omit intermediate unstable states from s_0 to s_1 and to s_2). We correct rules r_8 and r_9, replacing them with r'_8: **on** *ExtLgtLow* **if** *lSlp*=0 **do set** (*intLgt*, 6) and r'_9: **on** *ExtLgtLow* **if** *lSlp*=1 **do set** (*intLgt*, 0); resulting in model PN_{fc}. Checking this new model against for confluence, we find that the number of bad states decreases from 72,644 to 24,420. After investigation, we determine that the remaining problem is related to rules r_2 and r_3. After

changing rule r_2 to **on** *SecElp* **if** ((*lgtsTmr* ≥ 1 **and** *lgtsTmr* ≤ 359) **and** *lMtn* $= 0$) **do increase** (*lgtsTmr*, 1), the model passes the check. This demonstrates the effectiveness of counterexamples to help a designer debug a set of ECA rules.

We then turn our attention to larger models, which extend our original model by introducing four additional rules and increasing variable ranges. In PN_1 and PN_2, the external light variable *ExtLgt* ranges in $[0, 20]$ instead of $[0, 10]$; for PN_4, it ranges in $[0, 50]$. PN_2 also extends the range of the light timer variable *lgtTmr* to $[0, 720]$; PN_3 to $[0, 3600]$. We observe that, when verifying termination or confluence, the time and memory consumption tends to increase as the model grows; also, our symbolic algorithm scales much better than the best explicit approach when verifying confluence. For the relatively small state space of PN_t, enumeration is effective, since computing TC is quite computationally expensive. However, as the state space grows, enumerating the unstable states consumes excessive resources. We also observe that the supposedly improved explicit confluence algorithm sometimes makes things worse. The reason may lie in the fact that a random selection of a state from the frontier has different statistical properties than for the original explicit approach, and also in the fact that operation caches save many intermediate results. However, both explicit algorithms run out of memory on PN_3 and PN_4. Comparing the results for PN_3 and PN_4, we also observe that larger state spaces might require less resources. With symbolic encodings, this might happen because the corresponding MDD is more regular than the one for a smaller state space.

7 Conclusion

Verifying termination and confluence of ECA rule bases for reactive systems is challenging due to their highly concurrent and nondeterministic nature. We proposed an approach to verify these properties using a self-modifying PN with inhibitor arcs and priorities. Our approach is general enough to give precise answers to questions about other properties, certainly those that can be expressed in CTL. As an application, we showed how a light control system can be captured by our approach, and we verified termination and confluence for this model using SMART. In the future, we would like to improve our approach in the following ways. The confluence algorithm must perform constrained state space generation starting from each unstable state, which is not efficient if $\mathcal{S}_{unst}$ is large. In that case, a simulation-based falsification approach might be more suitable, using intelligent heuristic sampling and searching strategies. However, this approach is sound only if the entire set $\mathcal{S}_{unst}$ is explored. Another direction to extend our work is the inclusion of abstraction techniques to reduce the size of the state space.

Acknowledgement
Work supported in part by UC-MEXUS under grant *Verification of active rule bases using timed Petri nets* and by the National Science Foundation under grant CCF-1018057.

References

1. D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.
2. A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of database production rules: termination, confluence, and observable determinism. *Proc. ACM SIGMOD*, pp. 59–68. ACM Press, 1992.
3. J. C. Augusto and C. D. Nugent. A new architecture for smart homes based on ADB and temporal reasoning. *Toward a Human-Friendly Assistive Environment*, vol. 14, pp. 106–113, 2004.
4. E. Baralis, S. Ceri, and S. Paraboschi. Improved rule analysis by means of triggering and activation graphs. *Rules in Database Systems*, LNCS 985, pp. 163–181. 1995.
5. E. Baralis and J. Widom. An algebraic approach to static analysis of active database rules. *ACM TODS*, 25(3):269–332, Sept. 2000.
6. E.-H. Choi, T. Tsuchiya, and T. Kikuno. Model checking active database rules under various rule processing strategies. *IPSJ Digital Courier*, 2:826–839, 2006.
7. G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. *Proc. Modelling Techniques and Tools for Computer Performance Evaluation*, LNCS 2794, pp. 78–97. 2003.
8. S. Comai and L. Tanca. Termination and confluence by rule prioritization. *IEEE TKDE*, 15:257–270, 2003.
9. K. G. Kulkarni, N. M. Mattos, and R. Cochrane. Active database features in SQL3. *Active Rules in Database Systems*, pp. 197–219. Springer, New York, 1999.
10. E. A. Lee. Computing foundations and practice for cyber-physical systems: A preliminary report. Technical Report UCB/EECS-2007-72, UC Berkeley, May 2007.
11. X. Li, J. M. Marín, and S. V. Chapa. A structural model of ECA rules in active database. *Proc. Second Mexican International Conference on Artificial Intelligence: Advances in Artificial Intelligence*, pp. 486–493. Springer, 2002.
12. D. McCarthy and U. Dayal. The architecture of an active database management system. *ACM SIGMOD Record*, 18(2):215–224, 1989.
13. T. Murata. Petri nets: properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–579, Apr. 1989.
14. D. Nazareth. Investigating the applicability of Petri nets for rule-based system verification. *IEEE TKDE*, 5(3):402–415, 1993.
15. I. Ray and I. Ray. Detecting termination of active database rules using symbolic model checking. *Proc. East European Conference on Advances in Databases and Information Systems*, pp. 266–279. Springer, 2001.
16. R. Valk. Generalizations of Petri nets. *Mathematical foundations of computer science*, LNCS 118, pp. 140–155. 1981.
17. D. Varró. A formal semantics of UML statecharts by model transition systems. *Proc. Int. Conf. on Graph Transformation*, pp. 378–392. Springer, 2002.
18. Y. Zhao and G. Ciardo. Symbolic CTL model checking of asynchronous systems using constrained saturation. *Proc. ATVA*, LNCS 5799, pp. 368–381. 2009.
19. Y. Zhao and G. Ciardo. Symbolic computation of strongly connected components and fair cycles using saturation. *ISSE*, 7(2):141–150, 2011.
20. Y. Zhao, J. Xiaoqing, and G. Ciardo. A symbolic algorithm for shortest EG witness generation. *Proc. TASE*, pp. 68–75. IEEE Comp. Soc. Press, 2011.

Soundness of Workflow Nets with an Unbounded Resource is Decidable*

Vladimir A. Bashkin[1] and Irina A. Lomazova[2,3]

[1] Yaroslavl State University, Yaroslavl, 150000, Russia
v_bashkin@mail.ru
[2] National Research University Higher School of Economics, Moscow, 101000, Russia
[3] Program Systems Institute of the Russian Academy of Science, Pereslavl-Zalessky, 152020, Russia
i_lomazova@mail.ru

Abstract. In this work we consider modeling of workflow systems with Petri nets. A resource workflow net (RWF-net) is a workflow net, supplied with an additional set of initially marked resource places. Resources can be consumed and/or produced by transitions. We do not constrain neither the intermediate nor final resource markings, hence a net can have an infinite number of different reachable states.
An initially marked RWF-net is called sound if it properly terminates and, moreover, adding any extra initial resource does not violate its proper termination. An (unmarked) RWF-net is sound if it is sound for some initial resource. In this paper we prove the decidability of both marked and unmarked soundness for a restricted class of RWF-nets with a single unbounded resource place (1-dim RWF-nets). We present an algorithm for computing the minimal sound resource for a given sound 1-dim RWF-net.

1 Introduction

Petri nets constitute a popular formalism for modeling and analysis of distributed systems. In this paper we consider workflow systems, or, to be more precise, workflow processes. To model workflow processes a special subclass of Petri nets, called WF-nets [1, 2], is used.

In the context of WF-nets a crucial correctness criterion is soundness [1, 3]. We say that a workflow case execution terminates properly, iff its firing sequence (starting from the initial marking with a single token in the initial place) terminates with a single token in the final place (i.e. there are no "garbage" tokens after the termination). A model is called sound iff a process can terminate properly starting from any reachable marking.

* This work is supported by the Basic Research Program of the National Research University Higher School of Economics, Russian Fund for Basic Research (projects 11-01-00737, 12-01-31508, 12-01-00281), and by Federal Program "Human Capital for Science and Education in Innovative Russia" (Contract No. 14.B37.21.0392).

Soundness of WF-nets is decidable [1]. Moreover, a number of decidable variations of soundness are established, for example, k-soundness [9], structural soundness [14] and soundness of nested models [11].

One of important aspects in workflow development concerns resource management. Resources here is a general notion for executives (people or devices), raw materials, finances, etc. To take resources into account different extensions of a base formalism where introduced, having different versions of soundness criteria.

In [5, 6] a specific class of WFR-nets with decidable soundness is studied. In [10, 13] a more general class of Resource-Constrained Workflow Nets (RCWF-nets) is defined. Informally, the authors impose two constraints on resources. First, they require that all resources that are initially available are available again after terminating of all cases. Second, they also require that for any reachable marking, the number of available resources does not override the number of initially available resources.

In [10] it is proven that for RCWF-nets with a single resource type soundness can be effectively checked in polynomial time. In [13] it is proven that soundness is also decidable in general case (by reducing to the home-space problem).

In all mentioned papers resources are assumed to be permanent, i.e. they are used (blocked) and released later on. Resources are never created, nor destroyed. Hence the process state space is explicitly bounded.

To study a more general case of arbitrary resource transformations (that can arise, for example, in open and/or adaptive workflow systems), in [8] we defined a notion of WF-nets with resources (RWF-nets). RWF-nets extend RCWF-nets from [10] in such a way that resources can be generated or consumed during a process execution without any restrictions (cf. [7]). For RWF-nets we defined notions of resources and controlled resources and studied the problem of soundness-preserving resource replacement (this problem is important for adaptive workflows).

Unfortunately, even sound RWF-nets are not bounded in general, hence existing soundness checking algorithms cannot be applied here. In [8] the decidability of soundness for RFW-nets was declared as an open problem.

In this paper we consider a restricted case — RWF-nets with a single resource place (1-dim RWF-nets). One resource type is sufficient for many practical applications (memory or money are typical examples of such resources). Note that 1-dim RWF-nets are, generally speaking, not bounded and hence this case cannot be reduced to finite-state WF-nets with resources, such as RCWF- or WFR-nets.

In this paper we use graph-theoretic properties of RWF-net control automaton to prove decidability of soundness for marked, as well as unmarked 1-dim RWF-nets. We present also an algorithm for computing minimal sound resource for a given sound 1-dim RWF-net.

The paper is organized as follows. In Section 2 basic definitions of multisets and Petri nets are given. In Section 3 we give definitions of sound RWF-nets. In Section 4 the class of 1-dim RWF-nets is defined and studied, algorithms for

checking marked soundness, soundness and finding the minimal sound resource are given. Section 5 contains some conclusions.

2 Preliminaries

Let S be a finite set. A *multiset* m over a set S is a mapping $m : S \to Nat$, where Nat is the set of natural numbers (including zero), i. e. a multiset may contain several copies of the same element.

For two multisets m, m' we write $m \subseteq m'$ iff $\forall s \in S : m(s) \leq m'(s)$ (the inclusion relation). The sum, the union and the subtraction of two multisets m and m' are defined as usual: $\forall s \in S : (m+m')(s) = m(s)+m'(s), (m \cup m')(s) = max(m(s), m'(s)), (m - m')(s) = m(s) \ominus m'(s)$ (where $\ominus$ denotes the truncated subtraction). By $\mathcal{M}(S)$ we denote the set of all finite multisets over S.

Non-negative integer vectors are often used to encode multisets. Actually, the set of all multisets over finite S is a homomorphic image of $Nat^{|S|}$.

Let P and T be nonempty disjoint sets of *places* and *transitions* and let $F : (P \times T) \cup (T \times P) \to Nat$. Then $N = (P, T, F)$ is a *Petri net*. A *marking* in a Petri net is a function $M : P \to Nat$, mapping each place to some natural number (possibly zero). Thus a marking may be considered as a multiset over the set of places. Pictorially, P-elements are represented by circles, T-elements by boxes, and the flow relation F by directed arcs. Places may carry tokens represented by filled circles. A current marking M is designated by putting $M(p)$ tokens into each place $p \in P$. Tokens residing in a place are often interpreted as resources of some type consumed or produced by a transition firing. A simple example, where tokens represent molecules of hydrogen, oxygen and water respectively is shown in Fig. 1.

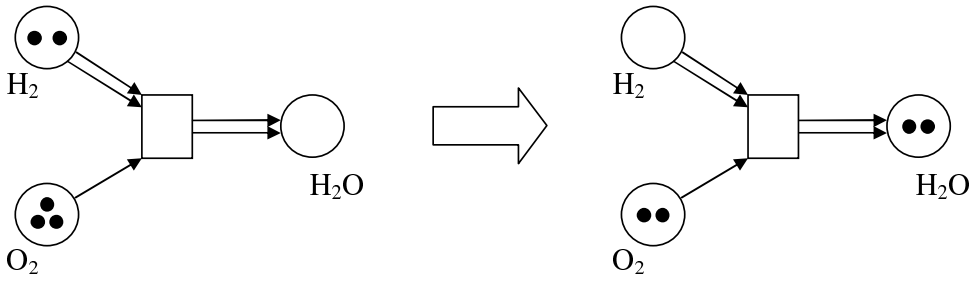


Fig. 1. A chemical reaction.

For a transition $t \in T$ an arc (x, t) is called an *input arc*, and an arc (t, x) — an *output arc*; the *preset* ${}^\bullet t$ and the *postset* $t^\bullet$ are defined as the multisets over P such that ${}^\bullet t(p) = F(p, t)$ and $t^\bullet(p) = F(t, p)$ for each $p \in P$. A transition $t \in T$ is *enabled* in a marking M iff $\forall p \in P\ M(p) \geq F(p, t)$. An enabled transition t may *fire* yielding a new marking $M' =_{\text{def}} M - {}^\bullet t + t^\bullet$, i. e. $M'(p) = M(p) - F(p, t) +$

$F(t,p)$ for each $p \in P$ (denoted $M \xrightarrow{t} M'$, or just $M \to M'$). We say that M' is reachable from M iff there is a sequence $M = M_1 \to M_2 \to \cdots \to M_n = M'$. For a Petri net N by $\mathcal{R}(N, M_0)$ we denote the set of all markings reachable from its initial marking M_0.

3 WF-nets with resources

In Petri nets with resources we divide Petri net places into control and resource ones.

Definition 1. A *Petri net with resources is a tuple* $N = (P_c, P_r, T, F_c, F_r)$, *where*

- P_c *is a finite set of control places;*
- P_r *is a finite set of resource places,* $P_c \cap P_r = \emptyset$*;*
- T *is a finite set of transitions,* $P_c \cap T = P_r \cap T = \emptyset$*;*
- $F_c : (P_c \times T) \cup (T \times P_c) \to Nat$ *is a multiset of control arcs;*
- $F_r : (P_r \times T) \cup (T \times P_r) \to Nat$ *is a multiset of resource arcs;*
- $\forall t \in T\ \exists p \in P_c : F_c(p,t) + F_c(t,p) > 0$ *(each transition is incident to some control place).*

Note that all transitions are necessarily linked to control places — this guarantees the absence of "uncontrolled" resource modifications.

A marking in a Petri net with resources is also divided into control and resource parts. For a multiset $c + r$, where $c \in \mathcal{M}(P_c)$ and $r \in \mathcal{M}(P_r)$, we write $c|r$.

Definition 2. *For a net* N *a* resource *is a multiset over* P_r*. A* controlled resource *is a multiset over* $P_c \cup P_r$*.*

Workflow nets (WF-nets) are a special subclass of Petri nets designed for modeling workflow processes. To study resource dependencies in workflow systems we consider WF-nets with resources.

Definition 3. *A Petri net with resources* N *is called a* WF-net with resources (RWF-net) *iff*

1. *There is one source place* $i \in P_c$ *and one sink place* $o \in P_c$ *s. t.* ${}^\bullet i = o^\bullet = \emptyset$*;*
2. *Every node from* $P_c \cup T$ *is on a path from* i *to* o*, and this path consists of nodes from* $P_c \cup T$*.*

Fig. 2 represents an example of a RWF-net, where resource places r_1 and r_2 are depicted by ovals, resource arcs — by dotted arrows.

Every RWF-net $N = (P_c, P_r, T, F_c, F_r)$ contains its *control subnet* $N_c = (P_c, T, F_c)$, which forms a RWF-net with the empty set of resources.

A *marked net* is a net together with some initial marking.

Definition 4. *A marked RWF-net* $(N, c|r)$ *is called* sound *iff* $\forall s \in \mathcal{M}(P_r)$, $\forall M \in \mathcal{R}(N, c|r + s)$ *we have:*

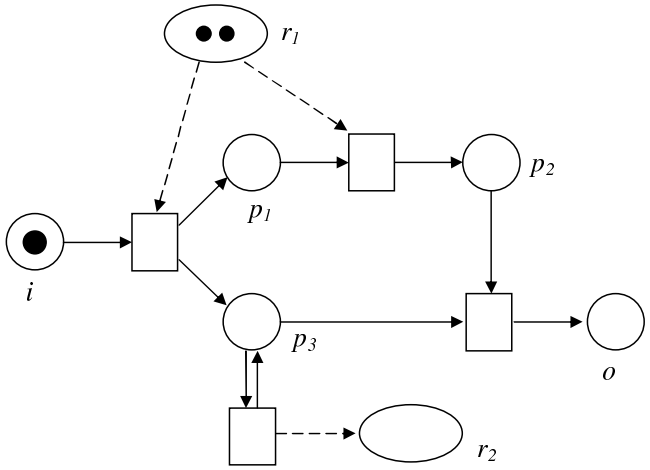


Fig. 2. WF-net wth resources.

1. $\exists s' \in \mathcal{M}(P_r) \ : \ o|s' \in \mathcal{R}(N, M)$;
2. $c'|r' \in \mathcal{R}(N, M) \ \Rightarrow \ c' = o \ \vee \ c' \cap o = \emptyset$.

Thus soundness for a RWF-net means that, first, this workflow net can terminate properly from any reachable state, and, additionally, adding any extra resource does not violate the proper termination property.

Note that our definition is substantially different from the definition of sound RCWF-nets (Resource-Constrained Workflow net) in [10]. We do not forbid creating and spending of resources. Thus, in RWF-nets resources may be produced and consumed during a process execution. This implies the possible unboundedness of sound RWF-nets.

The following statement is analogous to Lemma 5 in [10].

Proposition 1. *[7] If a marked RWF-net $(N, i|r)$ is sound, then its control subnet N_c with the initial marking i is also sound.*

The proof of this proposition is given in the Appendix.

The converse statement is not true: there may be RWF-nets with sound control subnets, for which sound resources do not exist. An example of such a net is given in Fig. 3.

Let N be a RWF-net. By $C(N)$ we denote *the set of all control markings reachable in* N_c, i. e. $C(N) = \mathcal{R}(N_c, i)$.

Proposition 2. *[7] If a marked RWF-net $(N, i|r)$ is sound, then*

1. *for any reachable control marking $c \in C(N)$ there exists a resource r', such that $(N, c|r')$ is sound;*
2. *for any two control markings $c_1, c_2 \in C(N)$ we have $c_1 \not\subset c_2$ and $c_2 \not\subset c_1$.*

The proof of this proposition is given in the Appendix.

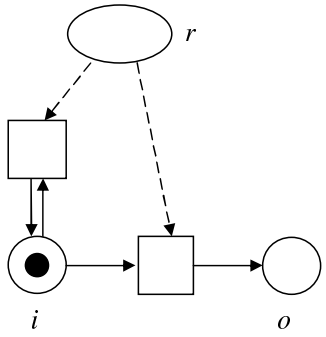


Fig. 3. RWF-net with a sound control subnet, which is not sound for any resource.

Further, we call a RWF-net N sound (without indicating any concrete resource) iff a marked RWF-net $(N, i|r)$ is sound with some initial resource r.

From the second statement of Proposition 2 and the well-known Dickson's lemma we obtain the following

Corollary 1. *For a sound RWF-net N the set $C(N)$ of all its reachable control markings is finite.*

Note 1. Since the control subnet of a sound RWF-net N is bounded, the set $C(N)$ can be effectively constructed (e. g. by constructing a coverability tree).

Definition 5. *Let N be a RWF-net, $c \in C(N)$. We define:*

1. $res(c) =_{def} \{r \in \mathcal{M}(P_r) \,|\, (N, c|r)$ *is sound*$\}$ — *the set of all sound resources for c;*
2. $mres(c) =_{def} \{r \in res(c) \,|\, \nexists r' \in res(c) : r' \subset r\}$ — *the set of all minimal sound resources for c.*

Then from Dickson's Lemma we immediately obtain:

Proposition 3. *[7] For any sound RWF-net N and any control marking $c \in C(N)$ the set $mres(c)$ is finite.*

The questions of computability of $mres(c)$ and decidability of soundness for RWF-nets remain open. In the next section positive answers to these two questions are given for a restricted case — RWF-nets with a single resource place.

4 Soundness of 1-dim RWF-nets

Let $N = (P_c, P_r, T, F_c, F_r)$ be an RWF-net with $P_r = \{p_r\}$, i.e. with just one resource place. By 1-dim RWF-nets we denote the subclass of RWF-nets with single resources. An example of such a net is given in Fig. 4. In the following sections we consider only 1-dim RWF-nets.

If a control subnet of N is not sound, then N is also not sound. So, we suppose that the control subnet of N is sound, and hence bounded.

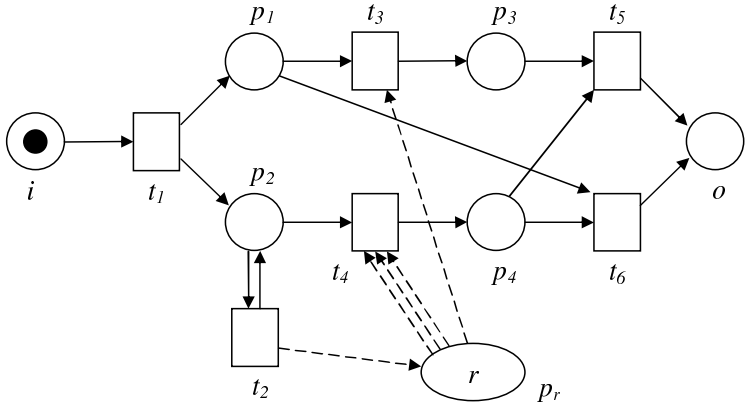


Fig. 4. 1-dim RWF-net N_1.

4.1 Control automaton

It is easy to note that a bounded control subnet can be represented as an equivalent finite automaton (a transition system). This automaton is an oriented graph with two distinguished nodes – a source node i and a sink node o.

In this control automaton states are exactly the elements of $C(N)$, transitions — transitions of the given net N. But now it will be more convenient to consider a transition t as a pair (c_1, c_2) of control states, where $c_1 \stackrel{t}{\to} c_2$. Every transition t of the automaton is labeled by an integer $\delta(t)$, defining a "resource effect" of transition firing. A positive $\delta(t)$ means that the firing of t increments the marking of a (single) resource place p_r by $\delta(t)$, a negative $\delta(t)$ means that t is enabled in a state $(c|r)$ iff $r(p_r) \geq |\delta(t)|$, and that the firing of t decrements the resource by $|\delta(t)|$. Formally,

$$\delta(t) =_{\text{def}} \begin{cases} -F_r(p_r, t) & \text{for } F_r(p_r, t) > 0; \\ F_r(t, p_r) & \text{for } F_r(t, p_r) > 0. \end{cases}$$

The value $\delta(t)$ is called an *effect* of t (denoted $\mathit{Eff}(t)$). Note that for simplicity we exclude loops, when both $F_r(p_r, t) > 0$ and $F_r(t, p_r) > 0$; such loops can be simulated by two sequential transitions.

A *support* of t is the amount of the resource required for a firing of t. It is defined as:

$$\mathit{Supp}(t) =_{\text{def}} \begin{cases} 0, & \delta(t) \geq 0; \\ |\delta(t)|, & \delta(t) < 0. \end{cases}$$

Thus, a 1-dim RWF-net N can be transformed into a control automaton $\mathit{Aut}(N)$, which can be considered as a one-counter net (e.g. [4]) or, alternatively, a 1-dim Vector Addition System with States (VASS [12]) with a specific workflow structure: one source state, one sink state, and every state is reachable from the source state, as well as the sink is reachable from every state. Note that the control automaton $\mathit{Aut}(N)$ is behaviorally equivalent to N in the branching-time semantics.

Consider an example depicted in Fig. 5. This is a control automaton for the 1-dim RWF-net from Fig. 4. States are denoted by octagons, labelled with the corresponding control markings of the net. Transitions are labelled with the corresponding names and effects.

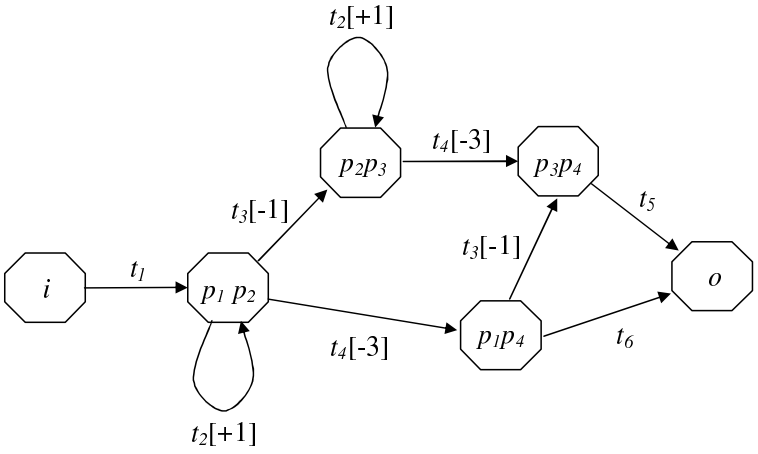


Fig. 5. Control automaton for N_1.

A control automaton (a one-counter net) is a digraph with arcs labeled by integers. Recall some basic notion from graph theory.

A walk is an alternating sequence of nodes and arcs, beginning and ending with a node, where each node is incident to both the arc that precedes it and the arc that follows it in the sequence, and where the nodes that precede and follow an arc are the head and the tail of this arc.

We consider only non-empty walks, containing at least one arc.

A walk is *closed* if its first and last nodes coincide.

A path is a walk where no arcs are repeated (nodes may be repeated).

A simple path is a path where no nodes are repeated.

A cycle is a closed path.

A simple cycle is a closed path where no nodes are repeated (except the first/last one).

A walk in a control automaton corresponds to some sequence of firings in 1-dim RWF-net. Now we inductively define an effect and a support of a walk. Intutively,

Let t be a transition and σ a walk, such that the ending node of t is the beginning node of the first transition of σ. Let $t\sigma$ denote a walk, constructed by linking t and σ. We define:

$$\mathit{Eff}(t\sigma) =_{\text{def}} \mathit{Eff}(t) + \mathit{Eff}(\sigma); \quad \mathit{Supp}(t\sigma) =_{\text{def}} \mathit{Supp}(t) + (\mathit{Supp}(\sigma) \ominus \mathit{Eff}(t)),$$

where $\ominus$ denotes the truncated subtraction.

A positive (resp., *negative*) *walk* is a walk with a positive (resp., negative) effect. Obviously, the effect of a cycle does not depend on a choice of a starting node.

A node q is called *a positive generator*, iff there exists a simple positive path from q to q (a simple positive cycle) with a zero support.

Lemma 1. *Any simple positive cycle contains at least one generator.*

Proof. Note that without loss of generality we can consider only cycles of even lengths, having alternating positive and negative arcs. Then the proof is straightforward, by induction on the length of a cycle.

A node q is called *a negative generator*, iff there exists a simple negative path θ from q to q (a simple negative cycle), such that $Supp(\theta) = -Eff(\theta)$.

Lemma 2. *Any simple negative cycle contains at least one generator.*

Proof. Similar to the previous lemma.

4.2 Decidability of soundness for marked nets

Let $(N, i|r_0)$ be an initially marked 1-dim RWF-net. By abuse of notation we denote by N also the control automaton of N, represented as a one-counter net. Recall that $i \in C(N)$ denotes the initial control state, $r_0 \in Nat$ denotes the initial value of a counter (the single resource place), and $\mathcal{R}(N, (i|r_0))$ denotes the set of all reachable states.

Note that a marked RWF-net $(N, i|r_0)$ with a sound control subnet is not sound if and only if it does not always terminate with a final control state o for some larger initial resource $r_0 + s$:

$$\exists (c|r) \in \mathcal{R}(N, i|r_0 + s) \text{ such that } (o|s') \notin \mathcal{R}(N, c|r) \text{ for any } s' \in Nat.$$

So we consider both two kinds of possible undesirable (not properly terminating) behaviours of a Petri net, namely, deadlocks and livelocks.

Definition 6. *A state $(c|r) \in C(N) \times Nat$ is a* deadlock *iff $c \neq o$ and there is no transition $t \in T$ s.t. $(c|r) \xrightarrow{t} (c'|r')$ for some c', r'.*

A finite set $L \subset C(N) \times Nat$ of states is a livelock *iff*

1. $|L| > 1$;
2. *for any $(c|r), (c'|r') \in L$ there is a finite transition sequence $\sigma \in T^*$ s.t. $(c|r) \xrightarrow{\sigma} (c'|r')$;*
3. *for any $(c|r) \in L$ and $t \in T$ s.t. $(c|r) \xrightarrow{t} (c''|r'')$ we have $(c''|r'') \in L$.*

A livelock state *is a state that belongs to some livelock.*

Note that by definition $(o|r) \notin L$ for any r;.

Proposition 4. *If a state $(c|r)$ is a deadlock then for any $t \in T$ s.t. $c \xrightarrow{t} c'$ we have $Supp(t) > r$.*

Proof. Straightforward.

Note that Prop. 4 implies $\delta(t) < 0$ and hence we can reformulate it:

Corollary 2. *If a state $(c|r)$ is a deadlock then:*

1. *$\forall t \in T$ s.t. $c \xrightarrow{t} c'$ for some c' we have $\delta(t) < 0$;*
2. *$r < min\{|\delta(t)| \ : \ c \xrightarrow{t} c' \ \ for\ some \ \ c'\}$.*

So deadlocks can occur (1) just for control states with only negative outgoing transitions; (2) only for a finite number of different resources – when there are no enough resources for firing any of the successor transitions.

Proposition 5. *The set of deadlock states is finite.*

Proof. The set of "potential deadlock" control states (nodes with only negative outgoing transitions) is finite. For a given "potential deadlock" control state the set of applicable deadlock states (natural numbers smaller than the smallest required resource for a successor transition) is also finite.

Hence, all deadlocks can be detected by checking control states with only negative outgoing transitions.

Now let us consider livelocks.

Proposition 6. *If $L \subset C(N) \times Nat$ is a livelock then there is a state $(c|r) \in L$ and a negative transition $t \in T$ with $c \xrightarrow{t} c'$, such that $\delta(t) < -r$.*

Proof. Straightforward, since the control subnet of RWF-net N is sound.

Proposition 7. *The set of livelocks is finite.*

Proof. First note that if $(c|r), (c|r+x) \in L$ with $x > 0$ then L is not a livelock. Indeed, in this case the transition sequence $(c|r) \xrightarrow{\sigma} (c|r+x)$ corresponds to a positive cycle, that can generate an infinite number of states — a contradiction to the finiteness of livelocks. So, every control state can occur in a given livelock at most once.

Now assume the converse: there are infinitely many livelocks. Then there are infinitely many livelocks with the same set of control states, which differ only in their resource value. Hence, this set includes a livelock with an arbitrarily large resource, and we can take a livelock with a resource big enough to reach the final state o. This implies that o belongs to the livelock — a contradiction with the definition of a livelock.

Thus, all livelocks can be easily detected by checking finite systems of states, closed under transition firings (strongly connected components) and satisfying the property from Prop. 6.

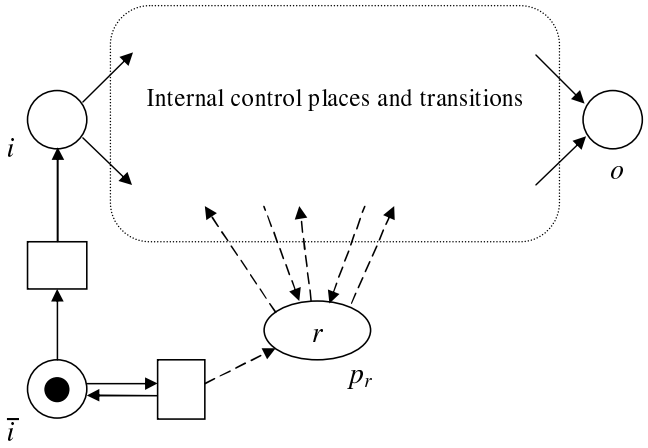


Fig. 6. Modified RWF-net $\overline{N}$.

Theorem 1. *Soundness is decidable for marked 1-dim RWF-nets.*

Proof. The following proof is similar to the proof of decidability of structural soundness in [14].

For a given 1-dim RWF-net N construct the modified RWF-net $\overline{N}$ by adding a new initial place $\overline{i}$ and two new transitions, as depicted in Fig. 6. The original 1-dim RWF-net $(N, i|r)$ is sound iff neither deadlocks nor livelocks are reachable in 1-dim RWF-net $(\overline{N}, \overline{i}|r)$ (otherwise some large enough initial resource would produce the same undesirable situation in the given net N).

Since the sets of deadlocks and livelocks are finite and computable, the problem of soundness of a marked 1-dim RWF-net can be reduced to a finite number of instances of a reachability problem for a 1-counter Petri net. This reachability problem is decidable.

4.3 Decidability of soundness for unmarked nets

Theorem 1 gives us only a semidecision procedure for soundness of a net. One can check the soundness of a given initial marking, but if the answer is negative, it is not known whether there exists a larger sound marking.

Definition 7. *An unmarked RWF-net N is called* sound *iff $(N, i|r)$ is sound for some resource r.*

Corollary 2 gives us only a necessary condition of a deadlock, reachable from *some* initial marking. Now we prove a stronger theorem, which gives a sufficient and necessary condition for existence of a soundness-violating deadlock (i.e. a deadlock that is reachable from an infinite number of different initial markings).

Theorem 2. *An unmarked 1-dim RWF-net is not sound with deadlocks iff there exist a deadlock state $(c|r)$, a negative generator q and a simple path $q \xrightarrow{\sigma} c$ such that $Eff(\sigma) \ominus Supp(\sigma) \leq r$.*

Proof. ($\Leftarrow$) It is sufficient to show that for any (large enough) initial resource r_0 there exists a larger initial resource $r_0 + x$, such that a deadlock is reachable from $(i|r_0 + x)$.

Consider an arbitrary (large enough) initial resource r_0 s.t.

$$(i|r_0) \xrightarrow{\tau} (q|s)$$

for some path τ and resource s (it is always possible to find such a resource since the control net is sound, and therefore any control state is reachable for some sufficiently large initial resource). Let $\theta = qc_1 \dots c_j q$ be a simple negative cycle with generator q, i.e. $Supp(\theta) = -Eff(\theta)$. Denote $z = s$ **mod** $Supp(\theta)$ and consider a larger initial resource $r_0 + z + Supp(\sigma)$.

We have

$$\begin{gathered}\big(i|r_0 + z + Supp(\sigma)\big)\\ \downarrow\ \tau\\ \big(q|s + z + Supp(\sigma)\big)\\ \downarrow\ \theta^{\big((s+z)/Supp(\theta)\big)}\\ \big(q|Supp(\sigma)\big)\\ \downarrow\ \sigma\\ \big(c|Eff(\sigma) \ominus Supp(\sigma)\big)\end{gathered}$$

and hence a deadlock.

($\Rightarrow$) Assume the converse: the net is unsound with a deadlock, but for any given deadlock state, it is impossible to find a negative generator that satisfies the conditions in the theorem.

The number of deadlock states is finite, hence some deadlock state $(c|r)$ is reachable from an infinite number of different initial states (initial resource values).

Every transition sequence $\sigma = t_1.t_2.\dots.t_n$ from $(i|r_0)$ to $(c|r)$ corresponds to a walk σ in the control automaton graph. Since there are infinitely many deadlock-generating initial states, the set of corresponding walks is also infinite. Each of these walks can be decomposed into a sequence of alternating simple cycles and acyclic simple paths:

$$\sigma = \tau_1(\theta_1)^{k_1}\tau_2(\theta_2)^{k_2} \dots \tau_{n-1}(\theta_{n-1})^{k_{n-1}}\tau_n.$$

Note that this decomposition is not unique: $ababa$ can be considered both as $(ab)^2a$ and $a(ba)^2$. To fix ideas, we only consider "decomposition from the right to the left", so $a(ba)^2$.

Let us show that among these walks there is a walk with a negative last cycle θ_{n-1}. Indeed, if the last cycle is positive (or neutral) with an effect x, we can consider a larger initial resource $r_0 + x * k_{n-1}$ and a shorter walk

$$\sigma' = \tau_1(\theta_1)^{k_1}\tau_2(\theta_2)^{k_2} \dots \tau_{n-1}\tau_n,$$

having the same ending — a deadlock. Now, the new walk σ' can be decomposed into simple cycles and simple paths, then the last cycle, if it is positive, can be removed by increasing the initial resource, and so on. At the end of this process we will obtain either a walk with a negative "last cycle" or a completely acyclic walk (simple path from i to c). There are only finitely many acyclic paths in the graph, but infinitely many deadlock-generating initial markings (and hence deadlock-generating walks from i to c), so we necessarily obtain a walk with a negative last cycle.

Consider such a deadlock-walk σ'', ending with a suffix $\theta^k\tau$, where θ is a negative cycle and τ is acyclic. Let $\theta = c_1c_2\dots c_i\dots c_mc_1$, where c_i is a negative generator (from Lemma 2 such c_i always exists). The path $\big((c_i\dots c_mc_1)\tau\big)$ is simple (remember that we decompose "from the right to the left" and hence $\theta\tau$ cannot contain cycles other than θ). Since the final state of the whole walk σ'' is $(c|r)$, for any suffix ϕ of σ'' we have

$$\mathit{Eff}(\phi) \ominus \mathit{Supp}(\phi) \le r.$$

It holds for $\big((c_i\dots c_nc_1)\tau\big)$ as well. But this is a simple path that leads from a negative generator to a deadlock control state – Q.E.D.

A result similar to Th. 2 is valid for livelocks:

Theorem 3. *An unmarked 1-dim RWF-net is not sound with livelocks iff there exist a livelock state $(c|r)$, a negative generator q and a simple path $q \xrightarrow{\sigma} c$ such that $\mathit{Eff}(\sigma) \ominus \mathit{Supp}(\sigma) \le r$.*

Proof. Similar to Th. 2.

Corollary 3. *Soundness is decidable for unmarked 1-dim RWF-nets.*

Proof. All simple (negative) cycles can be found by Tarjan algorithm, deadlock and livelock states — by searching for states, satisfiyng Prop. 4 and Prop. 6 respectively. The set of simple paths is finite (and easily computable).

4.4 Computability of a minimal sound resource

Now we propose a plain (and hence, may be, not the most effective) algorithm for computing the minimal resource r such that $(N, i|r)$ is sound: one tests soundness for incremented values of r until success. Note that this method can be applied only to sound nets, while soundness of the unmarked net can be checked with the algorithm given in Cor. 3.

5 Conclusion

In this paper we have investigated the soundness property for workflow nets with one (unbounded) resource place. We have proved that soundness is decidable

for marked and unmarked nets, and that the minimal sound resource can be effectively computed.

Our decision algorithms use the reduction to the reachability problem for unbounded Petri nets and hence cannot be considered efficient. However, the inefficiency could be unavoidable, since RWF-nets are expressively rather close to ordinary Petri nets (VASS).

Further research will concern decidability of soundness for the general case of RWF-nets. It is also quite interesting to apply some alternative notions of soundness to our infinite-state workflow nets. The so-called *relaxed soundness* is of a particular interest. Relaxed soundness has been proposed as a weaker than soundness property.

Some other interesting variants of soundness property are k-soundness, generalized and structural soundness [3].

The authors would like to thank the anonymous reviewers for their helpful comments.

References

1. W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
2. W.M.P. van der Aalst, K.M. van Hee. *Workflow Management: Models, Methods and Systems*, MIT Press, 2002.
3. W.M.P. van der Aalst, K.M. van Hee, A.H.M. Hofstede, N. Sidorova, H.M.W. Verbeek, M. Voorhoeve, M.T. Wynn. Soundness of workflow nets: classification, decidability, and analysis, *Formal Aspects of Computing*, 23(3):333–363, Springer, 2011.
4. P. A. Abdulla, K. Čerans. Simulation is decidable for one-counter nets. In *Proc. CONCUR'98*, vol. 1466 of *Lecture Notes in Computer Science*, pages 253–268, Springer, 1998.
5. K. Barkaoui, L. Petrucci. Structural Analysis of Workflow Nets with Shared Resources. In *Proc. Workflow Management: Net-based Concepts, Models, Techniques and Tools (WFM98)*, volume 98/7 of *Computing Science Reports*, pages 82–95, Eidhoven University of Technology, 1998.
6. K. Barkaoui, R. Ben Ayed, Z. Sbaï. Workflow Soundness Verification based on Structure Theory of Petri Nets. *International Journal of Computing and Information Sciences*, Vol. 5, No. 1, 2007. P.51–61.
7. V. A. Bashkin, I. A. Lomazova. Petri Nets and resource bisimulation. *Fundamenta Informaticae*, Vol. 55, No. 2, 2003. P.101–114,
8. V. A. Bashkin, I. A. Lomazova. Resource equivalence in workflow nets. In *Proc. Concurrency, Specification and Programming, 2006*, volume 1, pages 80–91. Berlin, Humboldt Universitat zu Berlin, 2006.
9. K. van Hee, N. Sidorova, M. Voorhoeve. Generalized Soundness of Workflow Nets is Decidable. In *Proc. ICATPN 2004*, volume 3099 of *Lecture Notes in Computer Science*, pages 197–216. Springer, 2004.
10. K. van Hee, A. Serebrenik, N. Sidorova, M. Voorhoeve. Soundness of Resource-Constrained Workflow Nets. In *Proc. ICATPN 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 250–267. Springer, 2005.

11. K. van Hee, O. Oanea, A. Serebrenik, N. Sidorova, M. Voorhoeve, I.A. Lomazova. Checking Properties of Adaptive Workflow Nets. *Fundamenta Informaticae*, Vol. 79, No. 3, 2007. P.347–362.
12. J. Hopcroft, J.-J. Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theoretical Computer Science*, **8**(2), 1979, pages 135–159.
13. N. Sidorova, C. Stahl. Soundness for Resource-Contrained Workflow Nets is Decidable. In *BPM Center Report BPM-12-09*, BPMcenter.org, 2012.
14. F. L. Tiplea, D. C. Marinescu. Structural soundness of workflow nets is decidable. *Information Processing Letters*, Vol. 96, pages 54–58. Elsevier, 2005.

6 Appendix

Proof of Proposition 1.

The proof is by contradiction. Let $(N, i|r)$ be a sound RWF-net and let the net (N_c, i) be not sound. Then there exists a marking $c \in \mathcal{R}(N_c, i)$, such that either the final marking o is not reachable from c, or $o \in c$ and $c \neq \{o\}$.

Since for the control subnet the control marking c is reachable from the initial marking i via some sequence of firings, we can always take a resource s, sufficiently large to support the same sequence of firings for $(N, i|r + s)$ and to reach the same control state c. If for the control subnet the final state was not reachable from c, then adding resource places can't make it reachable for the net with resources, i. e. for $(N, i|r + s)$, in contradiction with the soundness of $(N, i|r)$. If, otherwise, $o \in c$ and $c \neq \{o\}$, then we also obtain a contradiction with the soundness of $(N, i|r)$, since the control state c is reachable for $(N, i|r + s)$.

Proof of Proposition 2.

(1) Similarly to the proof of the Proposition 1 we can always take a sufficiently large initial resource $r + s$.

(2) Suppose this is not true. Assume that for some $c_1, c_2 \in C(N)$ we have $c_2 = c_1 + c'$ for some $c' \neq \emptyset$. From the first statement of this proposition it follows that there exist resources r_1 and r_2 s. t. RWF-nets $(N, c_1|r_1)$ and $(N, c_2|r_2)$ are sound. Then nets $(N, c_1|r_1 + r_2)$ and $(N, c_2|r_1 + r_2)$ are also sound. Thus the final marking $o|r'$ is reachable from the marking $c_1|r_1 + r_2$, and (due to monotonicity property of Petri nets) the marking $o + c'|r'$ is reachable from the larger marking $c_2|r_1 + r_2$ — contradiction with the soundness for RWF-net $(N, c_2|r_1 + r_2)$.

Modeling Distributed Private Key Generation by Composing Petri Nets

Luca Bernardinello[1], Görkem Kılınç[1], Elisabetta Mangioni[1,2], and Lucia Pomello[1]

[1] Dipartimento di Informatica Sistemistica e Comunicazione,
Università degli studi di Milano - Bicocca,
Viale Sarca, 336 - Edificio U14 - I-20126 Milano, Italia
[2] Istituto per la Dinamica dei Processi Ambientali,
Consiglio Nazionale delle Ricerche (CNR-IDPA),
Piazza della Scienza, 1 - Edificio U1 - I-20126 Milano, Italia

Abstract. We present a Petri net model of a protocol for the distributed generation of id-based private keys. Those keys can then be used for secure communications. The components of the system are built as refinements of a common interface, by applying a formal operation based on a class of morphisms between Elementary Net Systems. It is then shown that we can derive behavioural properties of the composed system without building it explicitly, by exploiting properties of the given morphisms.

Keywords: Petri Nets, morphisms, local state refinement, composition, distributed private key generation

1 Introduction

In [1] we proposed a way to compose Elementary Net Systems (ENS) by identifying conditions, places, and events. The identification is ruled by a pair of morphisms from the two components to an *interface*. The interface is an ENS which can be seen as specifying the protocol of interaction between components, or a common abstraction.

This framework was first defined relying on N-morphisms, originally introduced in [11], [4]. Later, the same operation was defined over a new class of morphisms, called α-morphisms (see [3] and [1]).

An α-morphism from an ENS N_1 to an ENS N_2 corresponds to a relation of refinement: some subnets of N_1 refine conditions of N_2. This refinement may require that some events be duplicated. Such morphisms are defined and discussed in Section 3.

When composing two ENS, N_1 and N_2 over an interface N_I, the two morphisms towards the interface specify how each component refines parts of the interface. An uninterpreted example is given in Section 4.

One of the claimed advantages of this approach to design is the ability to derive properties of the composed systems from properties of the components and

of the morphisms, without the need to actually build and analyse the composed system.

Ideally, one would like to derive behavioural properties, like liveness and safety properties, by analyzing only the structure of the models involved, thus avoiding the potentially high cost of computing the reachable markings. This is not always possible. Hence, the method we propose uses some behavioural information about components and interface; however, this is limited to only a part of the models, and does not involve the whole system model.

Here, we test these ideas on a protocol for distributed generation of id-based cryptographic keys. The protocol, described in more detail in Section 5, requires the cooperation of several *private key generators* (PKGs) so that a client can build a private key. Basically, n PKG nodes come together to generate a master key pair consisting of a private and a public key. After that, each PKG node has a share for the master key pair. A client who wants to have a private key applies to k available PKG nodes. Each PKG node calculates a piece of the client's private key by using the unique id-string of the client and the share of the master private key which is held by that specific PKG node. On receiving k pieces, the client continues to extract its private key. The so called bulletin board is responsible for the initialization of the components in the system and broadcasting the public parameters. During both the distributed generation of the master key pair and the extraction of the clients' private keys, a verification can be performed by using the commitment values and public keys held and broadcast by the bulletin board. The id-based distributed private key generation protocol was proposed in [5]; an improved version is presented in [6]. In [7], a Petri net model for the protocol to be implemented on industrial control systems is presented.

In the next section, basic definitions related to ENS are recalled. Section 3 recalls the formal definition of α-morphisms and the properties they preserve or reflect and which are used in the rest of the paper. The definition of an operation of composition of ENS, based on α-morphisms, is informally recalled in Section 4 on the basis of an uninterpreted example. In the same section, the main result relating behavioral properties of the composed system to behavioral properties of its components is recalled. Section 5 presents the distributed private key generation protocol which is modelled by Petri nets in Section 6. In the same section, we analyze behavioural properties of the model. The paper is closed by a short concluding section.

2 Preliminary definitions

In this section, we recall the basic definitions of net theory, in particular Elementary Net Systems and unfoldings [13].

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 $X = B \cup E$.

The *preset* of an element $x \in X$ is ${}^\bullet x = \{y \in X | (y, x) \in F\}$; the *postset* of x is $x^\bullet = \{y \in X | (x, y) \in F\}$; 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 ${}^\circ N = \{x \in X : {}^\bullet x = \emptyset\}$ and the *out-elements* of N by $N^\circ = \{x \in X : x^\bullet = \emptyset\}$.

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 $N(A) = (B \cap A, E \cap A, F \cap (A \times A))$. Given a subset of conditions $A \subseteq B$, we say that N_A is the *subnet of* N *generated* by A if $N_A = (A, {}^\bullet A^\bullet, F \cap ((A \cup {}^\bullet A^\bullet) \times (A \cup {}^\bullet A^\bullet)))$. Note that given $A \subseteq B$, $N(A \cup {}^\bullet A^\bullet) = N_A$.

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 $m_0 \subseteq B$.

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)*.

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 $m\,[e\rangle$, if ${}^\bullet e \subseteq m$ and $e^\bullet \cap m = \emptyset$; the occurrence of e at m leads from m to m', denoted $m\,[e\rangle\, m'$, iff $m' = (m \setminus {}^\bullet e) \cup e^\bullet$.

Let ϵ denote the empty word in E^*. The firing rule is extended to sequences of events by setting $m\,[\epsilon\rangle\, m$ and $\forall e \in E, \forall w \in E^*, m\,[ew\rangle\, m' = m\,[e\rangle\, m''[w\rangle m'$; 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 $m_0\,[w\rangle\, m$. The *set of all reachable markings* of N is denoted by $[m_0\rangle$.

An EN System is *contact-free* if $\forall e \in E, \forall m \in [m_0\rangle$: ${}^\bullet e \subseteq m$ implies $e^\bullet \cap m = \emptyset$. An EN System covered by Sequential Components is contact-free [13]. 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 EN System is *deadlock-free* if no reachable marking is dead.

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 ${}^\bullet e_x \cap {}^\bullet e_y \neq \emptyset$, where F^* is the reflexive and transitive closure of F.

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.

An *occurrence net* is a net $N = (B, E, F)$ such that if $e_1, e_2 \in E, {e_1}^\bullet \cap {e_2}^\bullet \neq \emptyset$ then $e_1 = e_2$; F^* is a partial order; for any $x \in X, \{y : yF^*x\}$ is finite; $\#_N$ is irreflexive and the minimal elements with respect to F^* are conditions. Occurrence nets were introduced in [10]; in [13] they are called branching process nets.

A branching process of N is an occurrence net whose elements can be mapped to the elements of N. Let $N = (B, E, F, m_0)$ be an EN System, and $\Sigma = (P, T, G)$ be an occurrence net. Let $\pi : P \cup T \to B \cup E$ be a map. The pair (Σ, π) is a *branching process* of N if $\pi(P) \subseteq B$, $\pi(T) \subseteq E$; π restricted to the minimal elements of Σ is a bijection on m_0; for each $t \in T$, π restricted to ${}^\bullet t$ is injective and π restricted to $t^\bullet$ is injective and for each $t \in T$, $\pi({}^\bullet t) = {}^\bullet(\pi(t))$ and $\pi(t^\bullet) = (\pi(t))^\bullet$.

The unfolding of an EN System N, denoted by $Unf(N)$, is the maximal branching process of N, namely the unique, up to isomorphism, 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*.

3 α-morphisms

In this section we present the formal definition of α-morphisms [3, 2] for State Machine Decomposable Elementary Net Systems (SMD-EN Systems) and the structural and behavioural properties α-morphisms preserve and reflect.

Definition 1. *Let $N_i = (B_i, E_i, F_i, m_0^i)$ be a SMD-EN System, for $i = 1, 2$. An α-morphism from N_1 to N_2 is a total surjective map $\varphi : X_1 \to X_2$ 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* $\varphi({}^\bullet {e_1}^\bullet) = \{\varphi(e_1)\}$;
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 $\varphi^{-1}({}^\bullet {b_2}^\bullet) \subseteq E_{SC}$.*

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 φ.

In-conditions and out-conditions have different constraints, 5b and 5c respectively. As required by 5c, choices which are internal to a bubble can not 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 that pre-events of any in-condition are mapped on a subset of the pre-events of the condition it refines. 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, and the same holds for 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 [8]. Moreover, given that a bubble can be abstracted by a single condition no input event of a bubble is enabled whenever a token is within the bubble [8].

It is possible to show that the family of SMD-EN Systems together with α-morphisms forms a category [8].

In [8] and [3] structural and behavioral properties preserved or reflected by α-morphisms has been studied. In particular, sequential components are reflected in the sense that the inverse image of a sequential component is covered by sequential components and α-morphisms preserve reachable markings.

Moreover, stronger properties hold under additional constraints. In order to present them, we have to consider the following construction. Given an α-morphism $\varphi : N_1 \to N_2$, and a condition $b_2 \in B_2$ with its refinement, we define two new auxiliary SMD-EN Systems. The first one, denoted $S_1(b_2)$, contains the following elements: a copy of the subnet which is the refinement of b_2, i.e.: the bubble; its pre and post events in E_1 and two new conditions, denoted b_1^{in} and b_1^{out}. b_1^{in} is pre of all the pre-events, and b_1^{out} is post of all the post-events. The initial marking of $S_1(b_2)$ will be $\{b_1^{in}\}$ or, if there are no pre-events, the initial marking of the bubble in N_1. 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 $\{b_2^{in}\}$ or, if there are no pre-events, the initial marking of b_2. Define φ^S as a map from $S_1(b_2)$ to $S_2(b_2)$, which restricts φ 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 φ^S is an α-morphism. Let $Unf(S_1(b_2))$ be the unfolding of $S_1(b_2)$, with folding function $u : Unf(S_1(b_2)) \to S_2(b_2)$.

Consider the following additional constraints:

c1 the initial marking of each bubble is at the start of the bubble itself; formally, 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 $\varphi^{-1}(b_2) \cap m_0^1 = {}^\circ\varphi^{-1}(b_2)$;

c2 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, i.e.: $\varphi^S \circ u$ is an α-morphism from $Unf(S_1(b_2))$ (in which we put a token in the in-condition of the net) to $S_2(b_2)$;
c3 different bubbles do not "interfere" with each other; where we say that two bubbles interfere with each other when their images share, at least, a neighbour.

Note that the third constraint is not restrictive since the refinement of two interfering conditions can be done in two different steps.

Under **c1**, **c2**, and **c3**, the following properties can be proved [8]:

p1 reachable markings of N_2 are reflected:
for all $m_2 \in [m_0^2\rangle$, there is $m_1 \in [m_0^1\rangle$ such that $\varphi(m_1) = m_2$;
p2 N_1 and N_2 are weakly bisimilar:
by using φ, define two labelling functions 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; then (N_1, l_1) and (N_2, l_2) are weakly bisimilar $(N_1, l_1) \approx (N_2, l_2)$.

For a definition of weak bisimulation of EN Systems see [12] and [9].

4 Composition based on α-morphisms

In this section, we recall the composition of SMD-EN Systems based on α-morphisms as defined in [1], on the basis of an uninterpreted example given in Fig. 1.

The two systems to be composed, N_1 and N_2, must be mapped onto a common *interface*, which is another SMD-EN System N_I. The interface can be seen, intuitively, as a protocol of interaction, with which the components must comply, or as a common abstraction; in this second view, each component can refine some parts of the common abstraction. The two α-morphisms, from the components to the interface, determine how the two components refine the local states of the interface, and then which elements are to be identified and which events in the two components have to synchronize.

To compose two net systems, each must be *canonical* with respect to the corresponding morphism towards the interface. We say that a net system is canonical with respect to an α-morphism if each bubble contains a condition, called *representation*, that corresponds to the abstraction of that bubble. Examples of representations are $r_{N_1}(b_1)$ and $r_{N_2}(b_0)$ in Fig. 1. If a system is not canonical, it is always possible to construct its unique (up to isomorphism) canonical version by adding the missing representations, and marking them as their images, or by deleting the multiple ones. Because of the constraints on α-morphisms, and in particular of the ones on sequential components, point 5e of Def. 1, this construction does not modify the behaviour of the original system and the corresponding modified morphism is still an α-morphism.

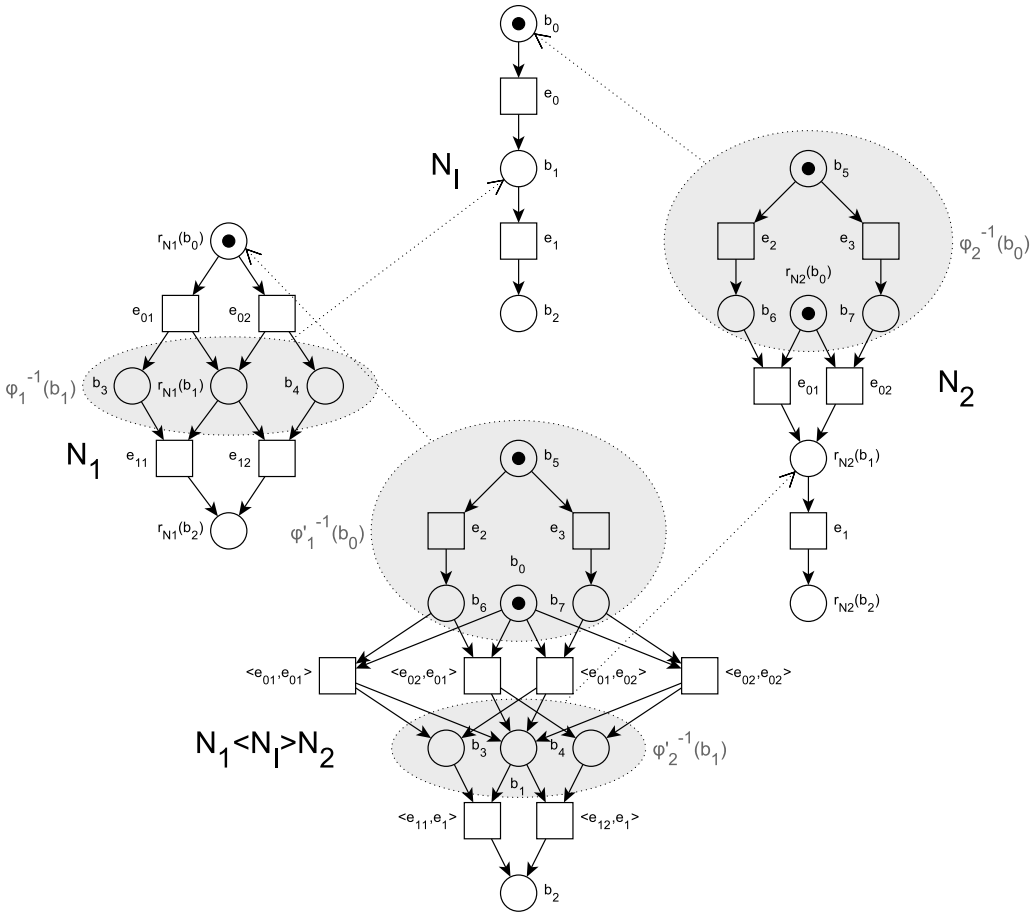


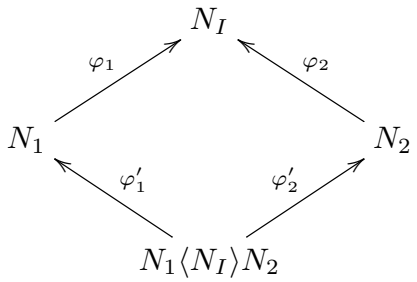
Fig. 1: An example of composition based on α-morphisms

In the example given in Figure 1 the interface N_I is a simple sequence of two events. The two components, N_1 and N_2, refine two different local states, b_1 and b_0, each one by a subnet, shown on a gray background.

The composed net $N = N_1\langle N_I\rangle N_2$ contains the refinement of each condition of the interface as it is in the two components, but for the representation, plus the condition itself, as we can see for condition b_0 and b_1 of the example. The rest of the net, not refined by the components, is taken as it is, but for the synchronizations of the events in the neighbourhood of the refinements/bubbles. Such events must be synchronized so that each possible pair composed by one event of a component and one event of the other component must be created, as we can see for events mapped on events e_0 and e_1 of Fig. 1. Then, also arcs between in- and out-condition of each bubble and its pre and post (synchronized) events must be created accordingly to the components. The initial marking is the union of the ones in the components. By construction, $N = N_1\langle N_I\rangle N_2$ is an EN System, and it is covered by sequential components [8].

This construction leads to the definition of a map φ_i' from $N = N_1\langle N_I\rangle N_2$ onto N_i, $i = 1, 2$, relating each element local to a component to the corresponding

representation and projecting synchronized events. In [8] it is proved that this map is an α-morphism and that the following diagram commutes.



These results say that the composed system refines both the components, as well as the interface. The main result relating behavioral properties of the composed system to behavioral properties of its components is stated in the following Proposition [8], [2].

Proposition 1. *Let $N_i = (B_i, E_i, F_i, m_0^i)$ be an SMD-EN System for $i = 1, 2, I$. Let φ_i, with $i = 1, 2$, be an α-morphism from N_i to N_I, and let $N = N_1\langle N_I\rangle N_2$ be be the composition of N_1 and N_2 using φ_1 and φ_2. If N_1 is weakly bisimilar to N_I then $N = N_1\langle N_I\rangle N_2$ is weakly bisimilar to N_2.*

Where, the labelling functions are derived from φ_1 and φ_2', respectively, in such a way that E_I and E_2 are all observable and the invisible events of E_1 and E are the ones which are mapped to conditions by φ_1 and φ_2', respectively.

This result tells us, in particular, that the composition of refinements N_1 and N_2, which are weakly bisimilar to a common interface N_I, yields a system N which is weakly bisimilar to N_I; and then, since bisimulation preserves deadlock-freeness, it is possible to deduce that N is also deadlock-free by verifying that N_I is deadlock-free. Remember that by **p2** it is possible to check weak bisimilarity between two systems related by an α-morphism by considering their behaviour only locally, as required by **c1**, **c2**, and **c3**.

5 Distributed private key generation for id-based cryptography

In an id-based cryptographic system, unlike in the other public key cryptographic systems, a publicly known string such as e-mail address, domain name, a physical IP address or a combination of more than one strings is used as public key. The idea of id-based cryptography was first proposed by Shamir in [14]. The proposed scheme enables users to communicate securely and to verify signatures without exchanging any private or public key. Consequently, there is no need for a certification authority to verify the association between public keys and users.

Basically, in an id- based cryptographic system there is a private key generator (PKG) which generates private keys for users. A PKG has a key pair which is referred as master key pair consisting of a master private key and a master

public key. A PKG generates a private key for a user basically by first hashing its publicly known unique identity string then signing hashed id by the master private key. Later, the user can verify its key by using the master public key.

Since the PKG can generate private keys for users, it can sign or decrypt a message for any user or it can make users' private keys public. This problem about private key generation is called the key escrow problem. Distributed private key generation (DPKG) is one of the effective solutions to the key escrow problem. In both schemes [5], [6] secret sharing methods are used for distributing private key generation among multiple PKGs.

In a DPKG there is a number of PKG nodes participating while they share the responsibility equally. In our work we followed the identity based distributed private key generation schemes presented in [5] and [6]. For more details about the algorithms and the terminology it is recommended to refer to these citations.

The components of a DPKG system are divided into two main groups as PKG nodes and clients. PKG nodes are responsible for generating private keys for clients in a distributed manner. There is also a third component called bulletin board which is responsible for managing the global system variables, collecting the commitments from PKG nodes, calculating the final commitment and broadcasting these commitments.

We can examine DPKG protocol in three steps: setup, distribution and extraction. Setup is a preparation step to create the system parameters and to get ready for extracting the master key pair distributively and extraction of private keys. In this step, bulletin board is given a security parameter and chooses some system variables according to this given security parameter; it then broadcasts public system parameters to be used by the other system components. It also initializes the commitment values to zero in order to set them to the values it will receive from PKG nodes in the distribution step. Final commitment is also set to zero which will be calculated using the received commitments and it will be broadcast later.

Distribution step is illustrated in Figure 2. In this step, n PKG nodes create a master private key together without using any dealer in a way that the key cannot be reconstructed without retrieving k shares from these n PKGs. k is the threshold number of PKG nodes needed to collaborate together in order to construct the key. To do this, an improved version of (n, k) Feldman's secret sharing scheme stated in [6] is used. The idea behind secret sharing without a dealer is to make each PKG node create a secret of their own and calculate subshares to distribute among other PKG nodes. At the end, each PKG node will have n subshares including the one it calculated for itself. The sum of these subshares will be the share of the PKG node for the master private key. During the calculation of the subshares each PKG node also creates commitments corresponding to the subshares calculated by them. These commitments are sent to the bulletin board to be used by the PKG nodes for the verification of the received subshares. Note that, in this DPKG system none of the PKG nodes knows the master secret key since each of them has only a part of it.

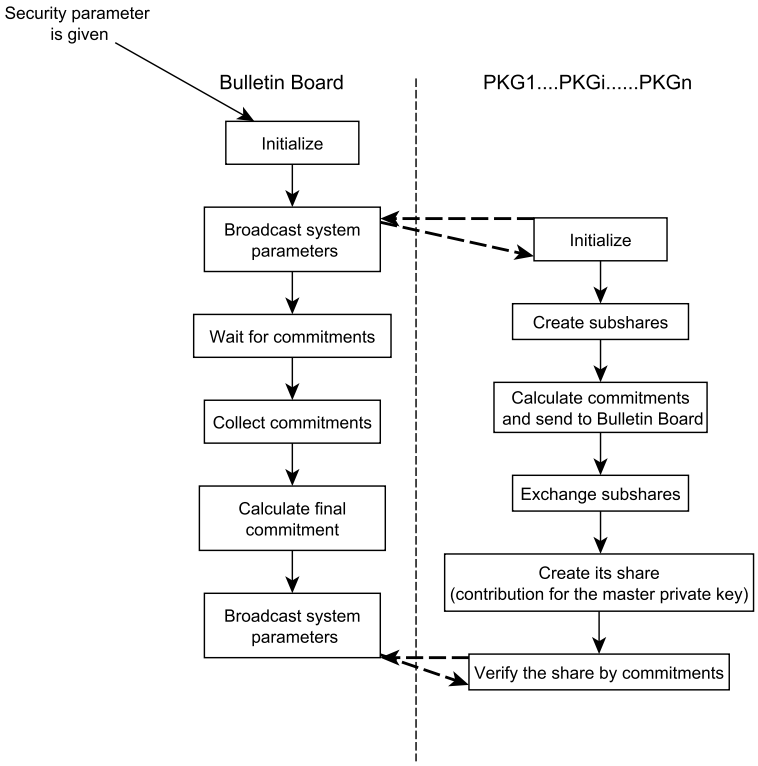


Fig. 2: Block schema of the distribution step of private key generation.

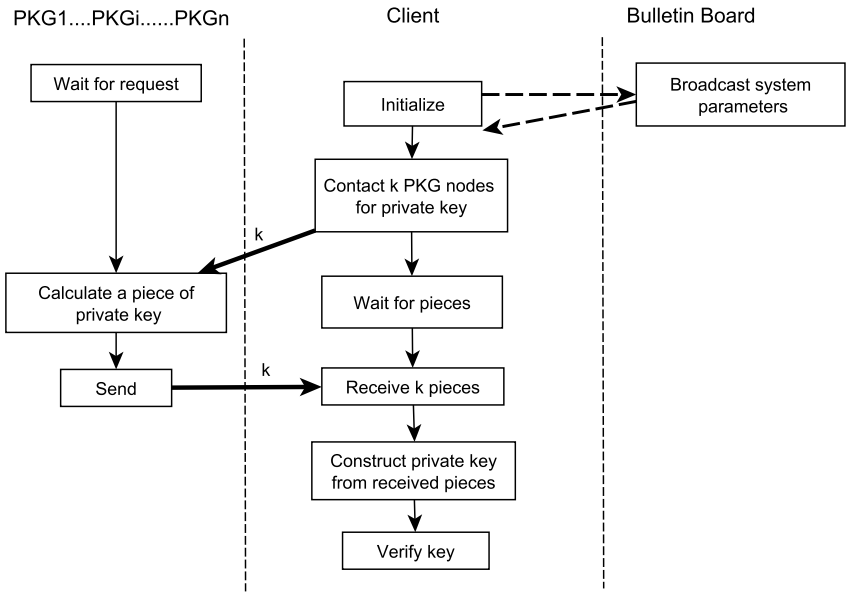


Fig. 3: Block schema of the extraction step.

In extraction step, as it is illustrated in Figure 3, a client with identity string ID contacts k available nodes from the PKG nodes pool. Each PKG_i signs hashed identity string of the client with its master private key share and returns a private key piece as $s_i\text{H}(ID)$ over a secure and authenticated channel. After receiving k pieces from k available PKG nodes, client constructs its private key. The client can verify the key by using bilinear pairings as it is stated in [6] and [7].

6 The model of DPKG

In this section, we present Petri net model of a simplified DPKG system with three PKG nodes while the threshold number is two. We fixed these numbers for the simplicity but the generated model is more generic and can easily be modified for different threshold and PKG node number as it will be discussed through this section. Our model consists of the following three nets: N_I, N_{PKG} and N_C. N_I is the common interface between N_{PKG} and N_C. It is an abstract model of the whole system which represents the interaction between the main components of the system. This model also includes the abstract behavior of the bulletin board which is basically responsible for managing the global system variables and commitments. N_{PKG} is the net representing the behavior of PKG nodes while N_C is the net representing the behavior of clients in the DPKG system. We aim to compose N_{PKG} and N_C using N_I as the common interface and prove that the composed net $N_{PKG}\langle N_I\rangle N_C$ and the interface N_I preserve and reflect some properties presented in Section 3 since there is an α-morphism both from N_{PKG} to N_I and from N_C to N_I.

N_I, which is the net representing the interface, is given in Figure 4. This net is an abstract model of the behavior of all three system components: bulletin board, PKG nodes and clients. The system is idle in the beginning. After event *init* occurs, system components are initialized and all PKG nodes are ready for generating a secret key distributedly. The event *init* includes the setup step of the protocol which is explained in Section 5. The condition *calculate shares* represents the whole process including calculating subshares and exchanging between PKG nodes in order to calculate their shares for the master private key. During this, each PKG node chooses a secret polynomial. It calculates the commitment corresponding to its secret polynomial and sends it to the bulleting board. It also calculates n subshares using its polynomial where n is the number of PKG nodes in the system. Each PKG node sends the subshare to the related PKG node. After exchanging is completed each PKG node will have $n-1$ subshares sent by other PKG nodes and one subshare of its own. By using these n subshares each PKG node calculates its share. When the condition *shares calculated* becomes true, it means that all the PKG nodes finished calculating share and each of them is holding a share.

Once PKG nodes have their shares, they can verify their shares using the final commitment value which is already calculated during the abstract event *calculate shares*. If all the shares are correctly verified, the condition *shares ver-*

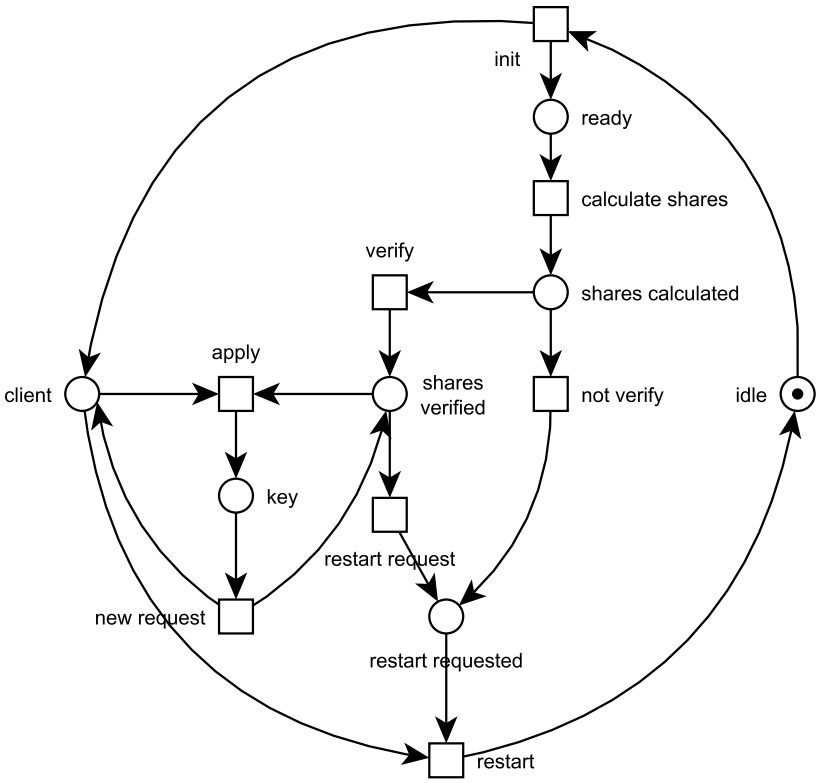


Fig. 4: N_I, the net representing the interface.

ified becomes true so a client can apply for extracting a private key. In this model, event *apply* includes choosing k available PKG nodes, receiving k pieces and calculating its private key using these pieces. When the condition *key* is true, the client has a key but we do not know if the key is correct or not by looking in this abstract model. In both cases *new request* event can occur or the system can continue with a restart which repeats the whole distributed private key generation. In case of a fail during the verification of shares, the system is forced to a restart without a key extraction.

N_I is a live and reversible net which means that from any reachable marking one can always get back to the initial state. These two properties are important because a DPKG system must always be alive to respond to the clients' key requests and key generation process must be restartable whenever it is needed. The net N_I is also covered by sequential components which is a requirement in order to be able to look for an α-morphism. The sequential components covering the net can be shown as lists of conditions: {*idle, ready, shares calculated, shares verified, key, restart requested* }, {*idle, client, key*}.

Figure 5 shows the net N_{PKG}. This net refines the interface with respect to PKG nodes' behavior. All the elements of N_{PKG} are mapped to the element with the same name in N_I but for the subnet circled by dashed line that is

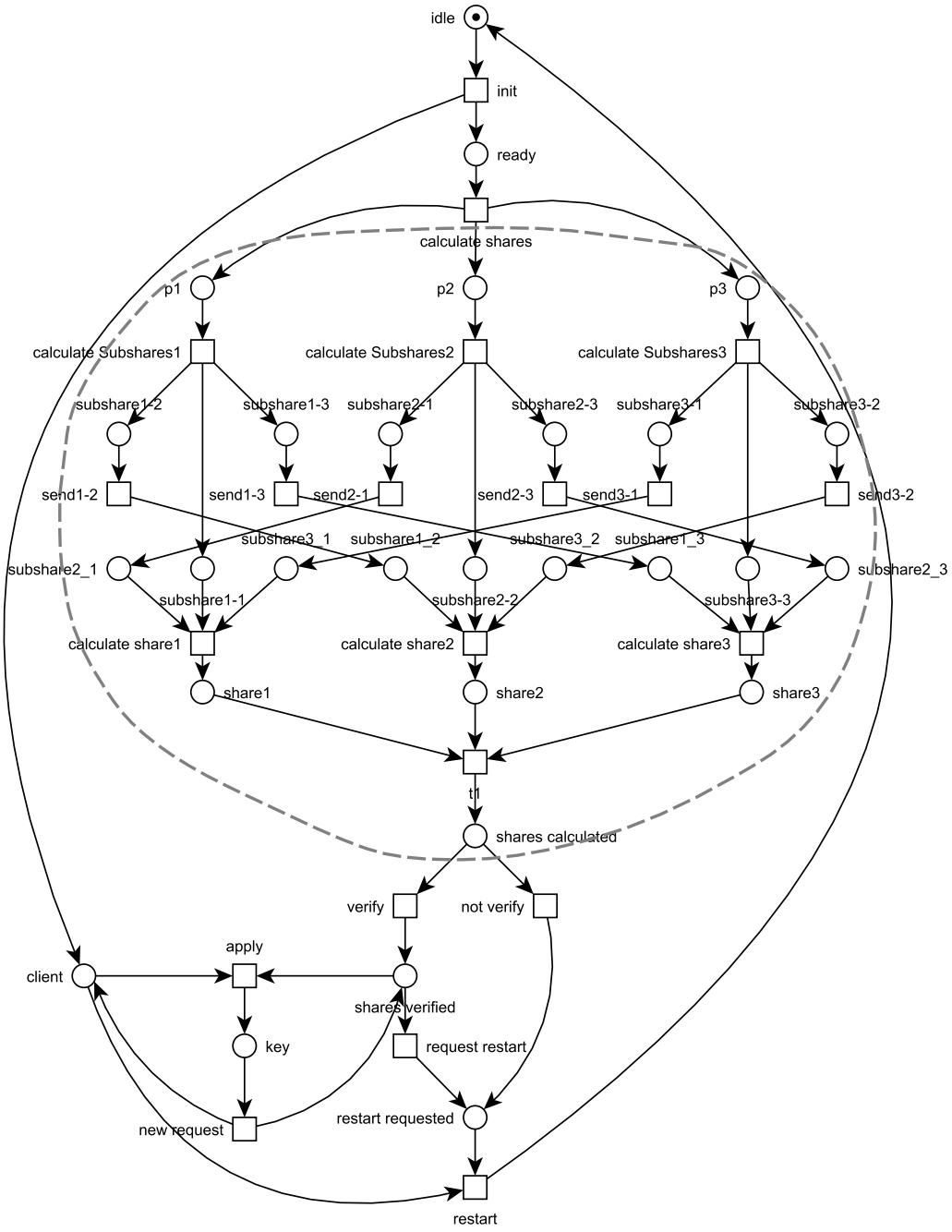


Fig. 5: N_{PKG}, the net representing the PKG nodes.

mapped to a single condition. This subnet forms a bubble which is a refinement of the condition *shares calculated* in N_I. The bubble shows the calculation and exchange of subshares between three PKG nodes and calculation of shares by each PKG node whereas in N_I the occurrence of the whole process is abstracted by one condition. If we model a system with n PKG nodes instead of three nodes, only the bubble will grow, the other elements of the net will remain the same.

N_{PKG} is also live, reversible and covered by sequential components like N_I. It is already shown that the conditions outside the bubble are covered by sequential components. Thus, here we will only show how the bubble is covered by sequential components. After event *calculate shares* the net branches into three paths and after each event *calculate subshares* i for $i = 1, 2, 3$ the net branches again into three paths. This fact results in having nine sequential components inside the bubble. Here we present only some of the sequential components as the lists of conditions that construct the components: {*p1, subshare 1-2, subshare 1_2, share 2, shares calculated*}, {*p1, subshare 1-1, share 1, shares calculated*}, {*p1, subshare 1-3, subshare 1_3, share 3, shares calculated*}. The paths starting with conditions *p2* and *p3* can also be constructed in the same way.

In order to prove that there is an α-morphism from N_{PKG} to N_I we have to show that the requirements in Definition 1 are satisfied. To begin with, the initial states of N_{PKG} and N_I are related. For all the events in N_{PKG} which are mapped to an event in N_I, also the pre-conditions and post conditions of these events are mapped to the pre and post-conditions of the related events in N_I. Moreover, for all the events in N_{PKG} that are mapped to a condition in N_I, all the pre and post-conditions of that event are also mapped to the same condition in N_I. We see that the nets satisfy the first four requirements of α-morphism. To continue with, we can see that the bubble in N_{PKG} is acyclic so 5a is satisfied. As seen in Figure 5 all the in-elements of the bubble are generated by the only one event entering the bubble which is mapped to the corresponding event in the interface, *calculate shares*. It is also seen that post-events of the out-condition of the bubble are exactly the same post-events of the corresponding condition in the interface. Thus, 5b and 5c are satisfied. 5d is also satisfied because the conditions that are internal to the bubble have pre-events and post-events which are all mapped to the refined condition *shares calculated* in N_I but for in and out-elements. Finally, as we already listed the sequential components of the net, it is easy to see that for each condition of the bubble there is a sequential component containing that condition and all the pre and post-events of the bubble, so requirement 5e is satisfied. In this way, we proved that there is an α-morphism from N_{PKG} to N_I.

The net shown in Figure 6, N_C, is the net representing the behavior of a client. While it includes the whole abstract model, it refines the key extraction process of a client. The bubble shown with a dashed line is the refinement of the condition *key* in the interface N_I. In this refinement, receiving two pieces from chosen PKG nodes, calculating the private key and verification of it is modeled in more details. In a DPKG system where the threshold number is two, when a client applies for a private key, it receives two pieces from two available PKG nodes. The client can verify the pieces it received. If both pieces are verified

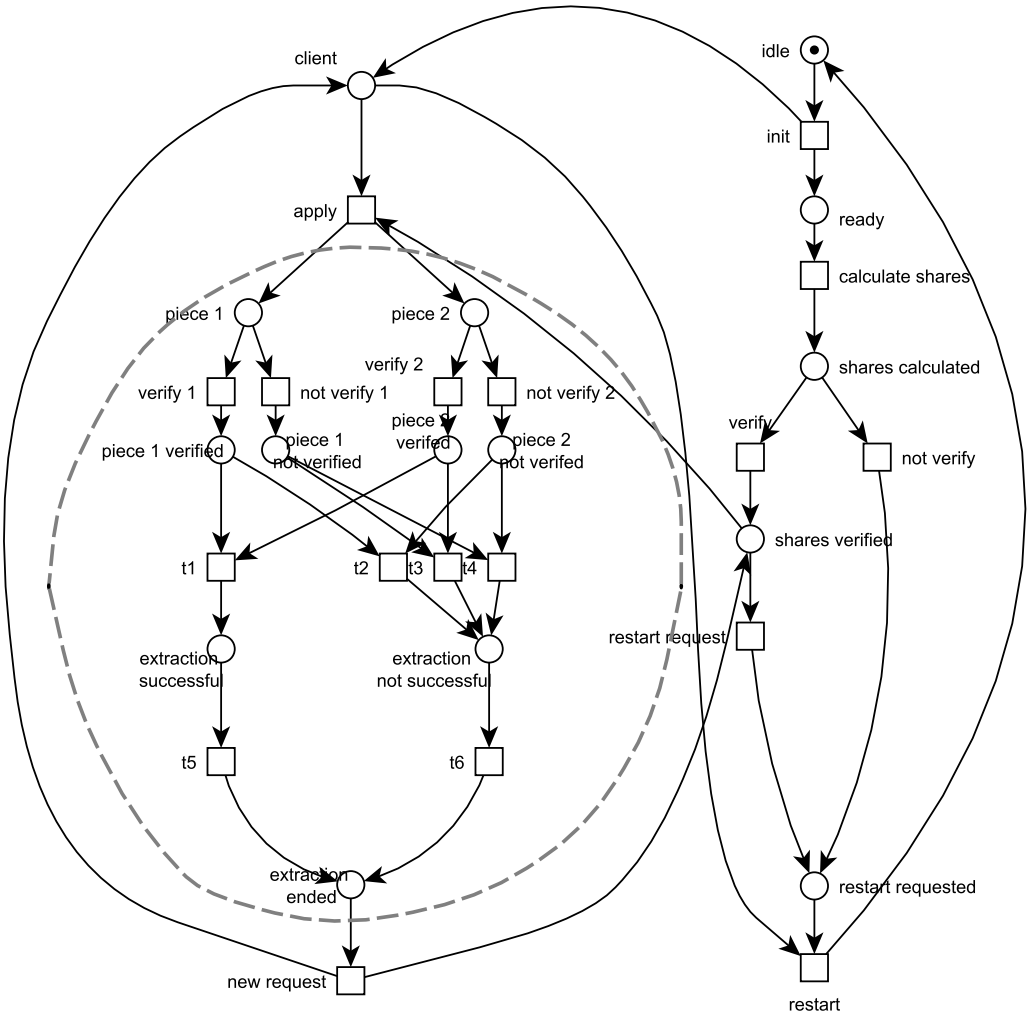


Fig. 6: N_C, the net representing the clients.

then the client can extract its private key by using these pieces and the system reaches a state where extraction is successful. In case at least one of the pieces are not verified then the condition *extraction not successful* becomes true. After both failed or successful extraction, the system reaches a state where *extraction ended* is true and a new key can be requested by the same client or by any other client in the system. Again if we improve the model for threshold value k instead of two, only the bubble will grow but the other elements of the net will remain the same.

This net is also live, reversible and covered by sequential components. Here we give the sequential components which are enough to cover the net as lists of conditions: {*idle, ready, shares calculated, shares verified, piece 1, piece 1 verified, piece 1 not verified, extraction successful, extraction not successful, extraction ended, restart requested*}, {*idle, ready, shares calculated, shares verified, piece 2, piece 2 verified, piece 2 not verified, extraction successful, extraction not successful, extraction ended, restart requested*}, {*client, idle, piece 1, piece 1 verified, piece 1 not verified, extraction successful, extraction not successful, extraction ended*}.

It is very easy to see that the first four requirements of α-morphism are already satisfied so we can continue with checking the rest of the requirements. The bubble contains no cycles so 5a is satisfied. All the in-elements of the bubble are generated by the only one event entering the bubble which is mapped to the corresponding event in the interface, *apply*. There is also only one post-event of out-condition of the bubble which empties the bubble and this event is mapped to the post-event of *key*. With these observation it is easy to see that 5b and 5c are satisfied. 5d is also satisfied because the conditions that are internal to the bubble have pre-events and post-events which are all mapped to the refined condition *key* in N_I but for in and out-elements.

Finally, as we already listed the sequential components of the net, it is easy to see that for each condition of the bubble there is a sequential component containing that condition and all the pre and post-events of the bubble, so requirement 5e is satisfied. Considering all the requirements, we can say that there is an α-morphism between N_C and N_I. Now that we proved that there is an α-morphism both from N_{PKG} to N_I and from N_C and N_I, we can prove that the composed net is weakly bisimilar to the interface by showing that some additional requirements which are stated as **c1**, **c2**, and **c3** in Section 3 are satisfied by N_{PKG} and N_C. Proposition 1 states that if both of the components are weakly bisimilar to the interface, then the composed net is also weakly bisimilar to the interface. Thus, here we first show that N_C is weakly bisimilar to the interface N_I. To do this, we follow the construction of the two auxiliary nets given in Section 3, i.e., we consider the bubble in N_C and the corresponding condition *key* in N_I and we add their pre and post-events to the subnets. We also add two more conditions to each subnet: one condition to be a pre-condition to all pre-events and another condition to be a post-condition to all post-events. Let us name these two subnets as $S_C(key)$ and $S_I(key)$. Finally, we build the unfolding of

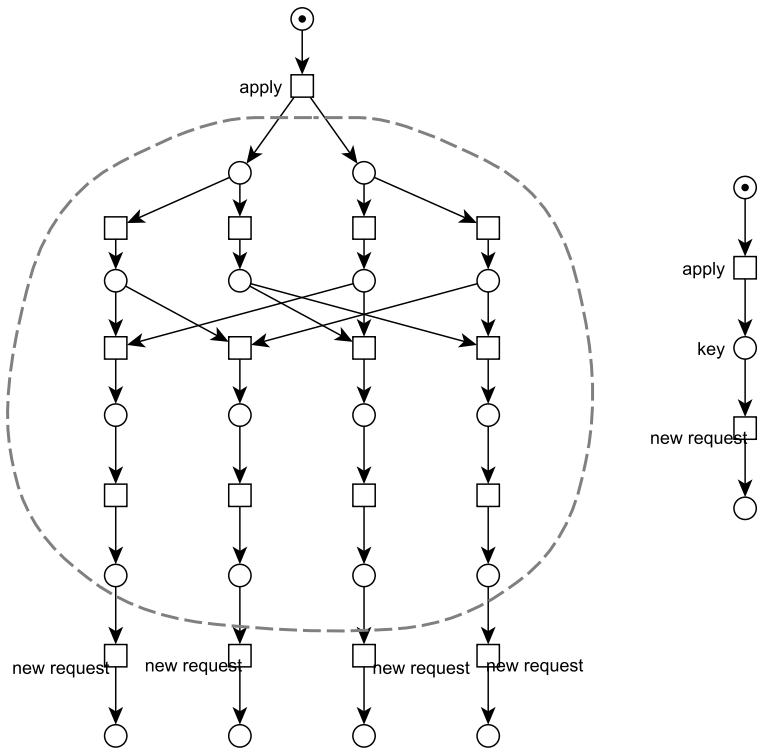


Fig. 7: $Unf(S_C(key))$ and $S_I(key)$

$S_C(key)$, represented as $Unf(S_C(key))$. The resulting nets are shown in Figure 7.

We follow the same procedure for N_{PKG} and we get two subnets $Unf(S_{PKG}(shares\ calculated))$ and $S_I(shares\,calculated)$ as in Figure 8.

When we examine these subnets, we see that no condition of the bubbles is in the initial marking. 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, so there is an α-morphism both from $Unf(S_C(key))$ to $S_I(key)$ and from $Unf(S_{PKG}(shares\,calculated))$ to $S_I(shares\,calculated)$. Thus, **c1** and **c2** are satisfied. Since there is only one bubble in both N_{PKG} and N_C, **c3** is automatically satisfied. Consequently, we can say that the additional properties **p1** and **p2** are held. Moreover, considering Proposition 1, we can conclude that the composed net $N_{PKG}\langle N_I\rangle N_C$ is weakly bisimilar to N_I.

Knowing that our nets satisfy the requirements of α-morphisms and the other three additional constraints, give us the ensurance that, in addition to weakly bisimulation, the nets preserve another important property stated in **p1**. The property of reflecting reachable markings gives us a big advantage in performing reachability analysis. Instead of analyzing the big composed net with respect to reachability of a specific marking we can analyze the interface for the corresponding marking in the interface. To give an example, we can consider the existence of the following situation in the composed net $N_{PKG}\langle N_I\rangle N_C$: the

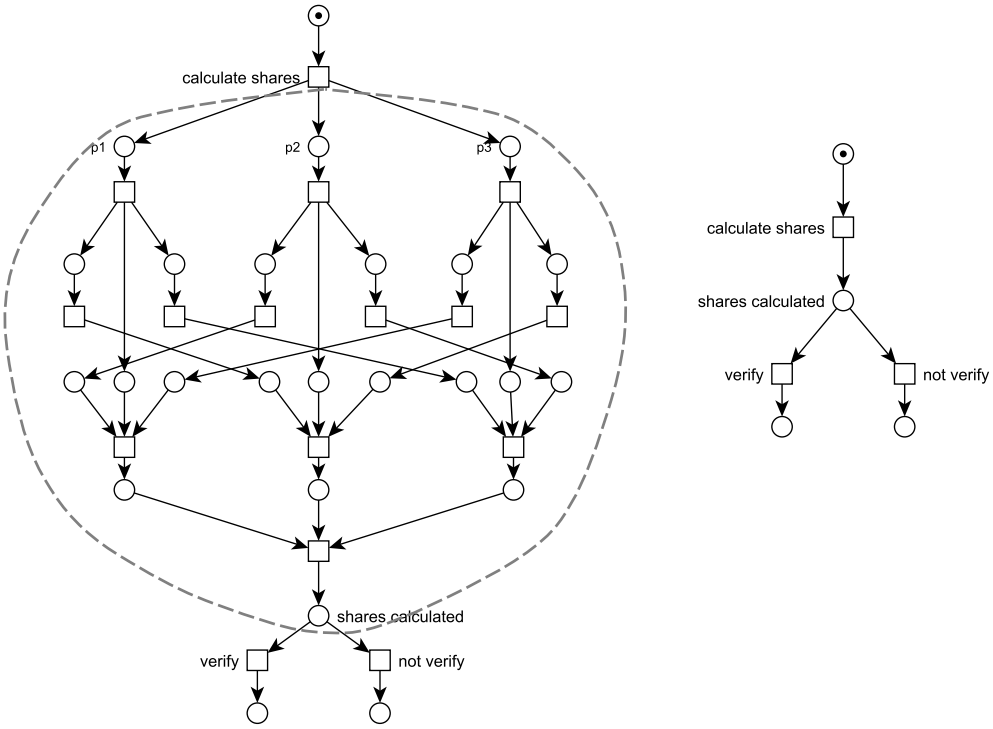


Fig. 8: $Unf(S_{PKG}(shares\,calculated))$ and $S_I(shares\,calculated)$

condition *shares verified* should not be true while there is at least one token in any bubble. Performing a reachability analysis on the composed net is complex in terms of time and space since both the net and the logic formula we have to use to represent the interested global state are big. Instead, the mentioned global state can be easily translated into a global state of the interface, N_I. Since each bubble in the composed net is mapped to a condition in the interface, reachability analysis becomes easier. The previously mentioned critical situation is reflected in the interface as the following: the condition *shares verified* cannot be true while *key* or *shares calculated* is true. Performing a reachability analysis for existence of this situation in N_I is easier than analyzing the composed net. Moreover, we do not even need to build the composed net.

7 Conclusion

We have developed a Petri net model of a protocol for distributed generation of private keys. The model has been obtained by composing two net models on a common interface. The first component models the interactions among PKG nodes, while the second component models clients of the key generator. Both components refine a common interface, representing the interactions among components.

We have then discussed behavioural properties of the model, directly derivable from properties of the components without generating the composed net. In particular, we have shown that some markings are not reachable.

On one hand, we have verified modeling and analysis capacity of the compositional approach proposed in [2] by means of a real world example. On the other side, we have proposed a model of distributed private key generation protocol by using the compositional approach.

We now plan to explore how to extend the approach to other classes of Petri nets, particularly PT nets and high-level nets. With respect to the model, we plan to improve it giving a less abstract specification in order to propose a formal verification of the protocol and to discuss its weak and strong aspects.

Acknowledgements

This work was partially supported by MIUR.

References

1. Luca Bernardinello, Elisabetta Mangioni, and Lucia Pomello. Composition of Elementary Net Systems based on α-morphisms. In Michael Köhler-Bußmeier, editor, *Joint Proc. of LAM'12, WooPS'12, and CompoNet'12, Hamburg, Germany, June 25-26, 2012*, volume 853 of *CEUR Workshop Proceedings*, pages 87–102. CEUR-WS.org, 2012.
2. Luca Bernardinello, Elisabetta Mangioni, and Lucia Pomello. Local state refinement and composition on elementary net systems: an approach based on morphisms. Transactions on Petri Nets and Other Models of Concurrency (ToPNoC), special issue based on the workshops of Petri nets 2012 (to appear), 2012.
3. Luca Bernardinello, Elisabetta Mangioni, and Lucia Pomello. Local State Refinement on Elementary Net Systems: an Approach Based on Morphisms. In Lawrence Cabac, Michael Duvigneau, and Daniel Moldt, editors, *Petri Nets and Software Engineering. International Workshop, PNSE'12, Hamburg, Germany, June 25-26, 2012. Proceedings*, volume 851 of *CEUR Workshop Proceedings*, pages 138–152. CEUR-WS.org, 2012.
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. Dan Boneh and Matthew K. Franklin. Identity-based encryption from the weil pairing. *SIAM J. Comput.*, 32:586–615, 2003.
6. Aniket Kate and Ian Goldberg. Asynchronous distributed private-key generators for identity-based cryptography. *IACR Cryptology ePrint Archive*, 2009:355, 2009.
7. Gorkem Kilinc, Igor Nai Fovino, Carlo Ferigato, and Ahmet Koltuksuz. A model of distributed key generation for industrial control systems. In *11th Workshop on Discrete Event Systems*, volume 11, pages 356–363, Guadalajara, Mexico, 2012.
8. Elisabetta Mangioni. *Modularity for system modelling and analysis*. PhD thesis, Università degli Studi di Milano-Bicocca, Dottorato di ricerca in Informatica, ciclo 24, 2013.

9. Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
10. Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part i. *Theor. Comput. Sci.*, 13:85–108, 1981.
11. Mogens Nielsen, Grzegorz Rozenberg, and P. S. Thiagarajan. Elementary transition systems. *Theor. Comput. Sci.*, 96(1):3–33, 1992.
12. 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.
13. 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.
14. A. Shamir. Identity-based cryptosystems and signature schemes. *Advances in Cryptology: Proceedings of CRYPTO 84, Lecture Notes in Computer Science*, 7:47–53, 1984.

Integrating Web Services in Petri Net-based Agent Applications

Tobias Betz, Lawrence Cabac, Michael Duvigneau, Thomas Wagner, Matthias Wester-Ebbinghaus

University of Hamburg
Faculty of Mathematics, Informatics and Natural Sciences
Department of Informatics
{betz,cabac,duvigne,wagner,wester}@informatik.uni-hamburg.de
http://www.informatik.uni-hamburg.de/TGI

Abstract. The context of this paper is given through a software engineering approach that uses Petri nets as executable code. We apply the particular understanding that Petri nets are not only used to model systems for design purposes but also to implement system components. Following this approach, we develop complex Petri net-based software applications according to the multi-agent paradigm. Agent-internal as well as agent-spanning processes are implemented directly as (high-level) Petri nets. These nets are essential parts of the resulting software application – alongside other parts (operational and declarative ones), which are implemented using traditional ways of programming.
One of our goals is to open our Petri net-based agent framework Mulan/Capa so that multi-agent applications can communicate and interoperate with other systems – especially with Web-based applications. For this cause, we present a gateway solution to enable Petri net-based applications to access Web services as well as to offer Web services to other applications: the *WebGateway*. Besides describing the WebGateway extension itself, we use its presentation to demonstrate the practicability of the Petri net-based software engineering approach in general. We emphasize two benefits: (1) Petri net models serve as conceptual models that progressively refine the constructed system from simple models to well-defined specifications of the systems. This improves the understanding of the systems. (2) Having essential parts of the software system being implemented with Petri nets allows to carry out (partial) verification of our application code by means of standard formal methods from the field of Petri net theory.

Keywords: Web Services, High-Level Petri Nets, Multi-Agent Systems, Mulan, Renew, P*aose

1 Introduction

One of the most frequent requirements for modern software applications is to open the access of the offered functionality to other entities in the World Wide

Web. In this paper we address the topic of meeting this requirement for Petri-net based software applications. We present a gateway solution for allowing Petri net-based applications to access Web services as well as offering Web services themselves.

The usefulness of Petri nets for software engineering has been recognized in the context of many paradigms like object-orientation [1], components / plug-ins [7,16,22] or agent-orientation [17,20]. However, in this paper we assume a particular understanding of *Petri net-based software*. In addition to using Petri nets for design-level artifacts and for verification of certain system properties, we utilize Petri nets as our *implementation language*. More concretely, we rely on the high-level Petri net formalism of *Java reference nets* [7,18] that allows to combine multi-level Petri net modeling (according to the *nets-within-nets* concept [25]) with Java programming. The formalism is supported by the Renew[1] tool (`http://www.renew.de`). We have developed the multi-agent system (MAS) framework Mulan[1] based on Java reference nets. It provides a powerful middleware for running distributed multi-agent applications on multiple instances of Renew. In addition, we have developed a Petri net-based agent-oriented software engineering approach (P*aose[1]) for the construction of such multi-agent applications.

With the *WebGateway* extension, we introduce the latest addition to our Petri net-based software engineering framework Mulan/Capa. While the Mulan model is often referred to as the reference architecture, Capa (Concurrent Agent Platform Architecture) is an extension and implementation of Mulan. Capa [11] provides convenient ontology-based message processing and an infrastructure for FIPA-compliant agent management and IP-based transport services. Capa is, thus, one implementation of the reference architecture Mulan. It allows to integrate Mulan applications into Web-based environments via Web services. This *opens* Mulan applications in the sense that Mulan agents can now access resources external to the agent world in a uniform way via Web Services instead of having to be equipped with proprietary connectors. In the other direction, Mulan agents publish and offer their own services also as Web services and thus can equally be accessed uniformly from anywhere across the Web. Again, we stress our specific understanding of Petri net-based software, which carries over to the integration with Web services. Usually, work on Petri nets and Web services deals with providing semantics to modeling notations that are used within the Web context, like BPEL [13] and some translations [15]. Our approach is to provide a way to offer Web services that are actually *realized by the execution of Petri net models*. The other way round, the *execution of Petri net-based applications can include the access of arbitrary Web services*.

The concrete aim of this paper is twofold. Firstly, we introduce the WebGateway itself as a solution for bringing Petri net-based applications and Web services together. Secondly, we use the WebGateway extension as an example

[1] For more detailed information about Renew, Mulan and P*aose see [5] and `http://www.paose.net`.

for the general benefits that underlie our approach of Petri net-based software engineering. We claim that these benefits are basically:

1. We follow an engineering approach, in which we move from conceptual models as design artifacts to refined, technical models as software artifacts. In our opinion, this approach of *implementation through specification* allows to iteratively build models/code in a documented and comprehensible way. In addition, core features of a system can be determined early on and maintained in further refinements.
2. By using *Petri nets as code* we can verify our application code. Of course, this can only happen within certain limits. Both the nets-within-nets nature of our models and the use of Java inscriptions prohibit a comprehensive verification. Nevertheless, we can define abstractions (e.g. P/T net abstractions) of our program code and verify specific properties (e.g. properties of sound workflows) with respect to them. This allows at least partial verification of our application code (which could possibly be supplemented with unit testing mechanisms for reference nets, cf. Section 8 and [8]).

The outline of the paper is as follows. In Section 2, we present the conceptual model of our WebGateway extension for bringing Petri net-based (agent-oriented) applications and Web services together. Based on this, we present details of the WebGateway implementation in Section 3. In combination, these two sections demonstrate the benefits and practicability of our *implementation through specification* approach. In Section 4, we demonstrate how the WebGateway and one particular Petri net-based Web service are deployed in the context of our integrated project management environment (IPME) on a day-to-day basis. Section 5 presents a further insight to the implementation in order to demonstrate the refinement process of the model. We discuss the results of the paper in Section 6, position them in the context of related work in Section 7 before we conclude the paper in Section 8.

2 Conceptual Gateway Architecture

The WebGateway extension to our multi-agent framework Mulan/Capa is realized by a *WebGateway agent*. This agent is coupled to a (jetty) Web Server and thus brings the two worlds of the (Web service-based) Internet and multi-agent systems together. In Section 3, we address technical details of the WebGateway realization. In this section, we focus on the conceptual architecture. Along the way, we demonstrate the usefulness of applying a Petri net-based modeling approach. We progressively refine a simple architectural model to a meaningful and well-defined specification. This specification then represents the basis on which to actually implement the WebGateway agent's behavior. Due to space limitations, we cannot address the whole functionality of the WebGateway. Instead we concentrate on handling Web requests and responses. Further topics and challenges will be addressed in the following section.

Figure 1 illustrates our starting point. In order to provide communication in such a heterogeneous setup consisting of Web and multi-agent system parts, the communication has to be facilitated. The WebGateway provides the translation as an adapter between the two worlds of communication. For this it provides two interfaces: one for Web-based communication and one for FIPA[2]-compliant communication. The interfaces are depicted in Figure 1 as white rectangles. The Internet is shown as a cloud and the MULAN reference model (cf. [17]) stands as an exemplary multi-agent system.

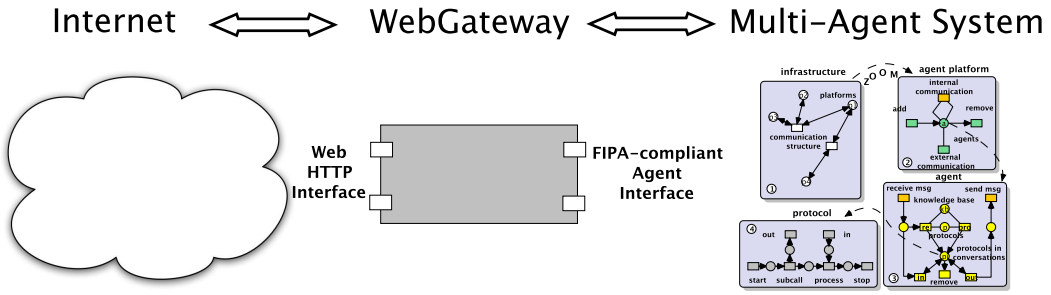


Figure 1. WebGateway context.

In the following we will consecutively refine the WebGateway as a Petri net model. Figure 2 shows the first step of this refinement. The WebGateway's main functionality of translating messages from one domain to the other is represented as two transitions that are included in the transformation component. The two interfaces are now depicted as transitions.[3] Messages may enter through the interface transitions with outgoing arcs, are received on the buffer places and ready for transformation processing.

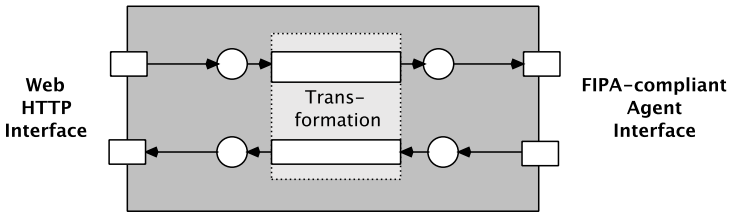


Figure 2. Simple WebGateway architecture model

[2] Foundation for Intelligent Physical Agents: `http://www.fipa.org`.

[3] The notion of transitions being interfaces fits nicely with an object-oriented paradigm, if one presumes that these (on one side open) transitions are one port of synchronous channels.

Translation is an important and already technically challenging part (cf. the following section) but by far not the only task of the WebGateway. In addition, it has to make sure that responses are matched to requests across the heterogeneous setup. In our approach the WebGateway keeps a copy of a request message in order to be able to provide this matching. Thus responses can be routed to the right recipient. In Figure 3 an exemplary conversation direction is modeled: a request from the Web to an agent-implemented service. The original request (e.g. from a Web browser) is a JSON message (JavaScript Object Notation). The WebGateway translates this message to FIPA-SL (Semantic Language), which can be understood by agents in the multi-agent system. A copy of the message is kept within the gateway, which allows the gateway to route the response message to the requester after the answer has been translated back from FIPA-SL to JSON. Please note that JSON stands for just one possibility of a Web message's content. Content types like XML or HTML form data can be translated in a similar fashion.

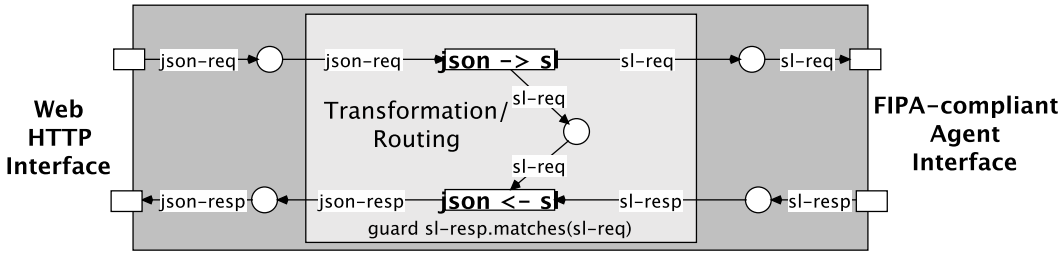


Figure 3. WebGateway architecture model for handling Web-client requests

While Figure 3 covers one exemplary interaction type, namely a request sent by a Web-based client to an agent-based service, the WebGateway also provides the possibility that a Web service request is initiated from the agent's side. In this case an SL encoded request will be translated to (for instance) JSON, a copy of this message will be kept for later routing and the answer from a Web application will be translated from JSON to SL in order to deliver it to the requesting agent. Both initiating directions are supported by the WebGateway and use the same interface as depicted in Figure 4.

In addition to the request interactions covered so far, the WebGateway also supports a uni-directional communication (*inform* interaction), which is not discussed here.

While this conceptual Petri net model of the WebGateway architecture shows the basic (internal) behavior of the WebGateway, it neither specifies the interactions between WebGateway and Web applications or agents nor does it present a realistic level of detail for the implementation of the WebGateway agent. These details are covered in the following section.

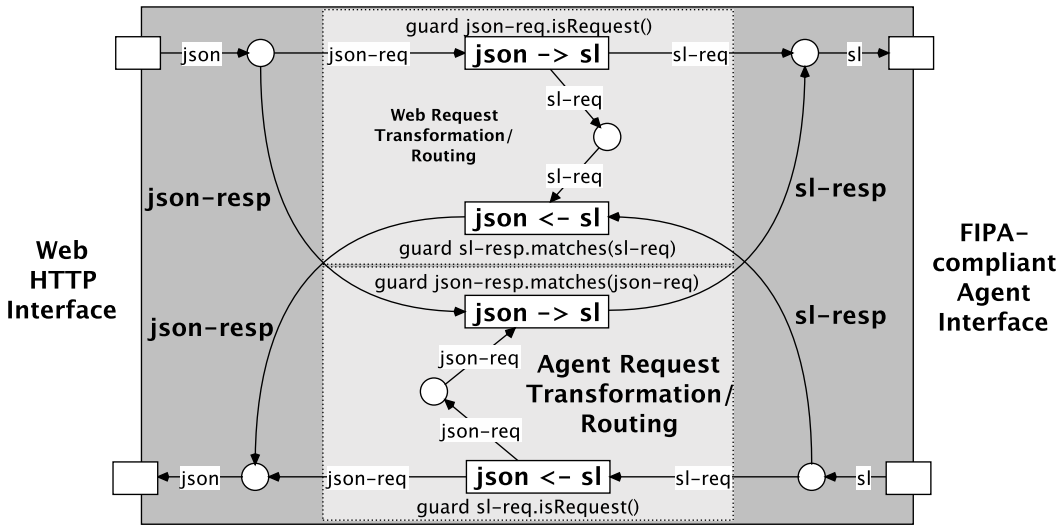


Figure 4. WebGateway architecture for two-way service request handling

3 WebGateway: Integration and Details

In order to achieve a concrete implementation of the abstract architecture described in the previous section we need to combine multiple technologies, which are well established in the world of multi-agent systems and Web services. In this section, we describe how those technologies are combined for the implementation of the WebGateway in order to obtain the desired integration of both application domains. For this we present two parts of the adapter functionality of the WebGateway. The first is concerned with the message routing and translation as well as service registration. It focuses on the Web interface side, which is – from the perspective of the muulti-agent system – the external interface. The realization of this interface, which requires the integration of the required technologies, is presented in Section 3.1. The second part focuses on the WebGateway as a part of the agent system and its communication with other agents. Hence, Section 3.2 introduces the communication protocols providegd with the WebGateway.

3.1 Integration of the Required Technologies

An initial requirement is that the WebGateway must be able to interact with communication partners of both worlds. For that reason the WebGateway provides two communication interfaces as shown in Figure 5 where we have included the conceptual architecture model from the previous section in order to illustrate its relation to the actual WebGateway implementation.

As the WebGateway is realized as an agent itself, the agent interface for communication with other agents is inherently part of the underlying multi-agent system framework (Mulan/Capa in our case). Consequently, the *ordinary* FIPA-compliant agent communication infrastructure of our framework enables

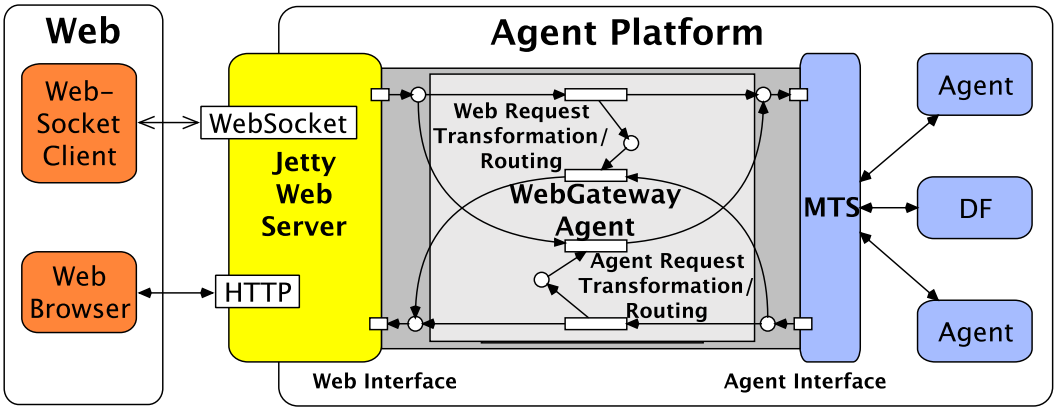


Figure 5. Integration of the WebGateway in the multi-agent system

communication with other agents both on the same platform and on remote platforms. This part can be considered as the WebGateway's *internal communication interface*.

In addition, the WebGateway agent needs a Web interface that serves the communication with Web services and Web clients. It is realized using a Web server (Jetty, `http://www.eclipse.org/jetty`). This is where we have extended our framework. For each Mulan/Capa host, one Web server is launched. A Mulan/Capa host may include multiple agent platforms, but typically we have a one-to-one correspondence between a host and a platform. For each platform, a WebGateway agent is launched and connects automatically to the platform's (/host's) Web server. This part can be considered as the WebGateway's *external communication interface*. The Web server enables communication between the WebGateway agent and Web services/clients using the well established HTTP protocol (Hypertext Transfer Protocol, `http://www.w3.org/Protocols`) as well as the HTML5 WebSocket protocol (`http://dev.w3.org/html5/websockets/`). In contrast to HTTP, a WebSocket connection allows to exchange messages asynchronously between client and server. This allows more flexibility for browser-based Web applications and fits quite well with the agent paradigm as it traditionally relies on asynchronous interactions.

Besides mediating communication *technically*, there remain further challenges in order for the WebGateway to really provide a transparent and bidirectional communication between the agent and the Web service world. We identify the following required key features.

1. Routing and management of messages between the different interfaces.
2. Registration and management of agent services that are published as Web services and vice versa.
3. A two-way translation of the supported message encodings.

The first mentioned feature is specifically addressed by the conceptual architecture for the WebGateway from the previous section. More (technical) details

of our solution concerning the cross-technological tracking and routing of messages can be found in [4].

For the second mentioned feature, our approach supports – and actually is limited to – RESTful Web services. The **RE**presentational **S**tate **T**ransfer (REST) architecture [12] gained increased attention because of its simplicity of publishing and consuming services over the Web. The architecture is based on resources that identify the participants of service interactions and that are addressed with globally unique URIs[4]. Such a resource can be manipulated over a uniform interface with a fixed set of operations (GET, POST, PUT, DELETE, etc.) that are traditionally part of the underlying HTTP networking protocol. Resources are also decoupled from their representations so that their content can be accessed in a variety of formats e.g. HTML, JSON[5], XML or even JPEG images. For our purposes, we treat artifacts from the multi-agent world (hosts, platforms, agents, agent services) as REST resources in the Web world. The technical counterparts for these *agent-based REST resources* on the Web server side are implemented as Java Servlets[6]. These are responsible for providing the resource representations and also act as connection endpoints for HTTP and WebSocket connections, forwarding all incoming messages to the responsible WebGateway agent. In [4], we provide more details on addressing agent-based REST resources and on how to provide suitable presentations.

The last mentioned feature was also briefly addressed in the previous section (translation between FIPA-SL and JSON/XML/HTML form data). We will not cover this topic here, but again refer to [4].

3.2 The WebGateway as a Mulan/Capa Agent

So far we have mainly focused on Web technologies that are necessary for realizing the WebGateway according to the conceptual architecture presented in the previous section. However, we have already stressed the fact that the WebGateway is actually realized as an *agent* in our Mulan/Capa framework. We have presented the development approach (P*aose) for Mulan/Capa multi-agent systems together with corresponding tools on other occasions (cf. [5,6]). Basically, a Mulan/Capa agent is designed in terms of three aspects: agent knowledge, agent-internal processes, agent-spanning processes. These aspects eventually manifest in three types of software artifacts for agent implementation: a *knowledge base*, *decision components* (DCs) for managing agent-internal processes and *protocols* for managing interactions with other agents. In this paper, we will not comprehensively cover all details of developing the WebGateway agent but provide an overview of the necessary parts.

Basically, the conceptual architecture described in Section 2 provides the groundwork, on which the agent's decision components are designed. In the previous subsection, we have covered the technologies that are needed to flesh out the conceptual architecture in order to arrive at an actual implementation.

[4] Uniform Resource Identifier

[5] Javascript Object Notation (JSON)

[6] `http://www.oracle.com/technetwork/java/index-jsp-135475.html`

One central aspect of agent design is its interactions with other system parts. Interactions between the WebGateway and Web applications take place via HTTP/WebSockets. For interactions between the WebGateway and other agents we have to provide equally well-defined protocols. Basically, the WebGateway agent offers five protocols for this purpose:

- WebGateway_registerAgent for registering agent services as Web services.
- WebGateway_sendrequest for forwarding request from Web applications to application agents
- WebGateway_receiverequest for forwarding request from application agents to Web applications
- WebGateway_sendinform and WebGateway_receiveinform for sending inform messages in both directions

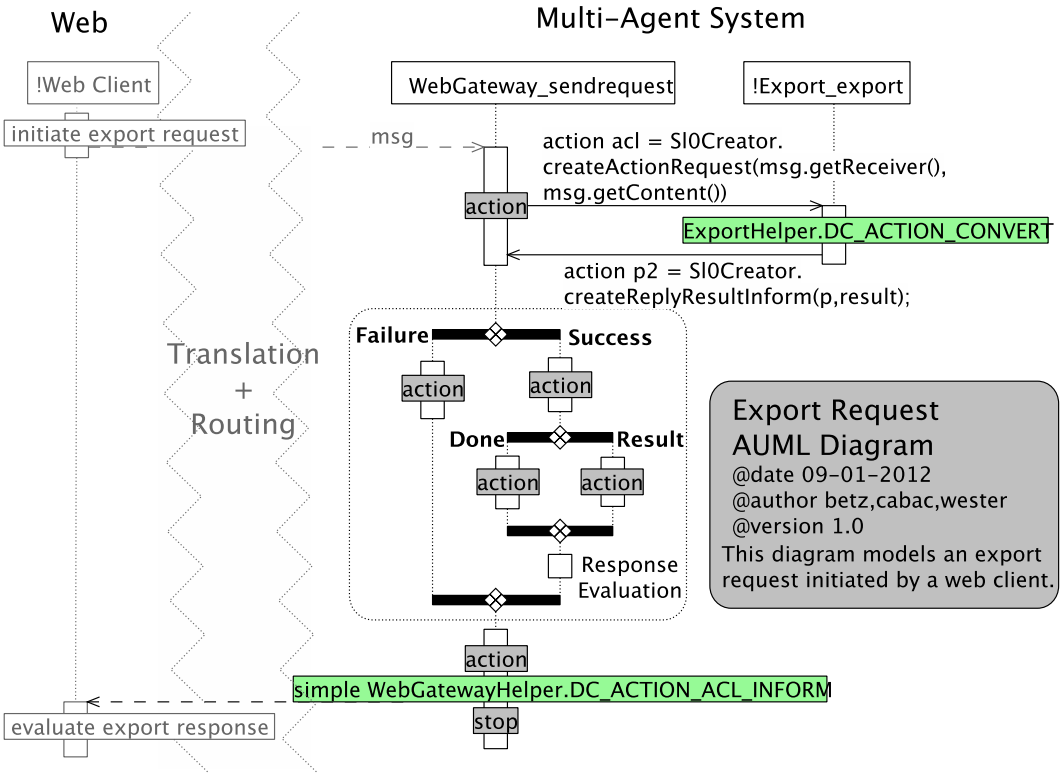


Figure 6. A model of the export request interaction initiated by a web client.

We cannot cover all of these interactions but will turn to one example. Figure 6 shows an AUML diagram for the case of a WebGateway_sendrequest. The AUML (Agent UML, see [9,6]) Interaction Protocol diagrams are derived from Sequence Diagrams. They allow to fold several sequences into one scenario by providing modeling elements for alternatives or concurrency – similar to UML2 Interaction Diagrams. Here, we regard the sample case where a Web application

requests the export functionality of an application agent and the WebGateway agent acts as the mediator. We address the export application scenario more deeply in the next section.

In the P*AOSE approach, we use these AUML diagrams to (semi-)automatically generate the interaction parts for each party as Petri nets (cf. [9]). These resulting *protocol nets* are then directly used for the implementation of agent interactions. It is important to note that the WebGateway_sendrequest protocol net is generic. Here, it is shown in the context of the export example. But is designed to be applicable for arbitrary requests sent from a Web application to an application agent. For instance, we have developed a web component-based GUI framework for browser applications that relies on exchanging web component events between browsers and agents (so called *Agentlets*).

For each new way of making use of the WebRequest_sendrequest protocol net, the two interaction sides have to *fit* together. This means that the composition of the WebRequest_sendrequest protocol net and its counterpart protocol net has (at least) to result in a sound protocol.[7]

4 Application of the WebGateway / Export Example

As a real world example we present an application that utilizes the WebGateway's functionality to provide a Web service. The *Export Service* takes a representation of a Petri net in the form of a serialized Renew drawing or as PNML and returns an image representation of the model.[8] The *Diff Service* service takes two representations of models and returns a graphical diff [10] of the two models.

People can access these Web services directly through a Web page interface. But the main application for the Web services is currently a different one. In the context of software engineering, in which we use our models, a tight integration of available tools ensures the efficiency and therefore the acceptance of the available tools among developers. Thus we have integrated the two Web services in our preferred integrated project management environment (IPME, see [3]).

Figure 7 shows a schematic model of the setup of our IPME. The IPME – in this case: *Redmine* (`http://www.redmine.org/`) – runs on a standard Web server shown in the center of the model. It includes several plugins for the access of the source code management system (SCM, possibly located on another server) and the Export/Diff Web services (again located on another server). Developers can interact with the source code repositories to introduce new versions

[7] Although the soundness property is not well-defined for protocol nets, we conceive this property in analogy to soundness of workflow nets.

[8] In fact the service takes any file type that can be read by Renew, e.g. Renew nets (.rnw), JHotDraw drawings (.draw), PNML, several diagram types used within P*AOSE (.aip,.arm) and also Lola net files (.net). The RENEW import/export system is also extensible, so any envisioned file type in the context of Petri nets and UML modeling can easily be implemented.

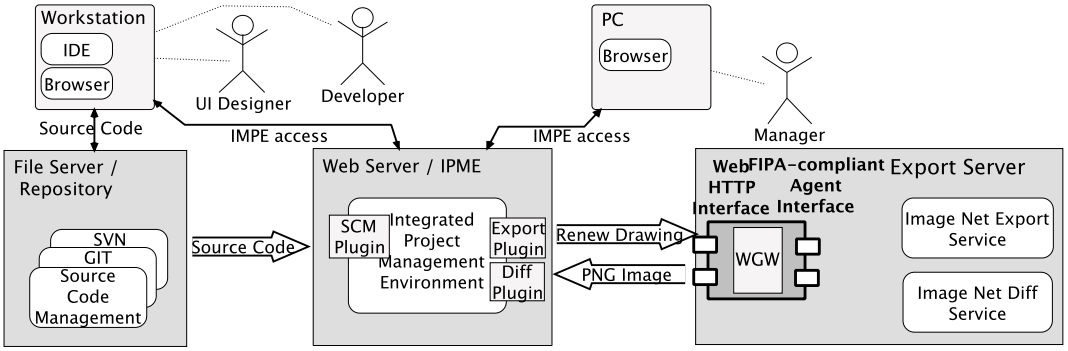


Figure 7. A schematic architecture for the export Web Service / Redmine plugin.

of artifacts, to examine the commit history or to examine differences of the selected versions and so on. Managers as well as developers can in addition use the IPME – besides of using the planning and documentation features – to investigate the source repository comfortably in a Web browser. One main part of the functionality provided by the IPME is that developers and project managers can browse quickly through the source code and choose to display a diff of versions of the source code in a Web browser. However, the default browsing and diffing functionality of IPMEs works on text-based source code only while a large part of our code base is Petri net-based. A textual representation of diagrams – for example in PNML – is not very significant for human readers. Moreover, a text based diff of versions of the diagram's text representations is completely useless. Thus, the Export and Diff plugins take the text representations for diagrams from the SCM, hand them to the Export and Diff Web services and integrate the returned images smoothly into the Web page-based display for the developer. Figure 8 shows a screenshot of the integration of the Diff Service in the IPME Redmine. The screenshot shows The Redmine user interface in a Web browser. The diff of revisions 9044 and 9545 of the *Receiver_chat* protocol net is displayed. The differences are highlighted in red (removals) and green (additions).[9] All other graphical elements are faded to a foggy gray leaving a shadow of the original net.

Consequently, within our development environment the Web services are used by the IPMEs[10] and are thus provided to the developers in an automated way. A server instance of Renew is running and provides the Web services using the Mulan/Capa framework with its WebGateway extension. A publicly accessible Export/Diff Web page and a demonstration page of the Redmine integration can be accessed from the P*aose Web Site (`http://www.paose.net/`).

Figure 9 shows a screenshot of the presented multi-agent application showing an export interaction. The MulanViewer on the left shows the multi-agent system's status in terms of all started agents, their decision components, knowledge

[9] In black & white printing the location of the manipulated parts are still recognizable, although it becomes impossible to distinguish removals from additions.

[10] We provide plugins for Redmine and Trac (`http://trac.edgewall.org/`).

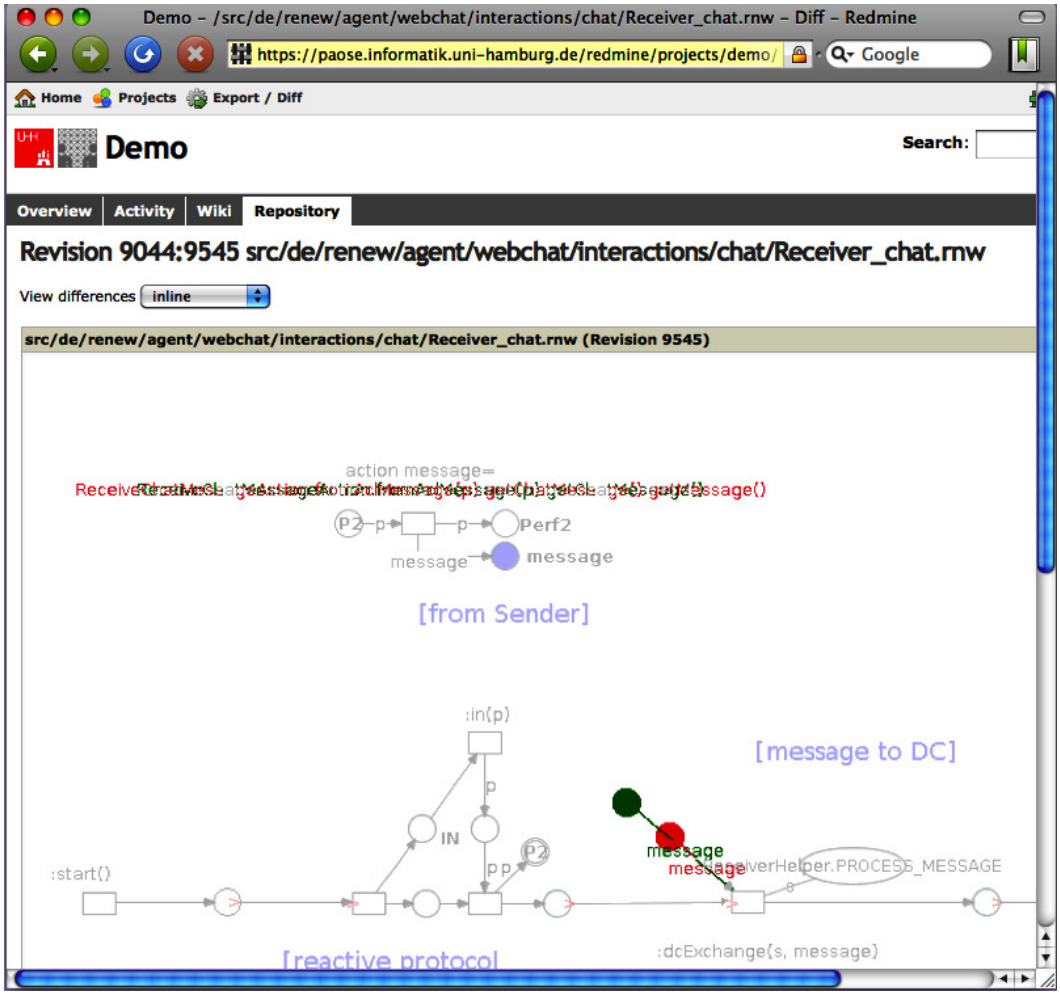


Figure 8. Screenshot of a diff image integrated in Redmine (Demo).

bases and currently executed protocol nets. In the back, parts of the involved nets are shown: these are – from top to bottom – the *transformation* decision component of the WebGateway and the *export* protocol net of the Export agent. The *sendrequest* protocol net of the WebGateway is not shown but listed in the MulanViewer's tree view. During this interaction the WebGateway *sends* the request to the Export agent after the request has been *received* from the Web client. On the right hand side of the screenshot are two frames showing a deep inspection of tokens, which are located as indicated on the highlighted places. The first one shows the *request* message in FIPA-SL, waiting to be matched with the response for routing purposes (cf. Section 2). The second is the response message, which is just about to be sent from the Export agent to the WebGateway.[11]

[11] Although the messages could also be inspected in String representation, the UML representation is much clearer and more concise.

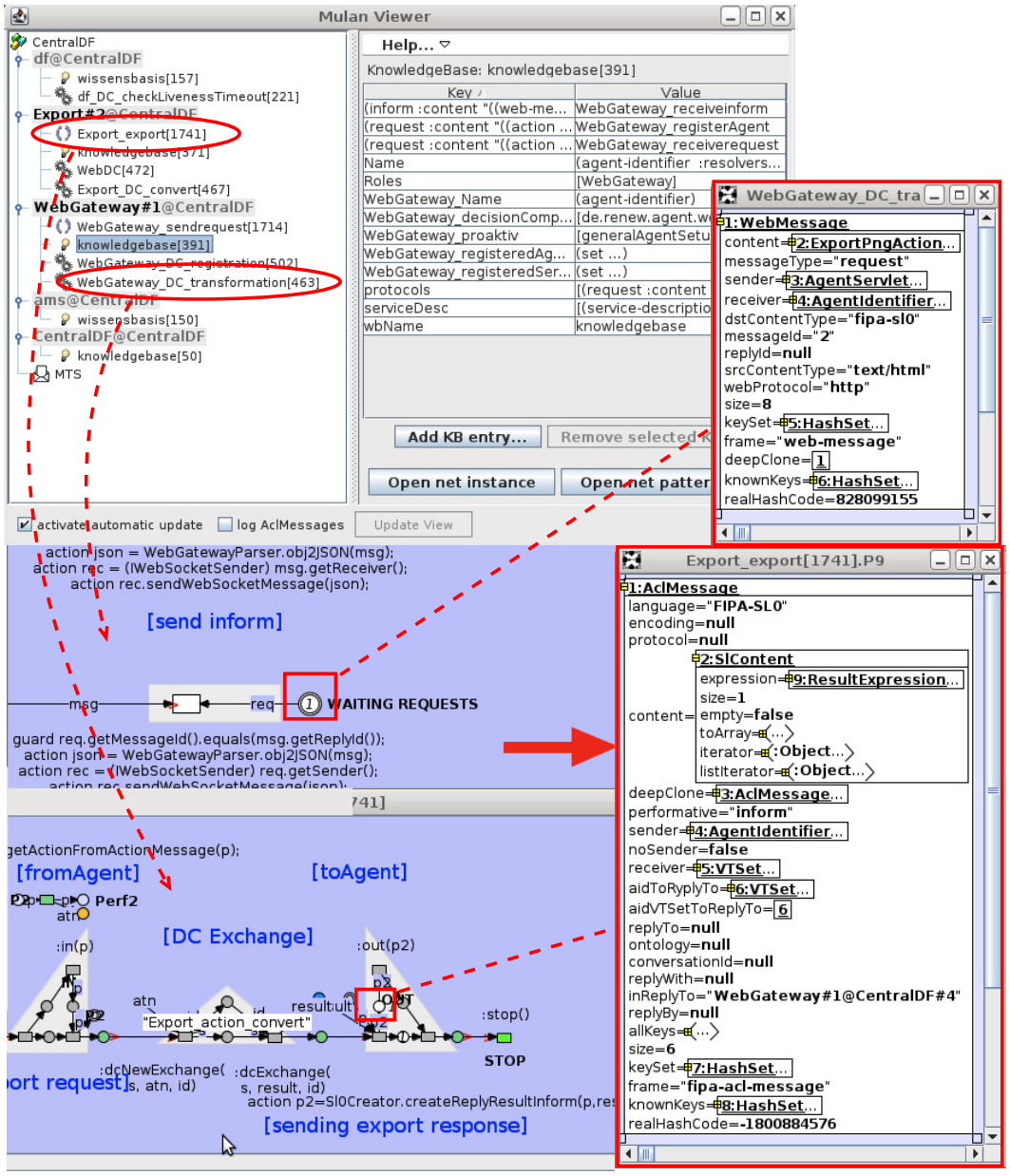


Figure 9. A screenshot of the agent application while running the export interaction.

The availability and the robustness of the Export and Diff Web services provided through a running instance of a Petri net application shows that our framework is already beyond a pure proof of concept.

5 Implementation

Although executable models tend to grow to a size that cannot be presented in all details, we present the implementation of the transformation component in

order to discuss the specification refinement that leads to the executable model. Often the process that leads incrementally to the executable model has been presented as *implementation through specification*. In this executable model we do not discuss the inscriptions and certain technical details such as the preparation and selection of the messages. Figure 10 shows an executable version of the WebGateway Transformation Component as an overview.[12] The details of the main parts are presented again in Figure 11. Compared to the abstract model shown in Figure 4 this model shows several refinements. First of all the interfaces – indicated by the dashed boxes – of the component have been duplicated. This results from the fact that our implementation allows for additional communication protocols and two connection types.

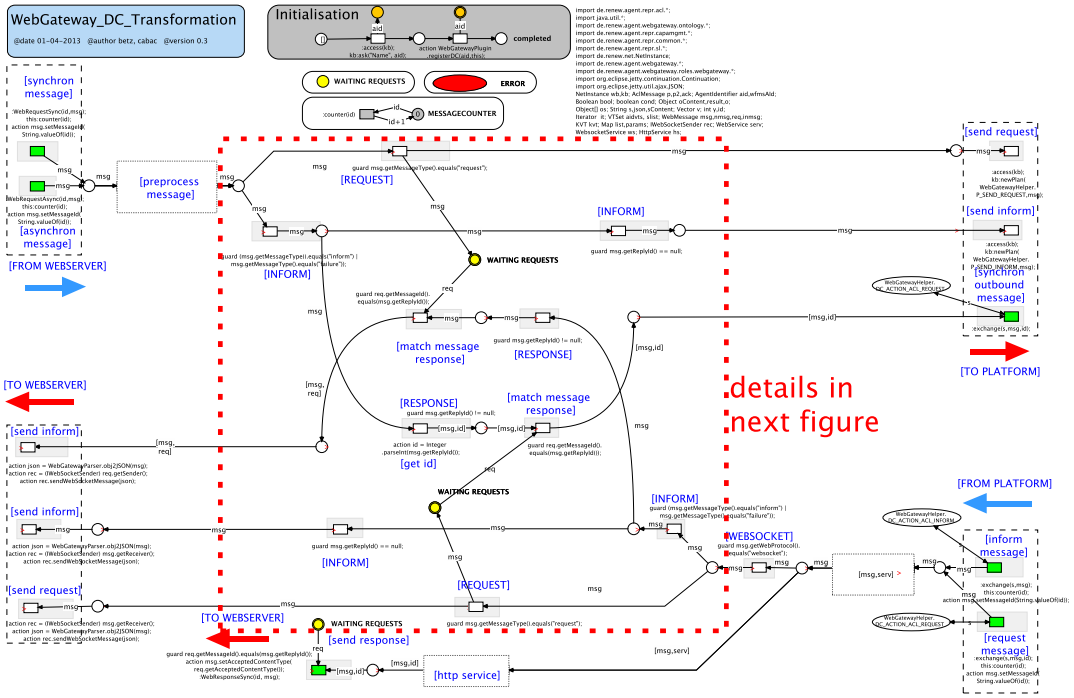


Figure 10. Implementation of the WebGateway Transformation Component.

In the abstract model we described the possibility to serve a typical request protocol. Thus, one party can send a request message and receive an answer to this in the form of an inform message. This protocol may be triggered from either side the web server interface or the agent system interface. Additionally, the implementation also allows for a simple inform message that has not been triggered and does not expect any follow-ups. Consequently, we have three out-

[12] The Petri nets is presented as a whole, in order to show the final result of the refinement process. Most of the details, such as declarations are only of minor importance. The main details are presented in a magnified version in Figure 11 for convenient inspection.

going transitions for these three different communication possibilities (request, inform as answer, simple inform) on each outgoing interface side. Additionally, we included the possibility to connect as a Web service in two possible ways. The first is the well-known http connection, the second, which is a websocket connection, allows for asynchronous communication through a bidirectional permanent connection link.

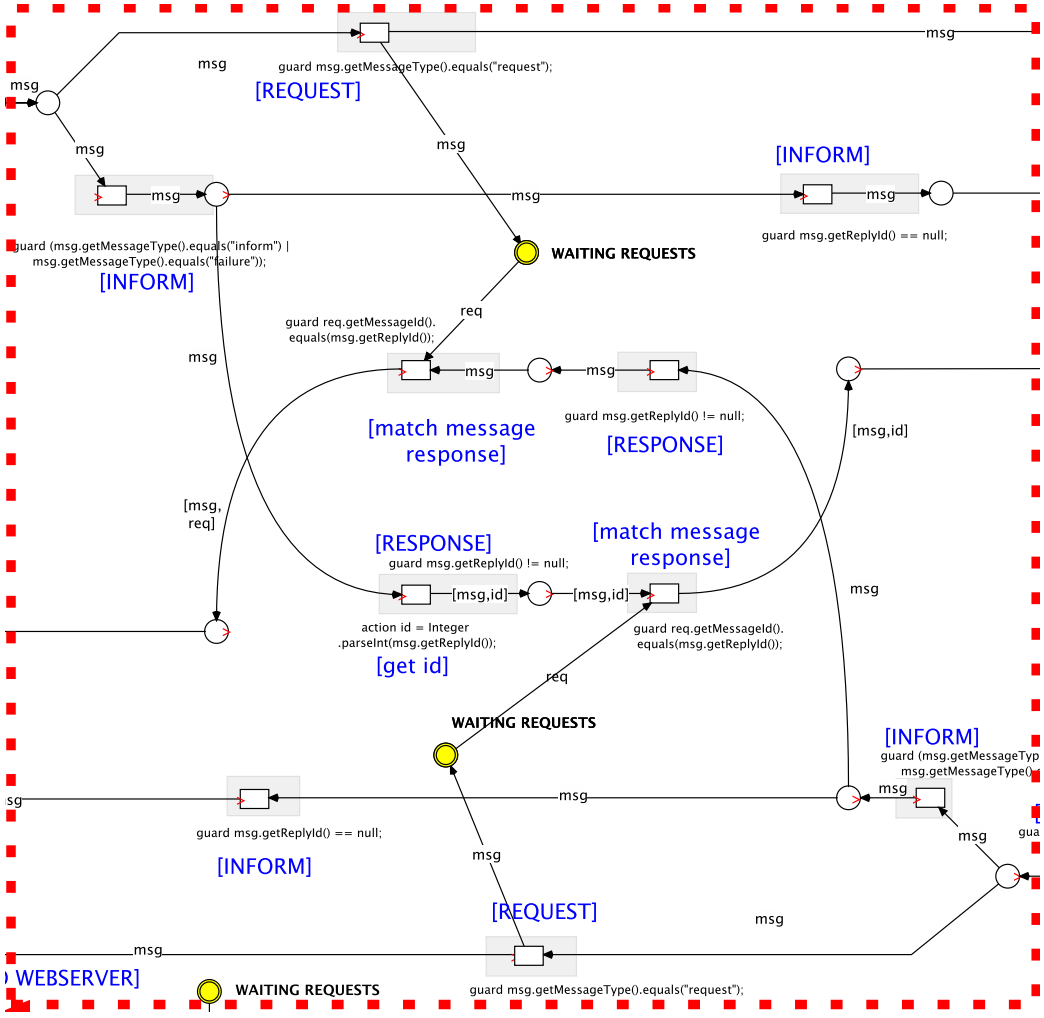


Figure 11. Fragment of the WebGateway Transformation Component.

In the center of the net – displayed in more detail in Figure 11 – one can prominently conceive the buffer places that hold the messages for matching answers to request and thus to determine the receiver. The buffer places are filled during the processing of the requests – constituting the first branch leading to the first of an outgoing interface. The answer collects the waiting message during processing – indicated by an arc – leading to the second interface transition. The last interface transition serves for the simple inform message. We have hidden

several parts of the original net – indicated by large dotted transitions. These transitions hide message processing and routing as well as the processing of the http answers.

6 Discussion

The presented example and the publicly available Web services and Web site demonstrate that the presented application is more than just a proof of concept. Especially the availability of the Export and Diff services as Web services has proven their usefulness in the straight-forward and seamless inclusion within several used instances of IPMEs (internal as well as public). We have also briefly mentioned the realization of Web-based GUIs where Web events related to the GUI components are transmitted between a browser and application agents (Agentlets). While this application of the WebGateway is still under development and still in an experimental stage, we have successfully made use of it in academic projects where students develop browser-based applications for collaboration support.

The choice of RESTful Web services in combination with WebSockets brings more flexibility to our gateway than strict WSU stack-based gateways can provide. Currently, we use rather simple service descriptions, which might hinder the automation of service workflows. A possible improvement in this area could be the integration of WADL[13], which plays a similar role for RESTful Web service as WSDL for SOAP Web services.

7 Related Work

Service oriented architectures especially in combination with Web applications is currently a popular field in the research community. Hence, the integration of Web services into multi-agent systems is a well researched topic with many interesting solutions. Such an approach that impacts on our introduced architectural design is presented by Greenwood et al. [14]. They introduce a *Web Service Integration Gateway Service* (WSGIS), which acts as a broker between the service participants and provides translation and routing mechanisms. Shafiq et al. [24] offers a slightly different solution that addresses the interconnection of FIPA-compliant multi-agent systems and Web services. Their approach rely on a middleware software that handles the communication of the service participants without any modification on the respective communication systems. In contrast to theses approaches, which are based on Web services that use the standard WSU[14] stack, the approach of Soto [19] makes use of the advantages of Web services that comply with the RESTful architecture [23]. He provides a *Web Service Message Transport Service* (WSMTS) for JADE platforms that is capable of handling FIPA-SL messages in XML representation. These messages

[13] Web Application Description Language (WADL): `http://java.net/projects/wadl`
[14] WSDL, SOAP, UDDI (WSU)

are extended with additional information that ensure an end-to-end communication with only one single message encoding. In this case, agents are able to register themselves with a specific address to publish their services as a REST service.

A still problematic issue, concerning the use and composition of RESTful services, is the change of state of a resource and the corresponding update of clients. Especially Web clients have to constantly send *GET* requests to check if the state has changed, which will result in heavy traffic and unnecessarily high load of Javascript. For a bidirectional communication, as used for instance in a User Interface Framework, Aghaee et al. [2, Section 5.2] recommend the use of W3C WebSockets [26]. The WebSocket API provides a mechanism to sent data in various representations (JSON, XML, etc.) to clients when the resource has changed.

An approach concerning the modeling of Web services using high-level Petri nets was given by Moldt et al. [21]. The authors introduce a four layer architecture that focuses on modeling the internal behavior of a Web service and provide a proposal for lifecycle management and interconnection of Web services.

With the approach presented in this paper we provide a prerequisite that enables us to verify the soundness of internal Web service processes by examination of agent interactions. In order to examine the composition of Web services we have to extend our approach to the external Web service interactions and their interfaces. A related approach is presented by Wolf [27] and his team at the University of Rostock. They provide formal models based on Petri nets to describe service interfaces and tools[15] that support the discovery and synthesis of matching service partners.

8 Conclusion and Future Works

In this paper, we present a gateway architecture that makes it possible to interconnect FIPA-compliant multi-agent systems and RESTful Web services. More specifically, it creates a bridge between Petri net-based (Mulan) agents and arbitrary Web service providers or clients. Its suitability for daily use has been proven by coupling image conversion and comparison services (provided by Mulan agents) with an integrated project management environment (running as a classic web server, using these services). Besides its useful functionality, the gateway as an artifact exemplifies the benefits of engineering Petri net-based software. The gateway architecture, its message routing core and its multi-agent interface are modeled and implemented in Java reference nets. We present the design of the core gateway functionality as coarse Petri net models, the integration of concrete functionality into these Petri nets – thus turning them into application code –, and the validation of certain application properties by using well-known Petri net analysis techniques.

On the practical side, the gateway broadens the range of applications for FIPA-compliant agents (and especially Mulan agents). Their functionality be-

[15] Service-Technology: `http://service-technology.org/tools/`

comes available for any Web service client, and they can refer to functionality provided by any other web service. The interaction with web services is restricted to the request-response pattern or just unidirectional information distribution, and thus not as feature-rich as the speech act-based communication in the multi-agent world. Nevertheless, these simple interaction patterns form the basis of any complex interaction and can thus be considered as sufficient for everyday use.

On the Petri net-based software engineering side, the tools and methods of the P*AOSE approach are evolving while we use them for the design and implementation of applications like the WebGateway. A major focus is currently put on validation and testing of the application's Petri net-based code artifacts.

The further use of our approach in future student projects and the continuous advancement of our agent-based collaboration platform will help to improve the gateway functionality and software engineering techniques step by step.

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. Saeed Aghaee and Cesare Pautasso. Mashup development with HTML5. In *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*, Mashups '09/'10, pages 10:1–10:8, New York, NY, USA, 2010. ACM.
3. Tobias Betz, Lawrence Cabac, and Matthias Güttler. Improving the development tool chain in the context of Petri net-based software development. In Michael Duvigneau, Daniel Moldt, and Kunihiko Hiraishi, editors, *Petri Nets and Software Engineering. International Workshop PNSE'11, Newcastle upon Tyne, UK, June 2011. Proceedings*, volume 723 of *CEUR Workshop Proceedings*, pages 167–178. CEUR-WS.org, June 2011.
4. Tobias Betz, Lawrence Cabac, and Matthias Wester-Ebbinghaus. Gateway architecture for Web-based agent services. In Franziska Klügl and Sascha Ossowski, editors, *Multiagent System Technologies*, volume 6973 of *Lecture Notes in Computer Science*, pages 165–172. Springer Berlin / Heidelberg, 2011.
5. Lawrence Cabac. *Modeling Petri Net-Based Multi-Agent Applications*, volume 5 of *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2010.
6. Lawrence Cabac, Till Dörges, Michael Duvigneau, Daniel Moldt, Christine Reese, and Matthias Wester-Ebbinghaus. Agent models for concurrent software systems. In Ralph Bergmann and Gabriela Lindemann, editors, *Proceedings of the Sixth German Conference on Multiagent System Technologies, MATES'08*, volume 5244 of *Lecture Notes in Artificial Intelligence*, pages 37–48, Berlin Heidelberg New York, 2008. Springer-Verlag.
7. Lawrence Cabac, Michael Duvigneau, Daniel Moldt, and Heiko Rölke. Modeling dynamic architectures using nets-within-nets. In Gianfranco Ciardo and Philippe Darondeau, editors, *Applications and Theory of Petri Nets 2005. 26th International Conference, ICATPN 2005, Miami, USA, June 2005. Proceedings*, volume 3536 of *Lecture Notes in Computer Science*, pages 148–167, 2005.
8. Lawrence Cabac, Michael Duvigneau, Daniel Moldt, and Matthias Wester-Ebbinghaus. Towards unit testing for Java reference nets. In Robin Bergenthum and Jörg Desel, editors, *Algorithmen und Werkzeuge für Petrinetze. 18. Workshop AWPN 2011, Hagen, September 2011. Tagungsband*, pages 1–6, 2011.

9. Lawrence Cabac, Daniel Moldt, and Heiko Rölke. A proposal for structuring Petri net-based agent interaction protocols. In Wil van der Aalst and Eike Best, editors, *24th International Conference on Application and Theory of Petri Nets, Eindhoven, Netherlands, June 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 102–120. Springer-Verlag, June 2003.
10. Lawrence Cabac and Jan Schlüter. ImageNetDiff: A visual aid to support the discovery of differences in Petri nets. In *15. Workshop Algorithmen und Werkzeuge für Petrinetze, AWPN'08*, volume 380 of *CEUR Workshop Proceedings*, pages 93–98. Universität Rostock, September 2008.
11. Michael Duvigneau, Daniel Moldt, and Heiko Rölke. Concurrent architecture for a multi-agent platform. In Fausto Giunchiglia, James Odell, and Gerhard Weiß, editors, *Agent-Oriented Software Engineering. 3rd International Workshop, AOSE 2002, Bologna. Proceedings*, pages 147–159. ACM Press, July 2002.
12. Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. *ACM Trans. Internet Technol.*, 2:115–150, May 2002.
13. OASIS (Organization for the Advancement of Structured Information Standards). BPEL: Web services business process execution language. Available at: http://bpel.xml.org/, 2012. Release 2.0: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.
14. D. Greenwood and M. Calisti. Engineering Web service - agent integration. In *Systems, Man and Cybernetics, 2004 IEEE International Conference on*, volume 2, pages 1918 – 1925 vol.2, october 2004.
15. Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming BPEL to Petri nets. *Lecture Notes in Computer Science*, 3649:220–235, 2005.
16. Ekkart Kindler, Vladimir Rubin, and Robert Wagner. Component tools: Integrating Petri nets with other formal methods. *Lecture Notes in Computer Science : Petri Nets and Other Models of Concurrency - ICATPN 2006, Volume 4024, 2006*, pages 37–56, 2006.
17. Michael Köhler, Daniel Moldt, and Heiko Rölke. Modelling the structure and behaviour of Petri net agents. In J.M. Colom and M. Koutny, editors, *Proceedings of the 22nd Conference on Application and Theory of Petri Nets 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 224–241. Springer-Verlag, 2001.
18. Olaf Kummer, Frank Wienberg, Michael Duvigneau, Jörn Schumacher, Michael Köhler, Daniel Moldt, Heiko Rölke, and Rüdiger Valk. An extensible editor and simulation engine for Petri nets: Renew. In Jordi Cortadella and Wolfgang Reisig, editors, *Applications and Theory of Petri Nets 2004. 25th International Conference, ICATPN 2004, Bologna, Italy, June 2004. Proceedings*, volume 3099 of *Lecture Notes in Computer Science*, pages 484–493, Berlin Heidelberg New York, June 2004. Springer.
19. Esteban León Soto. Agent communication using Web services, a new fipa message transport service for jade. In Paolo Petta, Jörg Müller, Matthias Klusch, and Michael Georgeff, editors, *Multiagent System Technologies*, volume 4687 of *Lecture Notes in Computer Science*, pages 73–84. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-74949-3_7.
20. T. Miyamoto and S. Kumagai. An agent net approach to autonomous distributed systems. In *Proc. of 1996 IEEE Systems, Man, and Cybernetics, 14-17 October 1996, Beijing, China*, pages 3204–3209, October 1996.
21. Daniel Moldt, Sven Offermann, and Jan Ortmann. A Petri net-based architecture for web services. In Lawrence Cavedon, Ryszard Kowalczyk, Zakaria Maamar, David Martin, and Ingo Müller, editors, *Workshop on Service-Oriented Computing*

and Agent-Based Engineering, SOCABE 2005, Utrecht, Netherland, July 26, 2005. Proceedings, pages 33–40, 2005.
22. Julia Padberg and Hartmut Ehrig. Petri net modules in the transformation-based component framework. *Journal of Logic and Algebraic Programming, Vol 97 (1-2)*, pages 198–225, 2006.
23. Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big" web services: making the right architectural decision. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 805–814, New York, NY, USA, 2008. ACM.
24. M. Omair Shafiq, Ying Ding, and Dieter Fensel. Bridging multi agent systems and Web services: towards interoperability between software agents and semantic Web services. In *Enterprise Distributed Object Computing Conference, 2006. EDOC '06. 10th IEEE International*, pages 85 –96, october 2006.
25. Rüdiger Valk. Object Petri Nets – Using the Nets-within-Nets Paradigm. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets: Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 819–848. Springer-Verlag, Berlin Heidelberg New York, 2004.
26. World Wide Web Consortium (W3C). The websocket api editor's draft 6. Website, June 2011. http://dev.w3.org/html5/websockets.
27. Karsten Wolf. Does my service have partners? *LNCS ToPNoC*, 5460(II):152–171, March 2009. Special Issue on Concurrency in Process-Aware Information Systems.

Petri nets as a means to validate an architecture for time aware systems

Francesco Fiamberti, Daniela Micucci, and Francesco Tisato

D.I.S.Co., University of Milano-Bicocca, Viale Sarca 336, 20126, Milano, Italy
{francesco.fiamberti, daniela.micucci, francesco.tisato}@disco.unimib.it

Abstract. Time aware systems claim for an explicit representation of time-related concepts, so that they can be observed and possibly controlled at run-time. The paper identifies a set of architectural abstractions capturing such concepts related to time and classifies the base activities performed by a time aware system. Our proposal has been formalized using two different modeling techniques: UML and Petri nets. The former has been chosen to model the static structure of both the abstractions and the entities performing time aware activities. The latter have been exploited to model the dynamics of a time aware system.

Keywords: real-time, architectural abstractions, UML, Petri nets

1 Introduction

Time aware systems [1] deal with *time-related* issues when accomplishing domain-related tasks. For example, a time aware system includes activities whose activation is *time driven*, activities that need to reason on *timestamped facts*, and activities that need to know *what time it is*. Therefore, time ought to emerge as a first-class concept because of its relevance in the application domain.

As stated in [2], only recently model-based development has begun focusing on timing aspects of a system in addition to its functional and structural ones. However, the proposed approaches tend to specify the requirements with respect to timing focusing on a specific solution. Moreover, the satisfaction of timing requirements is verified only during the test phase of the development process. As stated in [3], this is due to the fact that model-driven approaches applied to embedded systems in early design phases do not always rely on a systematic and rigorous methodology that includes specifying and verifying timing requirements.

To overcome the above drawbacks, several modeling techniques have been proposed. MARTE [4] is an UML profile designed to face real-time aspects of a system from a model-based perspective. UML [5] is a standardized general-purpose modeling language used to specify the artifacts of a software system. A UML profile customizes UML for a specific purpose or domain by using extension mechanisms able to modify the semantics of the meta-model elements [5]. [6] exploits MARTE (the logical time concept) and the CCSL language [7] to specify the causal and temporal characteristics of the software as well as the hardware

parts of the system. However, modeling capabilities need to be supported by tools that directly implement the system.

Languages like Giotto [8] and SIGNAL [9] extend existing paradigms to include time-related issues. Close to Giotto, PTIDES [3] is a programming model for distributed embedded systems based on a global, consistent notion of time. However, such approaches allow time-related issues to be managed at compile time only, preventing the temporal behavior of the system from being adaptive.

The key idea behind our proposal is that time-related concepts should be first-class concepts, which directly turn into basic architectural abstractions [10] supported by a running machine. In this way, it is possible to explicitly treat time-related aspects from the analysis of the requirements to the test phase of the life cycle of a system. Even if UML [5] is a well-known modeling language in the software engineering area, the features of Petri nets [11] make them a suitable tool to model the dynamics of a time aware system. Indeed, when a set of time driven entities must be performed, they must be enabled in a deterministic way. However, once they have been enabled, their actual execution can be nondeterministic, that is, their execution order should have no significance and should not affect the system behavior. Therefore, we used UML to describe the architectural abstractions and the time-related activities (static structure), and Petri nets to describe the dynamics of a time aware system.

The paper is organized as follows. Section 2 introduces time-related abstractions by means of UML class diagrams. Section 3 identifies the three base entities performing time aware activities by means of UML class and state diagrams. Section 4 discusses the dynamics of a time aware system exploiting Petri nets. Finally, Section 5 presents concluding remarks.

2 Time-related abstractions

To give a flavor of the proposed model, the following simplified example will be used. Consider a road gate equipped with a camera. The gate provides access to an area with traffic restrictions. In particular, car transits are only allowed at night and for a restricted set of vehicles. Every second, the camera must acquire a frame, which must be stored with the acquisition timestamp. Finally, the acquired frames are elaborated offline to detect infractions.

The described scenario is an example of a time aware system. A *time aware* system reifies the following time aware activities:

- a *time driven* activity is triggered by events that are assumed to model the flow of time. In the proposed example, the acquisition activity must be time driven in order to acquire frames at predefined time instants.
- a *time observer* activity observes "what time it is". Thus, the acquisition activity in the example must also be time observer, since it needs the correct timestamp for every acquired frame.
- a *time conscious* activity reasons on facts placed in a temporal context, no matter when the computation is realized. In the example, the infraction

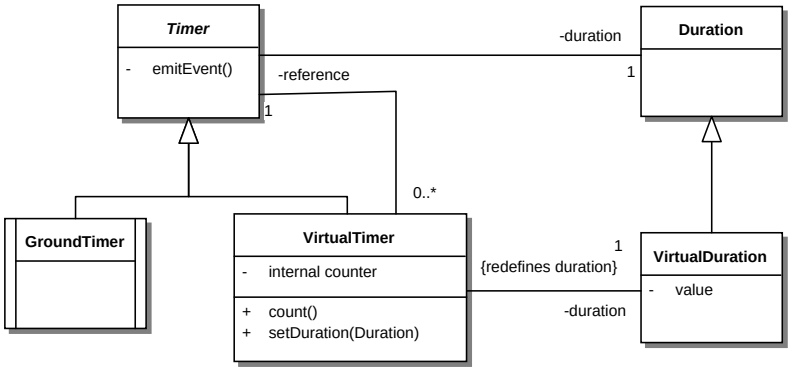


Fig. 1: Concepts related to timers

detection activity is time conscious, as it performs an offline elaboration of timestamped frames.

Drivenness, *observability*, and *consciousness* can be enabled by means of three well distinguished architectural abstractions: *Timer*, *Clock*, and *Timeline*.

2.1 Timer

A *Timer* is a cyclic source of events, all of the same type: two successive events define a *duration*. A timer generates events by means of its *emitEvent* operation.

A *Virtual Timer* is a timer whose event generation is constrained by the behavior of its *reference* timer: it counts (by means of the *count* operation) the number of events it receives from its reference timer and generates an event when this number equals a predefined *value*. The duration is specialized to *virtual duration*. Timers can thus be arranged in hierarchies, in which every descendant timer has exactly one reference timer. The root of every hierarchy is a *Ground Timer*, which is a timer whose duration is not constrained by the duration of another timer. Therefore, the duration of a ground timer can be interpreted as intervals of the real external time, so that the events generated by a ground timer can be interpreted as marking the advance of time. Finally, the *setDuration* operation allows the duration of a virtual timer to be modified, thus varying the speed at which events are generated. Figure 1 sketches the described concepts.

2.2 Clock

A *Clock* counts (by means of its *increment* operation) the events it receives from the associated timer. The event count is interpreted as the clock's *current time* (see Figure 2). Thus, time is not a primitive concept but it is built from events.

2.3 Timeline

A *Timeline* is a data structure (thus intrinsically discrete) constituting a static representation of time as a numbered sequence of *grains*. A grain is an elementary unit of time identified by its *index* and whose interior cannot be inspected.

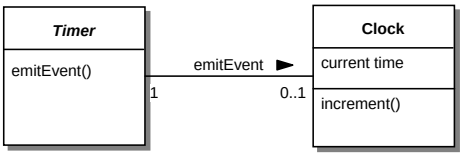


Fig. 2: Concepts related to clocks

A *Time Interval*, defined on a timeline, is a subset of contiguous grains belonging to that timeline. A *virtual timeline* is a timeline whose grains (*virtual grains*) have a *duration* that can be expressed as a time interval in the associated *reference* timeline. Timelines can thus be arranged in hierarchies. The root of every hierarchy is a *Ground Timeline*, which is a timeline whose grain durations are not constrained by the grains of another timeline. In each hierarchy, the ground timeline is therefore the only one whose grains can be interpreted as an elementary time interval in an arbitrary ground reference time (e.g., the "real" time from the application viewpoint).

A *Fact* is an assertion regarding the system domain. A *Timed Fact* is a fact associated to a time interval representing the fact's interval of validity. Therefore, timelines are histories of timed facts. Figure 3 sketches all the described concepts.

By connecting a clock with a timeline, it is possible to interpret as *present time* on the associated timeline the grain whose index equals the clock's current time (see Figure 4). Every time the clock receives an event from the connected timer, it advances the present time on the corresponding timeline by one grain. The clock also defines the concepts of *past* and *future* in the associated timeline: the grains with *index* less than *current time* belong to the past and the grains with *index* greater than *current time* belong to the future.

3 Time aware entities

The identified abstractions enable the design of the following *time aware entities* (see Figure 5), which reify the activities of a time aware system:

- *Time driven entity*: an entity whose activation is triggered by a virtual timer
- *Time observer entity*: an entity that reads current time from clocks
- *Time conscious entity*: an entity that reads/writes timed facts on a timeline without any reference to when such a management is actually realized

More articulated behaviors can be obtained by combining the three basic entities. For example, a *time driven time conscious entity* is an entity that is triggered by a virtual timer (time driven) and reads/writes timed facts (time conscious).

3.1 Time driven entities

A time driven entity is associated to its *activating* timer, and it may be in two states: running and idle. A time driven entity enters the *running* state when its

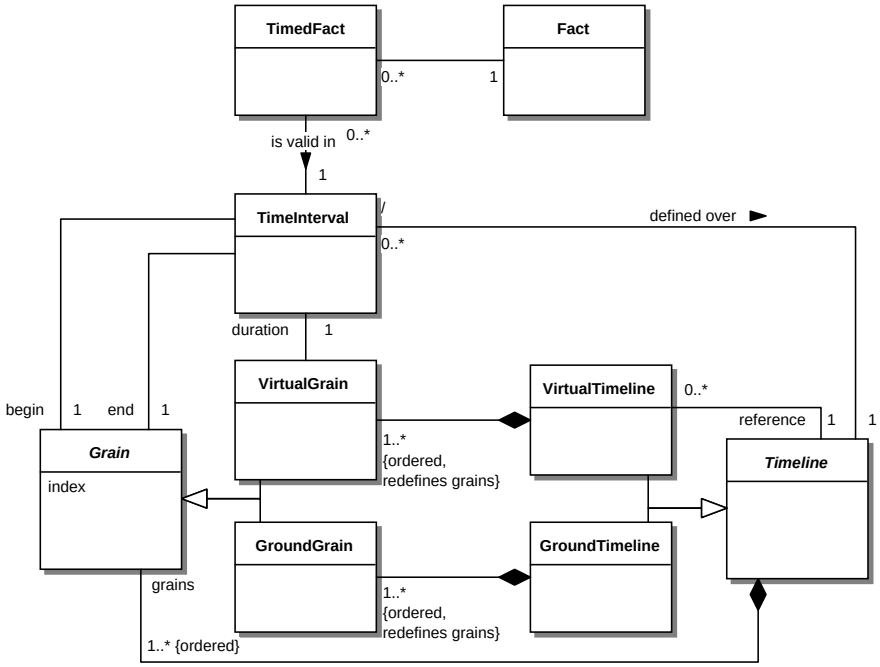


Fig. 3: Concepts related to timelines

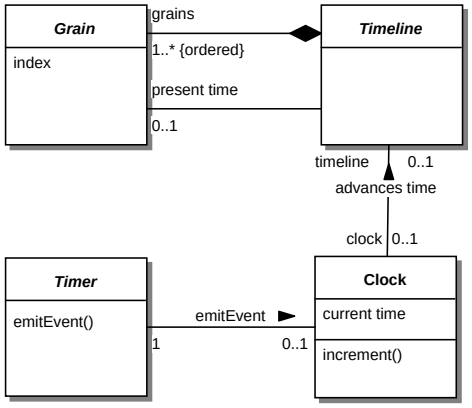


Fig. 4: Connection of a clock to a timeline

activating timer emits an event. In this state, it performs its domain-dependant operation. At the end of the execution, the entity goes back to the *idle* state.

This simple model assumes that the deadline for an execution coincides with the beginning of the next execution. To adapt the model to the general case where deadlines temporally precede the beginning of the next execution, it is possible to associate a second timer to each time driven entity, as sketched in Figure 6. When the deadline timer emits an event, the associated time driven entity must have already completed the *perform* operation. It follows that a time driven entity must include an additional state, denoted *terminated*.

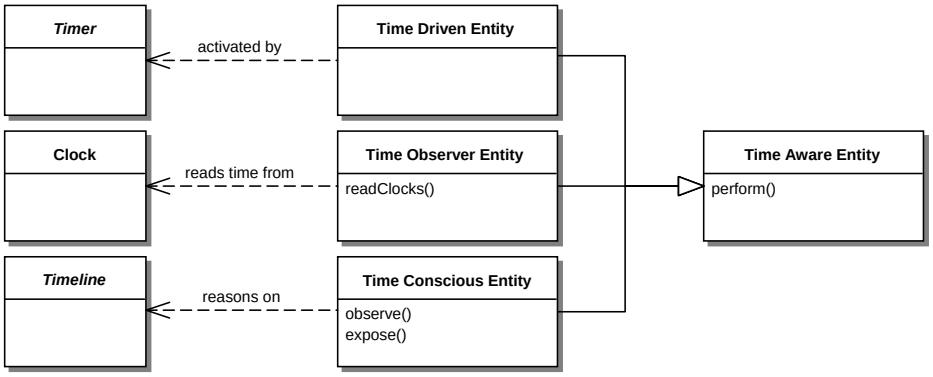


Fig. 5: Entity classification according to the relation with the basic concepts

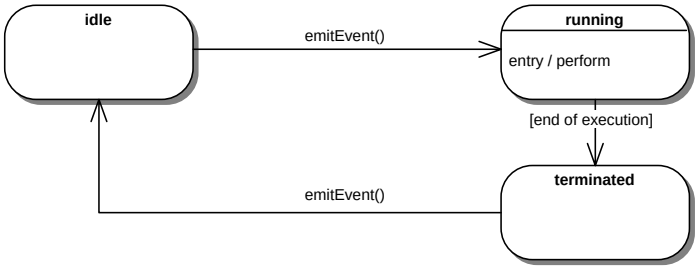


Fig. 6: State diagram for a time driven entity

3.2 Time driven time conscious entities

Some care must be used to guarantee consistency when designing entities that are both time driven and time conscious. In fact, it is desirable that the behavior of all the entities that are triggered simultaneously does not depend on the order in which the executions are actually managed, which may be affected by low-level details such as the number of available cores or the particular scheduling algorithm that is being used. Therefore, it is necessary to guarantee that all the time driven time conscious entities that are triggered simultaneously share the same view of the timelines, to avoid the situation of an entity that reads timed facts written by another entity triggered simultaneously just because the latter was granted higher execution priority by the low-level scheduler.

A possible solution is that all entities read timed facts immediately when they are activated by the activating timer and write timed facts only when they receive an event by the deadline timer, even if the actual execution ends before the deadline. The state diagram in Figure 7 enriches Figure 6 by introducing effects in the transitions triggered by timers: the effect of an event from the activating timer is the reading of facts by means of the *observe* operation, whereas the effect of an event from the deadline timer is the writing of facts by means of the *expose* operation. In an actual implementation, the concrete component in charge of the execution of entities must guarantee that when the execution of a set of entities is triggered, all the entities read timed facts before any one of

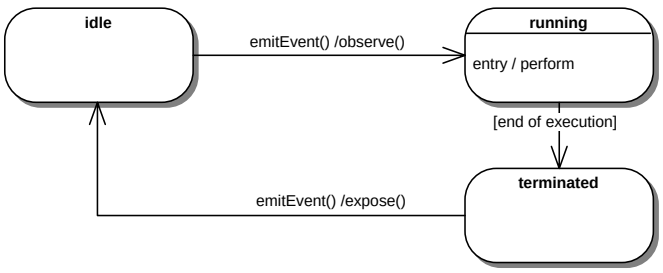


Fig. 7: State diagram for a time driven time conscious entity

them is allowed to start the actual execution, and that every entity writes timed facts only at the deadline for its execution.

4 Time aware systems

This section presents the behavior of time aware systems exploiting Petri nets. First, the dynamics of a timer hierarchy will be discussed. Afterwards, the model of the activation of time driven and time conscious entities will be presented.

4.1 Timer hierarchy

Before describing a complete timer hierarchy, we will detail the internal behavior of a virtual timer. Figure 8 shows the subnet modeling the timer `T2`, of duration 3, without descendant timers. In the initial marking, a token is present in place `p1`. The first time a token is put into place `Event to T2`, transition `t1` is enabled and fires, putting a token into both `p2` and `T2 updated`. The structure made of `p*` and `t*` works as an internal counter, and the token to `T2 updated` is needed to allow the external system to be notified that the timer completed its update operations (simple increment of the internal counter or event generation). When the internal counter equals the timer's duration, transition `Emit event` is fired. At the end, a token is put into place `T2 updated`. If a clock is connected to the timer, after every generated event, the corresponding transition `increment clock time` is fired before the end of the timer's update. Note also that the gray arcs in Figure 8 are required for the correct behavior of the Petri net, but do not have any particular time-related semantics. For example, the arcs from the place `Event to T2` to transitions `t1`, `t2` and `t3` allow such transitions to fire only one at a time, when a token is present in place `Event to T2`.

Figure 9 shows the subnet of `T1`, a timer with descendants (`T3` and `T4`). Unlike in the previous case, transition `Emit event` puts tokens into all the places `Event to T*` of the descendant timers, whose places `T* updated` are joined in transition `Join descendant timers`, after which the system's behavior is the same as in the case without descendant timers. No assumptions are made on the order in which descendant timers are updated.

Figure 10 shows a Petri net modeling a four-level hierarchical timer structure. Place `Ground timer event` receives a token that is interpreted as the flow

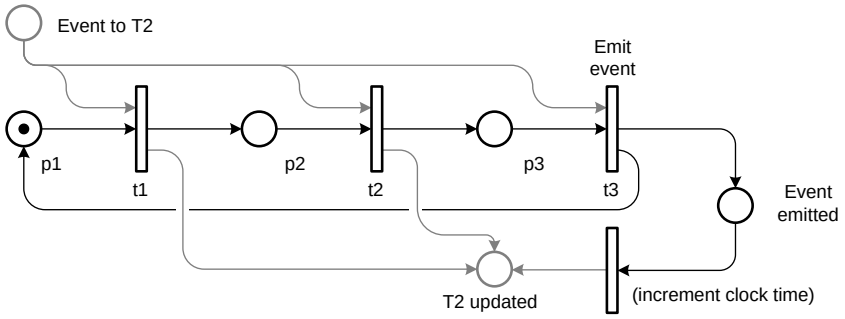


Fig. 8: Subnet for a timer with no descendants

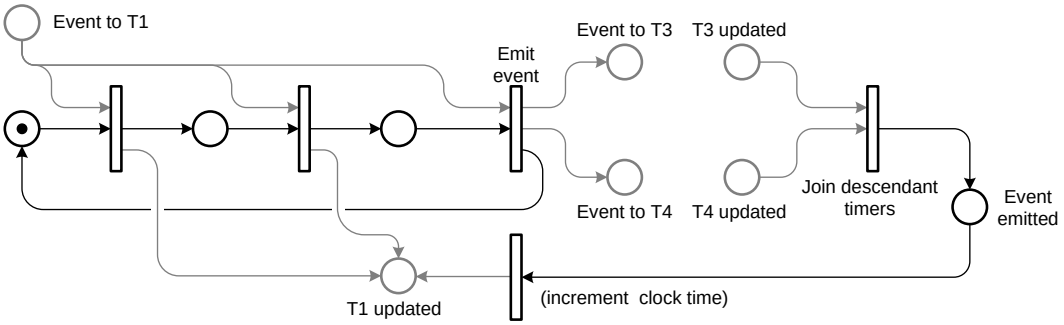


Fig. 9: Subnet for a timer with descendants

of real time. If the previous update of the system has been completed (a token is present in place `All timers updated`), transition `Start execution` is enabled. When the transition is fired, a token is sent to place `Event to T*` of every virtual timer directly connected to the ground timer, enabling internal update operations. When a timer emits an event, its update can terminate only when all its possible descendant timers' updates have been triggered and completed. Once the timer has been updated, a token is put into the corresponding place `T* updated`. Places `T* updated` are joined in transition `Join direct descendants of ground timer`, whose firing terminates the atomic update of all timers by putting a token into place `All timers updated`. This token enables transition `Start execution` when the next token is produced in place `Ground timer event`. Thanks to the recursive structure of virtual timers, only direct descendants of the ground timer need to be joined in transition `Join direct descendants of ground timer` to ensure atomicity of all timers' updates. In Figure 10, for clarity the details regarding the internal structure of timers have been hidden and represented by means of simple transitions shown in gray.

4.2 Time driven entities

Time driven entities can be executed by associating them to timers. Every time a timer emits an event, it sends a signal to all its associated entities, which behave consequently according to their internal state. To make the concepts clear, we

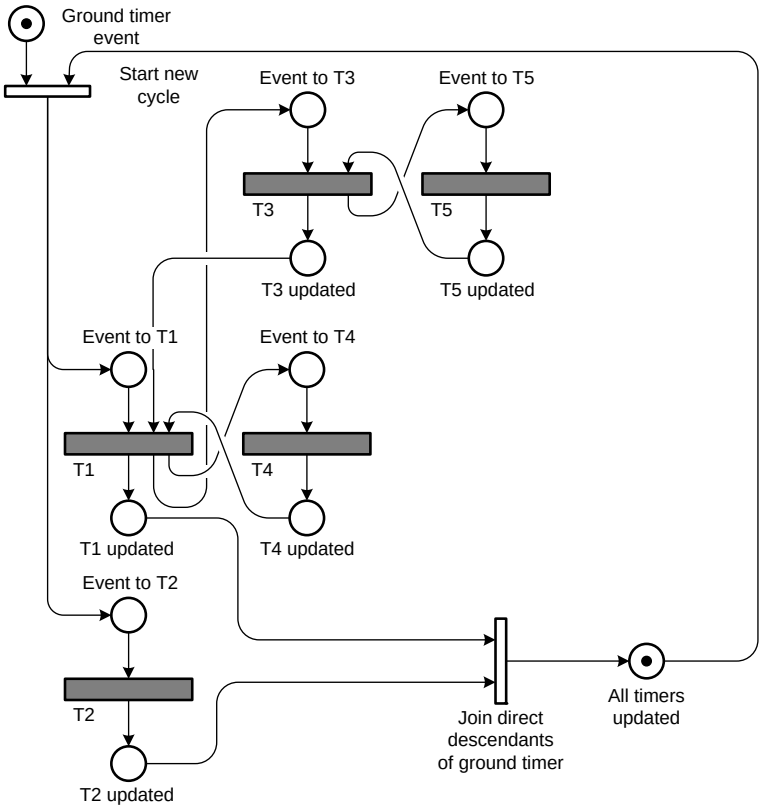


Fig. 10: Example of hierarchical timer structure

assume that the relative deadlines of the time driven entities coincide with the beginning of the next execution. Figure 11 sketches an example of such a system, where timers `T3`, `T4`, and `T5` have been hidden for the sake of readability. The assumption is made that the execution of a time driven entity is an instantaneous action. Thus, if the actual time taken by an entity is not negligible with respect to the smallest time scale in the system, the execution considered here is made only of the (instantaneous) set of operations needed to decouple the actual actions from the main system flow (e.g., the operations needed to start a new thread where threads are available).

Figures 12 and 13 show the structures of timers `T2` and `T1` respectively in presence of time driven entities: when an event is generated, a token is put into place `T* time driven entities to be enabled`, to enable the execution of the entities associated to `T*`. Note that habilitation does not mean immediate execution: the actual execution of all the time driven entities can be started only once all the timers have been updated, as will be explained later. To ensure that the entities have been enabled, a token is required in place `T* time driven entities enabled` for the transition in input to place `T* updated` to fire.

Figure 14 shows how atomicity of all timers'update can be granted. A copy of place `All timers updated` is available for every subnet modeling a group of time driven entities associated to the same timer. Every entity group has a `T* time`

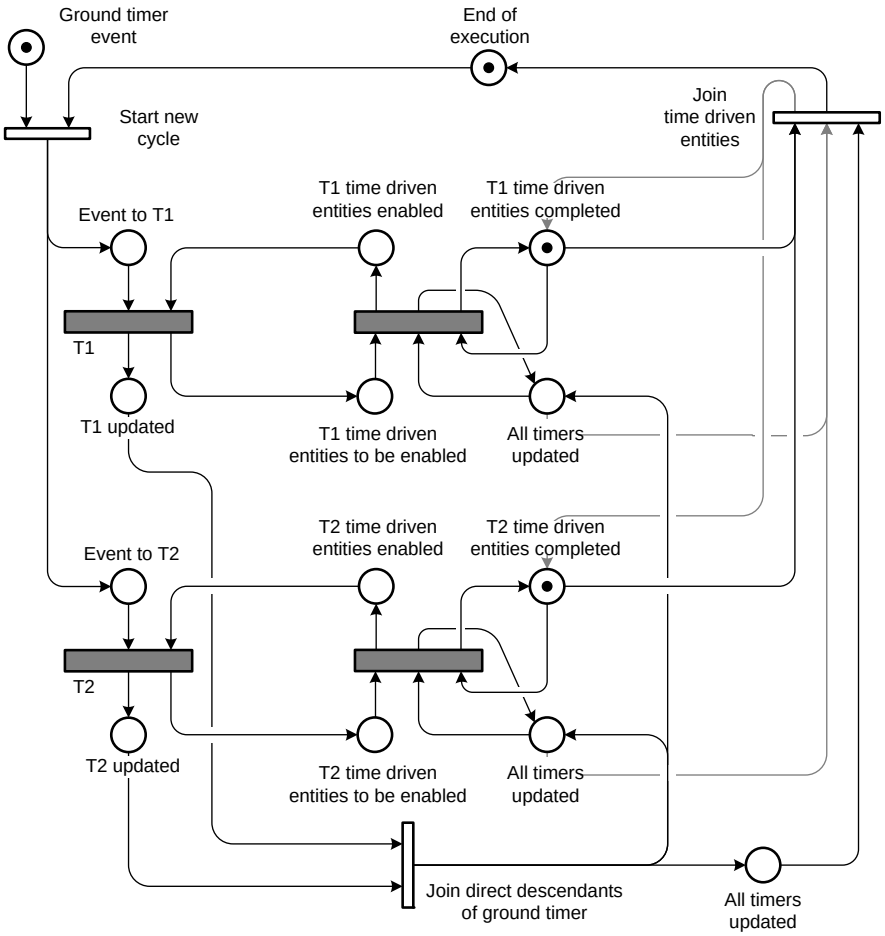


Fig. 11: Time aware system with time driven entities

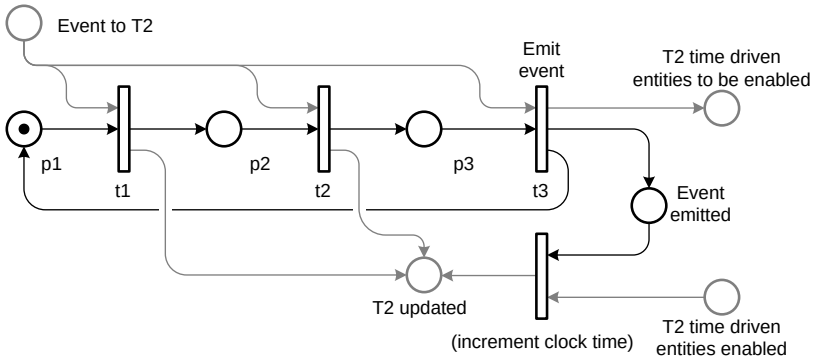


Fig. 12: Subnet for a timer with associated time driven entities

`driven entities completed` place, where the presence of a token indicates that no additional actions are required for the group. This is needed to deal with the (typical) situation where only a subset of all the timers emit an event. All places `T* time driven entities completed` are initialized with a token, and a token

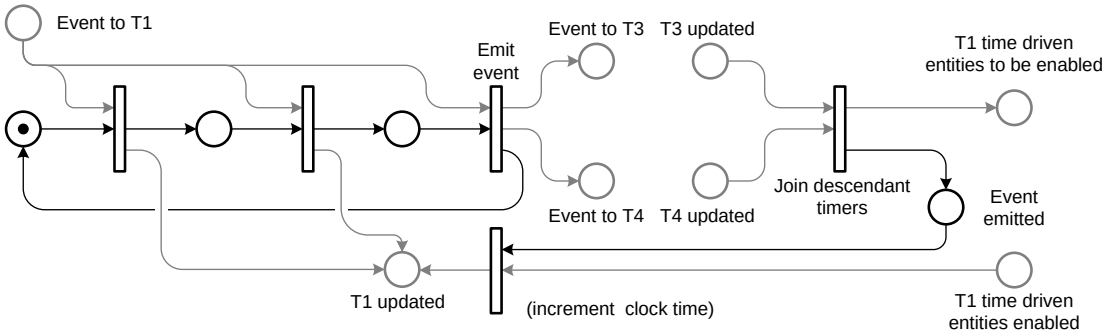


Fig. 13: Subnet for a timer with descendants and associated time driven entities

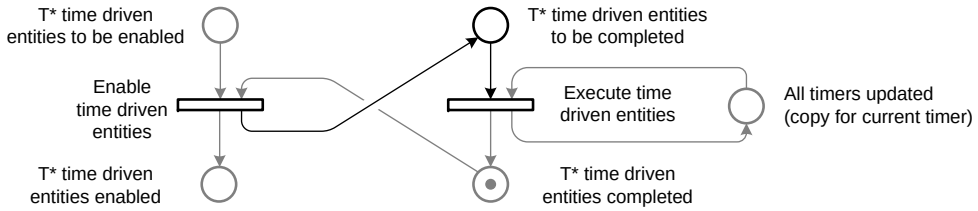


Fig. 14: Model of a group of time driven entities

is always present at the end of every execution. Only in case of an event from the associated timer, place `T* time driven entities completed` is cleared as a consequence of the habilitation of time driven entities. In fact, when a token is put into place `T* time driven entities to be enabled`, transition `Enable time driven entities` fires, removing the token from place `T* time driven entities completed` and putting it into place `T* time driven entities to be completed`. Transition `Execute time driven entities` is not enabled until a token is present in the current timer's copy of place `All timers updated`, that is, until all the timers have been updated. All places `T* time driven entities completed` are joined in transition `Join time driven entities` (see Figure 11). At the end of each global timer update, the time driven entity groups that do not require execution (because their timer did not emit an event) already have a token in place `T* time driven entities completed`.

Transition `Join time driven entities` is not enabled only if some of the time driven entities must still be executed. If this is the case, all the corresponding transitions `Execute time driven entities` are now enabled, removing the token from place `T* time driven entities to be completed` and putting it into `T* time driven entities completed`. No assumptions are made on the possible order in which time driven entities are executed. Once all the executions have been completed, the transition `Join time driven entities` is enabled. Note that in order to prevent an early firing of this transition in the case where no entity groups need to be executed, a copy of place `All timers updated` is present as an input to `Join time driven entities`, so that the global update

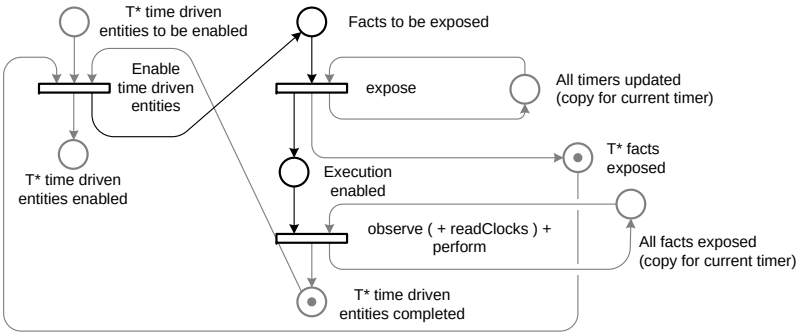


Fig. 15: Model of a group of time driven time conscious entities

of timers must be terminated first even though all the places `T* time driven entities completed` contain a token.

4.3 Time driven time conscious (observer) entities

As stated in subsection 3.2, some care must be used when designing entities that are time driven and time conscious, or time driven, time conscious and time observer. The two cases can be analyzed together, since the differences are limited to the presence of clocks and to the need to read the relevant clocks' current times before starting the executions. Figure 15 contains the model of a group of time driven time conscious entities associated to the same timer, while an example of time aware system is shown in Figure 16.

As introduced in Subsection 3.2, the sequence of operations of a time driven time conscious entity can be organized in three blocks: `expose` (that puts on the right timelines the timed facts computed during the previous execution), `observe` (that reads timed facts from the timelines of interest), and `perform` (the actual execution of the entity's actions). Since no constraints are put on the order in which entities are executed, consistency and predictability of the system's behavior require that timed facts are added on a timeline by an entity only at the end of the time grain of the timer to which the entity is associated, and before any other entity in the system reads facts from the same timelines. So the exposition of all the facts generated by all entities must be realized as an atomic action after all timers have been updated and before the execution of the `perform` of any time driven time conscious entity.

With reference to Figure 15, for every group of entities, place `T* facts exposed` is initialized with a token. When the associated timer emits an event, the token put into place `Time driven entities to be enabled` enables transition `Enable time driven entities`, which removes the tokens from places `Time driven entities completed` and `T* facts exposed`, putting a token into `Facts to be exposed`. Once all the timers have been updated, so that a token is put into the copy of place `All timers updated` for every group of entities, transition `expose` is fired, putting a token into the corresponding places `T* facts exposed` and `Execution enabled` (whose presence prevents unrequested

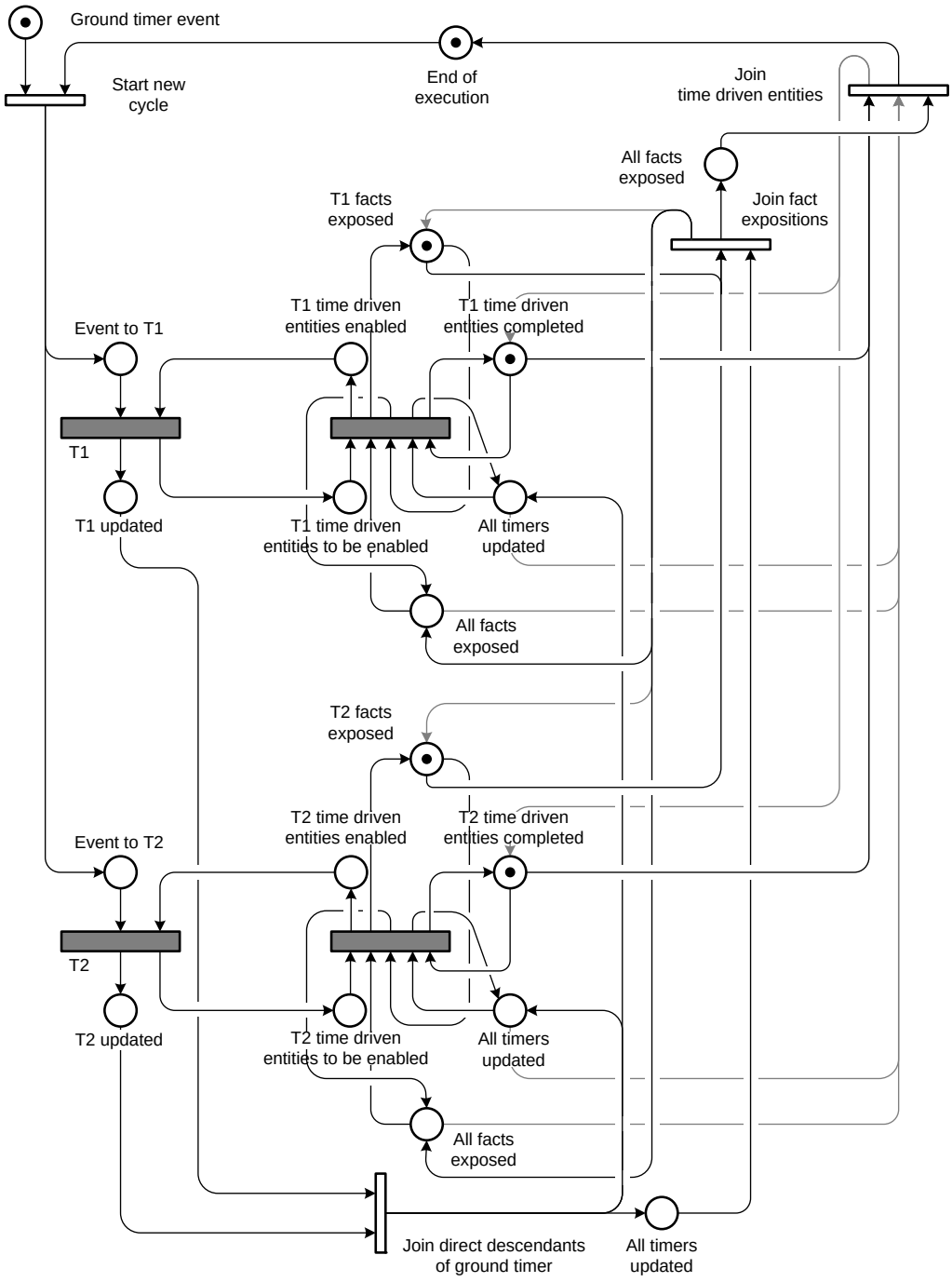


Fig. 16: Time aware system

firing of transition `observe (+ readClocks) + perform` that would be otherwise triggered by the simple presence of a token in place `All timers updated`). All places `T* facts exposed` are joined, together with a copy of `All timers updated`, in transition `Join fact expositions`. Only after all the expositions have been completed, this transition can fire, putting a token into the copy of

Property	Y/N	Property	Y/N
Pure (PUR)	No	Covered by P-invariants (CPI)	Yes
Ordinary (ORD)	Yes	Strongly Covered by T-invariants (SCTI)	Yes
Homogeneous (HOM)	Yes	Structurally Bounded (SB)	Yes
Non-Blocking Multiplicity (NBM)	Yes	Bounded (kB)	Yes
Conservative (CSV)	No	Safe (1-B)	Yes
Structurally Conflict-Free (SCF)	No	Dynamically Conflict-Free (DCF)	Yes
FT0, TF0, FP0, PF0	Yes	No Dead States (DSt(0))	Yes
Connected (CON)	Yes	No Dead Transitions (DTr)	Yes
Strongly Connected (SC)	Yes	Live (LIV)	Yes
Deadlock-Trap Property (DTP)	No	Reversible (Rev)	Yes
Covered by T-invariants (CTI)	Yes		

Table 1: Properties of the proposed Petri nets

place `All facts exposed` for all groups of entities. This enables all transitions `observe (+ readClocks) + perform` (for time observer entities, `readClocks` is executed on all the clocks of interest before `perform`), whose firing puts a token into the corresponding place `T* time driven entities completed`. At this stage, as in the case of pure time driven entities, places `Time driven entities completed` are joined in transition `Join time driven entities`, which fires when all the entity executions have been completed. The gray arcs in Figure 16 involve groups of entities: one connects place `All facts exposed` of each group and transition `Join time driven entities`, to ensure that tokens do not pile up in these places when the corresponding timer does not emit an event. The other is between transition `Join fact expositions` and place `T* facts exposed` of every group, needed to recharge the token for the next execution.

5 Final remarks

The time-related abstractions and the dynamics of a time aware system could have been fully described by means of UML diagrams (i.e., class and state to model the basic abstractions and sequence to model the dynamics). Initially, this was the direction we followed, but we soon realized that the resulting sequence diagrams would be complex and difficult to read. So we decided to use Petri nets, because of their suitability to model the dynamics of a system. The obtained result consists in a set of Petri nets that are simpler and more readable with respect to the corresponding UML sequence diagrams, notwithstanding the need for additional places and transitions that do not have an application semantic but are required for the Petri nets to behave correctly.

Table 1 summarizes the properties of the proposed Petri nets, as defined in [12,13]. Some of the properties cannot be satisfied because of the intrinsic nature of time aware systems (e.g., the proposed nets are not pure because of the presence of loops, which are required to obtain a cyclic behavior).

The proposed models supported the implementation of a Java framework named Time Aware Machine (TAM) [14] that has been used for the experimental testing of the Space Integration Services platform [15]. Currently, the framework is being used in ALARM [16], an architecture based on a time driven mechanism that verifies hypotheses about domain entities against previsions.

References

1. Fiamberti, F., Micucci, D., Tisato, F.: An architecture for time-aware systems. In: 2011 IEEE 16th Conference on Emerging Technologies & Factory Automation (ETFA), IEEE (2011)
2. Buckl, C., Gaponova, I., Geisinger, M., Knoll, A., Lee, E.A.: Model-based specification of timing requirements. In: Proceedings of the tenth ACM international conference on Embedded software. EMSOFT '10, ACM (2010) 239–248
3. Zhao, Y., Liu, J., Lee, E.A.: A programming model for Time-Synchronized distributed Real-Time systems. In: 13th IEEE Real Time and Embedded Technology and Applications Symposium, 2007. RTAS '07, IEEE (2007) 259–268
4. OMG: MARTE Modeling and Analysis of Real-Time and Embedded systems. http://www.omg.org/spec/MARTE/1.1/PDF/
5. OMG: Unified Modeling Language (UML), Superstructure. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/
6. Peraldi-Frati, M., DeAntoni, J.: Scheduling multi clock real time systems: From requirements to implementation. In: Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2011 14th IEEE International Symposium on. (2011) 50 –57
7. Mallet, F.: Clock constraint specification language: specifying clock constraints with UML/MARTE. Innovations in Systems and Software Engineering **4**(3) (2008) 309–314
8. Henzinger, T., Horowitz, B., Kirsch, C.: Giotto: a time-triggered language for embedded programming. Proceedings of the IEEE **91**(1) (2003) 84–99
9. Gamatié, A., Gautier, T., Guernic, P.L., Talpin, J.P.: Polychronous design of embedded real-time applications. ACM Trans. Softw. Eng. Methodol. **16**(2) (2007)
10. Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., Zelesnik, G.: Abstractions for software architecture and tools to support them. IEEE Transactions on Software Engineering **21**(4) (1995) 314–335
11. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4) (1989) 541–580
12. Starke, P.H.: Analyse von Petri-netz-modellen. Teubner BG GmbH (1990)
13. Starke, P.H.: INA Integrated Net Analyzer. http://www2.informatik.hu-berlin.de/lehrstuehle/automaten/ina/manual.html
14. Fiamberti, F., Micucci, D., Tisato, F.: An object-oriented application framework for the development of real-time systems. In: 50th International Conference on Objects, Models, Components, Patterns (TOOLS 2012), Springer (2012) 75–90
15. Bernini, D., Fiamberti, F., Micucci, D., Tisato, F.: Architectural Abstractions for Spaces-Based Communication in Smart Environments. Journal of Ambient Intelligence and Smart Environments **4**(3) (2012)
16. Fiamberti, F., Micucci, D., Mobilio, M., Tisato, F.: A Layered Architecture based on Previsional Mechanisms. In: ICSOFT 2013 - Proceedings of the 8th International Joint Conference on Software Technologies (accepted for publication). (2013)

Part III

PNSE'13: Short Presentations

A Framework for Efficiently Deciding Language Inclusion for Sound Unlabelled WF-Nets

D.M.M. Schunselaar*, H.M.W. Verbeek*, W.M.P. van der Aalst*, and H.A. Reijers*

Eindhoven University of Technology,
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
{d.m.m.schunselaar, h.m.w.verbeek, w.m.p.v.d.aalst, h.a.reijers}@tue.nl

Abstract. We present a framework to efficiently check language inclusion between two sound unlabelled WF-nets. That is, to efficiently check whether every successfully terminating transition sequence of one sound unlabelled WF-net also is a successfully terminating transition sequence of a second sound unlabelled WF-net. Existing approaches for checking language inclusion are typically based on the underlying transition systems of both nets, and hence are subject to the well-known state-explosion problem. As a result, these approaches cannot check language inclusion on sound unlabelled WF-nets in polynomial time. Our framework allows for efficient language inclusion checks even if parallelism is present, by comparing specific net abstractions that can be computed and compared in polynomial time. For sound unlabelled WF-nets that are free-choice and do not contain loops, we prove that our approach is complete.

1 Introduction

Language inclusion refers to the problem of deciding whether a first language is included in a second language, that is, whether every word of the first language is also a word in the second language. For this paper, we assume that a language is described by a sound unlabelled WF-net (a subclass of Petri nets), and that every word in the language corresponds to a successfully terminating trace in that WF-net. As a result, the problem can now be rephrased as deciding whether every successfully terminating trace of a first sound unlabelled WF-net is also a successfully terminating trace of a second sound unlabelled WF-net.

There already is a large body of work on equivalences and/or similarities between Petri nets, e.g., [1–3]. However, in some cases we do not need to decide on language equivalence as language inclusion is already sufficient to investigate relevant properties. For example, suppose we have a large repository of Petri nets that are all different. To reduce the number of Petri nets we need to maintain, we might want to remove any Petri net that has less behaviour than some other

* This research has been carried out as part of the Configurable Services for Local Governments (CoSeLoG) project (http://www.win.tue.nl/coselog/).

Petri net. This requires us to determine whether the language of a Petri net is included in the language of another Petri net.

Conformance checking is another interesting application area for language inclusion. In conformance checking, one seeks to validate an event log against a Petri net, i.e., does every trace in the event log correspond to a successfully terminating transition sequence of the Petri net [4]? If we can create a Petri net that precisely captures the traces from the event log, then this question corresponds to whether the language of this created Petri net is included in the language of the original, given, Petri net.

Another interesting application of language inclusion comes from the CoSeLoG project[1]. In this project, we are cooperating with municipalities of different sizes and with different characteristics. If, for instance, a first, small, municipality wants to cooperate with a second, large, municipality, then the second municipality might have more generic process models (that is, Petri nets) that also support the first municipality. Although, language equivalence is unlikely to hold, language inclusion might very well hold, which is already sufficient for a fruitful cooperation. Apart from comparing the Petri nets of these municipalities, we can also combine them into a more generic Petri net [5], and check whether a Petri net of a third municipality is already included in this generic Petri net. If so, then this third municipality can be supported by this generic Petri net as well, while it might not have been supported by the Petri net of the first or the second municipality only. In the context of [5], we already have standardised the Petri nets, i.e., same activities names, and same level of abstraction.

The main problem with Petri-net-based language inclusion is that it is defined on the state-space underlying the Petri net (encoded as a transition system) and is PSPACE-complete. Due to the state-space explosion problem, such a transition system may be extremely large. Therefore, in general, it is computationally expensive to decide on language inclusion using the underlying transition systems.

We present a framework for deciding whether and how Petri-net-based language inclusion can be decided in a more efficient way for sound unlabelled WF-nets. The framework is based on the general pattern: If a sound unlabelled WF-net adheres only to some characteristics (e.g., is acyclic), then we can map this net to an alternative representation using a mapping function λ. Using a comparator $\leq_\lambda$, we may then efficiently decide whether language inclusion holds or not. As an example, Fig. 1 depicts a mapping function that maps every net to its set of transitions. Using the subset operator as comparator, we can already conclude that language inclusion does not hold, i.e., that the transition set of the first net is not a subset of the transition set of the second net.

For the case, sound unlabelled WF-nets that are free-choice and contain no loops, we prove that our approach is complete. If the net is not free-choice or contains loops, we present mapping functions and corresponding comparators that can signal that language inclusion does *not* hold.

[1] http://www.win.tue.nl/coselog/

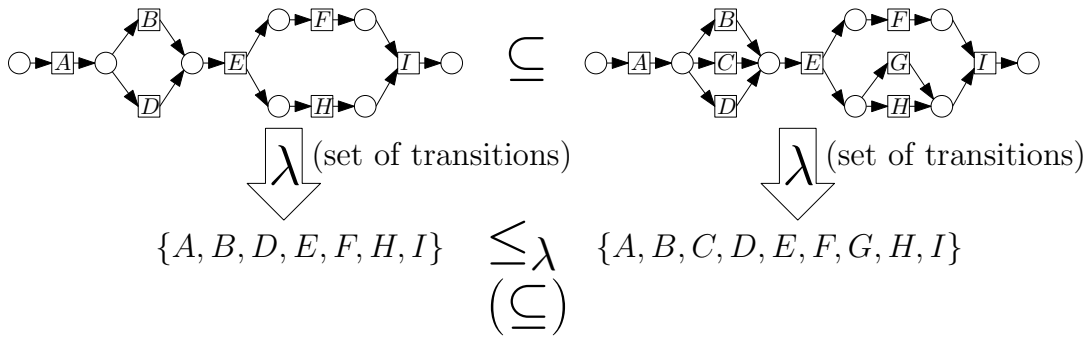


Fig. 1: General pattern of the framework: First, we map every net to an alternative representation (like its set of transitions); second, by efficiently comparing these alternative representations (like the standard subset comparator on the sets of transitions), language inclusion may be decided.

The structure of the paper is as follows. In Sect. 2, we present the approaches from literature and present the mappings found in literature. Preliminaries are provided in Sect. 3. Section 4 presents the general framework for our approach, which includes different mapping functions and comparators to tackle the language inclusion problem at hand. Finally, the conclusions and directions for future work are elaborated on in Sect. 5.

2 Related Work

There are solutions for checking whether the language of a first Petri net is included in the language of a second Petri net, see [6–8]. However, these solutions use the transition systems underlying both Petri nets. Unfortunately, these transition systems may become very large in the presence of parallelism. Therefore, as mentioned, we are designing a framework for deciding whether and how Petri-net-based language inclusion can be decided in a more efficient way. An important element of this framework is mapping a net to an alternative representation. In the remainder of this section, we present different candidates for such mappings as found in the literature. Note that this list is not exhaustive, but that our framework can easily incorporate other, new, mappings.

The first mapping we consider is based on the α-relation [9]. The α-relation denotes direct succession between activities and was originally designed for process discovery. In [10], a characterisation is given for the nets discoverable by the α-relation. Extensions have been made to the α-relation in [11] to be able to discover a larger set of nets.

The α-relation denotes direct succession, the relation used in behavioural profiles [3,12,13] denotes the eventual succession. Based on the behavioural profiles, work has been done for language equivalence, but language inclusion has not been addressed. A more generic approach based on behavioural profiles is presented in [1], where eventual succession is still considered, but the distance between occurrences of transition is bounded. Using behavioural profiles, [14]

considers an abstraction, but this abstraction is not language-equivalence preserving, i.e., the language of the abstraction is different from the language of the original net.

An approach similar to behavioural profiles is the approach using causal footprints [15]. Causal footprints denote causality between activities. In [15], causal footprints are used to deduce structural properties of sound WF-nets. Using causal footprints, [16] denotes the similarity between nets. This similarity can be used to deduce language equivalence (similarity of 1), but this approach is not tailored towards language inclusion.

Causal nets [4] allow us to model processes in such a way that causality between activities is made explicit. Language inclusion of causal nets has been defined in [17]. Furthermore, [4] presents an operational algorithm to transform a net to a causal net by transforming place and transition into activities.

Process nets [18] allow us to encode causal runs. Such a causal run is expressed as a marked graph. The approach was originally designed for the validation of nets. Therefore, no language equivalence or language inclusion is defined on process nets.

The last mapping we mention is the mapping from a net to its basis of semi-positive transition invariants [19]. The intuitive notion of such an invariant is that if we start at a given marking, and execute all the transitions from the invariant as many times as the invariant indicates, we return to the marking we started with. As such, every behavioural cycle in the Petri net is covered by such an invariant, but this not work the other way around: It may be possible that an invariant does not correspond to any behavioural cycle (due to the fact that we may not have sufficient tokens to execute exactly the transitions from the invariant).

In this paper, we consider the α-mapping, and the work that has been done on behavioural profiles, due to the fact they can be computed efficiently (in case of free-choice Petri nets [20]) on the structure of the net. The other abstractions can be used, but some do no always run in a time polynomial with respect to the size of the net. Causal footprints are closely related to behavioural profile. Therefore, using causal footprints yields similar results as using behavioural profiles. Hence, we do not use causal footprints. To use causal nets, we would need an approach to deal with the places introduced in the transformation as the same place might have different names in different process models. To use process nets, we would need an approach to deduce language inclusion between sets of sets of marked graphs. Furthermore, the number of process nets might be exponential in the number of choices, as there is no choice in a marked graph and every trace needs to be encoded in at least one marked graph. Finally, the main problem with semi-positive transition invariants is that the basis of invariant may be exponential in the size of the net.

3 Preliminaries

We use the following general notions: $\mathcal{A}$ denotes the set of actions, and $\mathbb{N}$ denotes the set of non-negative integers.

First, the notions of language, word of a language, and language inclusion are defined. A word is a sequence of actions from $\mathcal{A}$, a language is a set of words, and a language is included in another language if and only if all the words of the first language are part of the second language.

Next, we introduce the notion of a transition system, and the language of a transition system.

Definition 1 (Transition System). *A Transition System $TS = (S, \rightarrow, \alpha, \omega)$ is a tuple where:*

- *S is the set of states,*
- *$\rightarrow \subseteq S \times \mathcal{A} \times S$ is the set of transitions,*
- *$\alpha \in S$ is the initial state,*
- *$\omega \in S$ is the final state.*

A word of a transition system is a sequence of transitions starting from the initial state and ending in the final state. The language of a transition system is the set of all words which can be produced by the transitions system.

Now we introduce some general Petri net concepts.

Definition 2 ((Unlabelled) Petri net). *A Petri net $N = (P, T, F)$ is a tuple where:*

- *P is a set of places,*
- *$T \subseteq \mathcal{A}$ is a set of transitions, such that $P \cap T = \emptyset$,*
- *$F \subseteq (P \times T) \cup (T \times P)$ is a flow relation.*

The preset of a transition/place n, denoted by $\bullet n$, is the set of places/transitions in $\bigcup i \in P \cup T : (i, n) \in F$. The postset of a transition/place n, denoted by $n\bullet$, is the set of places/transitions in $\bigcup i \in P \cup T : (n, i) \in F$. A Petri net is free-choice if and only if for every two transitions, either the presets of every two transitions is disjoint, or they are the same.

Definition 3 (Bag over Set). *Let S be a set. A bag over S is a function from S to the natural numbers $\mathbb{N}$ such that only a finite number of elements from S is assigned a non-zero function value.*

Note that a finite set is also a bag, namely the function assigning 1 to every element in the set and 0 otherwise.

The set of all bags over S is denoted $\mathcal{B}(S)$. For a bag b over S and $s \in S$, $b(s)$ denotes the number of occurrences of s in b, often called the *cardinality* of s in b. We use brackets to explicitly enumerate a bag and superscripts to denote cardinalities. For example, $[a^2, b^3, c]$ is the bag with two a's, three b's and one c; the bag $[s^2 \mid P(s)]$, where P is a predicate on S, contains two elements s for

every s such that $P(s)$ holds. The sum of two bags b_1 and b_2, denoted $b_1 + b_2$, is defined as $[s^n \mid s \in S \wedge n = b_1(s) + b_2(s)]$. The difference of two bags b_1 and b_2, denoted $b_1 - b_2$, is defined as $[s^n \mid s \in S \wedge n = (b_1(s) - b_2(s)) \max 0]$. Bag b_1 is a subbag of b_2, denoted $b_1 \leq b_2$, if and only if $\forall s \in S : b_1(s) \leq b_2(s)$.

A marking of a Petri net is a bag over the set of places. A transition t is enabled in a marking M, denoted by $M[t\rangle$, if and only if the preset of t is a subbag of M. Firing an enabled transition t from a marking M resulting in another marking M' is denoted by $M[t\rangle M'$. Marking M' is obtained by taking the difference between M and the preset of t and summing it with the postset of t, i.e. $M' = M - {\bullet t} + t{\bullet}$.

Definition 4 (Transition System of a Petri net). *Let $N = (P, T, F)$ be a Petri net, let M_i be a marking of N, and let M_o be a marking of N, then the transition system $TS = TS(N, M_i, M_o)$, belonging to N, is constructed as follows:*

- $\alpha = M_i$,
- $\omega = M_o$,
- *S is the smallest set X, such that:*
  - $\alpha \in X$, $\omega \in X$, *and*
  - *if $s \in X \wedge s[t\rangle s'$ for some $t \in T$ and $s' \in \mathcal{B}(P)$, then $s' \in X$.*
- $\rightarrow = \{(s_1, t, s_2) \mid s_1, s_2 \in S \wedge t \in T \wedge s_1[t\rangle s_2\}$.

The language of a Petri net is the language of the transition system as constructed in Def. 4.

As mentioned, we limit ourselves to WF-nets [21]. A WF-net has a unique input and output place, i.e., the preset of the input place is empty and the postset of the output place is empty. Furthermore, every transition is on a path between the input place and the output place. With $TS(N)$, we denote the transition system belonging to the WF-net N with unique input place i and output place o (i.e., $TS(N) = TS(N, [i], [o])$).

We require a WF-net to be sound [21].

Definition 5 (Soundness [21]). *Let $N = (P, T, F)$ be a WF-net, let $TS = (S, \rightarrow, \alpha, \omega)$ be a transition system belonging to N, i.e., $TS = TS(N)$, then N is sound if and only if (note that α is $[i]$ and ω is $[o]$):*

- *$\forall m \in S : (m, \omega) \in \rightarrow^*$, where $\rightarrow^*$ is the transitive closure of $\rightarrow$,*
- *$\forall t \in T : \exists s, s' \in S : (s, t, s') \in \rightarrow$.*

In the remainder of this paper, $\mathcal{N}$ denotes the set of all sound unlabelled WF-nets. Note that whenever we say WF-net the unlabelled variant is intended unless stated differently.

As mentioned in the related work section, there exist approaches to compute language inclusion on the transition system. However, this computation does not need to be polynomial in the size of the Petri net. Therefore, we provide in Sect. 4 an approach to deduce language inclusion based on the structure of the Petri net.

4 Approach

In this section, we determine in which situations language inclusion can be efficiently computed on the transition system, and what can be done to avoid using the transition system in other situations. For the latter, we use our framework, i.e., when the net has certain characteristics, this net is mapped onto an alternative representation. Using a comparator, language inclusion can be decided, similar to the example given in the introduction.

Whether it is computationally efficient to use the transition system and, if not, which mapping can be used, depends heavily on the constructs used in the Petri nets, i.e., on the characteristics of both nets. For example, if both nets are sequential and acyclic, then language inclusion can be efficiently computed on the transition systems. If both nets are acyclic, then the *eventually-follows* relation between transitions may provide valuable information on language inclusion, where the eventually-follows relation between two transitions denote that the first can be followed by the second.

The remainder of this section is organised as follows: We first define an abstract mapping, then introduce characteristics, and finally define a comparator. Afterwards, we revisit the used (concrete) mappings from literature. Having the concrete mappings in place, the characteristics are defined, and a polynomial-time algorithm is provided to compute these characteristics on the free-choice WF-nets. Using the mappings and characteristics, we provide comparators and accompanying abstractions together, as they are tightly linked.

4.1 Abstract framework

In our abstract framework, an abstraction consists of three elements: (1) a mapping function, mapping a Petri net into an abstract representation, (2) a set of characteristics denoting when this abstraction can be applied, and (3) a comparator to compare the abstract representations (mapping to the booleans). The first and the last follow straightforward from the previous explanations, but the second requires some extra explanation and mainly on the *can be applied* part.

We say a Petri net adheres to certain characteristics if and only if these characteristics are valid for this Petri net. This allows abstractions to be used when characteristics are added, i.e., the addition of an characteristic does not invalidate the results presented in this paper.

Two different kinds of abstractions are considered: (1) a positive abstraction between N and N' denoting that: If the comparator yields true, N is guaranteed to be language included in N', and (2) a negative abstraction between N and N' denoting that: If the comparator yields false, N is guaranteed to be *not* language included in N'. This allows our framework to be more flexible.

4.2 Mappings

Having the abstract framework with abstractions in place, we now present the specific mappings used (based on the reasoning in Sect. 2) in the remainder of

Fig. 2: Two WF-nets with different languages but with the same mapping.

this paper. Please note that we do not claim that these mappings are exhaustive: Other mappings may exist that allow for efficient positive and/or negative abstractions that are not covered by this paper. However, these mappings (and corresponding comparator) can be easily added to our framework.

The T-mapping provides us with the set of transitions from a WF-net, as explained in the introduction (we denote this set as $\lambda_T(N)$).

The α-mapping denotes that two transitions can follow each other directly in a Petri net N, i.e., if (t_1, t_2) is part of the mapping (denoted by $\lambda_\alpha(N)$), then there is a trace in which t_1 is *directly* followed by t_2.

The ∞-mapping denotes that two transitions can follow each other eventually in a Petri net N, i.e., if (t_1, t_2) is part of the mapping (denoted by $\lambda_\infty(N)$), then there is a trace in which t_1 is *eventually* followed by t_2.

The presence of loops may be problematic for the ∞-mapping. For this reason, we also include the k-mapping (generalisation of the α-mapping), which denotes that there are *at most* k other activities between two transitions in a Petri net N. As such, this mapping (denoted by $\lambda_k(N)$) corresponds to the eventually-follows-within-k-steps-relation between transitions. As a result, transitions that occur in a loop of length of at least k are not automatically related in both ways by this mapping, e.g., assume a and b are in a loop, then (a, b) in $\lambda_k(N)$ but (b, a) does not need to be in $\lambda_k(N)$ while (b, a) would have been included in $\lambda_\infty(N)$.

These latter three specific mappings are all based on relations between two transitions, which poses a possible problem if a net only contains traces that contain only a single transition. Clearly, for such a net these follows relations are all empty, which yields equivalent mappings for the nets in Fig. 2. For this reason, we add artificial start and end activities $(\top, \bot)$ to any net. Note that this does not limit the expressivity of the process models.

4.3 Characteristics

Since our framework depends on the exact definition of the characteristics, this section defines the characteristics of sound WF-nets, and shows that we can compute these characteristics efficiently in case of free-choice nets.

The *sequential characteristic* denotes that from a state firing one transition, means the other transitions enabled in the state are no longer enabled.

Definition 6 (Sequential Characteristic). *Let $N = (P, T, F) \in \mathcal{N}$ be a sound WF-net, let $TS = (S, \rightarrow, \alpha, \omega)$ be the transition system belonging to N, i.e., $TS = TS(N)$, then N is sequential if and only if: $\forall t_1, t_2 \in T, s \in S$: $\neg(\bullet t_1 + \bullet t_2 \leq s)$*

The *acyclic characteristic* denotes that it is not possible to reach a previously visited state.

Definition 7 (Acyclic Characteristic). *Let $N = (P, T, F) \in \mathcal{N}$ be a sound WF-net, let $TS = (S, \rightarrow, \alpha, \omega)$ be the transition system belonging to N, i.e., $TS = TS(N)$, then N is acyclic if and only if:* $\forall n \in \mathbb{N}, s_0, \ldots, s_n \in S, t_1, \ldots, t_n \in T : n < 1 \vee s_0 \neq s_n \vee \exists 1 \leq k \leq n : (s_{k-1}, t_k, s_k) \notin \rightarrow$.

In general, these characteristics are defined on the transition system, and may be inefficient to compute. However, for free-choice nets we can compute them in an efficient way, as the following theorems show.

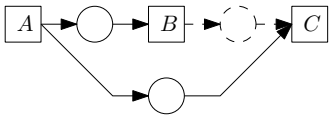


Fig. 3: With the dashed place and arrows, we have a sequential construct, else not.

In [1](Def. 26), a polynomial time algorithm is presented to compute the minimal structural successor between places and transitions (MSS) for a free-choice net, i.e., the minimal set of transitions and places between the execution of two transitions. For some characteristics, we require that the considered transitions can follow each other *directly*, thus a MSS has a size of 1, i.e., only a single place is between the transitions. Consider Fig. 3, without the dashed parts, B and C are non-sequential. However, with the dashed parts, B and C are sequential. In the first case, the size of the MSS between A and C is 1, in the latter case the size of the MSS between A and C is 3.

Theorem 1. *Let $N = (P, T, F)$ be a sound, free-choice WF-net, then N is sequential if and only if:* $\forall u_1, u_2, u_3 \in T : u_1 \bullet \cap \bullet u_2 = \emptyset \vee u_1 \bullet \cap \bullet u_3 = \emptyset \vee \bullet u_2 \cap \bullet u_3 \neq \emptyset \vee |MSS(u_1, u_2)| \neq 1 \vee |MSS(u_1, u_3)| \neq 1$.

Proof. We need to prove: $(\forall t_1, t_2 \in T, s \in S : \neg(\bullet t_1 + \bullet t_2 \leq s)) \Leftrightarrow (\forall u_1, u_2, u_3 \in T : u_1 \bullet \cap \bullet u_2 = \emptyset \vee u_1 \bullet \cap \bullet u_3 = \emptyset \vee \bullet u_2 \cap \bullet u_3 \neq \emptyset \vee |MSS(u_1, u_2)| \neq 1 \vee |MSS(u_1, u_3)| \neq 1)$.

$\Rightarrow$ We prove this by contradiction, assume the left-hand side is true, but the right-hand side is false, thus $\exists u_1, u_2, u_3 \in T : u_1 \bullet \cap \bullet u_2 \neq \emptyset \wedge u_1 \bullet \cap \bullet u_3 \neq \emptyset \wedge \bullet u_2 \cap \bullet u_3 = \emptyset \wedge |MSS(u_1, u_2)| = 1 \wedge |MSS(u_1, u_3)| = 1$. From the right-hand side it follows that u_2 and u_3 are both enabled after executing u_1 (by soundness, we know u_1 can fire). Furthermore, u_2 and u_3 can fire directly after u_1 by the MSS, and because their presets are disjoint there is no ordering between u_2 and u_3. Hence, there is a state s such that $\bullet u_2 + \bullet u_3 \leq s$.

$\Leftarrow$ Again we prove this by contradiction, assume the right-hand side is true, but the left-hand side is false, thus $\exists t_1, t_2 \in T, s \in S : \bullet t_1 + \bullet t_2 \leq s$. Since

we have a sound WF-net, we start with a single token in the input place. This means there is a sequence of transitions to reach a state s where two transitions are enabled say t_1 and t_2, let s be the first state. Then there is a transition t to reach state s, let t fire from state s', i.e., $s'[t\rangle s$. By the fact that t_1 and t_2 can fire directly from s, we know that $|MSS(t, t_1)| = 1$ and $|MSS(t, t_2)| = 1$. Now it remains to prove that the first 3 clauses are false, that is: $u_1 \bullet \cap \bullet u_2 \neq \emptyset$ and $u_1 \bullet \cap \bullet u_3 \neq \emptyset$ and $\bullet u_2 \cap \bullet u_3 = \emptyset$.

- Assume $t \bullet \cap \bullet t_1 = \emptyset$, this means that $s'[t\rangle$ and $s'[t_1\rangle$. Then there are two options: $\bullet t + \bullet t_1 \leq s'$, this means s' is an earlier state in which two transitions are enabled (not possible by assumption), and $\neg(\bullet t + \bullet t_1 \leq s')$. We now obtain by free-choiceness that $\bullet t = \bullet t_1$. From the fact that t_1 is still enabled after firing t, and the fact that $t \bullet \cap \bullet t_1 = \emptyset$, we obtain that $\bullet t + \bullet t_1 \leq s'$, this is not possible due to the free-choiceness and soundness as this allows t_1 to fire twice, and hence (by free-choiceness) it is possible to mark the output place with two tokens.
- The proof for $t \bullet \cap \bullet t_2$ goes analogous.
- We only have the case where $\bullet t_1 \cap \bullet t_2 \neq \emptyset$, thus $\bullet t_1 = \bullet t_2$. Again by soundness and free-choiceness it follows that t_1 can fire twice from state s, hence it is not sound.

□

Theorem 2. *Let $N = (P, T, F)$ be a sound, free-choice WF-net, then N is acyclic if and only if:* $\forall t_1, \ldots t_n \in T : \bullet t_1 \cap t_n \bullet = \emptyset \vee (\exists 1 \leq k < n : t_k \bullet \cap \bullet t_{k+1} = \emptyset)$.

Proof. We need to prove: $(\forall n \in \mathbb{N}, s_0, \ldots, s_n \in S, t_1, \ldots, t_n \in T : n < 1 \vee s_0 \neq s_n \vee \exists 1 \leq k \leq n : (s_{k-1}, t_k, s_k) \notin \rightarrow) \Leftrightarrow (\forall t_1, \ldots t_n \in T : \bullet t_1 \cap t_n \bullet = \emptyset \vee (\exists 1 \leq k < n : t_k \bullet \cap \bullet t_{k+1} = \emptyset))$.

$\Rightarrow$ This follows from the free-choice property, i.e., as soon as we can mark a place in the preset of a transition, then this token can only be removed after this transition has been enabled. As a result, if we mark a place in the preset of a previously enabled transition, then we can do this an infinite amount of times as this transition has to be able to fire. Since the state-space is finite (bounded net due to soundness) this means we have a path to a previously visited state.

$\Leftarrow$ When we do not have a sequence of transition to mark a place again, this means as soon as this place has been marked it will never be marked again. Hence, we cannot revisit a state.

□

From Thm. 1 and Thm. 2, we can conclude that, for free-choice WF-nets the characteristics can be efficiently computed, i.e., in polynomial time. As a result, we can use these characteristics in our framework.

4.4 Abstractions

This section builds on the previous sections by presenting concrete instantiations of the framework, consisting of selected abstractions, that can be used to efficiently decide language inclusion for nets with certain characteristics. Therefore, in the remainder, we display different approaches, each tailored towards nets with certain characteristics. We start with the approach which does not require an abstraction of the Petri net.

No abstraction needed In case the Petri net is sequential and free-choice, the transition systems will not suffer from the state-space-explosion problem. For this reason, we do not need an abstraction, as we can efficiently decide the language inclusion problem on the transition systems.

Negative abstraction using T-mapping The following negative abstraction is valid for all sound WF-nets.

Definition 8 (T-Comparator). *Let $N = (P, T, F)$ and $N' = (P', T', F')$ be sound WF-nets, let λ_T be the T-mapping, then $\lambda_T(N) \leq_T \lambda_T(N')$ if and only if: $\lambda_T(N) \subseteq \lambda_T(N')$*

Theorem 3. *$(\lambda_T, \leq_T, \emptyset)$ is a negative abstraction.*

Proof. As both nets are sound WF-nets, all transitions of N are included in $\mathcal{L}(N)$. That is, for every $t \in T$ there exists a trace $\sigma \in \mathcal{L}(N)$ such that $t \in \sigma$. Clearly, if there exists a transition $t \in T$ such that $t \notin T'$, then the traces in N that contain t are not traces of N'. Hence, language inclusion cannot hold. □

Negative abstraction using α-mapping The following negative abstraction is also valid for all sound WF-nets. However, to our knowledge, only for free-choice WF-nets there is a polynomial-time algorithm to compute the α-mapping. Hence we can use the negative abstraction to efficiently decide language inclusion either if we have a free-choice net or if the required α-mappings are given.

Definition 9 (α-Comparator). *Let $N = (P, T, F)$, $N' = (P', T', F')$, be sound WF-nets, let λ_α be the α-mapping, then $\lambda_\alpha(N) \leq_\alpha \lambda_\alpha(N')$ if and only if: $\lambda_\alpha(N) \subseteq \lambda_\alpha(N')$*

Theorem 4. *$(\lambda_\alpha, \leq_\alpha, \emptyset)$ is a negative abstraction.*

Proof. We need to prove that $\neg(\lambda_\alpha(N) \leq_\alpha \lambda_\alpha(N')) \Rightarrow \neg(\mathcal{L}(N) \subseteq \mathcal{L}(N'))$ which can be rewritten to $\mathcal{L}(N) \subseteq \mathcal{L}(N') \Rightarrow \lambda_\alpha(N) \leq_\alpha \lambda_\alpha(N')$ which follows straightforward from the definition of λ_α. □

Theorem 4 yields the following result: Given two sound WF-nets and their corresponding α-mappings, Def. 9 can be computed in polynomial time. Hence, Thm. 4 yields in polynomial time whether the language is *not* included, i.e., if

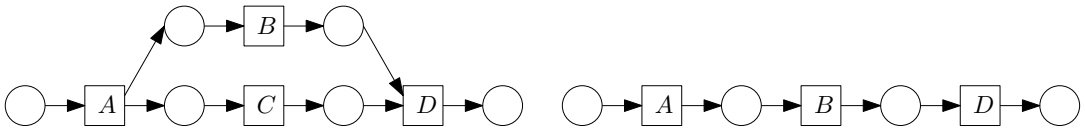


Fig. 4: Two process models where we have a false positive w.r.t. language inclusion if we simply use inclusion of the α-relation.

$\lambda_\alpha(N) \subseteq \lambda_\alpha(N')$ yields *false* then the inclusion does not hold. In case no α-mapping is provided, we can, for free-choice WF-nets, compute the α-mapping efficiently and determine if language inclusion does not hold.

Using the α-mapping, we did not obtain a positive abstraction, as we cannot do an inclusion of the relations between the different process models. Consider the models in Fig. 4. Here, it is easy to see that the relations of the right-hand model are a subset of left-hand model. That is, in the left model we have the relations $\lambda_\alpha(N) = \{(A, B), (A, C), (B, C), (C, B), (B, D), (C, D)\}$, and in the right model we have the relations: $\lambda_\alpha(N') = \{(A, B), (B, D)\}$. However, the word ABD is included in the right model, but not in the left model.

The problem is that we are omitting related activities, e.g., the A is followed, in Fig. 4, by a C, while with omitting relations, we also omit these dependencies.

Consider Fig. 5, at the bottom, for both models, we have transformed the parallel execution into a sequential execution such that λ_α(right model) $\subseteq \lambda_\alpha$(left model). In the top two models, we have done the same, but this yields a model allowing words not possible in the other model. So, we need to find an inclusion operator, which can either differentiate between both models or considers both models not to be included in the other. In the first case, we have false negatives (see bottom model), in the second case, we have false positive (see top model). The problem is: The α-mapping considers two activities exclusive if they cannot follow each other directly. Hence, we have to look at the transitivity of these relations.

Positive abstraction using ∞-mapping In case the WF-net is acyclic, we can use the ∞-mapping instead of the α-mapping. Using this mapping, we can compute the transitive closure of the α-relations to deduce the relations between all activities. However, similar to the α-mapping, a simple inclusion of the sets is not strong enough to deduce language inclusion. Therefore, we need an approach which takes into account the alternatives between different activities. Alternative activities are activities which do not occur together in a trace. So, if an activity does not occur in a word, then an alternative activity occurs in that word.

Theorem 5. *Let $N = (P, T, F)$ be a sound acyclic free-choice net, then:* $\forall a \in T, \sigma \in \mathcal{L}(N) : a \notin \sigma \Rightarrow \exists b \in T : b \in \sigma \wedge (a, b) \notin \lambda_\infty(N) \wedge (b, a) \notin \lambda_\infty(N)$.

Proof. Assume there is an $a \in T$ and $a \notin \sigma$ (by soundness we know a can be executed). This means there is a choice to not execute a. Since we have free-choice nets, this means somewhere there are two transitions t_1, t_2 ($\bullet t_1 = \bullet t_2$),

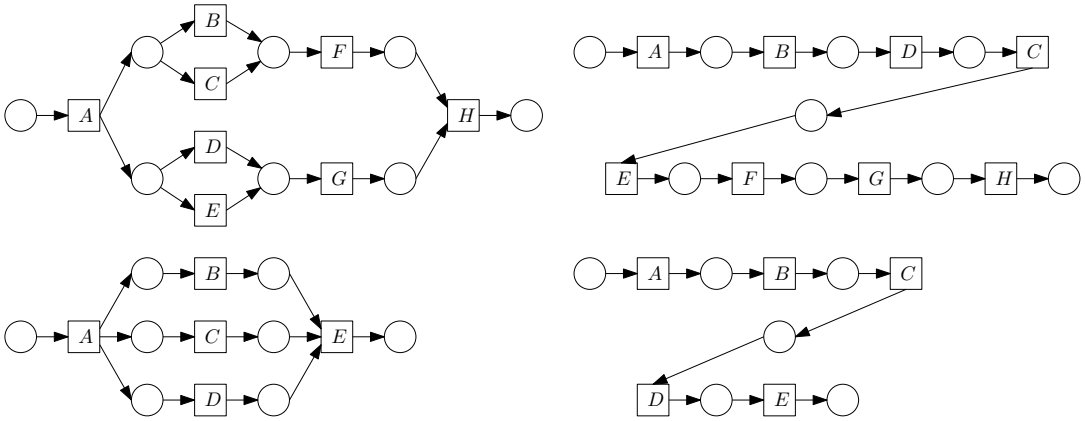


Fig. 5: Using the α-relation, we cannot differentiate between valid transformations from parallel to sequential (bottom) and invalid transformations from parallel to sequential (top).

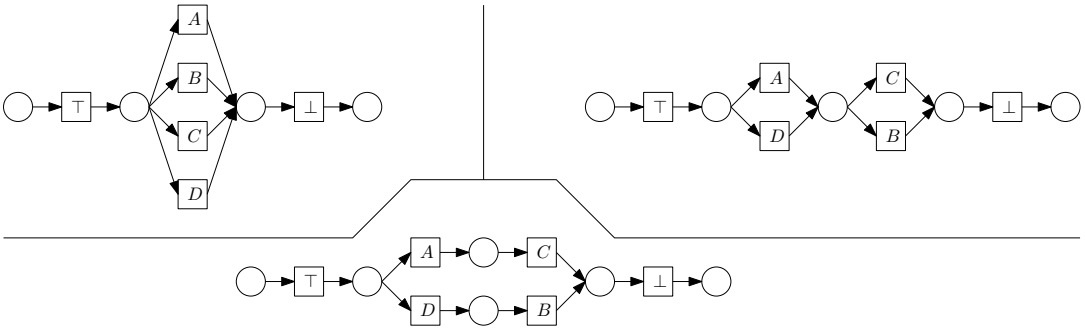


Fig. 6: The left and middle model are language included in the right model without clause 3 in Def. 10.

such that after t_1 a can still follow, but after executing t_2 a cannot follow. Due to the fact that the net is acyclic and free-choice, a cannot precede t_2, i.e., if a can, via firing some transitions, mark the preset of t_2 then also the preset of t_1 is marked and hence a can execute again. Since a is not in σ, we have chosen t_2, and thus $t_2 \in \sigma$. Also because a cannot follow t_2, and t_2 cannot follow a, $(a, t_2) \notin \lambda_\infty(N) \wedge (t_2, a) \notin \lambda_\infty(N)$. □

Now the comparator denotes that the activities have to be the same, the relations have to be a subset (similar to Def. 9), and there is always at least one alternative activity remaining. We have to require clause 3, as this allows us to only remove alternative activities. Consider Fig. 6, the lower model is language included in the right model, and this is valid since an alternative activity for B (namely C) follows A. However, the left model is not language included in the lower model, as there is no alternative activity for C following A. Without clause 3, the comparator would have denoted that the left model is included in the lower model.

Definition 10 (∞-Comparator). *Let $N = (P, T, F)$, $N' = (P', T', F')$ be sound FC-nets adhering to the {Acyclic} characterisation, let λ_∞ be the ∞-mapping, then $\lambda_\infty(N) \leq_\infty \lambda_\infty(N')$ if and only if:*

1. $T = T'$
2. $\lambda_\infty(N) \subseteq \lambda_\infty(N')$
3. $(\forall (a,b) \in \lambda_\infty(N') : (a,b) \in \lambda_\infty(N) \vee (b,a) \in \lambda_\infty(N) \vee (\exists c \in T : ((a,c) \in \lambda_\infty(N) \vee (c,a) \in \lambda_\infty(N)) \wedge (b,c) \notin \lambda_\infty(N') \wedge (c,b) \notin \lambda_\infty(N')))$

Theorem 6. $(\lambda_\infty, \leq_\infty, \{Acyclic\})$ *is a positive abstraction.*

Proof. We need to prove: $\lambda_\infty(N) \leq_\infty \lambda_\infty(N') \Rightarrow \mathcal{L}(N) \subseteq \mathcal{L}(N')$.

We prove this by contradiction, thus $\mathcal{L}(N) \not\subseteq \mathcal{L}(N')$ *but* $\lambda_\infty(N) \leq_\infty \lambda_\infty(N')$. By definition, this means there is a trace $\sigma \in \mathcal{L}(N)$ such that $\sigma \notin \mathcal{L}(N')$.

Now it remains to prove that this σ does not exist. We prove this by induction on the prefix of the trace. Let $\sigma = \langle t_1, \ldots, t_{k-1}, t_k, \ldots, t_m \rangle$ be a trace, the prefix of σ (denoted by σ_{prefix}) are transitions before position k (i.e., $\sigma_{\text{prefix}} = \langle t_1, \ldots, t_{k-1} \rangle$). It remains to show that if the prefix of σ corresponds to a prefix of a trace $\sigma_{N'} \in \mathcal{L}(N')$ then t_k has to be directly executable in N' after having executed this prefix.

Base case $k = 1$. In this case, the prefix is empty and by definition we always start with an unique start activity, hence both models can execute $\top$.

Inductions hypothesis: assume $\sigma_{\text{prefix}} = \langle t_1, \ldots t_{k-1} \rangle$ corresponds to a prefix of a trace in N', then there are two options: (1) t_k cannot follow σ_{prefix} in N', or (2) t_k cannot follow σ_{prefix} directly in N', i.e., there is at least on transition in between σ_{prefix} and t_k.

(1) If t_k cannot follow σ_{prefix} in N', then either $t_k \notin T'$ (not possible due to clause 1) or there is an alternative activity $t_i \in \sigma$ (Thm. 5). Note that this alternative activity has to be in σ_{prefix}, because else from σ_{prefix} it cannot follow that t_k cannot follow σ_{prefix} in N'. This latter option cannot happen due to clause 2 (else $(t_i, t_k) \in \lambda_\infty(N)$ *but* $(t_i, t_k) \notin \lambda_\infty(N')$). Hence t_k must be able to follow σ_{prefix} in N'.

(2) When t_k cannot directly follow σ_{prefix} this means there is an activity between σ_{prefix} and t_k in N'. We denote this activity with $b \in T'$. We know that $(b, t_k) \in \lambda_\infty(N')$ but also that $(t_k, b) \notin \lambda_\infty(N')$ (else b does not need to be between σ_{prefix} and t_k, by free-choiceness and soundness). Furthermore, by the absence of loops and soundness, we know that $b \notin \sigma_{\text{prefix}}$ (else b can be followed by b in N'). Then there are two options; (a) either b can follow σ_{prefix} in N, or (b) b cannot follow σ_{prefix} in N.

(a) Since $(t_k, b) \notin \lambda_\infty(N')$ it follows from clause 2, that $(t_k, b) \notin \lambda_\infty(N)$. This means that if b can follow σ_{prefix} in N, it has to be executed before t_k is executed. In σ this is not the case, i.e., there is no activity executed between σ_{prefix} and t_k, and Thm. 5 yields that there has to be an alternative activity for b in σ (note that it does not need to be an alternative activity for b in N') and by free-choiceness it has to be before t_k, thus in σ_{prefix}. From this it follows that b cannot follow σ_{prefix} in N. We only need to prove that part (b) yields a contradiction.

(b) Thm. 5 yields that somewhere we have made a choice in σ_{prefix} to not execute b in N, we call the activity after making this choice a. Since σ_{prefix} can be followed by b in N', we obtain that $(a, b) \in \lambda_\infty(N')$, but also $(a, b) \notin \lambda_\infty(N)$ (note that $(b, a) \notin \lambda_\infty(N)$, since this violates the acyclic property). From clause 3, we obtain there has to be an activity c such that $(a, c) \in \lambda_\infty(N) \vee (c, a) \in \lambda_\infty(N)$ and $(b, c) \notin \lambda_\infty(N') \wedge (c, b) \notin \lambda_\infty(N')$. Finally, from Thm. 5, we obtain that there has to be an alternative activity for b in σ, which c also is. Now it remains to show that c is in σ_{prefix}. By free-choiceness, we obtain that c has to be executed before t_k. If c can be executed after t_k, then dependent on the activities chosen before t_k, c can either be executed or not, which violates the free-choice property. Since $(c, t_k) \in \lambda_\infty(N)$, it follows that c has to be in σ_{prefix}, and thus no b can be between σ_{prefix} and t_k in N'.

From this it follows that both models can execute t_k directly after prefix σ_{prefix}, and hence both models are able to perform σ (note that termination is taken care of by including $\perp$ as unique end activity) and thus it follows that σ is also in $\mathcal{L}(N')$. Thus by contradiction, we have shown that $\lambda_\infty(N) \leq_\infty \lambda_\infty(N') \Rightarrow \mathcal{L}(N) \subseteq \mathcal{L}(N')$. □

In case $T = T'$, we can decide language inclusion based on the ∞-mapping. However, in general it is the case that $T \neq T'$. Therefore, we extend our approach to the case in which $T \subset T'$. Note that we do not need to consider the case that $T' \subset T$ as language inclusion cannot hold.

In order to achieve $T = T'$, we have the following reduction on N':

- Remove all transitions $t \in (T' \setminus T)$ from N'
- Remove all places not connected to any transition

If the reduced net is sound then we can use Def. 10 on the transformed net with $T = T'$. It is easy to see that language inclusion is preserved as we only remove transitions not present in T, i.e., we do not reduce the language w.r.t. the overlap with T. If the resulting model is not sound, then a part of the language of N cannot be part of N'.

Apart from showing the soundness of the abstraction using the ∞-mapping, we also want to show the completeness. This means that if the language of N is included in N', then our comparator always yields true.

Theorem 7. *$(\lambda_\infty, \leq_\infty, \{Acyclic\})$ is complete, i.e., for two acyclic free-choice WF-nets N and N', we have $\lambda_\infty(N) \leq_\infty \lambda_\infty(N') \Leftarrow \mathcal{L}(N) \subseteq \mathcal{L}(N')$.*

Proof. We prove this by contradiction. Assume $\mathcal{L}(N) \subseteq \mathcal{L}(N')$, but $\lambda_\infty(N) \leq_\infty \lambda_\infty(N')$ does not yield true.

Then, at least one of the clauses has to be false. Clause 1 and 2 follow in a straightforward way. Assume clause 1 is false, then there is an activity $x \in T$ which is not in T'. Since T is sound, this means x occurs in at least one trace and this trace cannot occur in $\mathcal{L}(N')$. Assume clause 2 is false, this means there is a relation between 2 activities $x, y \in T$ such that $(x, y) \in \lambda_\infty(N)$, which is not in $\lambda_\infty(N')$. By definition, this means there is a word such that

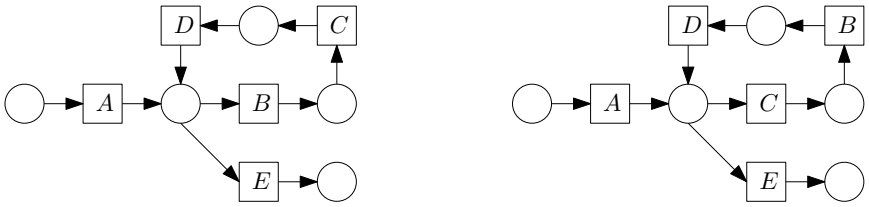


Fig. 7: Two process models where using the ∞-mapping gives a false positive.

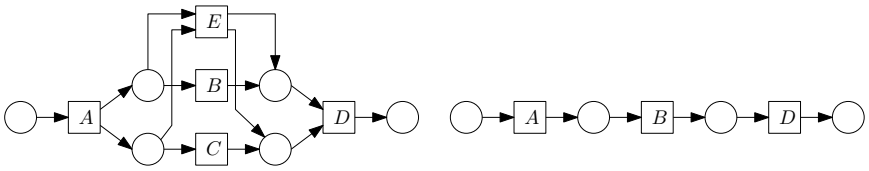


Fig. 8: Naive inclusion does not work for the k-mapping.

$\sigma = \langle \ldots, x, \ldots, y, \ldots \rangle$, but this trace cannot be in $\mathcal{L}(N')$, else this relation was also included in $\lambda_\infty(N')$.

Assume clause 3 is not valid, this means that: $\exists (a,b) \in \lambda_\infty(N') : (\forall c \in T : \neg((a,c) \in \lambda_\infty(N) \vee (c,a) \in \lambda_\infty(N)) \vee (c,b) \in \lambda_\infty(N') \vee (b,c) \in \lambda_\infty(N'))$ has to hold (negation of the clause 3). Recall that due to $\top$ and $\bot$, $\lambda_\infty(N')$ cannot be empty.

We choose an a and b, the universal quantifier denotes that either the activity c is exclusive to a, or can occur together with b in a trace. Thus, there is no alternative for b to occur together with a in the trace. If we chose $b = c$ then we have that either b does not occur with a in a trace in N, or $(b,b) \in \lambda_\infty(N')$ which is not possible due to the acyclicness. Thus, in N' a always occurs with b, while in N they are alternatives. Hence, language inclusion cannot hold.

We can conclude that Thm. 7 holds, and Def. 10 is complete. □

No abstraction using the *k*-mapping In the most general case, when the characterisation does not pose any constraints, we have the same problems as mentioned earlier, e.g., that a simple inclusion does not work. Furthermore, we now have difficulties introduced by the combination of cycles and non-sequential activities, e.g., short loops in the α-mapping. Furthermore, the ∞-mapping has problems with cycles. For instance, see Fig. 7 where both models have the same ∞-mapping. However, the trace $ACBDE$ is included in the right model, but not in the left model. Obviously, with equivalent ∞-mapping, we should have language equivalence, i.e., any comparator denotes that the sets are included in each other.

Therefore, we consider the k-mapping. Again, using a naive approach, using the inclusion of relations does not work. Consider, for instance, the models in Fig. 8, where the relations in the right model are a subset of the relations in the left model.

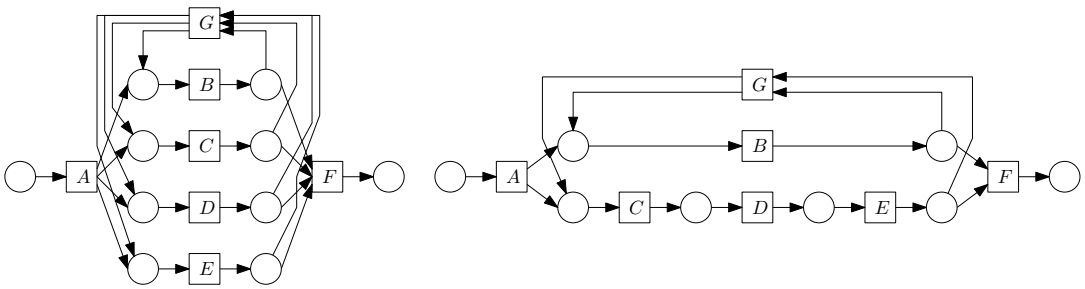


Fig. 9: For the k-mapping, we cannot find a value for k such that it yields that the right model is language included in the left model, but does not yield that the left model is language included in the right model.

Table 1: The framework for sound free-choice WF-nets

Abstraction	Characterisation	Pos/Neg	Complete
Trans. Syst.	S	Pos	✓
T	$\emptyset$	Neg	
α	$\emptyset$	Neg	
∞	A	Pos	✓

The problem with looking at k elements in advance is that this k is bounded by the shortest cycle. In other words, if k is larger than the shortest cycle then activities are considered in parallel when they are in sequence. Consider Fig. 9, from which it is clear that the right model is language-included in the left model. However, if we compute the relation between the activities, we need to have a relation between C and E (because present in right model, thus $k \geq 2$), but also $k < 2$ because $\langle ...EGC... \rangle$ is a proper part of a trace due to the cycle, while this would yield that C and E are in parallel in the right model, and thus that the left model is language included in the right model.

4.5 The complete framework

In Table 1, the total framework is listed. *Trans. Syst.* denotes that language inclusion can be efficiently computed on the transitions system. T is the negative abstraction only considering transitions. The negative abstraction using the α-mapping is listed under α. Finally, ∞ denotes the sound and complete positive abstraction using the ∞-mapping. Under characteristics: A denotes acyclic, and S denotes sequentiality. Pos/Neg denotes if it is a positive or negative approach, and complete denotes if the approach is complete.

5 Conclusion

We have presented a framework that supports the decision of language inclusion for sound unlabelled WF-nets in an efficient way. Apart from the framework,

we have provided a number of mappings, characterisations, and comparators to be used in the framework. If there is sequentiality present in both WF-nets, the language inclusion problem can be computed in polynomial time in terms of the underlying transition systems. If one of the WF-nets is non-sequential, then our framework may still efficiently decide language inclusion using a number of abstractions. The first abstraction uses the sets of transitions: If the set of transitions of the first net is not a subset of the set of transitions of the second, then the language of the first net cannot be included in the language of the second net. The second abstraction uses the directly-follows relation (also called the α-relation) between transitions: If the α-relation of the first net is not a subset of the α-relation of the second net, then the language of the first net cannot be included in the language of the second net. The third abstraction uses the eventually follows relation (also called the ∞-relation) between transitions: If the ∞-relation of the first free-choice net is a subset of the ∞-relation of the second free-choice net, and if some additional structural requirements hold, then the language of the first net is included in the language of the second net. For free-choice acyclic nets this third abstraction is complete. In other words, language inclusion can only hold if the ∞-relation of the first net is a subset of the ∞-relation of the second net, and if the additional structural requirements hold.

In case of free-choice unlabelled WF-nets, all relations can be computed polynomial in the size of the WF-net. On these relations, we mainly perform containment between sets. In the third abstraction, we search for an alternative activity which, naively, entails trying every activity and adds a polynomial factor. Since each of the steps is polynomial in the size of the WF-net, we can conclude that our computations can be done polynomial in the size of the WF-net.

The framework cannot efficiently decide on language inclusion in all cases. Especially for nets that are (1) non-sequential, (2) are cyclic or are not free-choice, and (3) for which language inclusion is known to hold, our current set of abstractions will not be able to provide an answer in polynomial time. Due to the fact that the comparators of the first two abstraction will yield true and the preconditions of the third abstraction are not met. However, we do not claim that our framework is complete. Yet, our framework can be easily extended based on the general pattern provided with new abstractions that may provide efficient answers for cases uncovered by the current abstractions: Given an abstraction, a characterisation, and a comparator, a positive (like the third abstraction) or negative (like the first two abstractions) abstraction can be formulated.

In some cases, our abstractions can easily be applied to general Petri nets. However, there does not exist, to our knowledge, a polynomial time algorithm to compute the abstractions on general nets. One can argue that computing the abstractions means analysing the behaviour in part or in full.

In the future, we plan to extend our framework with abstractions that also can deal with labelled WF-nets and with nets that are cyclic, for example by considering abstractions based on block-structured nets. For block-structured

nets (with a single entry and a single exit), the relations (like α and ∞) between transitions can often be derived efficiently from the block-structure.

Furthermore, we want to quantify the difference between two languages based on our framework, i.e., if language inclusion does not hold, what portion of the language in not included, and which part of the net is the main reason the language is not included. This can then be used as a quantification measure of inclusion between nets, and as a diagnostic result that can be used to align nets in such a way that language inclusion will hold.

References

1. Weidlich, M., Werf, J.M.E.M.v.d.: On Profiles and Footprints - Relational Semantics for Petri Nets. [22] 148–167
2. Glabbeek, R.J.v., Weijland, W.P.: Branching Time and Abstraction in Bisimulation Semantics. Journal of the ACM (JACM) (1996)
3. Kunze, M., Weidlich, M., Weske, M.: Behavioral Similarity - A Proper Metric. In Rinderle-Ma, S., Toumani, F., Wolf, K., eds.: BPM. Volume 6896 of Lecture Notes in Computer Science., Springer (2011) 166–181
4. Aalst, W.M.P.v.d., Adriansyah, A., Dongen, B.F.v.: Causal Nets: A Modeling Language Tailored towards Process Discovery. In Katoen, J.P., König, B., eds.: CONCUR. Volume 6901 of Lecture Notes in Computer Science., Springer (2011) 28–42
5. Schunselaar, D.M.M., Verbeek, H.M.W., Aalst, W.M.P.v.d., Reijers, H.A.: Creating Sound and Reversible Configurable Process Models Using CoSeNets. In Abramowicz, W., Kriksciuniene, D., Sakalauskas, V., eds.: BIS. Volume 117 of Lecture Notes in Business Information Processing., Springer (2012) 24–35
6. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
7. Abdulla, P.A., Chen, Y.F., Holík, L., Mayr, R., Vojnar, T.: When simulation meets antichains. In Esparza, J., Majumdar, R., eds.: TACAS. Volume 6015 of Lecture Notes in Computer Science., Springer (2010) 158–174
8. Dill, D.L., Hu, A.J., Wong-Toi, H.: Checking for language inclusion using simulation preorders. In Larsen, K.G., Skou, A., eds.: CAV. Volume 575 of Lecture Notes in Computer Science., Springer (1991) 255–265
9. Aalst, W.M.P.v.d., Weijters, A.J.M.M., Maruster, L.: Workflow Mining: Discovering Process Models from Event Logs. IEEE Transactions on Knowledge and Data Engineering **16**(9) (2004) 1128–1142
10. Badouel, E.: On the α-reconstructibility of workflow nets. [22] 128–147
11. Wen, L., van der Aalst, W.M.P., Wang, J., Sun, J.: Mining process models with non-free-choice constructs. Data Mining and Knowledge Discovery **15**(2) (2007) 145–180
12. Weidlich, M., Polyvyanyy, A., Mendling, J., Weske, M.: Causal Behavioural Profiles - Efficient Computation, Applications, and Evaluation. Fundamenta Informaticae **113**(3-4) (2011) 399–435
13. Weidlich, M., Polyvyanyy, A., Desai, N., Mendling, J., Weske, M.: Process compliance analysis based on behavioural profiles. Information Systems **36**(7) (2011) 1009–1025

14. Smirnov, S., Weidlich, M., Mendling, J.: Business process model abstraction based on behavioral profiles. In Maglio, P.P., Weske, M., Yang, J., Fantinato, M., eds.: ICSOC. Volume 6470 of Lecture Notes in Computer Science. (2010) 1–16
15. Dongen, B.F.v., Mendling, J., Aalst, W.M.P.v.d.: Structural Patterns for Soundness of Business Process Models. In: EDOC, IEEE Computer Society (2006) 116–128
16. Mendling, J., van Dongen, B.F., van der Aalst, W.M.P.: On the degree of behavioral similarity between business process models. In Nüttgens, M., Rump, F.J., Gadatsch, A., eds.: EPK. Volume 303 of CEUR Workshop Proceedings., CEUR-WS.org (2007) 39–58
17. Solé, M., Carmona, J.: A High-Level Strategy for C-net Discovery. In Brandt, J., Heljanko, K., eds.: ACSD, IEEE Computer Society (2012) 102–111
18. Desel, J.: Validation of process models by construction of process nets. In Aalst, W.M.P.v.d., Desel, J., Oberweis, A., eds.: BPM. Volume 1806 of Lecture Notes in Computer Science., Springer (2000) 110–128
19. Lautenbach, K.: Liveness in Petri Nets. Internal Report of the Gesellschaft für Mathematik und Datenverarbeitung. Bonn, Germany, ISF/75-02-1 (1975)
20. Desel, J., Esparza, J.: Free Choice Petri Nets (Cambridge Tracts in Theoretical Computer Science). Cambridge University Press (1995)
21. Aalst, W.M.P.v.d.: Verification of Workflow Nets. In Azéma, P., Balbo, G., eds.: ICATPN. Volume 1248 of Lecture Notes in Computer Science., Springer (1997) 407–426
22. Haddad, S., Pomello, L., eds.: Application and Theory of Petri Nets - 33rd International Conference, PETRI NETS 2012, Hamburg, Germany, June 25-29, 2012. Proceedings. In Haddad, S., Pomello, L., eds.: Petri Nets. Volume 7347 of Lecture Notes in Computer Science., Springer (2012)

Introducing Catch Arcs to Java Reference Nets

Lawrence Cabac, Michael Simon

University of Hamburg
Faculty of Mathematics, Informatics and Natural Sciences
Department of Informatics
{cabac,9simon}@informatik.uni-hamburg.de
http://www.informatik.uni-hamburg.de/TGI

Abstract. Modeling plays an important role during design and development of systems and processes. Petri nets allow for well-defined models that can be executed. For the implementation of these systems, however, still *normal* programming languages are used. In contrast, modeling languages – also if executable, such as Petri net formalisms – are not deemed fit for implementation. Besides the pragmatic power, one thing that modern programming languages offer and which Petri net formalisms are missing, is exception handling.
In this paper we present an approach that includes exception handling for Java reference nets. Our goal is to make the designed systems more robust and reliable. As a consequence, such executable models can be cleanly integrated into real execution environments.
Our approach provides the information of an exception being thrown to the level of modeling. We are thus able to model the exception handling explicitly within the model as it is done in many modern programming languages. This extension is conservative and does not alter the *normal* behavior of the model, leaving the Petri net semantics untouched. We discuss several possible extensions to our approach with respect to the modeling possibilities, the ease of implementation and their pragmatic usefulness.

Keywords: High-Level Petri Nets, Reference Nets, Exceptions, Catch Arcs, Renew

1 Introduction

High-level Petri net formalisms have often been extended by new primitives. This has been done mostly to improve modeling possibilities. The aim has been to increase comprehensiveness and compactness as in the case of *test arcs* and *inhibitor arcs* (see [2]) or *flexible arcs* (see [8]). We introduce a *catch arc* as a new primitive, in order to raise the tightness of integration with the inscription language. By this we improve the robustness of our executable models. With the new construct we are able to handle exceptions that might occur during execution of the model.

Renew[1] has been missing the possibility to handle exceptions on the model level, since it has been created. In this paper, we introduce a new kind of arc – the *catch arc* – which fills that gap. Its functionality is straight forward: if an exception is thrown during the execution of a transition inscription (in our case Java code) while firing, the exception object is put into the connected place. From this point, the exception can be treated in the model in an appropriate way. This is the reason why we call this arc a *catch arc*. The arc initiates a sequence of code that – in analogy to a catch statement in Java – follows the catch statement. If no exception occurs, the normal Petri net semantics is followed and the *catch arc* does not produce any token. We discuss in detail why this arc is suitable for handling exceptions and how the firing of a transition should be aborted on encountering an exception. Furthermore, we discuss a way to implement handling of exceptions that are thrown in one net without having to catch them all separately.

The structure of the paper is as follows: we introduce the *catch arc* and discuss its behavior by the means of Petri net modeling in Section 2. This is the central concept of this paper. Section 3 presents the complex process of the firing of a transition in Renew. It shows the inability to fully abort a transition firing and the consequent limitations on extensions of the Java reference net formalism. These limitations motivate the *try arc* as a solution. Section 4 introduces the *try arc* and discusses how a transition firing can be reverted after an exception. Section 5 presents ways to isolate the tokens involved in an erroneous firing for exception handling. Using the *catch arc* alone is compared to using it in combination with a *try arc*. Section 6 discusses the (conservative) extension of the *catch arc* by an expression whose result gets returned on catching an exception. In Section 7 we extend the notion of exception handling from transitions to net instances.

2 The Catch Arc

We extend reference nets [6] by adding *catch arcs* that put an exception into a place as object reference, if one occurred. Avoiding uncaught exceptions in *action inscriptions* is very complicated: the modeler would need to make sure that all *action inscriptions* only call Java methods that do not throw exceptions. This is not feasible, as the standard Java methods and well-written Java classes rely heavily on throwing exceptions. We do not want to worry about exceptions in every *action inscription*, but would rather prefer to have a simple way to handle those in general without having to consider each possible error case.

Up until now, the possibility of an exception being thrown was simply not covered by the reference net formalism. Thus, such an event was outside the scope of well-defined behavior of the simulator. On encountering an exception, the ingoing arcs would consume the bound tokens, but no token would be put out by the outgoing arcs. The simulator would log the incident, but ignore it

[1] Renew: The Reference Net Workshop [7], `http://www.renew.de`

in any other regard. By adding the possibility to catch and handle exceptions we pull this behavior into the model level. The behavior of the Petri net models itself is not changed as long as no exception occurs.

Unfortunately, there is no easy and clean way to completely reset a transition that has already begun to fire, as we will discuss in Section 4.

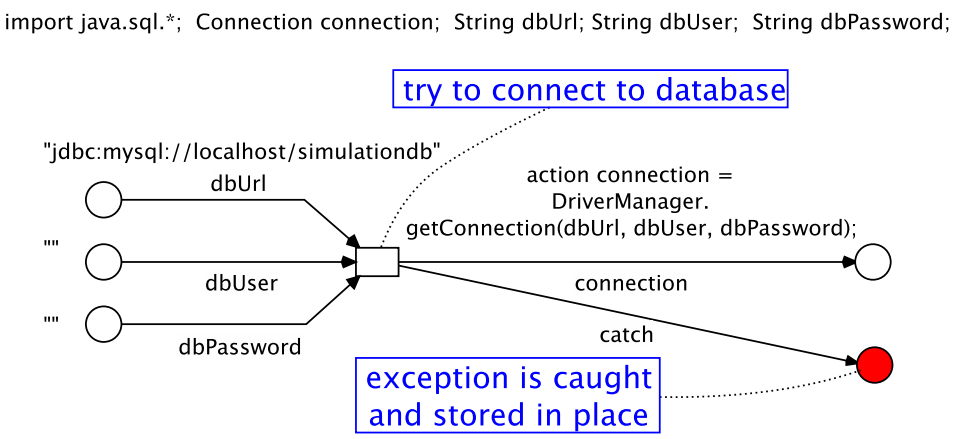


Fig. 1. Usage example.

Fig. 1 shows a typical use-case where *catch arcs* enable the modeler to express safe code with a few simple net elements, instead of modeling complex structures or exporting the error handling functionality to Java helper code. The *catch arc* can be identified by the *catch* inscription. The tokens in the places on the left are given, but they may also be dynamically derived by other net segments, for example through a user input dialogue. It is very difficult to ensure that all input tokens to the transition are valid. In this case, there are also external factors that determine the outcome of the *action inscription* as *dbUrl* is the URL of a database to open, which might not be available. In a scenario like this, catching exceptions is unavoidable to have a stable system. As already mentioned above, usually the handling of the possibly thrown exceptions is implemented in Java helper code that encapsulates the opening of the connection and catches exceptions on that level. With the *catch arc* we are not only able to pull the exception handling up to the model level, we also reduce the implementation of wrapper code. In fact, we are able to treat the exceptions as first order concepts within our models.

The reference net depicted in Fig. 2 illustrates the operational semantics of *catch arcs*. The transition holds an *action inscription* which converts a *String* into an *Integer* object. Naturally, this operation throws an exception, if the conversion cannot be achieved due to an invalid argument. Fig. 3 shows the result of executing the net in Fig. 2. The *String* `6` can be converted to an *Integer* which is put into **output**, while the attempt to convert `foo` throws *NumberFormatException* which is put out by the *catch arc*.

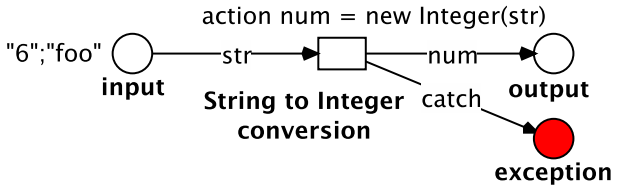


Fig. 2. Example of a safe *String* to *Integer* conversion with a *catch arc*.

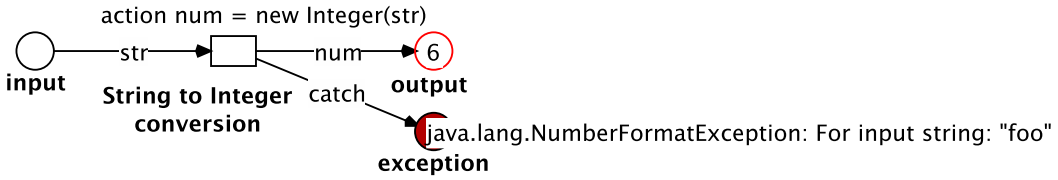
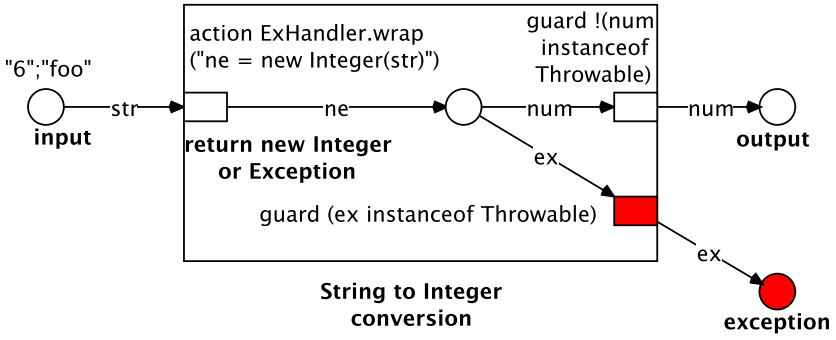


Fig. 3. Executed instance of the net in Fig. 2.

Fig. 4 is a conceptual model showing the execution semantics of the *catch arc*, simulated by a reference net model and some possible Java code. This model illustrates the implementation of the *catch arc*.



The *ExHandler.wrap(String)* method returns the same result as the expression given as *String*, if no exception is thrown. Otherwise, it returns the thrown exception.

Fig. 4. Net without *catch arcs* behaviorally equivalent to Fig. 2.

The transition **String to Integer conversion** in Fig. 4 is a refinement of the one in Fig. 2. The *ExHandler.wrap(String)* method is not implemented, but serves to illustrate how a wrapper that catches any error and returns it, or the result would be used to emulate the behavior of *catch arcs*.

The *catch arc* itself serves as a clear identifier of where the control flow for exception handling starts. All transitions that are dependent on the exception output token, are part of the exception handling control flow. By directing a

catch arc to a place that is only used for exceptions, it is easy to separate the control flow. Even though it is made easy, it is up to the modeler to separate the exception handling parts of a reference net from the rest, like any other software architecture design decisions. This is analogous to the catch block of a try-catch-construct in a textual programming language: what is done inside is up to the programmer. There is no difference between code that can be executed inside the catch block or outside. Still it is good practise to separate exception handling code from other code. The place to which a *catch arc* leads should be regarded in the same way as a catch block in a textual programming language.

We implemented an extension to the *catch arc*, allowing it to be accompanied by an expression. This is further discussed in Section 6.

Another possible improvement of the *catch arc* would be the ability to declare the class of the exception to catch. This would deepen the analogy with the common *try-catch* programming language construct and would simplify the implementation of different ways to handle different exceptions. On the other hand, this behavior is easy to achieve with guards that check the exception types. These would be inscribed to the transitions which handle the exceptions. Because of the flexibility that guard inscriptions offer, there is no need to catch only some types of exceptions. This refinement of the *catch arc* could be implemented by an expression that is given as an argument to the *catch* inscription. It has to evaluate to a (sub)class of *java.lang.Throwable* (the exception rootclass).

A *finally statement* is not needed in the extended Java reference net formalism. In Java it represents a section of code that is inserted into the control flow of the erroneous as well as the normal execution. It is executed in all cases. In the Java reference net formalism the control flow is explicitly modeled. The *catch arcs* start the control flow of the erroneous case and the normal output arcs start the control flow of the normal case. In Fig. 2 there is either one token put out to **output**, or to **exception**. To introduce some code that gets executed as a *finally statement*, we introduce a transition that is fired for every token in both places. This can be done using only classic Java reference net elements.

3 Firing a Transition in Renew

Table 1 provides an overview of the different steps of Renew's internal algorithm which fires transitions (compare with [6, Sec. 14.7]). We did not change this algorithm in any arc implementations presented in this paper. Before a transition is fired, a valid binding has to be found in phase 0. During the actual firing, the *early executables* are applied first. In phase 1 the *early executables* which mostly represent the ingoing arcs and *test arcs*, need to lock the associated places. This ensures that the tokens can not be taken by another transition firing. Then they verify that they can still be applied, as concurrent changes to the net instance state could have invalidated the found binding (phase 2). After this, the *early executables* can finally be executed (phase 3). Then the locks are unlocked again (phase 4) and the firing of the transition is already reported as successful, since the firing can not be aborted anymore (phase 5). In the end, the *late executables*

0	Binding search (before firing)
1	Lock *early executables*
2	Verify possibility of applying *early executables*
3	Execute *early executables*
4	Unlock *early executables*
5	Report success
6	Execute *late executables*

Table 1. Order of steps when successfully firing a transition in Renew.

can be executed (phase 6). They mostly represent outgoing arcs and *action inscriptions*.

If an exception is thrown while searching for a valid binding of a transition, the corresponding search branch is discarded and the transition will never fire that binding. Phase 0 from Table 1 is never left. If an *early executable* leads to an exception in phase 2 or 3, the firing of the transition is aborted. In a case where Java code is in an *action inscription*, on the other hand, it is not evaluated in the binding search (phase 0), but in a *late executable* (phase 6) after a valid binding was found. The firing can not be reverted and the exception has to be thrown.

Early executables are designed to model actions that can be aborted and reverted. They are executed first, so that as much error cases as possible lead to a rollback. *Late executables*, on the other hand, model actions that can not be reverted. Semantically these can not fail, but they can in practice, if they are not modeled safely, e.g., if an action inscription throws exceptions. Safe modeling is very complex and almost never achieved in practice. Thus, it might be tempting to move as much unsafe actions as possible into the *early executables*, so that the classic reference net semantics are never violated. However, a transition firing attempt should only be reverted, if the current binding can not be found again. This is possible, if the state of the net instance has evolved after the binding search. An example is a token that is consumed by another thread between the binding search and the current attempt to fire. In this case, the *early executable* representing the input arc will fail and cause a rollback. On the other hand, if there would be a case, where the current binding can be found again, the Renew simulator would attempt to fire it in an infinite loop. Since in the existing Renew implementation without any new arc implementations all changes of this firing are reverted, this firing itself can not change the net instance's state.

For a modeler who wants to use *catch arcs*, only *action inscriptions* are of concern. The exceptions thrown by all other inscribed Java code are already dealt with in the binding search. Renew does not fire bindings that lead to exceptions in the binding search.

4 Resetting a Transition Firing after an Exception

It is possible to avoid infinite loops when resetting a transition. One way to achieve this, is to have a special token which is consumed if an exception occurs. If only one such token existed, this binding can not be found again until another transition returns it.

We created an experimental implementation of this exception input arc, called the *try arc*. The existence of a *try arc* does not change the behavior of the classic *late executables* after an exception has been thrown. This includes that no tokens are put out. Thereafter, the *early executables* are rolled back, as if the exception had been thrown inside one of them. The *try arc* does nothing when executed normally, but consumes the bound token when rolled back. This behavior is the reverse of the normal input arcs. With this extension of the simulator the *late executables* behave like the *early executables*, because we have implemented a way to revert them.

Unfortunately, the possibilities of the *try arc* have their limits. Action inscriptions with side effects are not handled correctly. In this case, the transition would look reverted, but the side effects could still have occurred.

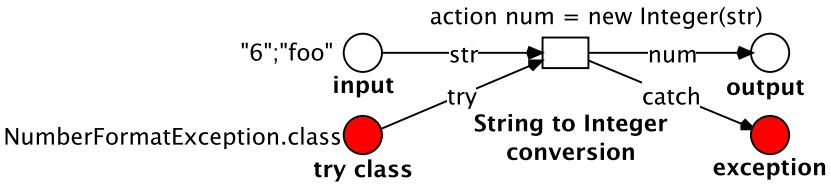


Fig. 5. Net in Fig. 2 extended by a *try arc*.

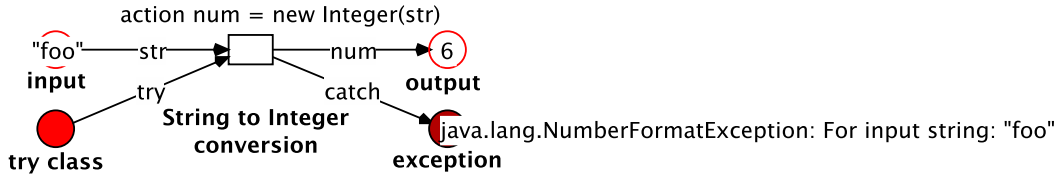


Fig. 6. One possible executed instance of the net in Fig. 5.

Fig. 5 extends the net in Fig. 2 by a *try arc*. Fig. 6 shows one of two possible executions of this net. In this execution the transition **String to Integer conversion** was first fired with the string `6` as input. Then `foo` was tried and the exception has been put out by the *catch arc*. The corresponding exception class token has been consumed and the rest of the transition firing has been reverted.

For this reason, the `foo` token is put back. The other possible execution of this net is that `foo` is tried first. In that case, the exception class is removed before `6` can be tried.

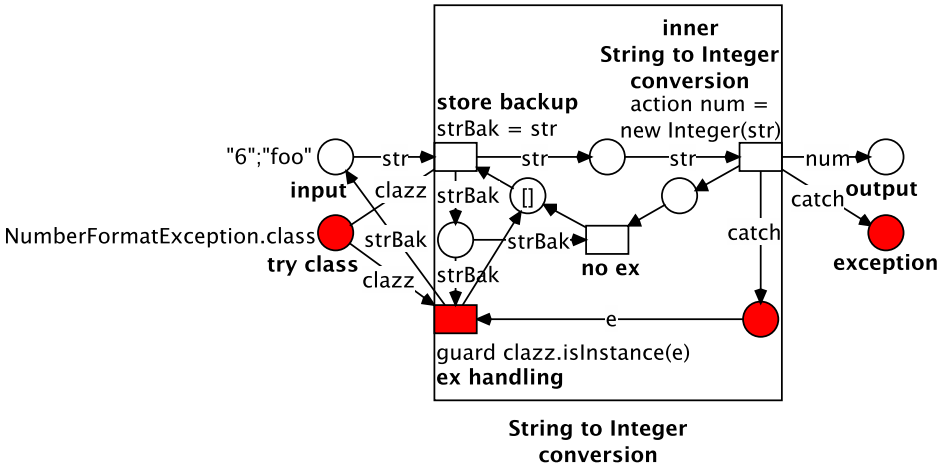


Fig. 7. Net without *try arcs* behaviorally equivalent to Fig. 5.

In Fig. 7 you can see that the behavior of *try arcs* can be emulated in the classic Java reference net formalism, extended only by *catch arcs*. In contrast to the net in Fig. 4, which was used to illustrate the semantics of *catch arcs*, we do not need any special, difficult to implement wrapper functionality in the Java inscriptions. In fact, this net is executable. First, one of the two strings is taken from **input** by **store backup** and passed on as *str*. The same string is put to the place below as *strBak*. The places in the triangle between **store backup**, **inner String to Integer conversion** and **no ex** can only hold one token in total at any time. The transition **inner String to Integer conversion** takes the string, attempts to convert it to an integer and returns the integer to **output**, or the thrown exception to **exception**. On the inside, it either returns a generic token to the white place in the lower left, or the *catch arc* returns another Java reference of the exception to the lower red place. Depending on this outcome, either **no ex** fires, consumes the just processed input string as *strBak* and returns the generic token to the place in the triangle, where it was originally, or **ex handling** fires and puts the input string back to **input**. In this process, it consumes the class token in **try class**. If this token is no longer present, **store backup** can not fire, because of the *test arc* to **try class**. This models the inability of **String to Integer conversion** in Fig. 5 to fire, if the *try arc* can not bind to an exception class token. Like **no ex**, **ex handling** also puts back the generic token. This would enable **store backup** to fire again, if **try class** contained another exception class token.

The model in Fig. 7 is only behaviorally equivalent to Fig. 5, if there are only Java references to one exception class in **try class**. It is more flexible, if we want

to use more than one class of exceptions as possible token for the *try arc*. In our current experimental *try arc* implementation, the *try arc* is first bound to an exception class token, before the transition starts firing (in phase 0 of Table 1). In every firing the *try arc* is bound to only one exception class. Thus, the *try arc* may not reset the transition, even if the class of the thrown exception exists as a token. One scenario, where this problem occurs, is a modification of Fig. 5 with a number of exception classes in **try class**. In Fig. 7, on the other hand, the **ex handling** transition can try every class in **try class**. The RENEW simulator would need to be changed considerably to implement this behavior for *try arcs*.

5 Handling Exceptions

In this chapter, we compare using the *catch arc* alone and along with a *try arc* to model exception handling. We show that the *try arc* does not provide a real advantage in this situation.

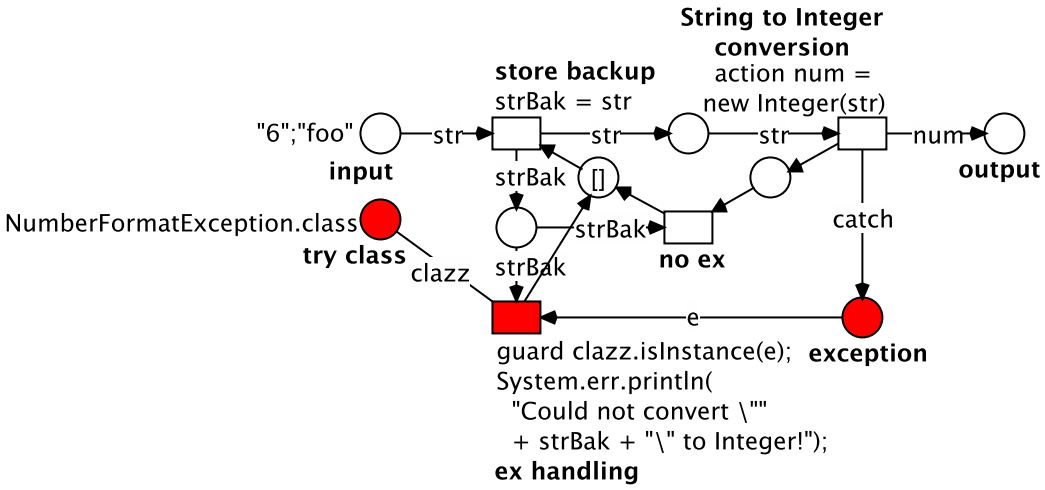


Fig. 8. Net in Fig. 7 modified to print out error.

Fig. 7 illustrates the behavior of the *try arcs* in Fig. 5. The fact that it is very complex, suggests that *try arcs* simplify error handling greatly. However, when handling the exception, one would normally want to ensure that it does not occur again. For this purpose, the involved tokens usually need to be identified and separated from the rest for special treatment. An example would be to generate feedback to calling code or the user. In Fig. 7 there is already a transition for exception handling present: **ex handling**. In order to generate feedback, instead of resetting the firing, we need to remove the output arc from **ex handling** to **input**, so no token is put back. We also need to change the arcs from **try class** to prevent the class token from being consumed. The Java code generating the output can be inscribed to **ex handling**. This is demonstrated in Fig. 8.

If we change the net in Fig. 5 to give feedback, we have more work to do. A possible implementation is presented in Fig. 9, which is very similar to Fig. 8. In both models it is important to reconstruct which input string has induced an exception. For this purpose, the capacity of the input place is restricted to 1. In Fig. 8 there can only be one token in the triangle between **store backup**, **inner String to Integer conversion** and **no ex**. In Fig. 9 there can only be one token in the input place and the place below. In contrast to Fig. 8, there is no need to store another Java reference of the input token, because this token is returned if an exception occurs. For this reason, a transition such as **no ex**, which is fired if no exception has occurred, is also not needed. Another difference is that the exception class token in **try class** gets consumed in the event of an exception and has to be replaced by **ex handling** when the exception is handled.

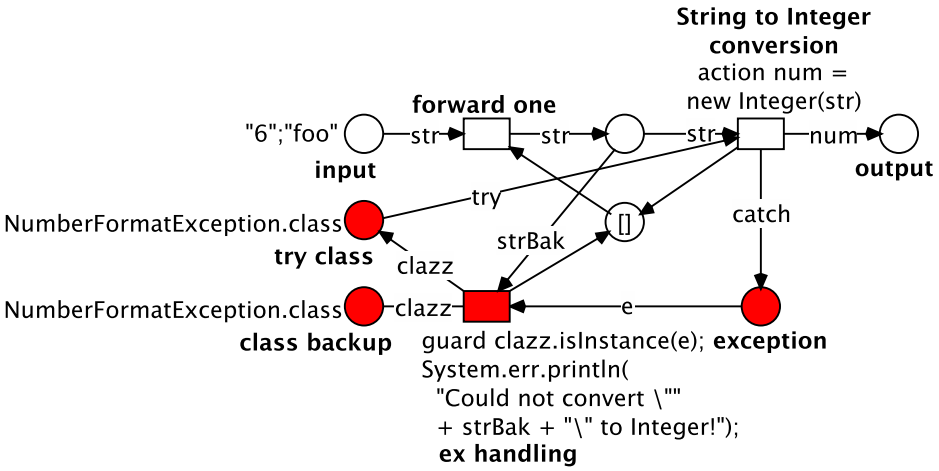


Fig. 9. Net with *try arc* modified to print out error (behaviorally equivalent to Fig. 8).

6 Catch Arc with an Expression

In order to further the expressiveness and thus simplify the scenarios, in which we would like to retain the tokens involved in an erroneous transition firing, we extended the *catch arc* by an expression whose result is produced alongside the exception. Similar to input arc inscriptions, this expression has to be fully bound before firing, and can thus not be dependent on any *action inscriptions*.

Fig. 10 emulates the *try arc* from Fig. 5. Unlike the model in Fig. 7, there is no need to limit the capacity of the input place of the transition, which can throw an exception. The involved input token can be reconstructed from the result of the *catch arc*'s expression, which gets returned to **exception** as tuple alongside the exception.

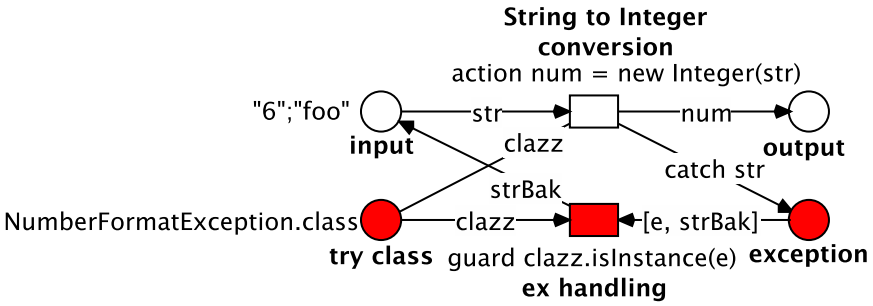


Fig. 10. Net without *try arcs*, but with a *catch arc* with an expression, behaviorally equivalent to Fig. 5 and 7.

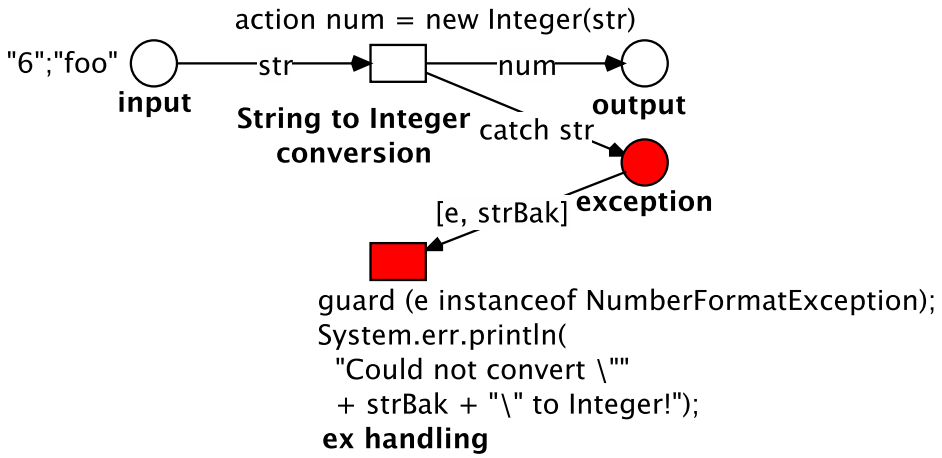


Fig. 11. Net in Fig. 10 modified to print out error (behaviorally equivalent to Fig. 8).

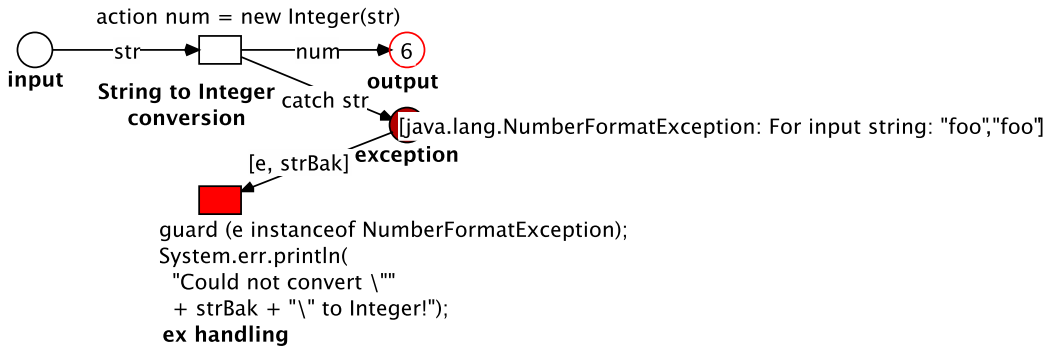


Fig. 12. Instance of the net in Fig. 11 after firing.

Fig. 11 prints out an error message, exactly like the nets in Fig. 8 and 9. The token put out by the *catch arc* can be seen in Fig. 12, which shows an executed instance of the former net after firing the transition **String to Integer conversion** twice. It is a 2-tuple that consists of the thrown exception and the result of the expression.

ex handling is a transition that fires after **String to Integer conversion** in the event of an exception. Because the transition **ex handling** takes in the string together with the exception, it behaves as if it had the same preset as **String to Integer conversion** and took the same token. If one wanted to fire a exception handling transition with all the same input token as the original transition, one could accompany the *catch arc* with a tuple of all input arc inscriptions.

7 Uncaught Exceptions and Exception Propagation

In classic programming languages exceptions propagate outward and escape all code sections, until they are finally caught and handled. If they are not handled in program code, the program crashes on the occurrence of an exception. In Renew's classic Java reference net formalism exceptions are logged, but otherwise ignored. All ingoing tokens are consumed and no tokens are written out. Since dependency is explicitly modeled in reference nets, it is reasonable to allow those parts of the simulation that are not dependent on the part where the exception occurred, to continue to run.

However, a concept of propagating exceptions upwards through a net hierarchy, can be realised. One step of this propagation can be achieved by a transition that binds exception tokens to an uplink, so another net instance which knows the current instance, can extract the exception. Binding to a downlink would also be possible, but then the exception handling net instance must be known to the current instance. (This would more accurately be described as *propagating downwards.*)

It is also possible to have a specific uplink channel to pass on exceptions that are not caught locally (for example *:catch(e)*). This can be done through a normal net channel that gets bound to all these unhandled exceptions. To achieve this, there can be one place in every net for unhandled exceptions and one transition that binds every token of this place to the uplink of the channel. All transitions without a *catch arc* receive one to this place. The only exception from this rule is at a transition with an uplink. The exceptions thrown by a firing that involves this transition, can be caught at the transition with the downlink. The place and the transition for unhandled exceptions could be hidden to the modeler, so they are only accessible through the channel. The *catch arcs* could be created automatically, where there does not already exist one in the model. If the modeler would like to manually mark an exception as unhandled (maybe because the class is not expected), she can add an uplink transition to the channel herself.

8 Related Work

There are many attempts to model the behavior of exceptions with various modeling languages. This kind of exception modeling includes concepts for expressing the behavior of exceptions that occur in systems. We call these exceptions model intrinsic. Examples are the *exception handler* in current versions of UML (Unified Modeling Language, current version 2.4.1, see [9, Sec. 12.3.23]) and the attempts to include exception handling to (hierarchical) Petri net formalisms (cf. [3] and [4]). While the above mentioned examples model the behavior of exceptions or errors, we strive for the treatment of exceptions that occur during execution of these models. We call these (execution) extrinsic exceptions.

Jannach and Gut [5] discuss the possibilities of exception / error handling in current modeling languages in detail and also point out the difference between modeled exception behavior (intrinsic) and exception handling during model execution (extrinsic). On the side of exception handling of executable models they discuss (among others) the possibilities in workflow execution (e.g., offered by YAWL, compare also with [1]) and WS-BPEL. The focus lies in both cases on the cancellation of processes / workflows and the compensation of undesired results. Cancellation of processes (or process regions) as by the *cancel arc* in YAWL is tightly related to *clear arcs* (*reset arcs*) in Petri net formalisms (i.e. Reset Nets). However, the cancellation / exception trigger comes from within the model – a cancel task has to be modeled explicitly and triggered. In our approach we tighten the integration of the executed model and the underlying expression language.

9 Conclusion and Future Works

We presented an approach and a first implementation of the exception handling as an extension of the Java reference net formalism and the Renew simulation engine. The *catch arc* behavior can be expressed by a combination of net refinement and code wrapper implementation. Our first approach which has been implemented within Renew, constitutes already a powerful and also conservative extension to the execution semantics of Java reference nets. The *try arc* was discussed as a possible further extension. It was introduced and motivated by the idea of resetting a erroneous transition firing. We have shown that it does not add much to the expressiveness of the formalism in the context of exception handling. We do not plan to incorporate it in our practical implementation. The possibility of adding an expression to a *catch arc* whose result is returned alongside the exception, is a conservative extension of the original *catch arc* concept. In difference to the *try arc*, it greatly adds to the expressive power. It can be used to provide relevant details of the binding, in which a firing of a transition failed and allows for concise net models for detailed exception handling. This concept can also effectively emulate the resetting of an erroneous transition firing and thus, supersedes the *try arc* concept. Especially the questions of exception propagation in reference net systems and the adequate compensation modeling is, however, not satisfyingly resolved and needs to be further investigated.

References

1. Michael James Adams. *Facilitating dynamic flexibility and exception handling for workflows*. PhD thesis, Queensland University of Technology, 2007.
2. Søren Christensen and Niels Damgaard Hansen. Coloured petri nets extended with place capacities, test arcs and inhibitor arcs. In Marco Ajmone Marsan, editor, *Application and Theory of Petri Nets*, volume 691 of *Lecture Notes in Computer Science*, pages 186–205. Springer, 1993.
3. W.L.A. de Oliveira, N. Marranghello, and F. Damiani. Exception handling with petri net for digital systems. In *Integrated Circuits and Systems Design, 2002. Proceedings. 15th Symposium on*, pages 229–234, 2002.
4. M. Doligalski and M. Adamski. Exceptions handling in hierarchical petri net based specification for logic controllers. In *Systems Engineering (ICSEng), 2011 21st International Conference on*, pages 459–460, 2011.
5. Dietmar Jannach and Alexander Gut. Exception handling in web service processes. In Roland Kaschek and Lois M. L. Delcambre, editors, *The Evolution of Conceptual Modeling*, volume 6520 of *Lecture Notes in Computer Science*, pages 225–253. Springer, 2008.
6. Olaf Kummer. *Referenznetze*. Logos Verlag, Berlin, 2002.
7. Olaf Kummer, Frank Wienberg, Michael Duvigneau, and Lawrence Cabac. Renew – the Reference Net Workshop. Available at: `http://www.renew.de/`, March 2012. Release 2.3.
8. Wolfgang Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets*. Springer-Verlag New York, October 1997.
9. UML. Unified modeling language: Superstructure. `http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF`, August 2011.

A System Performance in Presence of Faults Modeling Framework Using AADL and GSPNs

Belhassen MAZIGH[1] and Kais BEN FADHEL[1]

Department of Computer Science, Faculty of Science of Monastir,
Avenue of the environment, 5019, Monastir - Tunisia
belhassen.mazigh@gmail.com

Abstract. The increasing complexity of new-generation systems which take into account interactions between hardware and software components, particularly the fault-tolerant systems, raises major preoccupations in various critical application domains.These preoccupations concern principally the modeling and analysis requirements of these systems.Thus, designers are interested in the verification of critical proprieties and particularly the Performance and Dependability analysis.
In this paper, we present an approach for modeling and analyzing systems with hardware and software components in the presence of faults: an approach based on Architecture Analysis and Design Language (AADL) and Generalized Stochastic Petri Nets (GSPN). This approach starts with the description of the system architecture in AADL. This description is enhanced by the use of two annexes, the existing Error Model Annex and the Activity Model Annex (proposed Annex). By applying existing transformation rules of GSPN models, we build two models: GSPNs Dependability and Performance models. Finally, we apply our composition algorithm, to obtain a global GSPN model allowing to analyze Dependability and Performance measurements.

1 Introduction

The quantity and complexity of systems continues to rise and generally the cost of manufacturing these systems is very high. To this end, many modeling and analysis approaches are more and more used in industry with the aim to control this complexity since the design phase. These approaches must be accompanied by languages and tools. An explicit approach presents the building of a GSPN of a complex system from the GSPNs of its components, taking into account the interactions between the components, is presented in [1]. These approaches are referred to as block modeling approach and incremental approach respectively.
AADL is among the languages having a growing interest in industry-critical systems. This language has been standardized by the "International Society of Automotive Engineers" (SAE) in 2004 [2, 3] , to facilitate the design and analysis of complex systems, critical, real-time in areas such as avionics, automotive and aerospace [4]. It provides a standardized graphical and textual notation to describe the hardware and software architectures. It is designed to be extensible

in order to adapt and analyze architectures execution that the core language does not fully support. The extensions may take the form of new properties and notations specific to analysis that may be associated with the architectural description in the form of annexes. Among these approaches, the one proposed in [5] allows specialists in AADL to obtain Dependability measures, based on a formal approach. This approach aims to control the construction and validation of Dependability models in the form of GSPN. But in reality the designers want to have a final model of the system that allows them to analyze Dependability and Performance which take into account functional and dysfunctional aspects of the system. In this paper we propose an extension to this approach so that the final model of the system allows us to analyze the attributes of Dependability and Performance measures. The outline of the paper is as follows. In Section 2, we define the AADL concepts. Then we present our approach in Section 3 and its application on a simple example in Section 4. We conclude in section 5.

2 AADL concepts

The AADL core language is used to analyze the impact of different architectural choices on the properties of the system [6] and [7]. An architecture specification in AADL describes how components are combined into subsystems and how they interact. Architectures are described hierarchically. Components are the basic bricks of AADL architectures. They are grouped into three categories: 1) software (process, subprogram, data, and thread), 2) hardware (processor, memory, device, bus) and 3) composite (system). AADL components may be composed of sub-components and interconnected through features such as ports. These features specify how the components are interfaced with each other. Each AADL system component has two levels of description: the component type and the component implementation. The type describes how the environment sees that component, i.e., its properties and features. Examples of features are **in** and **out** port that represent access points to the component. One or more component implementations may be associated with the same component type.
As mentioned in the introduction, AADL is designed to be extensible in order to adapt and analyze architectures execution the core language that does not fully support. The Error Model Annex is a standardized annex [3] that completes description of the capabilities of the core language AADL, providing a textual syntax with a precise semantics to be used to describe the characteristics of Dependability related to AADL architectural models. AADL error models are defined in libraries and can be associated with software and hardware components as well the connection between them. When an error model is associated with a component, it is possible to customize it by setting component-specific values for the arrival rate or the probability of occurrence of error events and error propagation declared in the error model.
In the same way as for AADL components, the error model has two levels of description: the error model type and the error model implementation. The error model type declares a set of error states, error events and error propagation

circulating through the connections and links between architecture model. In addition, the user can declare properties of type **Guard** to control the propagation. The error model implementation declares states transitions between states, triggered by events and propagation declared in the error model type [8]. Note that **in** and **out** features identify respectively incoming propagation and outgoing propagation. An **out** propagation occurs in an error model source with property of occurrence specified by the user. The error model source sends the propagation through all ports and connections of the component to which error model is associated. As a result, **out** propagation arrives at one or more error models associated with receptor components. If an error model receiver, declares **in** propagation with the same name as the **out** propagation received, the **in** propagation can influence its behavior.

3 Modeling approach

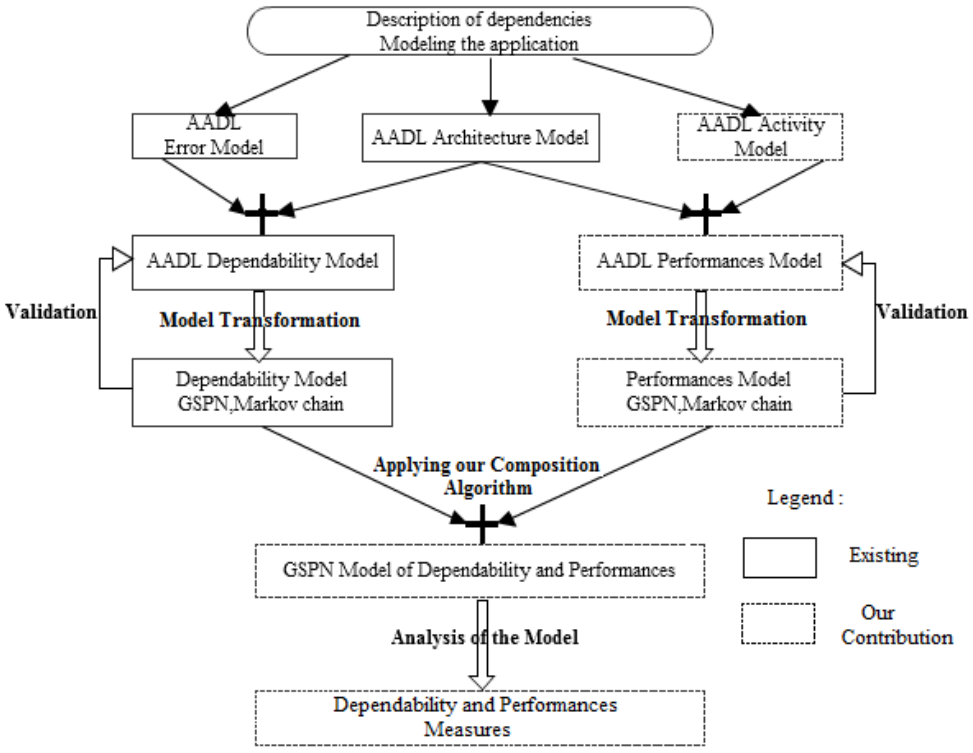


Fig. 1. Proposed Approach

We can describe our method, presented in Figure 1, into five main steps:

1. The first step focuses on the modeling of the system architecture in AADL.

2. The second step concentrates on building an AADL Dependability Model and a AADL Performance Model:
   - Building Dependability model is done by the association of AADL Error models to the components of AADL Architectural Model, see [5] for more details.
   - Building performance model is done by the association of our AADL Activity models to the components of AADL Architectural Model. An Activity Model is similar to the Error Model, this model works as an Error Model, but the state change is performed by passing from a reliable state to another with integration of performance metrics associated to the properties that must be defined in AADL language. Syntactically it is inspired by the standard Error Model. Activity models are devoted to describe the components activities.
3. The third step is to build two GSPN models, of Dependability and Performance, from two AADL models using the transformation rules presented in [5].
4. The fourth step is dedicated to the application of our composition algorithm. This algorithm receives as an input two GSPN models, a GSPN model of Dependability and a GSPN model of Performance, each model is composed of sub-networks of components and sub-networks of dependencies. We obtain a global GSPN model which allows to analyze Dependability and Performance measurements for hardware and software systems in the presence of faults.
5. The fifth step is dedicated to analyzing the global GSPN model to obtain measures of Dependability and Performance. This last step is entirely based on classical algorithms for processing GSPN models and it is not the subject of this work, and therefore will not be detailed here.

The next section presents the application of our approach to a simple example.

4 Application of our approach

To illustrate our approach, we use a simple system constituted by a processor PR which executes a user process P. The processor allocation is made according to the following policy: we define a quantum of time (e.g. q) for the processor PR. A process P sends an allocation request for the processor PR. If the processor is free then it accepts the request, the process pass to the execution state. The process can pass into a blocking state (blocked) if it expects other resource (e.g. end of an input output). If the execution time is smaller than the quantum then the process completes its task and passes to termination state otherwise the processor interrupt the execution of process, so that another process could be executed (the processor is retained for the current process until the end of quantum). When the process passes into the blocking state the processor can be allocated to another process. The processor is not necessarily free. The process then moves to the ready state. The ready state is the standby state of the processor. It is clear

that there is a structural dependence between the processor PR and process P. Defects in materials could be propagated and influence behavior of software associated with it.
We will first establish an AADL model of Dependability, with the corresponding transformation steps into GSPN and we do the same thing to develop an AADL Performance model. Finally, we apply our composition algorithm to obtain a global Performance model in presence of software and hardware faults.

4.1 Construction of Dependability model

```
Error Model comp_hard
   features
   Error_Free : initial error state;
   Activation : error state;
   Erroneous : error state;
   Failed : error state;
   Fault : error event {Occurrence => poisson λ1h};
   Temp_Fault : error event {Occurrence => poisson λ2h};
   Perm_Fault : error event {Occurrence => poisson λ3h};
   Restart : error event {Occurrence => poisson λ4h};
   Recover : error event {Occurrence => poisson λ5h};
End comp_hard;
Error Model implementation comp_hard.general
   transitions
   Error_Free -[Fault]-> Activation;
   Activation -[Perm_Fault]-> Failed;
   Activation -[Temp_Fault]-> Erroneous;
   Failed -[Restart]-> Error_Free;
   Erroneous -[Recover]-> Error_Free;
End comp_hard.general;
```

Fig. 2. Error Model of hardware component

We propose generic error models (without propagation) for the hardware and software components (Figure 2 and 3) inspired by the works [8], [9], [10] and [11]. These two models are respectively associated with the implementation of the processor PR and the process P components. Because the fact that the process P is running on the processor PR, the errors in the processor (hardware faults) can affect the process executed as follows:

- If the fault is temporary, it can transmit errors in the process. The error sent by the processor leads the process in state relating to the activation of fault (state '*Detect_ERR*').
- If the fault is permanent, this failure has two consequences: stopping the software components and synchronizing the restoration actions since the

relaunch of software components is conditioned by the end of the repair of hardware component on which they were executed.

```
Error Model comp_soft
   Features
   Error_Free : initial error state;
   Detect_ERR : error state;
   ERR_ND : error state;
   ERR_D : error state;
   Erroneous : error state;
   Failed : error state;
   Fault : error event {Occurrence => poisson λ1s};
   NonDetect : error event {Occurrence => poisson λ2s};
   Eliminate : error event {Occurrence => poisson λ3s};
   PerceiveFail : error event {Occurrence => poisson λ4s};
   Detect : error event {Occurrence => poisson λ5s};
   Temp_Fault : error event {Occurrence => poisson λ6s};
   Perm_Fault : error event {Occurrence => poisson λ7s};
   Restart : error event {Occurrence => poisson λ8s};
   Recover : error event {Occurrence => poisson λ9s};
End comp_soft;
Error Model implementation comp_soft.general
   transitions
   Error_Free -[Fault]-> Detect_ERR;
   Detect_ERR -[NonDetect]-> ERR_ND;
   Detect_ERR -[Detect]-> ERR_D;
   ERR_ND -[Eliminate]-> Error_Free;
   ERR_ND -[PerceiveFail]-> Failed;
   ERR_D -[Temp_Fault]-> Erroneous;
   ERR_D -[Perm_Fault]-> Failed;
   Failed -[Restart]-> Error_Free;
   Erroneous -[Recover]-> Error_Free;
End comp_soft.general;
```

Fig. 3. Error Model of software component

Figure 4 shows only what is added to the error model associated with the processor in order to describe the structural dependency after a recovery failure. The error model type for processors, *comp_hard*, is completed with lines R_O^1 and R_O^2 in order to include 'out' error propagation declarations ('PR_Failed','PR_Ok'), 'PR_Failed' causes the processes failures while 'PR_Ok' is used to synchronize the repair of the processor with the restart of the process. The error model implementation, *comp_hard.general*, takes into account the sender side of the recovery dependency, it declares one transition triggered by each of the two newly introduced 'out' propagation (see lines R_O^{11} and R_O^{22} of Figure 4). When one of the 'out' propagation occurs, the processor remains in the same state and the propagation remains visible until the processor leaves this state. Figure 5 shows what is added to the error model associated with a process in order to describe the structural dependency. The error model type, $comp\_soft$, is com-

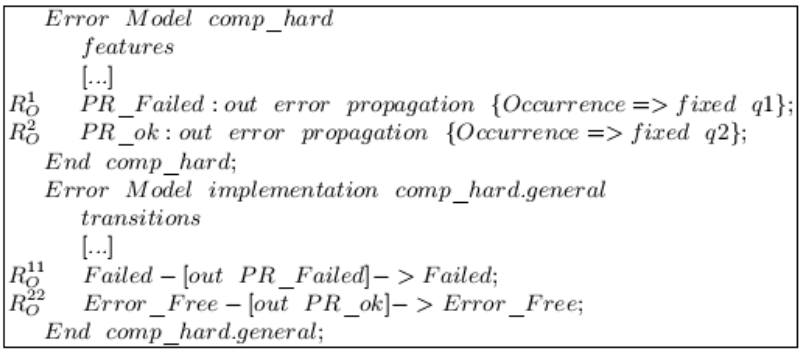


Fig. 4. Error Model for processor component

pleted with lines $L_0, L_1 and\ L_2$. Line L_0 declares an additional state in which the process is allowed to restart and Lines L_1 and L_2 declares 'in' propagation. The error model implementation, $comp\_soft.generale$, takes into account the recipient side of the structural dependency by declaring five transitions triggered by the 'in' propagation 'PR_Failed' (see lines $L_1^1, L_1^2, L_1^3, L_1^4$ and L_1^5 of Figure 5) and leading the process from each state (other than '$Failed$') to the '$Failed$' state. The process is authorized to move from the '$Failed$' state to '$InRestart$' state only when it receives the 'PR_Ok' propagation (see line L_2^1 of Figure 5). Since the recovery procedure is now engaged by the '$InRestart$' state , the AADL transition ($Failed - [Restart] - > Error\_Free$) will be replaced by the transition ($Failed - [PR\_Ok] - > InRestart$, see line R).

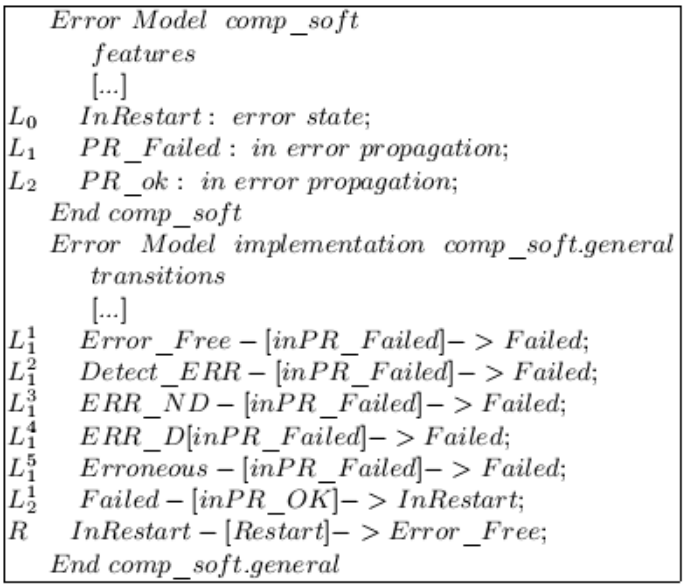


Fig. 5. Error Model for process component

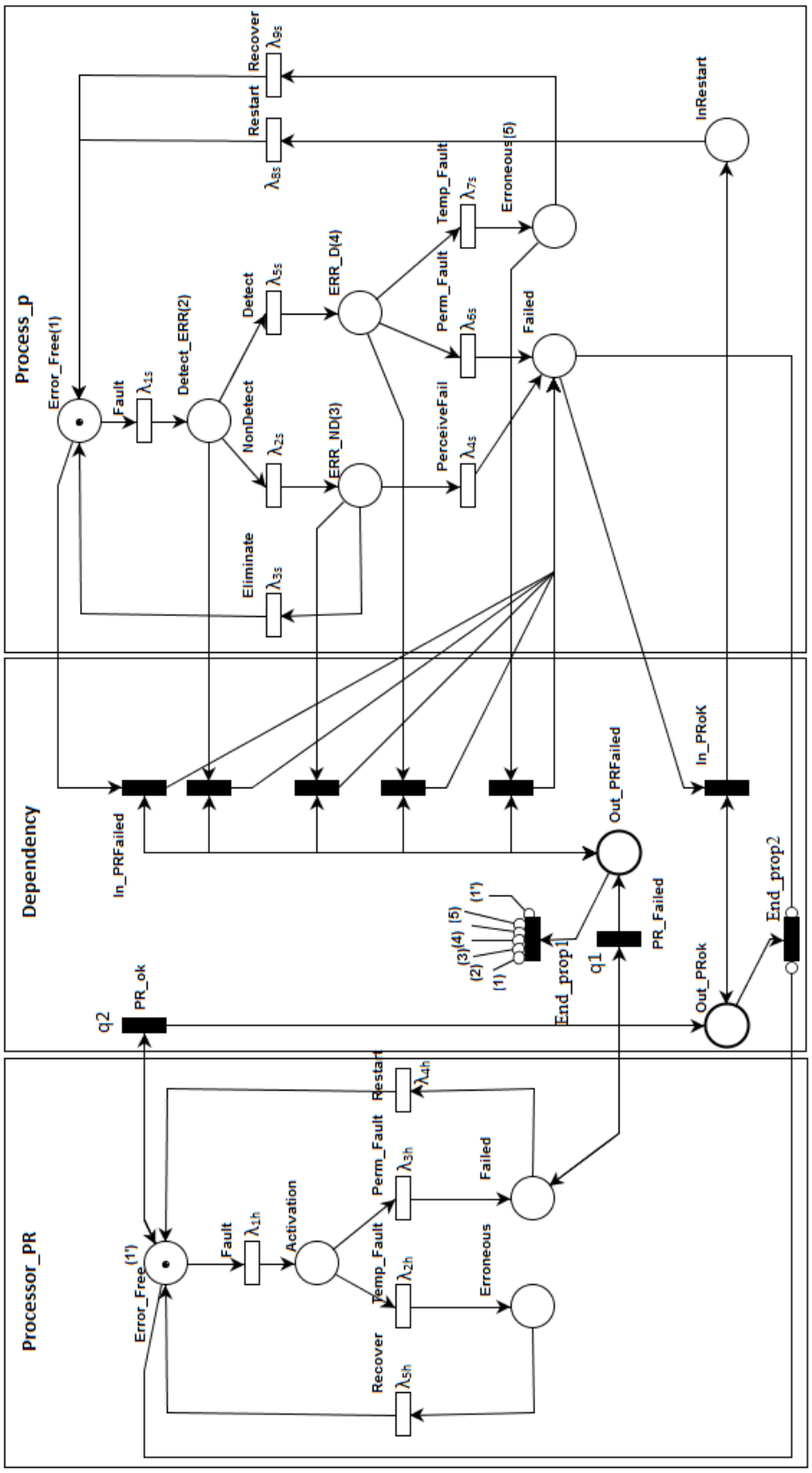


Fig. 6. Dependability GSPN Model

By applying transformation rules presented in [5], Figure 6 shows the GSPN obtained when transforming the AADL model corresponding to the process P linked to the processor PR. We note that places with dark circles are places with capacity of one token.

4.2 Construction of Performances model

The following section presents the steps for constructing the AADL Performance model with corresponding transformation steps. For more clarity, we model each component (process P or processor PR) in the presence of internal events and 'in' propagation and then we integrate the 'out' propagation following the description of the system. Figure 7 shows the activity model associated with the processor. As for the error model, we associate the activity model, *processor_PR.imp*, to the implementation of the component processor. The processor is initially free. It will be occupied if it receives an allocation request, '*request*'. It can go from '*Busy*' state to the '*Exp_termination*' state if the '*End_quantum*' event is activated with a rate λ_{6h}. Or it can pass to the '*Free*' state if it receives an 'in' propagation '*FreePR*'. From the '*Exp_termination*' state it can return to its initial state with a rate λ_{7h}.

Activity Model processor_PR
Features
Free : initial state;
Busy : state;
Exp_termination : state;
End_quantum : event {occurrence => poissonsλ_{6h}};
Initialization : event {occurrence => poissonsλ_{7h}};
request : in propagation;
FreePR : in propagation;
End processor_PR;
Activity Model implementation processor_PR.imp
transitions
Free − [in request]− > Busy;
Busy − [End_quantum]− > Exp_termination;
Exp_termination − [Initialization]− > Free;
Busy − [in FreePR]− > Free;
End processor_PR.imp;

Fig. 7. Activity Model for the processor component

The GSPN model of the processor PR (Figure 8) is obtained by applying the transformation rules (incomplete model).

Figure 9 shows the Activity Model associated with the process P. Initially, the process is in '*Ready*' state. It passes to '*Execution*' state when it receives an 'in' propagation '*Grant*' (it means that the processor starts its execution). It can go from '*Execution*' state to a *Ready* state if it receives an 'in' propagation

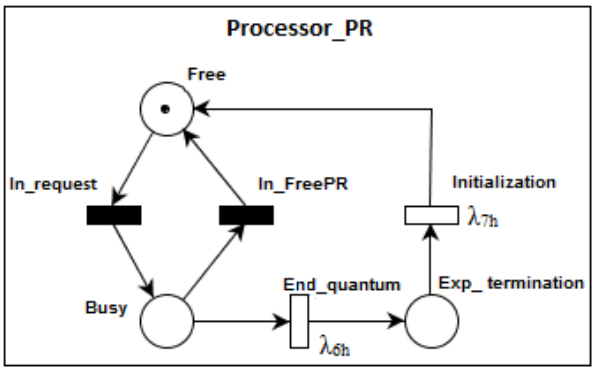


Fig. 8. GSPN model for the processor component

'*FQ*' (it is temporarily suspended to allow the execution of another process). It passes from the '*Execution*' state to the '*blocked*' state if the '*Even_R*' event is activated with a rate λ_{11s}. When the '*F_Even_R*' event occurs it passes to '*Ready*' state. It can go from the '*Execution*' state to a '*Ready*' state, if the '*End_T*' event is activated with a rate λ_{10s}.

Activity Model process_p
features
Ready : initial state;
Execution : state;
blocked : state;
Termination : state;
End_T : event {occurrence => poissons λ_{10s}*};*
Even_R : event {occurrence => poissons λ_{11s}*};*
F_Even_R : event {occurrence => poissons λ_{12s}*};*
Initialization : event {occurrence => poissons λ_{13s}*};*
FQ : in propagation;
Grant : in propagation;
end process_p;
Activity Model implementation process_p.general
transitions
Ready – [in Grant]– > Execution;
Execution – [in FQ]– > Ready;
Execution – [Even_R]– > Blocked;
Blocked – [F_Even_R]– > Ready;
Execution – [End_T]– > Termination;
Termination – [Initialization]– > Ready;
End process_p.general

Fig. 9. Activity Model for the process component

By applying the transformation rules we obtain the GSPN model of the process P (figure 10, incomplete model).

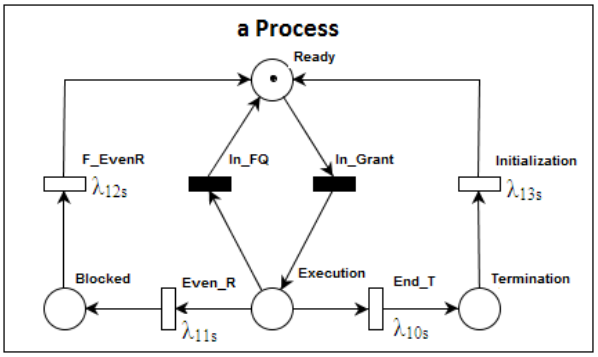


Fig. 10. GSPN model for the process component

Figure 11 shows just what is added to the activity model associated with the processor in order to describe the interaction between process P and the processor PR. The activity model type for processor, $processor\_PR$, is completed with lines L_O and L_1 (see figure 11) in order to include 'out' error propagation declarations such as:

- Line L_O declares an 'out' propagation 'FQ', which indicate the end of quantum. Its name matches the name of the 'in' propagation declared in the activity model type, $process\_p$ (see figure 9).
- Line L_1 declares an 'out' propagation '$Grant$', which indicates that the processor has given permission to move the process in '$Execution$' state. Its name matches the name of the 'in' propagation declared in the activity model type, $process\_p$ (see figure 9).

The activity model implementation, $processor\_PR.imp$, declares two transitions (lines L'_O and L'_1) triggered by the newly introduced 'out' propagation 'FQ' and '$Grant$'. When one of these two 'out' propagation occurs, the processor remains in the same state and the propagation remains visible until the processor leaves this state.

Similarly, Figure 12 shows what is added to the activity model associated with the process in order to describe the interaction between process P and the processor PR. The activity model type for process, $process\_p$, is completed with lines S_O and S_1 (see figure 12) in order to include 'out' error propagation declarations such as :

- Line S_O declares an 'out' propagation '$request$', to indicate that the process requires the processor. Its name matches the name of the 'in' propagation declared in the activity model type, $processor\_PR$ (see figure 7).

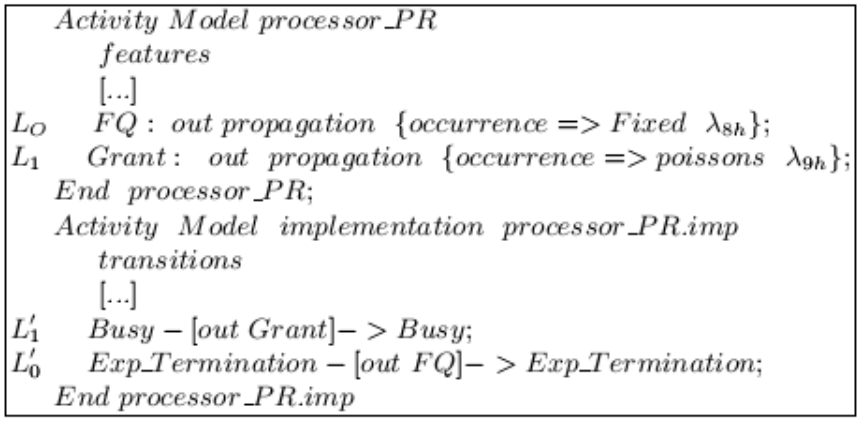


Fig. 11. Activity Model for processor component with interaction

– Line S_1 declares an 'out' propagation '$FreePR$'. Its name matches the name of the 'in' propagation declared in the activity model type, *processor_PR* (see figure 7).

The activity model implementation, *process_p.general*, declares three transitions (lines S_3, S_4 *and* S_5 of figure 12) triggered by the newly introduced 'out' propagation '*request*' and '$FreePR$'.

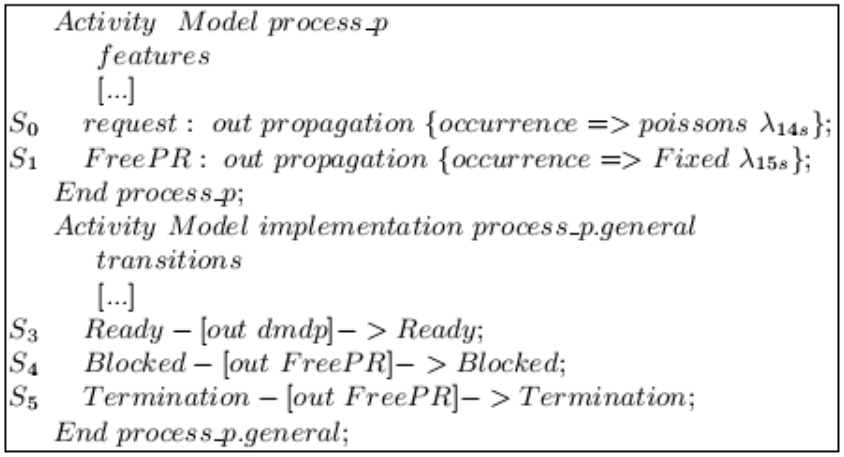


Fig. 12. Activity Model for process component with interaction

Figure 13 shows the GSPN model obtained by applying the transformation rules.

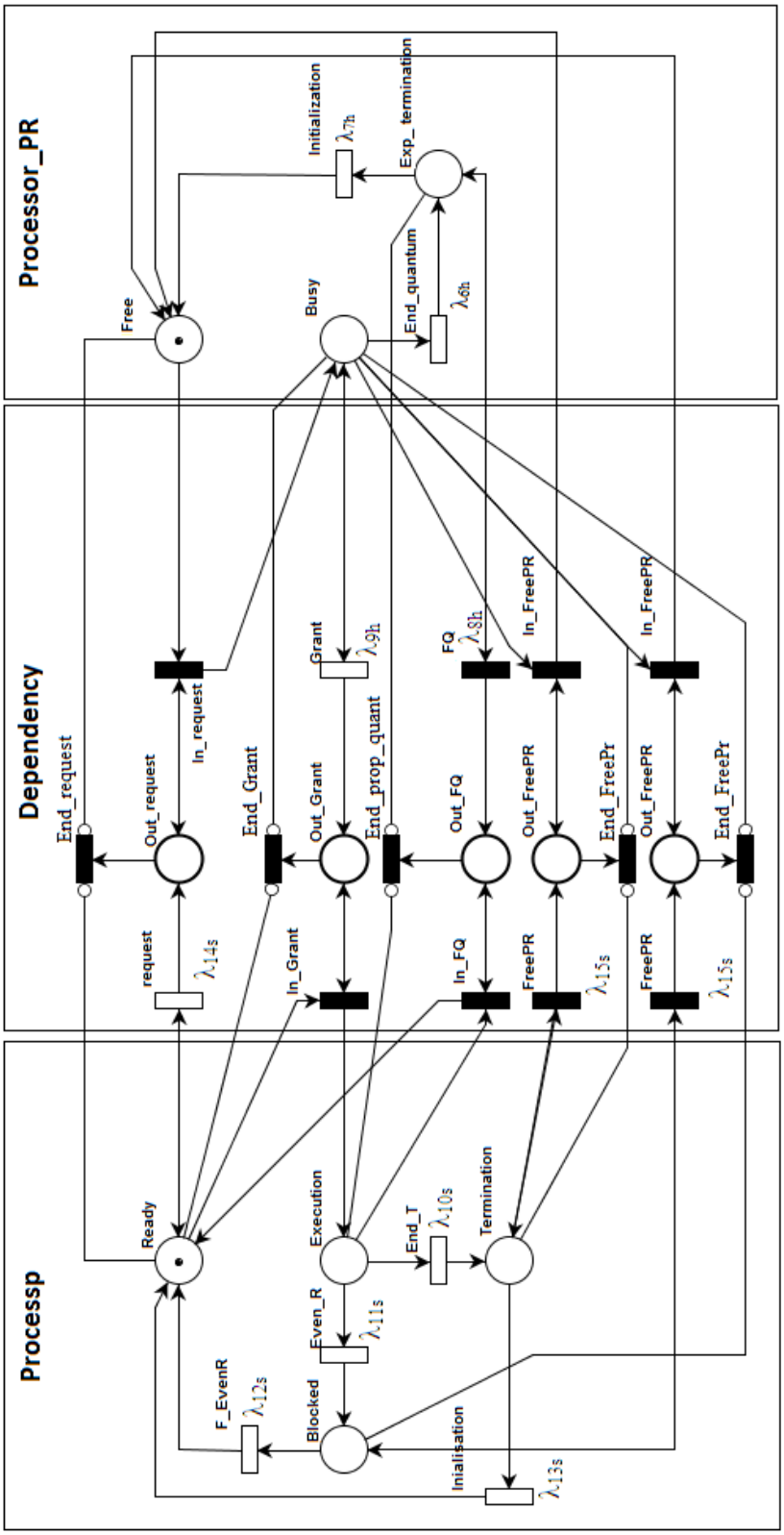


Fig. 13. GSPN Model of Performance

4.3 Application of composition algorithm

Now after the construction of the Performance and Dependability GSPN models, we apply our composition algorithm on these two models. Each model is composed of components sub-nets and dependencies subnets. Each component has a GSPN model of Dependability and a GSPN model of Performance. The basic idea of this algorithm is to connect each component sub-net of Performance with the corresponding component sub-net of Dependability. This algorithm is defined to ensure that the obtained global GSPN model is correct by construction (bounded, safe and no deadlock). The main steps of our composition algorithm are the following:

- For each sub-network component of Performance, if the component has no replicas, we add a bidirectional arc which connects the transitions of Performance model component with the place that represents the initial state of the corresponding Dependability model. This rule reflects that if the component is in a state of Performance model, it can move to another state only if there is no activation of a fault. Note that the number of tokens in a sub-net component is always 1 because at a given time a component can be only in one state. It is clear that if there is activation of a temporary fault, after adding a bidirectional arc, the component remains in waiting until the resolution of this fault since transitions in the Performance model are disabled. We note that if there is a place in a Performance model where the activation of a temporary fault does not exist, for all transitions that represent the output transitions of this place, the bidirectional arc is eliminated. In our case if the processor PR in a free state, a temporary fault will never be activated. Now if a permanent fault occurs, the component must regain its initial state. The rule of the link consists in adding timed transitions, and link with a bidirectional arc the initial place of Performance model with the transition *Restart* of Dependability model.
- if the component has replicas, each replica has the same GSPN model of Performance and Dependability. In first step, the addition of bidirectional arcs is applied to each replica. Then immediate transitions are added to represent the switching between the GSPN models.

By applying this algorithm on our models, we obtain Figure 14 which shows the GSPNs models of processor PR and Figure 15 which shows the GSPN models of the process P. The two models constitute the global model of the studied system.

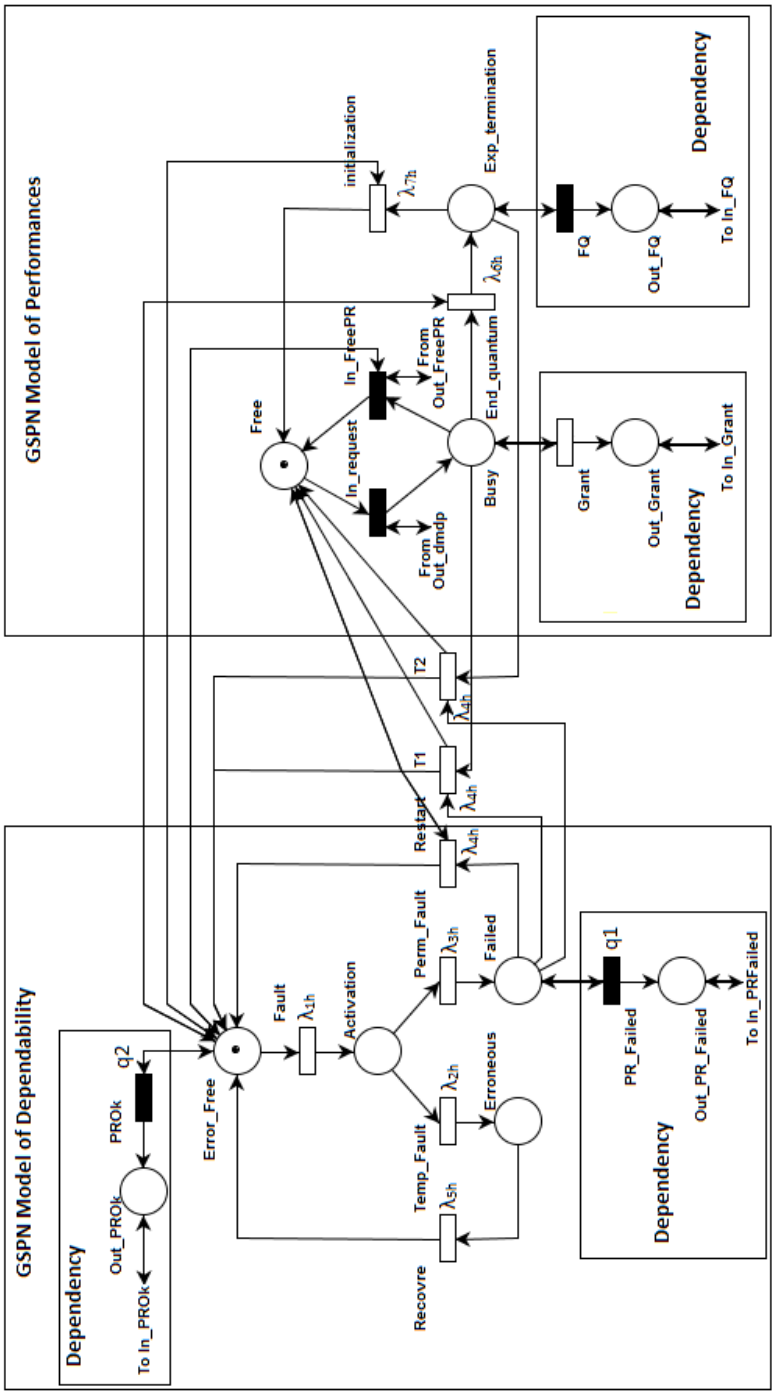


Fig. 14. The first part of the Global GSPN model: processor model

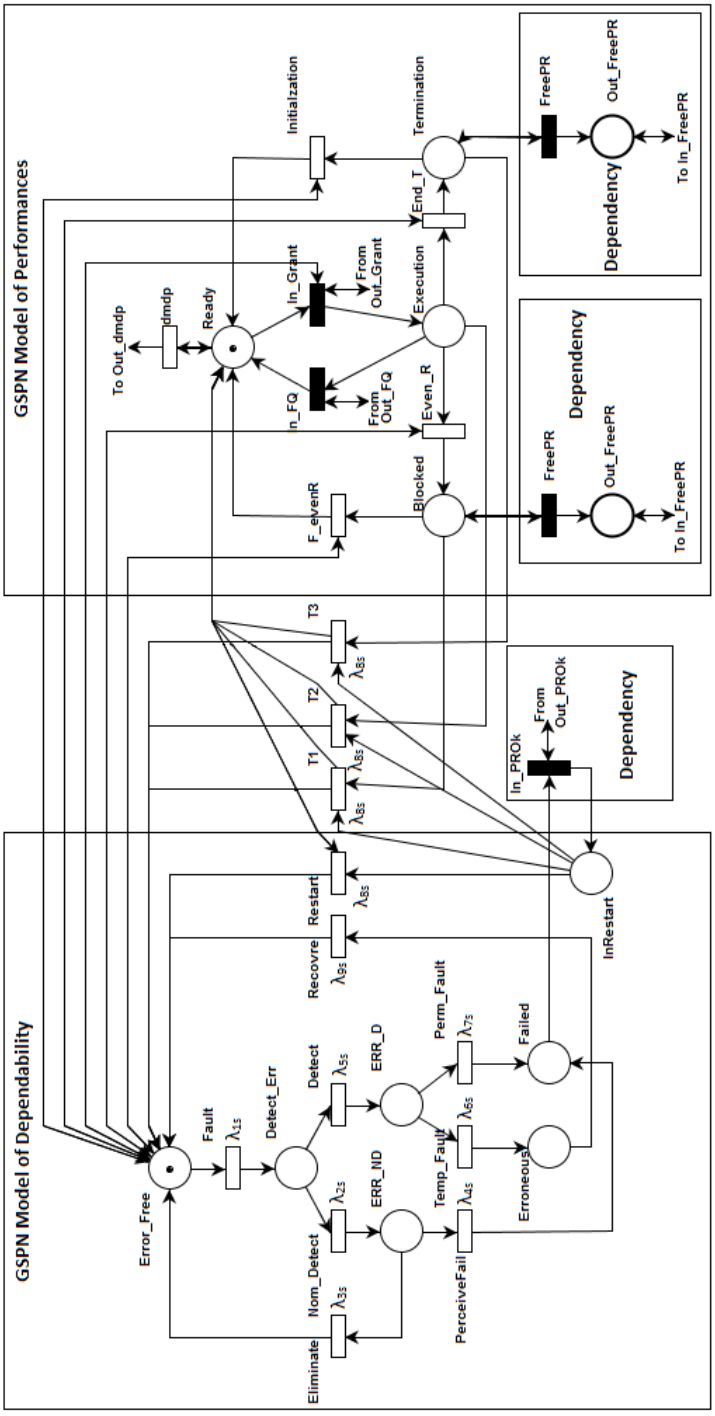


Fig. 15. The second part of the Global GSPN model: process model

5 Conclusion

In this paper, we have presented an approach based on AADL and Generalized Stochastic Petri Nets for modeling and analyzing Performance and Dependability of systems in the presence of faults. This approach consists of several steps. After modeling the system architecture in AADL, two AADL models are obtained, Dependability and Performance models. They are transformed in two GSPN models by applying transformation rules presented in [5]. Finally, by applying our algorithm we build the global model related to the studied system. Our composition algorithm was implemented in Java language. From Performance and Dependability models of hardware and software components, our algorithm builds a global GSPN model. The obtained GSPN model is a file type PNML exchange format which can be analyzed by tools that support this format such as Tina toolbox [12]. In [13] we applied this approach on the ABS anti-lock complex system.

References

1. K. Kanoun, M. Borrel, Fault-tolerant system Dependability: Explicit modeling of hardware and software component-interactions, IEEE Transactions on Reliability, 49, (2000).
2. SAE-AS5506. Architecture Analysis and Design Language, SAE, (2004).
3. SAE-AS5506/1, Architecture Analysis and Design Language Annex Volume 1. SAE, (2006).
4. Ana Elena Rugina, Karama Kanoun, Mohamed Kaaniche, System Dependability modeling using AADL langage, 15eme Congres de Maitrise des Risques et de Surete de Fonctionnement, Lille, (2006).
5. Ana-Elena Rugina, Dependability Modeling and Evaluation: From AADL to Stochastic Petri nets, PhD.D. thesis, Institut National Polytechnique de Toulouse, France, (2007).
6. Thomas Vergnaud. Modelisation des Systemes Temps reel Repartis Embarques pour la Generation Automatique d Application Formellement Verifiees. PhD thesis, Ecole Nationale Superieure des Telecommunications, (2006).
7. P. Feiler, B. Lewis, and S. Vestal. The SAE Architecture Analysis & Design Language (AADL) A Standard for Engineering Performance Critical Systems. In Proceedings of the 2006 IEEE Conference on Computer Aided Control Systems Design, (2006).
8. Rogerio de Lemos, Cristina Gacek, Alexander B. Romanovsky, Architecting Dependable Systems IV, Lecture Notes in Computer Science 4615, Springer, (2007).
9. Jean-Claude Laprie, Dependable Computing: Concepts, Limits, Challenges, Invited paper to FTCS-25, the 25th IEEE International Symposium on Fault-Tolerant Computing,Pasadena, California, USA, June 27-30, Special Issue, (1995).
10. M. Borrel, Interactions entre composants materiel et logiciel de systemes tolerant aux fautes - Caracterisation - Formalisation - Modelisation - Application a la surete de fonctionnement du CAUTRA, LAAS-CNRS, These de doctorat, Toulouse, France, (1996).

11. A. Bondavalli, S. Chiaradonna, F. D. Giandomenico, J. Xu, Fault-tolerant system Dependability: Explicit modeling of hardware and software component-interactions, Journal of Systems Architecture, 49, (2002).
12. B. Berthomieu, P.-O. Ribet, F. Vernadat, The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets, International Journal of Production Research, Vol. 42, No 14,(2004).
13. Kais Ben Fadhel, une approche de modelisation et d'analyse des performances et de la surete de fonctionnement : Transformation d'une specification AADL en GSPN, Master de recherche, faculte des sciences de Monastir, Tunisie, (2012).

Coloured Petri Nets Refinements

C. Choppy, L. Petrucci, and A. Sanogo

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

Abstract. Coloured Petri nets allow for modelling complex concurrent systems, but the specification of such systems remains a challenging task. Two approaches are generally used to lessen these difficulties: decomposition and refinement.
Charles Lakos proposed three kinds of refinements for coloured Petri nets: type refinement, subnet refinement, and node (place or transition) refinement. This paper proposes new rules widening the scope of both type and transition refinements.

1 Introduction

Coloured Petri nets are a specification language which presents the advantages of both a graphical description, giving an easy understanding of the model, and a formal semantics allowing for formal analysis techniques. However, as is the case for many specification languages, the specification of a system remains a difficult task. A way to alleviate these difficulties consists in using refinement techniques.

First, a rather simple model of the system is built, at a high level of abstraction, with few details. This abstract model constitutes a general description of the system. An incremental process of successive refinements is then applied, adding new details in a stepwise manner. The model is thus enhanced until all the expected behaviour and properties are taken into account. At each step, the abstract model is replaced by a refined one.

For defining refinements, the following questions should be addressed:

- which relation should exist between the abstract and the refined models?
- what are the necessary conditions on the coloured Petri net models for this relation to hold?
- which transformations of the abstract net satisfy these refinement conditions?

Since a model is characterised by its observed behaviour, the comparison between abstract and refined model will rely on it. Several notions of equivalence and order have been proposed in the literature. In particular, Lakos and Lewis [6, 8, 5, 7] propose that "*a model R is a refinement of a model A if all behaviour in R has a corresponding behaviour in A*" (thus a refinement, while it may introduce further details, cannot introduce behaviours that would be new

to the abstract model; this is useful to be able to explore the state space in a modular and still meaningful way). They express this relation between coloured Petri nets as a system morphism (behavioural morphism) from the refined model to the abstract one. Three kinds of refinements were proposed, and the system morphisms defined accordingly: type refinement, subnet refinement and node (place or transition) refinement. Transformation rules on the net structure, the colours and firing modes, that respect these morphisms, complete their work by providing practical refinement mechanisms.

This paper extends these coloured Petri nets refinements, and more specifically the type and transition refinements. Type refinement includes two constraints: the subtype relation defined by Liskov and Wing [9] as well as Lakos' refinement condition. Four operations on types are now permitted: addition of a component in a tuple, constraining a component value, addition and modification of functions. The conditions for these operations to satisfy the refinement constraints are checked. For node refinement, a new refinement rule satisfying Lakos's constraints is introduced: alternate transitions.

The paper is organised as follows. In Section 2 definitions of coloured Petri nets and system morphisms are recalled. Then, Section 3 recalls the refinements defined by Lakos and Section 4 the subtyping relation of Liskov and Wing. Our new refinement rules are then defined and proven correct in Sections 5 and 6. They are implemented in a tool for Petri net design, described in Section 7.

2 Coloured Petri Nets and Morphisms

In this section, we recall the necessary definitions and notations from [6].

2.1 Coloured Petri Nets

For a type universe Σ, we denote the set of functions from one type of Σ to another by $\Phi\Sigma = \{X \to Y \mid X, Y \in \Sigma\}$ and by $\mu X = \{X \to \mathbb{N}\}$ the set of multisets over a type $X \in \Sigma$.

Definition 1. *A* coloured Petri net *is a tuple* $N = \langle P, T, A, C, E, \mathbb{M}, \mathbb{Y}, M_0 \rangle$ *where:*

1. P *is a set of* places*;*
2. T *is a set of* transitions *such that* $P \cap T = \emptyset$*;*
3. A *is a set of* arcs *such that* $A \subseteq P \times T \cup T \times P$*;*
4. C *is a* colour function *which associates a type with each place and transition:* $C : P \cup T \to \Sigma$*;*
5. E *is a function associating an* expression *with each arc:* $E : A \to \Phi\Sigma$ *with* $E(p,t), E(t,p) : C(t) \to \mu C(p)$*;*
6. $\mathbb{M}$ *is the* set of markings*:* $\mathbb{M} = \mu\{(p,c) \mid p \in P, c \in C(p)\}$*;*
7. $\mathbb{Y}$ *is the* set of steps*:* $\mathbb{Y} = \mu\{(t,c) \mid t \in T, c \in C(t)\}$
8. $M_0 \in \mathbb{M}$ *is the* initial marking*.*

For a node $x \in P \cup T$, we denote by

- ${}^\bullet x$ the *preset* of x, i.e. ${}^\bullet x = \{y \in P \cup T | (y, x) \in A\}$;
- $x^\bullet$ the *postset* of x, i.e. $x^\bullet = \{y \in P \cup T | (x, y) \in A\}$.

In the following, E^- denotes the expressions of arcs from places to transitions and E^+ from transitions to places. The firing rule of a coloured Petri net is now defined.

Definition 2. *Let N be a coloured Petri net. A step $Y \in \mathbb{Y}$ is* firable *from a marking $M \in \mathbb{M}$, denoted $M[Y\rangle$ iff $M \geq E^-(Y)$.*

In order to replace a node of a Petri net by a subnet during the refinement process, the connections between these and their environment must be considered. Therefore, we now define the border nodes on each side, adapted from [6].

Definition 3. *Let N be a Petri net, and N' a subnet of N.*

1. *The* input border of N' *is* $inpbdr(N') = \{x \in P' \cup T' | \exists y \in (P \cup T) \setminus (P' \cup T') : y \in {}^\bullet x\}$;
2. *The* output border of N' *is* $outbdr(N') = \{x \in P' \cup T' | \exists y \in (P \cup T) \setminus (P' \cup T') : y \in x^\bullet\}$;
3. *The* border of N' *is* $bdr(N') = inpbdr(N') \cup outbdr(N')$;
4. *The* set of input environment nodes of N' *is* $inpenv(N') = \{y \in (P \cup T) \setminus (P' \cup T') | \exists x \in P' \cup T' : y \in {}^\bullet x\}$;
5. *The* set of output environment nodes of N' *is* $outenv(N') = \{y \in (P \cup T) \setminus (P' \cup T') | \exists x \in P' \cup T' : y \in x^\bullet\}$;
6. *The* set of environment nodes of N' *is* $env(N') = inpenv(N') \cup outenv(N')$.

Example: Let us consider the net in Figure 1, with the subnet in the large circle. Its border sets are:

- $inpbdr(N') = \{inp_1, inp_2\}$;
- $outbdr(N') = \{out_1, out_2\}$;
- $bdr(N') = \{inp_1, inp_2, out_1, out_2\}$;
- $inpenv(N') = \{te_1, te_2\}$;
- $outenv(N') = \{ts_1, ts_2\}$;
- $env(N') = \{te_1, te_2, ts_1, ts_2\}$.

2.2 System Morphisms

System morphisms were defined in [6] in order to capture behaviour compatibility between nets. In such a scheme, firing several transitions of N_r can correspond to the firing of a single transition in N_a. A step is then said to be complete if all border transitions of $N_r \setminus N_a$, i.e. the subset that remains after N_a has been removed from N_r, are fired with the same firing mode.

Definition 4. *Let $\phi : N_r \to N_a$. ϕ is a* system (behavioural) morphism *if:*

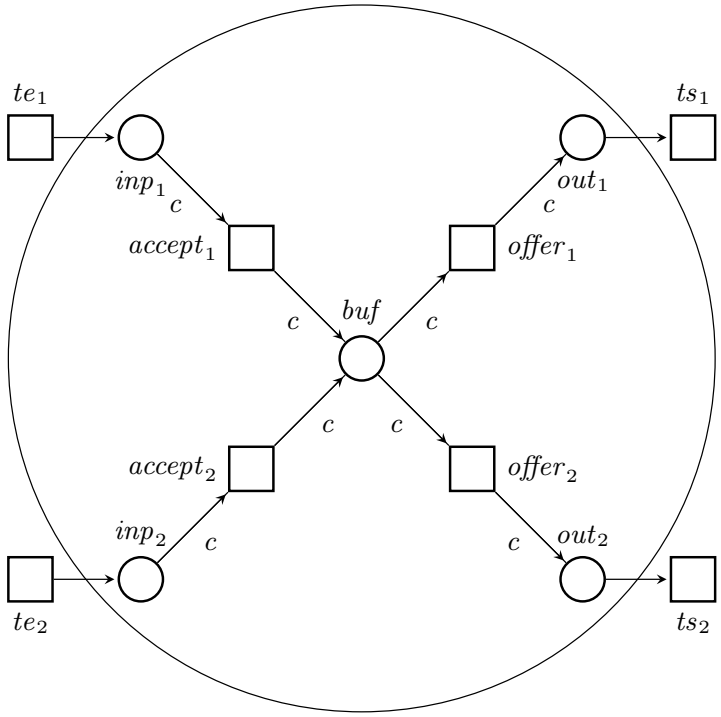


Fig. 1. Example of border nodes

1. *ϕ is surjective on P_a, T_a, A_a;*
2. *ϕ is linear and total on the set of markings $\mathbb{M}_r$ and the set of steps $\mathbb{Y}_r$;*
3. *$\forall M_r \in [M_0\rangle_r$, $\forall Y_r \in \mathbb{Y}_r$: if Y_r is complete, can be decomposed into Y_1, Y_2 ... Y_n, and is realisable from marking M_r, then $\phi(Y_r)$ can be decomposed into $\phi(Y_1), \phi(Y_2)$ ...$\phi(Y_n)$, and is realisable from marking $\phi(M_r)$.*
4. *$\forall M_r \in [M_0\rangle_r, \forall Y_r \in \mathbb{Y}_r : Y_r$ is complete $\Rightarrow \phi(M_r + E_r^+(Y_r) - E_r^-(Y_r)) = \phi(M_r) + \phi(E_r^+)(\phi(Y_r)) - \phi(E_r^-)(\phi(Y_r))$.*

In Definition 4, (3) indicates that when a step in N_r is decomposable, the corresponding step can be obtained by projection on N_a, while (4) states that only the effect of a step can be projected as well.

Definition 5. *Let $\phi : N_r \to N_a$ be a system morphism. A step Y_r in N_r is said to be* complete *if $\forall t_a \in T_a : \forall t_r \in bdr(T_r \setminus T_a) : Y_r(t_r) = \phi(Y_r)(t_a)$*

3 Existing Coloured Petri Nets Refinements

Three types of coloured Petri nets refinements are defined by Lakos in [6]: type refinement, node refinement, and subnet refinement.

3.1 Type Refinement

Type refinement is used when it is necessary to detail further the description of the information carried by tokens, and the transitions' firing modes. Refine-

ment then consists in replacing an abstract token type by another one, more detailed, called the refined type. The net structure remains unchanged. The type modification addressed by Lakos is the addition of a component.

3.2 Subnet Refinement

Subnet refinement consists in adding new elements (places, transitions and arcs) to the net.

[6] also considers extending type domains for places and transitions in the abstract net, as a subnet refinement.

3.3 Node Refinement

Node refinement is used when the modeller wishes to give additional information about one of the net elements (place or transition) by expliciting it further. It can thus be a place refinement if details concern a place (superplace) or transition refinement if a transition is concerned (supertransition). Lakos defined canonical refinements for both of these cases.

In its canonical form, a place refinement replaces a place by a place-bordered subnet, whereas a transition refinement replaces a transition by a transition-bordered subnet.

4 The Sub-typing Relation

A subtype relation was defined in [9] by Barbara Liskov and Jeannette Wing so that the supertype properties should be preserved by the subtype. The properties considered are invariants (that should be true for any object state) and "historical" properties (true on a state sequence). The substitutability principle states that a subtype object could substitute a supertype object.

Types are denoted by a triple $\langle O, V, M \rangle$ where O is a set of objects, V is a set of values, and M is a set of methods.

The type specification should contain the following information:

- the description of the set of authorised values;
- for each method, the description of (i) its signature (number and types of arguments, result type and exceptions list), (ii) its behaviour in terms of pre- and post-conditions.

B. Liskov and J. M. Wing distinguish three kinds of methods, *constructors* that return a new object of the type, *observers* that return values of other types, and *mutators* that modify object values.

They also identify two kinds of subtyping relations:

- *Extension subtype* where the subtype extends the supertype by introducing new methods and adding new states (or values).

– *Constrained subtype* where the subtype constrains the supertype with more details on the methods or on the supertype values. When the supertype specification keeps open several possibilities, subtyping may reduce or eliminate some of them.

Considering the two kinds of subtyping relations above, we consider here the following operations on types:

– *add a component* to a record
– *fix a component value*
– *add a method*
– *modify a method.*

These operations on types appear in the subtype relations as follows.

1. *Extension Subtype* The operations on types considered are the introduction of new methods and/or the addition of new states.
   – New method introduction
     • if the method introduced is a *constructor*, it should take into account the invariant preservation;
     • if the method introduced is an *observer*, this has no consequence on the fact that the subtype preserves the supertype properties;
     • if the method introduced is a *mutator*, this may have consequences on the fact that the subtype should preserve the supertype properties. After a mutator is invoked, the new object value should belong to the set of authorized type values (according to invariants and historical properties).
   – Addition of new object states (or adding a new variable in records): this has no effect on properties preservation, and the abstraction function forgets the added variable.
2. *Constrained Subtype* The operations considered here are constraints on the object values and/or method modification with the aim to add more precision.
   – Constraints on the object values (this may correspond to fixing a variable value in a record). This may be achieved by defining the set of values for the subtype objects as a subset of the supertype values and this preserves the supertype properties.
   – Method modification. This modification must comply with the subtyping rules defined in [11], that are a rule on the signature (same number of arguments, contravariance of arguments, covariance of result and exception rule), and a rule on methods.

Let us recall that if σ is a subtype of τ and m_τ is the τ method corresponding to method m_σ of σ :

– arguments contravariance states that m_τ and m_σ should have the same number of arguments, and type α_i of the ith argument in m_τ should be a subtype of type β_i of the corresponding argument in m_σ.
– either m_τ and m_σ return a result or do not return a result. If both methods return a result, the result covariance states that the type of the result returned by m_σ should be a subtype of the type of the result returned by m_τ.

5 New Rules for Type Refinement

An initial definition of *type refinement* was proposed by Lakos in [6] requiring that all behaviours of the refined type correspond to an abstract behaviour. This constraint will be referred to as LC in the following (Lakos's constraint). The only operation allowed by Lakos for type refinement is the addition of a component in a tuple. Although this kind of refinement is often used, it is still restrictive in practical cases which allow for more substantial type modifications and introduction of new operators on existing types.

Therefore, in this section, we extend the relation between the refined type and the abstract type by considering the sub-typing relation defined by Liskov and Wing in [10], as well as type modifications.

5.1 Type modifications and behaviour preserving

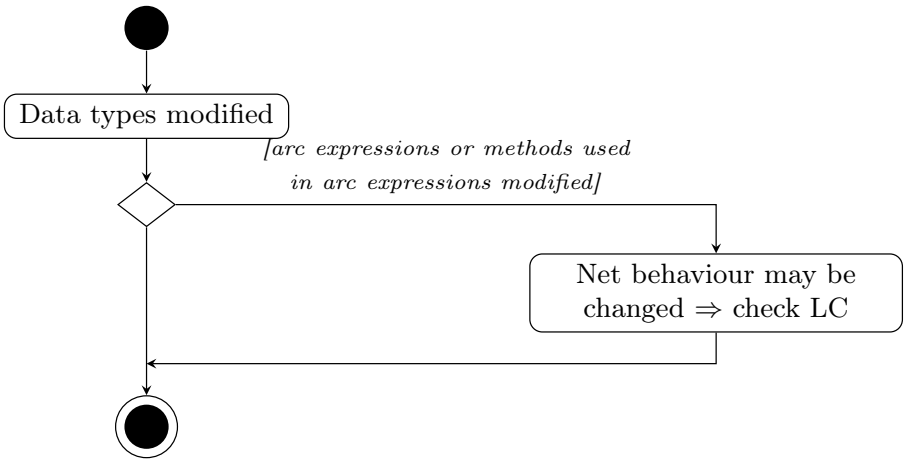


Fig. 2. Summary of arc expressions modifications impact

When performing type refinement, the net structure remains unchanged. Hence the only elements which may change the net behaviour are the values of arc expressions, and transitions guards.

Arc expressions Subsequently to a type refinement, an arc expression function might be modified. Figure 2 summarises the impact of type modification on arc expressions.

In the following, E_r denotes arc expressions in the refined net, and E_a the corresponding arc expression in the abstract net.

Unchanged arc expressions If the arc expression is the same, i.e. $E_r = E_a$, the following cases may occur:

1. adding a component to a tuple, setting the value of a component, adding a method, are changes that respect LC;
2. modification of a method leads to the following situations:
   - E_a does not refer to a method modified by the refinement. The values returned by E_r and E_a are thus the same, and the net behaviour remains unchanged ;
   - E_a refers to methods changed by the refinement. Hence, although the expressions are the same, the values returned by E_r and E_a may differ. Therefore the behaviour can be different as well, and we need to check that the effect of firing for each firing mode in the refined net is the same, once abstracted, as for the corresponding firing in the abstract net.

Modified arc expressions When an arc expression is modified due to the refinement process, values of E_r and E_a may differ, whatever the refinement operation. Hence, adding a component, setting the value of a component, adding a method may all modify the net behaviour. The expression modification should be performed so that it satisfies LC.

Guards associated with transitions Type refinement does not change the expression associated with a guard. However, two cases may occur (see Fig. 3):

- the guard does not refer to a method modified by the type refinement: it has thus no influence on the transition behaviour;
- the guard refers to methods modified by the type refinement. Hence the values returned by the refined and abstract guard expressions may differ. If the guard returns *true* in the refined net, it must also return *true* in the abstract net.

5.2 Example: request for material purchase

The net in Fig. 4(a) models a request for material issued from e.g. a service to the accountant. When material is needed, a request is issued. After the accountants check the request, they order the articles.

In a first approach to modelling the problem, the net describing the overall process can de considered. It uses a neutral colour, as shown in Fig. 4(a).

Then additional detail can be introduced, giving characteristics of the article to order, i.e. its *number*, its designation (*name*), its *origin*, the quantity required (*qty*) and the unitary *price*. The new type is thus obtained by adding components, as shown in the declarations below and Fig. 4(b). In this case, no arc or guard expression is changed, therefore LC is satisfied. It is also consistent with the subtyping conditions.

In order to take into account calls for offers when the price is above a certain level, a method is added which returns the total *amount* of the purchase

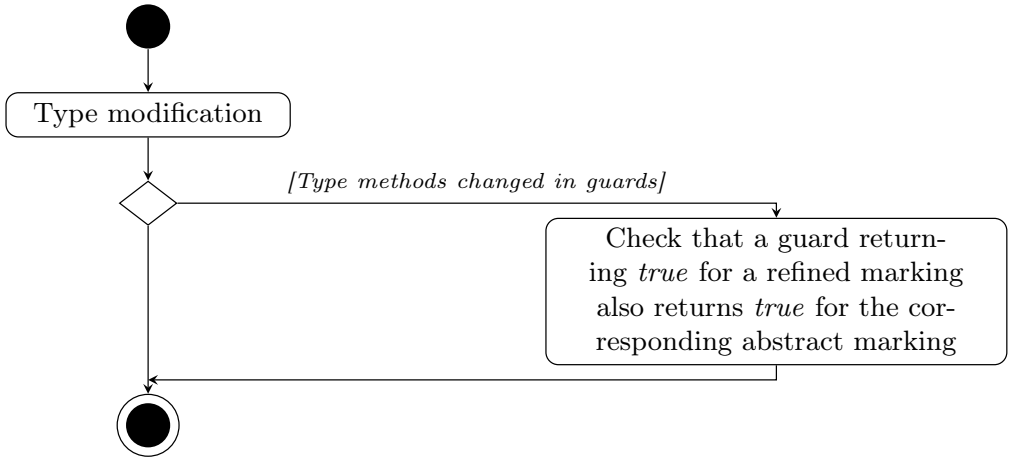
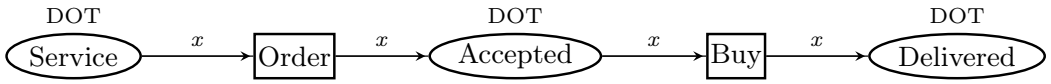
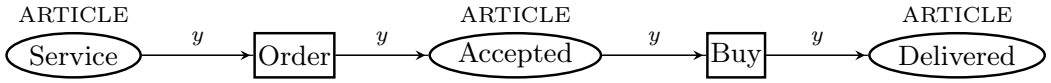


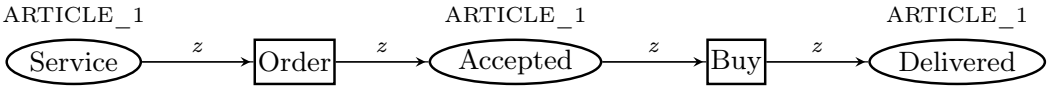
Fig. 3. Summary of the impact of type refinement on guards and net behaviour



(a) Request for material purchase modelled with the neutral type (DOT)



(b) Request for material purchase modelled with the ARTICLE type



(c) Request for material purchase modelled with the ARTICLE_1 type

Fig. 4. Type refinement of a net model of a request for material purchase

$price * qty$ (see type $ARTICLE\_1$) in the declarations and Fig. 4(c)). This modification adds an observer method which returns a value, and is consistent with the subtyping relation. It is not used in arc or guard expressions, and therefore LC holds.

```
Declarations:
DOT is a predefined neutral type
Colset ARTICLE = record number:NAT * origin:STRING *
                 name:STRING * price:Rat * qty:Rat;

Colset  ARTICLE_1 = record number:NAT * origin:STRING *
                    name:STRING * price:Rat * qty:Rat;
fun amount = (#price ARTICLE_1 * #qty ARTICLE_1)::Rat;
var x : DOT;
var y : ARTICLE;
```

```
var z : ARTICLE_1;
var name : STRING;
var price : Rat;
var qty : Rat;
```

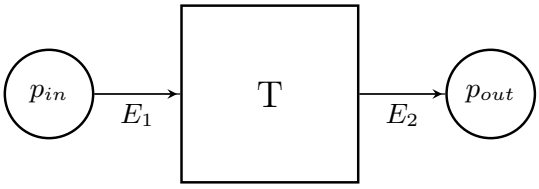
6 New Rules for Transition Refinement: Alternate Transitions

The canonical transition refinement proposed by Lakos, firing an abstract transition features firing a set of sequences of refined transitions, starting with transitions from the input border and ending with transitions from the output border of the refined part.

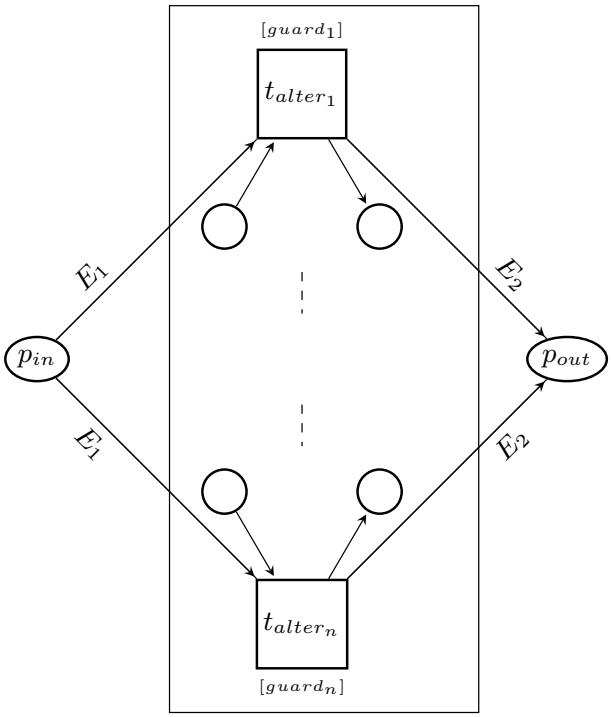
The underlying idea of the new refinement we propose here is to replace an abstract transition by a subnet containing alternative transitions plus internal places and transitions. Each of the alternative transitions is abstracted as the original abstract transition. This refinement aims at splitting a transition describing a general behaviour into several exclusive cases which then handle in more detail specific situations. For example, the net in Fig. 5(b) is a refinement of the one in Fig. 5(a): transition t_a is replaced by a subnet N_{t_a}. To improve readability, t_a has a single input place and a single output place (but this is not a constraint in the general case).

Definition 6. *Let* $N_{t_a} = (P_{t_a}, T_{t_a}, A_{t_a}, C_{t_a}, E_{t_a}, \mathbb{M}_{t_a}, \mathbb{Y}_{t_a}, M_{t_{a0}})$ *be a subnet. A morphism*
$\phi : N_r = (P_r, T_r, A_r, C_r, E_r, \mathbb{M}_r, \mathbb{Y}_r, M_{r0})$
$\quad \rightarrow N_a = (P_a, T_a, A_a, C_a, E_a, \mathbb{M}_a, \mathbb{Y}_a, M_{a0})$
is a refinement with alternative transitions *of* $t_a \in T_a$*, where* N_r *is the refined net obtained by replacing abstract transition* t_a *by* N_{t_a}*) in the abstract net* N_a *if:*

1. $\forall p_r \in P_r \setminus P_{t_a} : \forall t_r \in T_r \setminus T_{t_a} :$
   $\phi(p_r) = p_r \wedge \phi(t_r) = t_r \wedge$
   $(p_r, t_r) \in A_r \Rightarrow ((p_r, t_r) \in A_a \wedge E_r(p_r, t_r) = E_a(p_r, t_r)) \wedge$
   $(t_r, p_r) \in A_r \Rightarrow ((t_r, p_r) \in A_a \wedge E_r(t_r, p_r) = E_a(t_r, p_r))$. *Apart from transition* t_a *the abstract net remains unchanged.*
2. $T_{t_a} = T_{alternative} \cup T_{other}$ *The transitions replacing* t_a *are of two kinds: the alternatives, and the others.*
3. $\forall t \in T_{alternative}, \ {}^\bullet t_a = {}^\bullet t \setminus P_{t_a} \wedge \forall p \in {}^\bullet t_a : E_a(p, t_a) = E_r(p, t)$
   All input places of t_a *are also input places of all alternative transitions, and are the only such abstract places.*
4. $\forall t \in T_{alternative}, t_a^\bullet = t^\bullet \setminus P_{t_a} \wedge \forall p \in t_a^\bullet : E_a(t_a, p) = E_r(t, p)$
   All output places of t_a *are also output places of all alternative transitions, and are the only such abstract places.*
5. $\forall M \in \mathbb{M}_r : \forall t' \in T_{alternative} :$
   $\phi(M)[t_a > \ \wedge \ M[t' > \Rightarrow \forall t \in T_{alternative} \setminus \{t'\}, \neg(M[\ t >)$
   There is at most one alternative transition that is firable in a given marking (so that the token flow is preserved). This can be ensured by guards or internal places of the N_{t_a} *subnet.*



(a) Abstract transition to be refined



(b) Refinement of transition t_a

Fig. 5. Refinement with alternative transitions

6. $\forall t \in T_{alternative} \wedge \forall c \in C_r(t) : \phi(1\ `(t,c)) = 1\ `(t_a, \phi(c))$
   Firing an alternative transition has the same effect as firing t_a. Firing a step in the refined net has the same effect as in the abstract net.
7. $\forall p \in P_r : \forall c \in C_r(p) : \phi(1\ `(p,c)) = 1\ `(p,c)$ *if* $p_r \in P_r \setminus P_{t_a}$, $\phi(1\ `(p,c)) = \emptyset$ *otherwise.*
   The internal marking of N_{t_a} is ignored by the refinement.

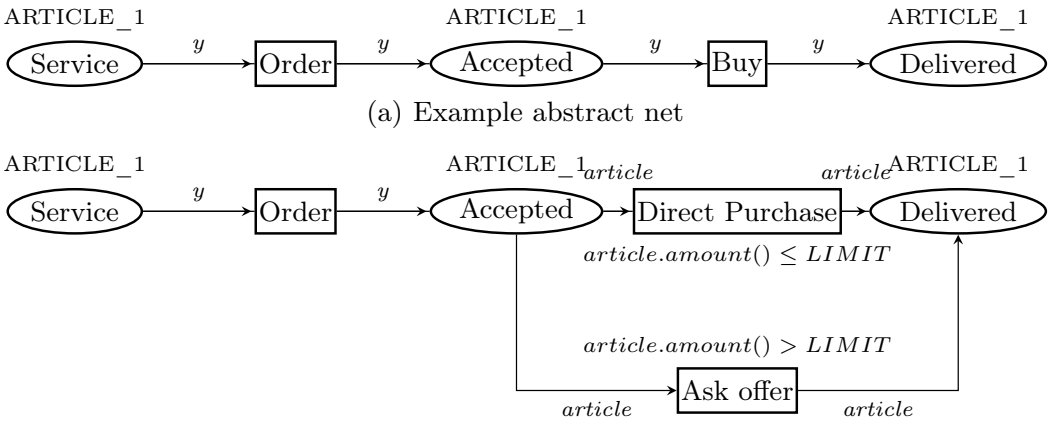
Proposition 1. *A refinement with alternative transitions $\phi : N_r \rightarrow N_a$ is a systems morphism.*

Proof.
According to definition 6(1) ϕ is surjective over P_a, T_a, A_a.

From (8), the abstract marking is obtained by ignoring the subnet marking. Hence, ϕ is linear and total over $\mathbb{M}_r$.
From (6), the firing of an alternative transition corresponds to firing the abstract transition. Hence, ϕ is linear and total over $\mathbb{Y}_r$.
From (6) $\forall Y_r \in \mathbb{Y}_r$, Y_r is complete since only one of the alternative transitions can be fired in a given marking.
From (3), (4) and (7), $\forall Y_r \in \mathbb{Y}_r, \forall M_r \in \mathbb{M}_r$ *si* $M_r \geq E_r^-(Y_r)$ then
$\phi(M_r) \geq \phi(E_r^-)(\phi(Y_r))$ and
$\phi(M_r + E_r^+(Y_r) - E_r^-(Y_r)) = \phi(M_r) + \phi(E_r^+)(\phi(Y_r)) - \phi(E_r^-)(\phi(Y_r))$.
$\phi(M_r + E_r^+(Y_r) - E_r^-(Y_r))$ is the effect after abstraction of the firing of the refined step Y_r from the refined marking M_r and $\phi(M_r) + \phi(E_r^+)(\phi(Y_r)) - \phi(E_r^-)(\phi(Y_r))$ the effect of the firing of the abstract step $\phi(Y_a)$ from the abstract marking $\phi(M_a)$.

Let us consider again the example of Sect. 5.2, with the abstract net in Fig. 6(a). We now want to explicit the accounting rules: if the amount of the purchase is greater than some limit, a call for offers must be issued (see Fig. 6(b)).

Transition *Buy* has been replaced by two alternative transitions: *Direct purchase* et *Ask offer*.



(a) Example abstract net

(b) Refined net including the accounting rules

Fig. 6. Refinement of the net model for purchases

7 CPN Refiner: Tool Support for Modelling

We designed CPN Refiner so as to support the development of coloured Petri nets following the approach proposed in [3] and the refinement techniques proposed in this paper.

Let us recall that the Petri net development method presented in [3] proposes the following steps: (i) analyse the text describing the problem and extract the

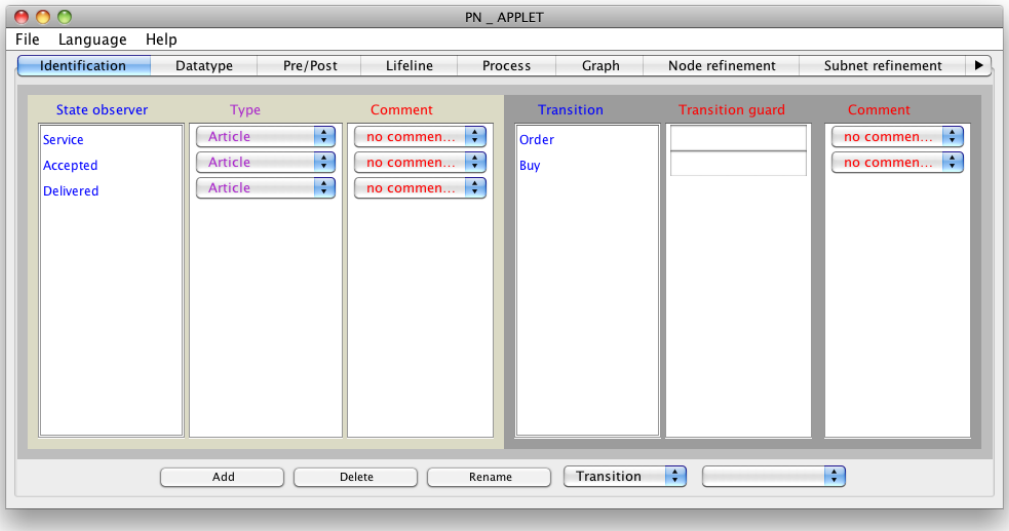


Fig. 7. Workshop example: state observers and transitions

events (yielding a state change), state observers, data types, and possibly the modules, (ii) establish the system properties (in particular pre and postconditions of events), (iii) build the coloured Petri net, (iv) check the resulting net properties (some are built-in, but others can be model checked), and update it if necessary.

CPN Refiner supports both the coloured Petri net development given the (typed) state observers and the events (together with their pre and postconditions), and its refinement. CPN Refiner supports node refinement and subnet refinement according to the principles stated in this paper. Type refinement is supported via a mere type modification.

Figure 7 shows the screen where the user entered the state observers and the events for a workshop example, and the generated Petri net is shown in Figure 8. A refined Petri net obtained through alternate transition refinement is shown in Figure 9.

CPN Refiner is build using Java Swing (Eclipse), API JDOM to handle the XML resulting file, library JGraph for the graphical presentation Petri nets.

8 Conclusion and Perspectives

Refinement is used to build a more detailed model (the refined model) from an abstract model. Several refinement notions have been proposed for different aims and various languages, and the work of Lakos deals with refinement for coloured Petri nets. An important point he states is that a model R is a refinement of a model A if for each behaviour of R there is a corresponding behaviour of A. More specifically, he defines three kinds of refinement, type refinement, node

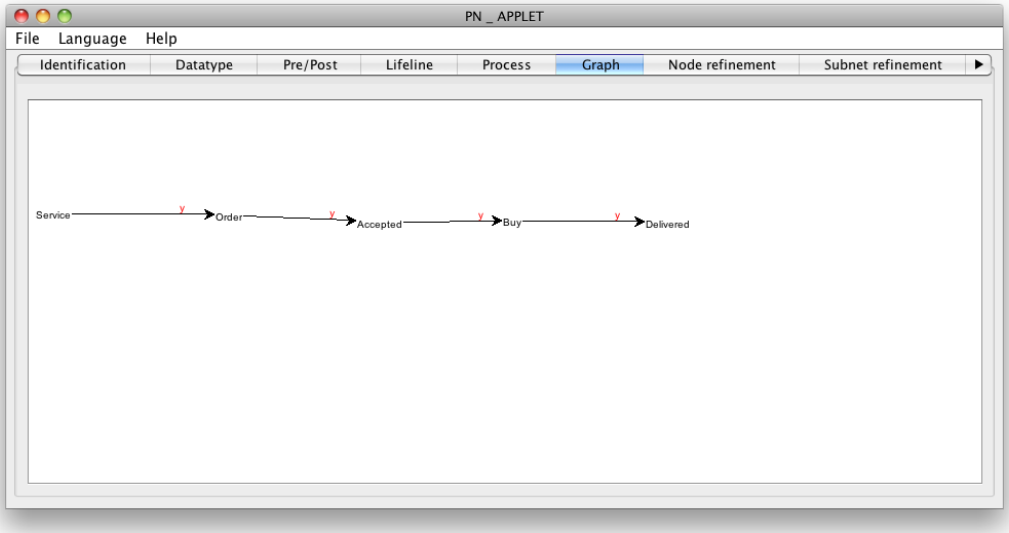


Fig. 8. Workshop example: coloured Petri net

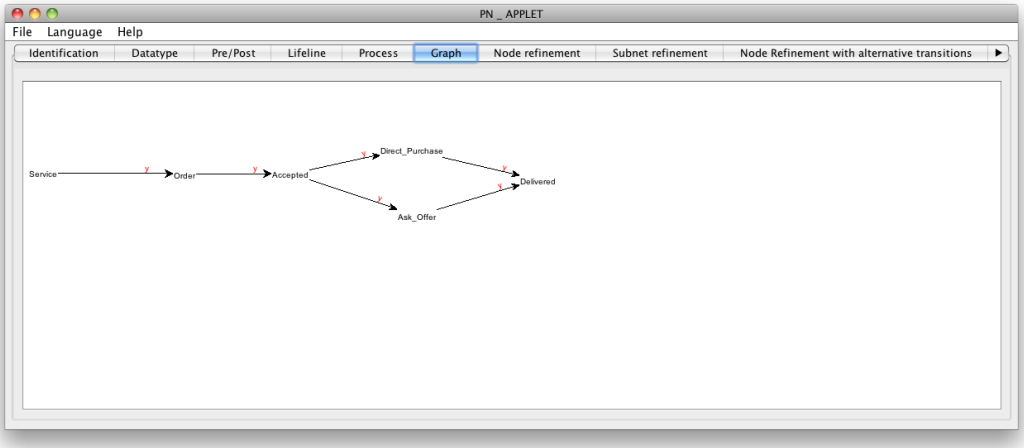


Fig. 9. Workshop example: refined Petri net

refinement (for places or transitions), and subnet refinement, so as to insure this behaviour correspondence.

In this paper, we extend the type refinement and the node refinement proposed by Lakos. For type refinement, we propose two constraints. The subtype relation as defined by Liskov and Wing that should exist between the refined and the abstract type, and Lakos' refinement relation between the refined and the abstract model. We consider four operations on types, that are to add a component, to fix a component value, to add a method, and to modify a method. We studied the conditions that ensure that these operations guarantee both Liskov and Wing subtyping relation and Lakos refinement relation. We also extend the node refinement with a new rule for transitions refined by alternate transitions. This new rule complies with Lakos' refinement principle.

A model development method was proposed for coloured Petri nets in [3]. We developed a tool, CPN-Refiner, for model development of coloured Petri nets following this method, and integrated in this tool the refinement techniques presented here [12]. In the future, we plan to develop large case studies using our method and our tool. We also plan to work on property refinement. Interface with other tools like CPN Tools [1] or CosyVerif [4] is also subject of future work, which should be eased since CPN Refiner uses PNML [2].

References

1. CPN Tools Homepage. `http://cpntools.org/`.
2. PNML reference site. `http://pnml.org/`.
3. C. Choppy, L. Petrucci, and G. Reggio. Designing coloured Petri net models: a method. In *Proc. Workshop on Practical Use of Coloured Petri Nets*, 2007.
4. CosyVerif group, The. CosyVerif home page. `http://cosyverif.org`.
5. Charles Lakos. On the abstraction of coloured Petri nets. In *18th Int. Conference on Application and Theory of Petri Nets ICATPN '97*, volume 1248 of *Lecture Notes in Computer Science*, pages 42–61. Springer-Verlag, 1997.
6. Charles Lakos. Composing abstractions of coloured Petri nets. In Nielsen, M. and Simpson, D., editors, *21st Int. Conf. on Application and Theory of Petri Nets (ICATPN)*, volume 1825 of *Lecture Notes in Computer Science*, pages 323–345. Springer-Verlag, 2000.
7. Charles Lakos and Glenn Lewis. A catalogue of incremental changes for coloured Petri nets. Technical report, Department of Computer Science, University of Adelaide, 1999.
8. Glenn Lewis. *Incremental specification and analysis in the context of coloured Petri nets.* PhD thesis, University of Hobart, Tasmania, 2002.
9. Barbara Liskov and Jeannette M. Wing. Family values: A semantic notion of subtyping. Technical report, Pittsburgh, PA, USA, 1992.
10. Barbara Liskov and Jeannette M. Wing. A new definition of the subtype relation. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, pages 118–141, London, UK, 1993. Springer-Verlag.
11. Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. on Programming Languages and Systems*, 16(6):1811–1841, 1994.
12. Alfred Sanogo. *Raffinement des réseaux de Petri colorés.* PhD thesis, Université Paris 13, 2012.

Petri Nets-Based Development of Dynamically Reconfigurable Embedded Systems

Tomáš Richta, Vladimír Janoušek, Radek Kočí

Brno University of Technology, Faculty of Information Technology,
IT4Innovations Centre of Excellence
Božetěchova 2, 612 66 Brno, The Czech Republic
{irichta,janousek,koci}@fit.vutbr.cz

Abstract. This paper deals with the embedded systems construction process based on the system specification modeled as a set of Petri nets. Modeling of the system starts with Workflow Petri Nets specification describing the main modules and processes within the system. Workflow model is then transformed to the multilayered Reference Nets structure, that is finally used for the code generation of the target system. The main goal of our approach is to provide for dynamic reconfigurability of the system deployment according to the changes within its specification. Dynamic reconfigurability means the possibility of system changes within its runtime. This is achieved by the decomposition of the whole functionality of the system to small interpretable pieces of computation. This approach also introduces several layers of reconfigurability using different translation rules operating on each layer. The heart of the system lies within the reference nets hosting platform called Petri Nets Operating System (PNOS) that includes the Petri Nets Virtual Machine (PNVM) that performs the very Reference Nets interpretation.

Keywords: workflow modeling, reference nets, embedded systems, model-based software engineering, code generation, model transformation

1 Introduction

Control systems are important border technology lying between the physical and information world. The whole control process is described as a control loop that consists of reading data from sensors and triggers a number of actuators installed within the physical environment controlled by the system. Most of the control systems are constructed using a set of programmable logic controllers with appropriate suitable software installation. At a higher level of abstraction, programmable logic controller and its software installation could be seen as an embedded system. In this paper the considered target platform for system installation is set of minimalistic and low energy consumption hardware devices, e.g. Atmel microcontrollers equipped with wireless transmission modules that are often used within Wireless Sensor Networks (WSN).

A control system implementation could be divided into the hardware and software part. The hardware part starts with selection of the proper set of modules and its installation within the physical environment, including the sensors and actuators attachment. When there are multiple controllers, the hardware part must also take into account the communication among them. The software part then follows with the programming and installation of each control unit with appropriate application that controls hardware. The main purpose of this paper is to describe the software part of this construction process with the focus on dynamic reconfigurability of the resulting system using executable models and model continuity approach. The reconfigurability is necessary for the ability of the system to adapt itself to changes in environment and also to enable the system maintainer with the possibility to change the system behavior without the necessity of its complete destruction and reconstruction.

Because there is strong demand on proper coverage of the system complexity at the beginning of the construction process, there is a need for suitable description tools that preserve the user requirements semantics. During the system lifetime there are also strong demands on its dynamic reconfiguration according to the new requirements and also according to the changes within the physical environment. The dynamic system specification change and following reconfiguration requirements are not easy to satisfy. Within this paper we introduce our solution of the described problem using the Workflow Petri Nets model [1] and MULAN-like multilayered Reference Nets control system structure[5], which is constructed according to the workflow model and then translated into the executable form. The system prototype then runs within the target platform simulator that deduces the requirements for the hardware part of system installation. The main characteristics of the system - its dynamic reconfigurability - is based on the ability of nets to migrate among places as tokens, which was inspired by [5]. The new or modified nets could be sent over another nets to its target place to change the system behavior. Within our solution, the nets are maintained by Petri Nets Operating System (PNOS) and interpreted using the Petri Nets Virtual Machine (PNVM).

Next two sections describe used formalisms and the reconfigurable system architecture. The following section describes the whole system development process using a running example, and the last section contains the evaluation and conclusion.

2 Formalisms and Tools

2.1 Workflow Management

Will van der Aalst defined the way to construct workflow models using Petri Nets[1]. His work is also well formally defined and so the workflow models could be used for the processes verification and validation purposes. The way of modeling the system in this way is also similar to the BPMN workflow models, so it could be easily used by the business process modeling experts. For that reason we decided to use the YAWL notation[2] and Workflow Nets formalism[1] in

the beginning of the embedded control system construction process. There are two main concepts from this theory that we use at the moment - basic transition categories (AND-split, AND-join, OR-split and OR-join) and the concept of workflow subprocess.

2.2 Reference Nets

The second part of the system construction then consists of the transformation of Workflow Petri Nets into the multilayered reference Nets model of the system that comply with the nets-within-nets concept defined by Rudiger Valk [3] formalized as Reference Nets[4]. The problem of generating code from formal specification to the running prototype of target system is mainly based on the decomposition of the whole net to a set of subnets, which is also called partitioning problem. For this purpose we use similar concept to the MULAN architecture defined by Cabac et al.[5]. This architecture divides the model into four levels of abstraction - infrastructure, agent platform, agents, and protocols. Our architecture is very similar, we use layers for the infrastructure, platform, main processes and subprocess. Each of those layers is mapped to the target platform and the transformation is used for the code generation. The main goal of the architecture is to enable changes within the system specification during its run-time. This is mainly achieved by the platform, process and subprocess abstraction levels that specify the functionality of the system.

3 Reconfigurable Architecture

Reference Nets allow to construct a system hierarchically, in several levels. Such an idea is a basis of the MULAN (MultiAgent Nets) architecture developed by Cabac et al.[5]. Thanks to the nature of Reference nets, MULAN allows nets to migrate among places in other nets and thus it is possible to dynamically modify functionality of system components, specified by this kind of nets [5]. We use application-specific main processes and subprocesses, which are hosted on platform that is considered to be a part of the operating system of the node, PNOS (Petri Nets Operating System). The multi-layered nature of the system and responsibilities of particular levels are described in Figure 1.

The main part of the the system is installed over the hardware as a PNOS kernel with platform net, that are both able to host other nets. Each platform then hosts some number of main processes nets that hosts subprocesses. The whole communication is performed by sending messages using serial link. There is also theoretical possibility for subprocesses to contain other subprocesses etc. But presented example does not cover this.

The PNOS contains PNVM (Petri Net Virtual Machine) which interprets Petri Nets that are installed in the system in the form of a bytecode called Petri Nets ByteCode (PNBC). PNOS also provides the installed processes with the access to inputs and outputs of the underlying hardware that are connected

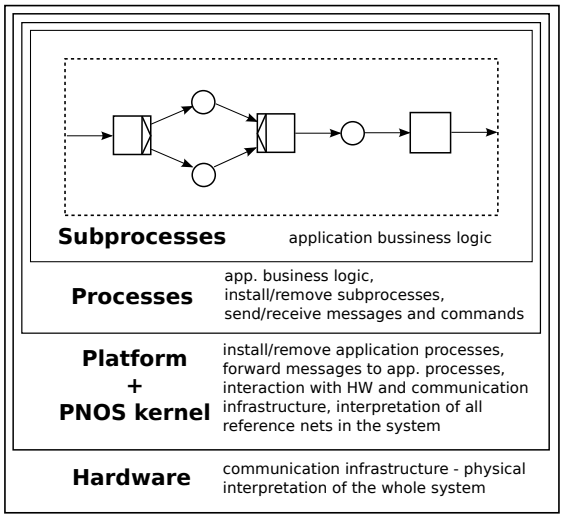


Fig. 1. System layers and their responsibility

to sensors and actuators, and also with the serial communication port that is connected to the wired or wireless communication module (e.g. ZigBee)[8].

The main net (first process) interpreted in PNOS is so called platform net. Platform net is responsible for interpretation of commands which are read from buffered serial line. These commands allow to install, instantiate, and uninstall other Petri nets. The Platform also allows to pass messages to the other layers, which are responsible for application-specific functionality. Since we need reconfigurability in all levels, the installation and uninstallation functionality is implemented in each level.

4 The Development Process

The whole process of system development is described in Figure 2. It starts with the specification of the main system workflow and its subprocesses. Resulting workflow model is then transformed to the layered architecture and might be further debugged using the Renew reference nets tool [6]. After this, the final set of Reference Nets is translated to Petri Nets ByteCode (PNBC) that is then used either for the target prototype simulation using SmallDEVS tool [7] and also to be transfered to the nodes of the system infrastructure. Here it serves as a reconfigurable part of the running system.

More detailed description of the whole PNOS architecture and functionality could be found in [8].

4.1 Running Example

As a running example, we use a subset of a home automation system. The home automation is partly based on the optimization of the energy consumption from multiple sources. There are diverse primary sources of the energy, but within

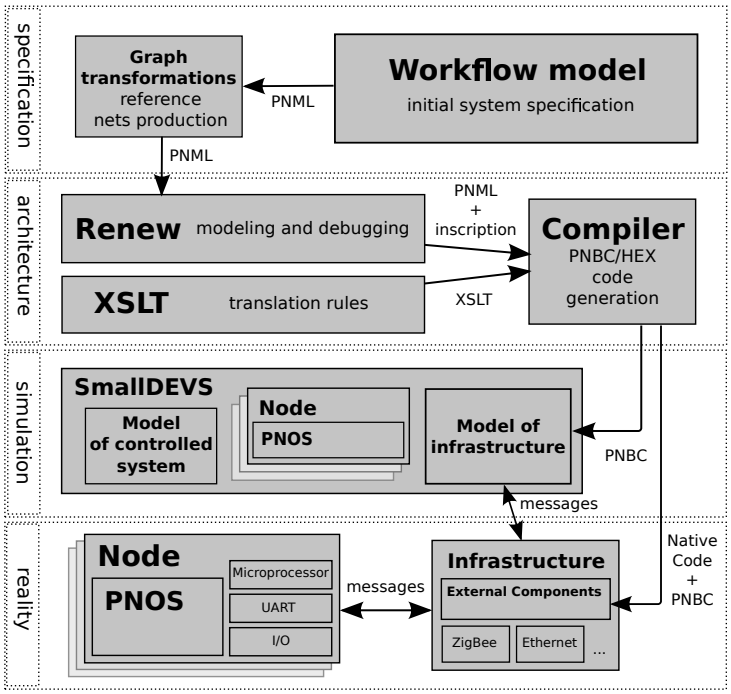


Fig. 2. System construction

our example we concern on the photothermic solar energy panels used for warm water and heating circuits energy supply. The home automation problem used as an example is described in more detail in [9], where also some preliminary ideas about the system design and code interpretation principles are proposed. In this paper we present refined and improved version of the design process and its evaluation.

Home automation process could be described as an workflow model using the Workflow Petri Nets described previously. Next section shows the workflow model and its description.

4.2 House Workflow Model

Within this section, the workflow model of the part of house automation system - the photothermic solar panel and hot water storage tank - is described, using the Workflow Petri Nets defined by Van der Aalst[1]. The Figure 3 describes two swimlines that represent two modules - solar panel and water tank. Each swimline consists of the main process of the module, that is constructed using a set of subprocesses. Within the solar panel module, there is a task of sending data and measure temperature subprocess. In the water tank module, there is a task of receiving the data and two subprocesses - measure temperature and adapt settings. Measure data subprocess and the receive task are connected with the adapt setting subprocess using the OR transition. Particular subprocesses descriptions are shown in next figures.

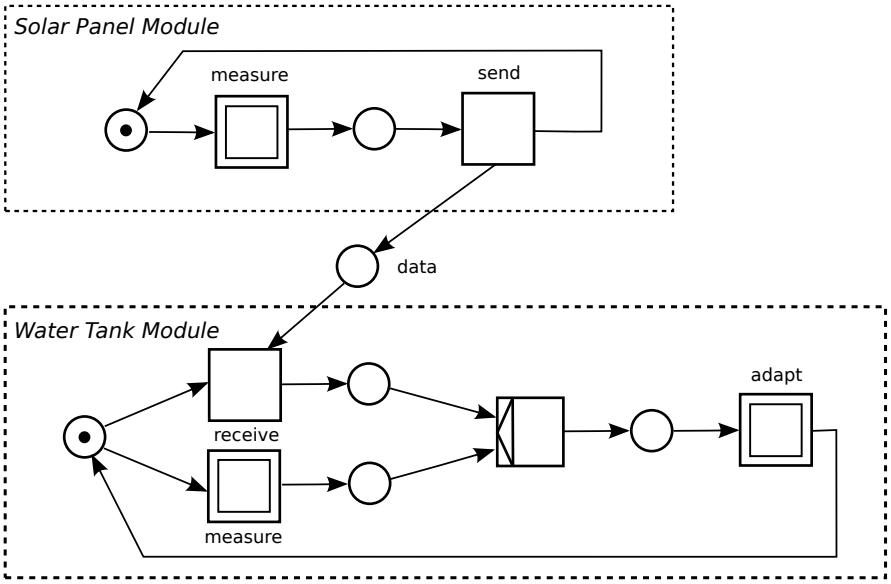


Fig. 3. House workflow example

In Figure 4 the measure subprocess was modeled also using the Workflow Petri Nets. It consists of two tasks - reading the data and converting it to the temperature value. Reading the data means getting the voltage from the input and the conversion means the necessary calculations to produce the human readable results.

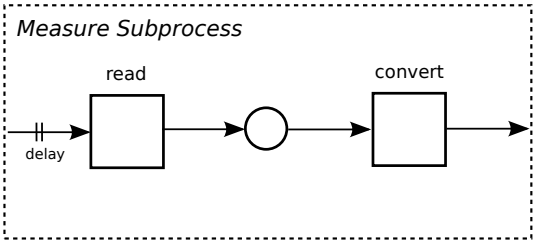


Fig. 4. Measure subprocess net

The other subprocess shown on Figure 5 consists of the task of temperatures reading and comparing them to use the result for the adequate reaction of the automation system. If there is higher temperature on the solar panel than within the water tank, corresponding circular pump is started to move the hot water form panel to the tank.

In this way the system specification is basically defined. But there are some other prerequisites, e.g. we need to know about the technical aspects of reading and writing the input/output data. This information should be obtained from the customer and must be included as a part of the PNOS system. At this moment, these rules are stored in a proprietary format alongside the nets specifications,

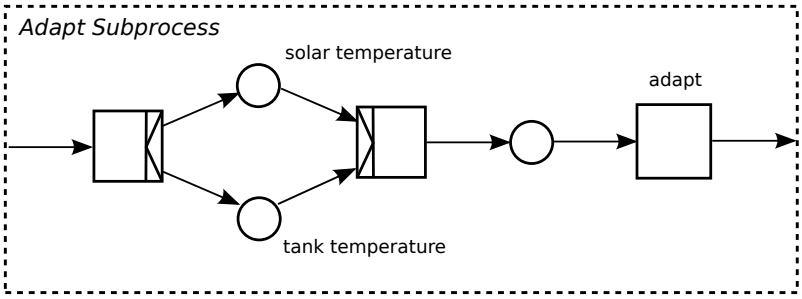


Fig. 5. Adapt subprocess net

but in future we plan to add them as a next layer of the system called drivers. The following section describes the derived four level reference nets architecture, which is produced from described workflow model. The process of conversion of workflow model into the multilayered Reference Nets system is done using some coarsely defined rules, but in future it should be based on formally defined translation rules.

4.3 Layered Reference Nets Architecture

The multilayered system architecture derivation starts with the subprocess nets. In Figure 6 there is the measure subprocess reference net derived from the measure subprocess. This net is constructed adding the initial and final uplinks and places. These uplinks serve as a starting and finishing transitions called from the main process of the module. There are also primitive system functions calls, that operate directly with the underlying operating system. Resulting value token is prepared and sent using uplink : *output*(). All the subprocess protocol nets are named using the name place and corresponding uplink.

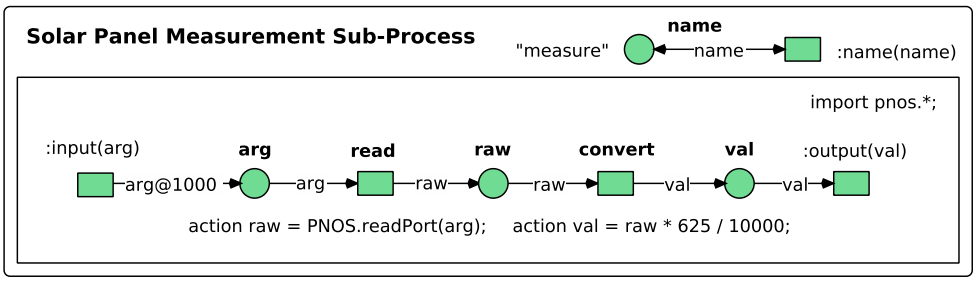


Fig. 6. Measure subprocess net

The solar panel main process described in Figure 7 is derived from the solar panel swimline in the workflow model. It consists of the place, where all the subprocess nets are stored and according to their names are called in particular order. Synchronization place is added between the subprocess protocol nets calls

matching the solar panel main process swimline place. The name of the protocol net is derived from the name of the workflow subprocess, and it is not necessary to be human readable.

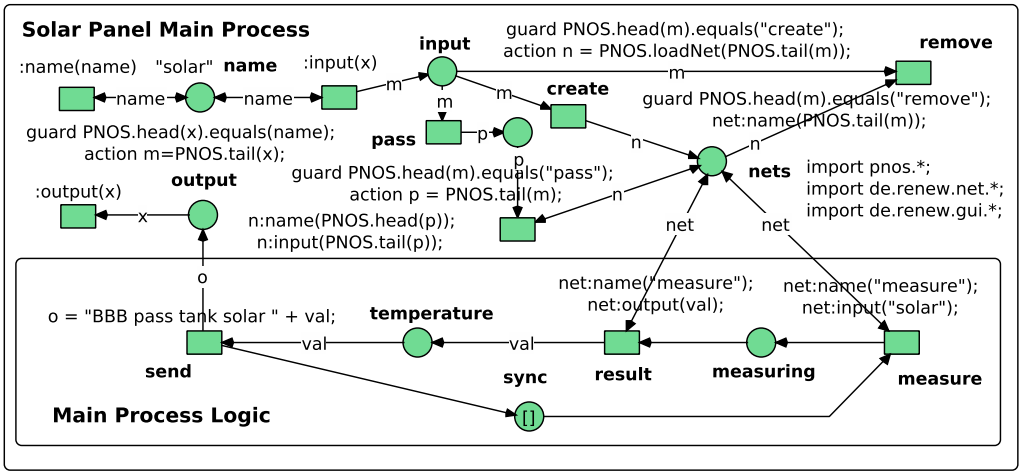


Fig. 7. Solar panel main process

The measurement subprocess protocol net has already been described, so the last net that remains is the settings adaptation subprocess protocol net. It is described in Figure 8 and communicates with the operating system calling the proper signals according to the decisions made in transitions.

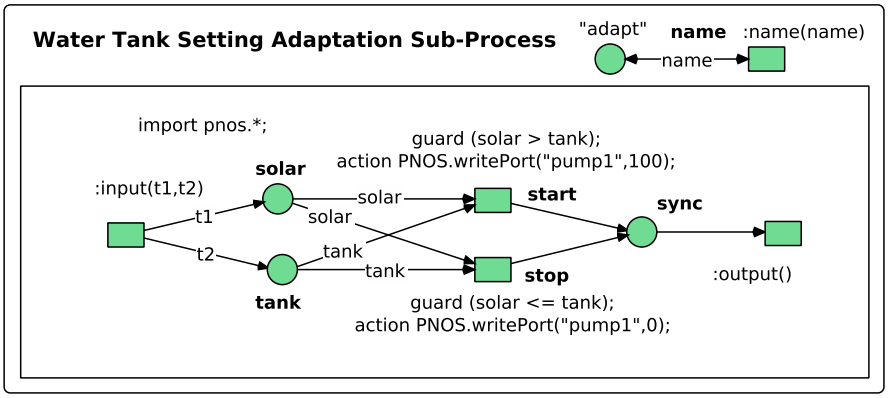


Fig. 8. Adapt subprocess net

The water tank main process reflects the main process in the workflow model. It calls all the subnets and performs the synchronization of subprocesses using two temperature places, that are then synchronized within the adapt subprocess. It is described in Figure 9.

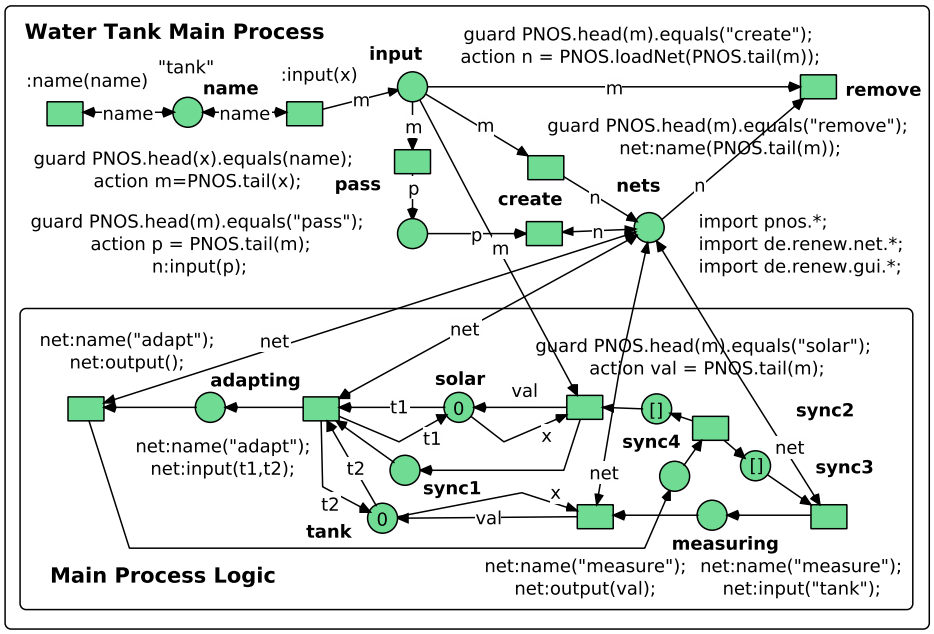


Fig. 9. Water tank main process

Above the last net, called infrastructure, there is a part of the underlying operating system called the platform net that describes the main required functions of the operating system needed by the application processes installed on it. The platform net is shown in Figure 10.

Finally the infrastructure layer, that is derived from the main workflow process description, is shown in Figure 11. In our example, it is very simple. Each swimline represents one place, where the module for hosting the platform, main process and protocols will be placed. The communication between the two subprocesses seated in different swimlines is represented here as an communication transition, that should internally call the final transition of the send task, that means the : *output*() downlink and the initial transition of the receive task, that means the : *input*() downlink. Those transitions are part of the platform layer and are propagated to the subprocesses nets.

4.4 Code Generation

Generally, in our approach, each layer of the system can be compiled to target code independently. There are two possibilities: first - the target code can be the native code of the controller processor, and second - target code is a bytecode that is interpreted by some virtual machine. Regardless on the way the code is generated, all the abstraction levels communicate with each other using uplinks and downlinks. The difference is, that levels deployed as interpreted bytecode are more flexible and dynamically changeable than the compiled ones. It is because such a modification needs a heavy compiler and (possibly) over-the-air programming of the node, that consumes a lot of energy. On the other hand

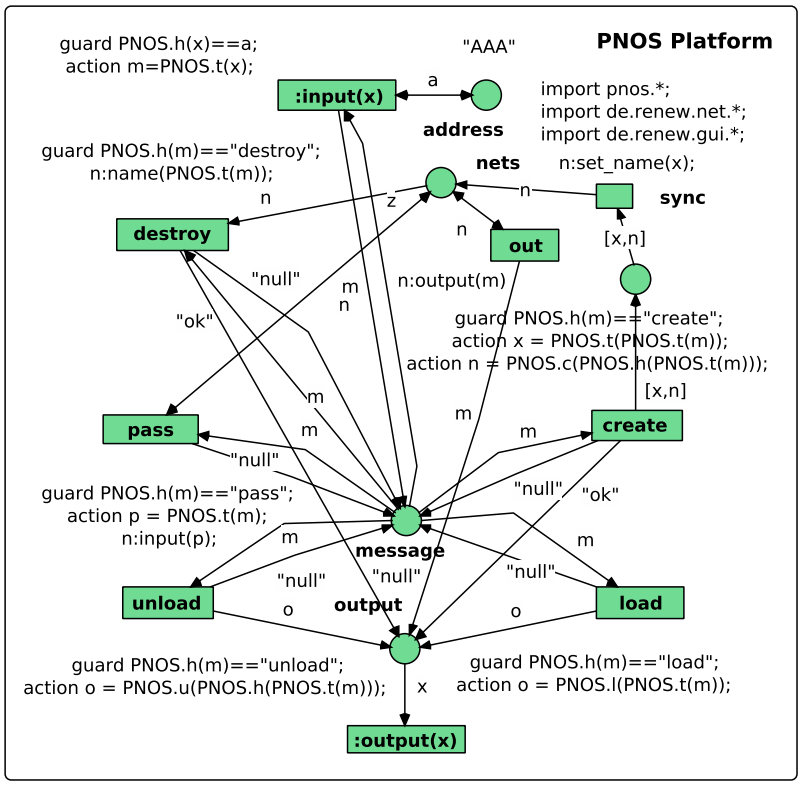


Fig. 10. Platform net

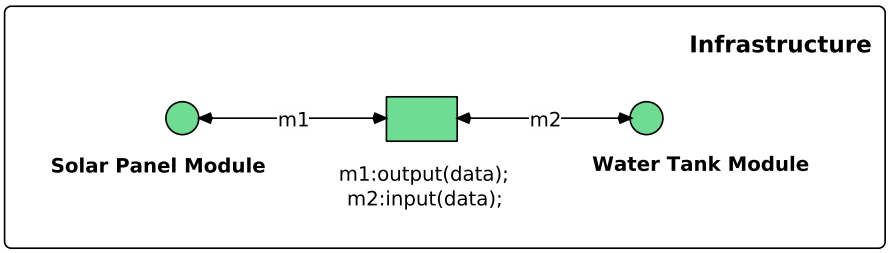


Fig. 11. Infrastructure net

it is possible to send the bytecode to the node as data. It thus allows for very high level of dynamic reconfigurability in the system runtime. E.g. new version of the measure subprocess is produced, then the corresponding Reference Net is derived and proper bytecode is generated. Finally the new version of the measure net is sent to the relevant node, and installed by its platform net.

We currently use the virtual machine and bytecode for all Petri Nets-based code. The only part which is implemented natively, is the PNOS kernel, including PNVM [8]. The example of bytecode follows. It represents simple net that reads data from some sensor and produces relevant output (the net is depicted in Figure 6). In fact, it is a human-readable version of the bytecode. In this representation, numbers are represented as a text and also some spaces and line

breaks are added. This means that the contents of the code memory is a bit more condensed. Each byte of the code is either an instruction for PNVM, or data.

```
(Nmeasure
 (measure)
 (arg/raw/val/name)
 (Uoutput(val)()(P3(B1)(V1)))
 (Uinput(arg)()(Y1(B1)(V1)(I1000)))
 (Uname(name)()(P4(B1)(V1)))
 (I(O4(B1)(S1)))
 (Tread(arg/raw)
  (P1(B1)(V1))
  (A(:(V2)(r(V1))))
  (O2(B1)(V2)))
 (Tconvert(raw/val)
  (P2(B1)(V1))
  (A(:(V2)(/(*(V1)(I625))(I10000))))
  (O3(B1)(V2))))
```

The bytecode contains symbols definitions and places definitions, allowed by a code for each uplink (U), initialization (I), and each transition (T). Each transition description consists of preconditions (P), guard (G), action (A) and postconditions (O), in a form of instructions for the PNVM. Each data element is a tuple consisting of a type and a value. Variables are declared as a part of transition code and identified by indexes. When the code of the net template is loaded to code memory of the PNVM, it is indexed in order to allow PNVM to quickly access particular parts of the code, especially places declarations, the uplinks and the transitions code. When the net template is instantiated, a specific part of runtime memory is allocated according to number of places. At the same time, the net transitions are scheduled for execution. Execution of a transition consists of reading its bytecode and attempting to satisfy all preconditions, downlinks and guards using recursive backtracking algorithm.

In guards and actions of transitions it is possible to call primitive operations of the underlying PNOS. Those operations are available in the Reference Nets inscription language as `PNOS.`*operation*, e.g., `PNOS.readPort("solar1")` reads data from virtual port named solar1, `PNOS.writePort("pump1",100)` writes value to the virtual port named pump1, `PNOS.h(m)` gets first space-separated substring from string m, and `PNOS.t(m)` returns the rest of the string m without the first substring.

Those primitive operations are directly mapped to the corresponding bytecode. We use a subset of the Reference Nets inscription language here. It works only on integers and strings as values with corresponding set of basic operations.

The important feature of the system is its reconfigurability. It is based on operations of the operating system that are designated for manipulations with nets (in the form of PNBC) and their instances. Nets could be sent to a node as a part of the command for its installation. The command is executed by Platform

net. Using other commands, the platform can instantiate a net, pass a command to it, destroy a net instance and unload a net template - see Figure 10. The PNOS Platform functionality is described in more detail in [8].

4.5 Simulation in SmallDEVS

PNOS-based nodes can be simulated in SmallDEVS environment [7], together with simulation models of sensors and actuators connected to the controlled physical process, as well as with simulation model of communication infrastructure. While Renew is used for application business logic debugging, SmallDEVS is used for realistic simulation of the system with its surroundings. Execution steps delays are incorporated to the simulation model in order to make simulation as realistic as possible. Statistics gathered from simulation experiments can be used for verification purposes and also can support decisions about type of hardware for target system implementation. Hardware-In-the-Loop simulation is also possible.

5 Evaluation

For the testing purposes we use the Arduino and Raspberry Pi hardware platforms with XBee modules for wireless communication. The Arduino is enabled with the ATmega328P chip that introduces some important restrictions to the implementation. The main one is the 2kB SRAM memory that makes extensive use of direct Petri Nets interpretation very difficult. There is a strong limitation for the number of nets and also for the complexity of problems solved. For that purposes we consider now for further testing of the system to use the Raspberry Pi platform, that offer much more memory for the interpretation purposes. The energy consumption of the ARM could be reduced by underclocking, that is part of our future work plans.

PNVM/PNOS prototype has been implemented in Smalltalk. The implementation resembles the way how it will be implemented in pure C language in order to make final implementation easy. Up to now, we do not have C implementation ready, because we are still doing minor improvements to the reference Smalltalk version. Nevertheless, the automated generation of C version of PNVM/PNOS is planned for very near future.

With the hardware limitations in mind, we have tested the PNVM/PNOS prototype with a model containing the platform net and other three simple nets (9 transitions in all nets), that are loaded and instantiated successively. The code of nets occupies 718 B, 679 B, 147 B, and 115 B, what is 1659 B of total used memory for code. The simulation generates 4 net instances, containing 14 places. The number of tokens is up to 31, and needs 1547 B of object memory. The history of memory occupation is shown in Figure 12. Peaks in the graph corresponds to receiving a net via serial line and its loading to code memory.

To investigate the time consumption of the simulation, we measured the time needed for each step execution. It comprises evaluation of all transitions in all

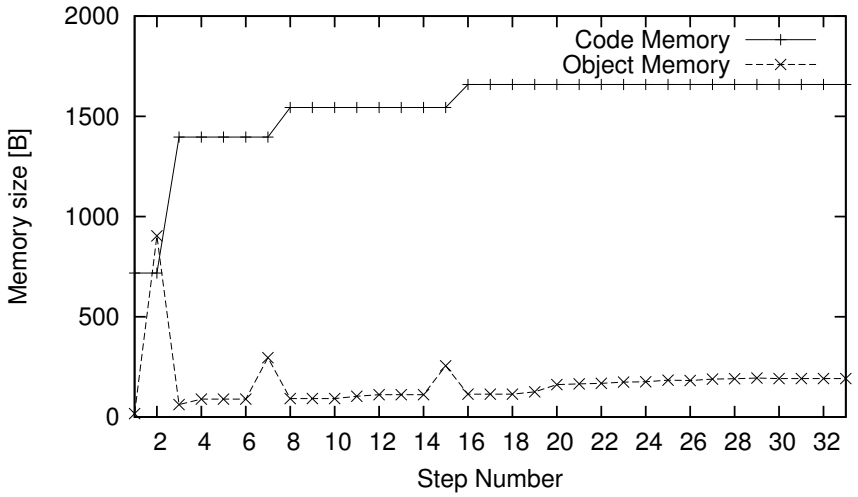


Fig. 12. Memory usage

ten instances. The simulation was executed for 50 times to get average step duration.

The history of simulation steps durations is shown in Figure 13. We can see, that the duration increases depending on number of instances because the number of transitions is increasing. Peeks in the graph correspond with net loading, net instantiation, and uplink execution. These experiments has been done on contemporary desktop computer. On Raspberry Pi the step duration is about 100 times higher, because of slower CPU and slower access to the memory. Nevertheless, we suppose that C version of the PNVM/PNOS for Raspberry Pi will run reasonably faster which will make Arduino and Raspberry Pi platforms well usable for Petri nets-specified control systems.

6 Related Work

The use of high-level languages, especially Petri Nets, allows to build and maintain control systems in a quite fast and intuitive way. There are many approaches to relate high-level languages with embedded devices or microcontrollers. One kind of that approaches is applicable in systems with not very limited resources. For example, Java can be used as a high-level language and works on architecture which can be successfully used in embedded systems [10]. To control robot application, hierarchical Petri Nets are used for middleware implementation in a RoboGraph framework [11]. Another approaches are focused to the devices with limited resources. They obviously use high-level languages or models for system design and the implementation is generated, usually into the C code. An examle is a usage of Timed Petri Nets for the synthesis of control software for embedded systems by generating C-code [12] or Sequential Function Charts [13]. All these approaches allow to design systems using high-level languages or models, but they either do not preserve models in the system implementation,

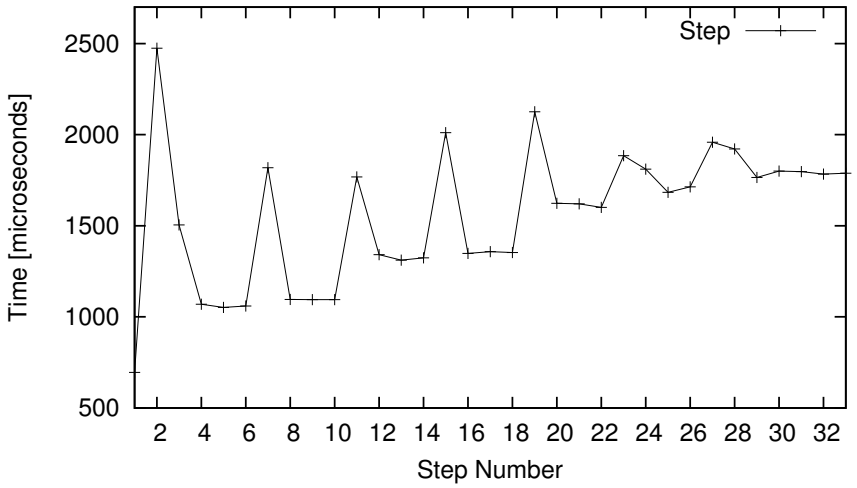


Fig. 13. Time overhead of simulation steps

or are not applicable for systems with limited resources. The approach presented in this paper allows not only for design of systems with limted resources, but also for systems implementation using a high-level language, particularly the Nets-within-Nets formalism, allowing for the dynamic reconfigurability.

7 Conclusion

In this paper, we described the process of system construction based on Petri Nets-based models transformations and target prototype code generation. This process starts with the workflow model defined according to the Van der Aalst's WF-Nets that describe the functionality of the system from the customers point of view. This model is then transformed to the multilayered architecture based on Reference Nets formalism. Each layer of the architecture is then translated to the specific target representation. The main part of the system is translated to the Petri Nets ByteCode (PNBC), that is interpreted by the Petri Nets Virtual Machine (PNVM) that is part of the Petri Nets Operating System (PNOS) which forms the remaining part of the system.

The whole system reconfigurability is based on the possibility of PNBC net replacement with the new version where the interpretation after reinstalling starts to perform the new version of the process. In the current version, the reconfigurability is considered to work on the granularity of processes and subprocesses. In further research, we plan to focus on more fine grained reconfiguration, including the platform primitive operations, and also on the processes migration.

Acknowledgment

This work has been 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. Van der Aalst, W.M.P., Van Hee, K.M. 2002. Workflow Management: Models, Methods, and Systems. IT press, Cambridge, MA.
2. Van der Aalst, W.M.P., Ter Hofstede, A.H.M. 2005. YAWL: yet another workflow language. Inf. Syst. 30, 4 (June 2005), 245-275.
3. Valk, R. 1998. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In Proceedings of the 19th International Conference on Application and Theory of Petri Nets (ICATPN '98), Jrgen Desel and Manuel Silva (Eds.). Springer-Verlag, London, UK, 1-25.
4. Kummer, O. 2001. Introduction to petri nets and reference nets. SozionikAktuell 1:2001 / Rolf von Lüde, Daniel Moldt, Rüdiger Valk (Hrsg.).
5. Cabac, L., Duvigneau, M., Moldt, D., Rölke, H. 2005. Modeling dynamic architectures using nets-within-nets. In Proceedings of the 26th international conference on Applications and Theory of Petri Nets (ICATPN'05), Gianfranco Ciardo and Philippe Darondeau (Eds.). Springer-Verlag, Berlin, Heidelberg, 148-167.
6. Kummer, O., Wienberg, F., Duvigneau, M., Köhler, M., Moldt, D., and Rölke, H. 2003. Renew - the Reference Net Workshop. Tool Demonstrations. Eric Veerbeek (ed.). 24th International Conference on Application and Theory of Petri Nets (ATPN 2003).
7. Češka M., Janoušek V., Vojnar T. PNtalk - A Computerized Tool for Object Oriented Petri Nets Modelling. Lecture Notes in Computer Science, vol. 1333, DE, p. 591-610, ISBN 3-540-63811-3, ISSN 0302-9743, 1997.
8. Richta, T., Janoušek, V. 2013. Operating System for Petri Nets-Specified Reconfigurable Embedded Systems, To appear in: Proceedings of the 14th Computer Aided Systems Theory, Las Palmas de Grand Canaria, LNCS, Springer Verlag.
9. Richta, T., Janoušek, V., Kočí, R. 2012. Code Generation For Petri Nets-Specified Reconfigurable Distributed Control Systems, In: Proceedings of 15th International Conference on Mechatronics - Mechatronika 2012, Praha, CZ, FEL Č, 2012, s. 263-269, ISBN 978-80-01-04985-3.
10. Pizlo, F., Ziarek, L., et al. 2010. High-level programming of embedded hard real-time devices. In Proceedings of the 5th European conference on Computer systems, ACM New York, 69-82.
11. Fernandez, J.L., Sanz, R., Paz, E., Alonso, C. 2008. Using hierarchical binary Petri nets to build robust mobile robot applications: RoboGraph. In IEEE International Conference on Robotics and Automation. 1372-1377.
12. Rust, C., Stappert, F., Kunnemeyer, R. From Timed Petri Nets to Interrupt-Driven Embedded Control Software. In International Conference on Computer, Communication and Control Technologies (CCCT 2003), Orlando, Florida, USA. p. 6.
13. Bayo-Puxan, O., Rafecas-Sabate, J., Gomis-Bellmunt, O., Bergas-Jane, J. 2008. A GRAFCET-compiler methodology for C-programmed microcontrollers, In Assembly Automation, Vol. 28 Iss: 1, Emerald Group Publishing. 55-60.

Decomposing Replay Problems: A Case Study

H.M.W. Verbeek and W.M.P. van der Aalst

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

Abstract. Conformance checking is an important field in the process mining area. In brief, conformance checking provides us with insights how well a given process model matches a given event log. To gain these insights, we typically try to *replay* the event log on the process model. However, this replay problem may be complex and, as a result, may take considerable time. To ease the complexity of the replay, we can decompose the given process model into a collection of (smaller) submodels, and associate a (smaller) sublog to every submodel. Instead of replaying the entire event log on the entire process model, we can then replay the corresponding sublog on every submodel, and combine the results. This paper tests this *divide-and-conquer* approach on a number of process models and event logs while using existing replay techniques. Results show that the decomposed replay may indeed be faster by orders of magnitude, but that success is not guaranteed, as in some cases a smaller model and a smaller log yield a more complex replay problem.

1 Introduction

In the area of process mining, we typically distinguish three different fields: *discovery*, *conformance checking*, and *enhancement* [1]. The discovery field is concerned with mining an event log for a process model (for the remainder of this paper, we assume that these process models are represented using Petri nets). We are given an event log, and we try to come up with some process model that nicely represents the observed behavior in the event log. The conformance checking field is concerned with checking whether an event log matches a process model. We are given an event log and a process model, and we try to provide insights how well these two match. The enhancement field is concerned with enhancing a process model using an event log. We are given an event log and a process model, and we copy data as found in the event log into the process model. A popular enhancement technique is to add durations to a process model based on timestamps in the log. This allows us to detect bottlenecks in a process model. This paper focuses on the conformance checking field [10].

As mentioned, for conformance checking, it is vital that we can match the event log with the process model. A popular approach for matching both is to replay the entire event log on the process model. Every trace of the event log is replayed in the process model in such a way that mismatches are minimized in

some way. Of course, our aim is to match every event in the trace to a possible action in the process model. However, this is not always possible. In some cases, we cannot match an event to any possible action in the process model, or vice versa, we cannot match an necessary action according to the model to an event in the log. The result of this matching process is an *alignment* [4] between every trace of the event log and the process model. A situation where the alignment cannot match an event to any action can be considered as a situation where we decided to skip a position in the log during replay. This is referred to as a *log move*: We handle the event in the log (by skipping it) but do not reflect this move in the model. Vice versa, a situation where the alignment cannot match an action to any event can be considered as a situation where we decided to skip the action. Such a situation is referred to as a *model move*. The remaining situation where an event and an action are matched is referred to as a *synchronous move*. By associating *costs* to these situations, we can obtain a best alignment by minimizing these costs. Based on this idea of associating costs, replay techniques have been implemented in the process mining tool ProM 6 [3].

Earlier work [5, 2] has shown that we can apply a *divide-and-conquer* approach to these cost-based replay techniques. Instead of replaying the entire event log on the entire process model, we first decompose the process model into a collection of submodels. Second, we create a sublog from the entire event log for every submodel. Behavior not captured by a submodel will be filtered out of the corresponding sublog, only behavior captured by the submodel will be filtered in. Third, we compute subcosts for every combination of submodel and sublog. Fourth and lasts, we replay every sublog on the corresponding submodel in such a way that the subcosts are minimized. It has been proven that the weighted accumulated subcosts are a lower bound for the actual overall costs [11]. Moreover, the fraction of fitting cases is the same with or without decomposition [11]. As a result, the accumulated subcosts is a lower bound for the actual costs.

Conceptually, the entire model can be regarded as a decomposition of itself into a single submodel, with a single sublog. Nevertheless, proper decompositions may exist as well. An example of a (most likely) proper decomposition is the maximal decomposition as described in [2, 11]. This maximal decomposition uses an equivalence class on the arcs in the process model to decompose the model into submodels. This equivalence class corresponds to the smallest relation for which the incident arcs of places, silent transitions, and visible transitions with a non-unique label are equivalent. As a result, the maximal decomposition results in submodels that only have visible transitions with unique labels in common. Nevertheless, other proper decompositions of a process model may exist (like [7]) which may be of interest for the decomposed replay problem.

Replaying all sublogs on the corresponding submodels has two possible advantages:

1. It may highlight problematic areas (submodels) in the model, that is, submodels with a bad fitness. This information can hence be used for diagnostic purposes.

2. It may be much faster than replaying the entire log on the entire model, while the fraction of fitting cases is the same for both. Obviously, this may save time while maintaining quality.

This paper focuses on the second advantage, that is on the possible speed-up of the entire replay when using divide-and-conquer techniques. As the overall model is bigger than any of its submodels, it is expected that the replay problem of the entire log is more complex than the replay problem of all sublogs. As a result, determining the actual costs may take simply way too much time, while determining a decent lower limit may take an acceptable amount of time. This paper takes four different process models with six corresponding different event logs, and several ways to decompose the process models into submodels, and checks whether determining the costs of more fine-grained decomposed replay problems are indeed faster than that of more coarse-grained decomposed replay problems. Furthermore, if it is indeed faster, it checks whether the obtained costs are still acceptable. Note that a technique that return a trivial lower bound (like 0) for the costs is very fast, but is not acceptable as we do not gain any insights by such a technique.

2 Preliminaries

This section introduces the basic concepts of process models and event logs as used in this paper. Furthermore, it details how we determine the cost associated by replaying an event log on a process model. For the remainder of this paper, we use $\mathcal{U}$ to denote the universe of labels.

A process model contains a labeled Petri net [9, 8] (P, T, F, l), where P a set of places, T is a set of transitions such that $P \cap T = \emptyset$, $F \subseteq (T \times P) \cup (P \times T)$ a set of arcs, and $l \in (T \not\to \mathcal{U})$ a partial function that maps a transition onto its label. For function l, we use $\mathrm{dom}(l)$ to denote the set of transitions that are mapped onto some label, and $\mathrm{rng}(l)$ to denote the set of labels that are mapped onto by some transition. A marking M of a net (P, T, F, l) is a multiset of places, denoted $M \in \mathcal{B}(P)$. The input set of a place or transition $n \in P \cup T$ is denoted $\bullet n$ and corresponds to the set of all nodes that have an arc going to n, that is, $\bullet n = \{n' | n' F n\}$. In a similar way, the output set is defined: $n\bullet = \{n' | n F n'\}$. Transition $t \in T$ is *enabled* by marking M in net $N = (P, T, F, l)$, denoted as $(N, M)[t\rangle$, if and only if M contains a token for every place in the input set of t, that is, if and only if $M \leq \bullet t$. An enabled transition $t \in T$ may *fire*, which results in a new marking M' where a token is removed from every place in the input set of t and a token is added for every place in its output set, that is, $M' = M - \bullet t + t\bullet$. We use $(N, M)[t\rangle(N, M')$ to denote that transition t is enabled by marking M, and that firing transition t in marking M results in marking M'. Let $\sigma = \langle t_1, t_2, \ldots, t_n \rangle \in T^*$ be a sequence of transitions. $(N, M)[\sigma\rangle(N, M')$ denotes that there is a set of markings $M_0, M_1, \ldots, M_n$ such that $M_0 = M$, $M_n = M'$, and $(N, M_i)[t_i\rangle(N, M_{i+1})$ for $0 \leq i < n$. A marking M' is *reachable* from a marking M if there exists a σ such that $(N, M)[\sigma\rangle(N, M')$. A transition $t \in \mathrm{dom}(l)$ is called *visible*, a transition $t \notin \mathrm{dom}(l)$ is called *invisible*. An occurrence of a

visible transition t corresponds to an observable activity $l(t)$. We can project a transition sequence $\sigma \in T^*$ onto its sequence of observable activities $\sigma_v \in \mathcal{U}^*$ in a straightforward way, where all visible activities are mapped onto their labels and all invisible transitions are ignored. We use $(N, M)[\sigma_v \rhd (N, M')$ to denote that there exists a $\sigma \in T^*$ such that $(N, M)[\sigma\rangle(N, M')$ and σ is mapped onto σ_v.

Furthermore, a process contains an initial state and a final state, that is a process model $\mathcal{P}$ is a triple (N, M_0, M_n), where $N = (P, T, F, l)$ is a labeled Petri net, $M_0 \in \mathcal{B}(P)$ is its initial marking, and $M_n \in \mathcal{B}(P)$ is its final marking. The set of *visible* traces for a process model $\mathcal{P} = (N, M_0, M_n)$, denoted $\phi(\mathcal{P})$, corresponds to the set of sequences of observable activities that start in the initial marking and end in the final marking, that is, $\phi(\mathcal{P}) = \{\sigma_v | (N, M_0)[\sigma_v \rhd (N, M_n)\}$.

An event log [1] L is a multiset of *traces*, where a trace is a sequence of activities. Thus, if $A \subseteq \mathcal{U}$ is the set of activities, then $\sigma \in A^*$ is a *trace* and $L \in \mathcal{B}(A^*)$ is an *event log*. An event log L can be projected onto some set of activities A', denoted $L{\restriction_{A'}}$, in a straightforward way: All events not in A' are filtered out, and all events in A' are filtered in.

The *replay problem* [4] for an event log L and a process model $\mathcal{P}$ can now be described as finding, for every trace $\sigma_L \in L$, the transition sequence $\sigma_\mathcal{P} \in T^*$ for which its corresponding sequence of observable activities $\sigma_{\mathcal{P},v} \in \phi(\mathcal{P})$ matches σ_L *best*. To determine which transition fits a trace best, we *align* σ_L and $\sigma_{\mathcal{P},v}$ and associate *costs* to every misaligned transition and/or event. A misaligned transition means that we need to execute a visible transition in the process model that is not reflected in the event log, and is called a *model move*. A misaligned event means that an activity has been executed and logged that is not reflected by a corresponding visible transition in the process model, and is called a *log move*. An aligned transition-event pair means that both the event log and the process model agree on the next activity, and is called a *synchronous move*. By associating costs to model moves and log moves, and by minimizing the costs, we can determine the transition sequence from the process model that best fits a trace from the event log.

For the *decomposed replay problem* for an event log L and a process model $\mathcal{P}$, we first decompose the process model into a collection of smaller process models $\{\mathcal{P}_1, \ldots, \mathcal{P}_M\}$. Second, we map the costs for replaying the entire log on the entire net to costs for every smaller process model $\mathcal{P}_i = ((P_i, T_i, F_i, l_i), M_{0,i}, M_{n,i})$. In earlier work [11], we have shown that these costs can be mapped in such away that the accumulated costs for the decomposed replay problem is a lower bound for the costs of the original replay problem. Third, we filter the log for every smaller process model into a smaller log $L_i = L{\restriction_{\mathrm{rng}(l_i)}}$. Fourth, we replay every smaller log L_i on the corresponding smaller process model $\mathcal{P}_i$, using the adapted costs. Fifth and last, we accumulate the resulting costs into a single costs, which is a lower bound for the actual costs.

In [11], we have shown how to decompose a process model $\mathcal{P} = ((P, T, F, l), M_0, M_n)$ in such a way that the decomposition is maximal. For this maximal decomposition, we introduce an equivalence class on the arcs of F. Arcs will

Table 1. Characteristics of process models

Process model	Transitions	Places	Arcs	Labels
RepairExample	12	12	26	8
A32	32	32	74	32
Bpic2012A	11	14	28	10
Bpic2012	58	44	124	36

end up in the same submodel $\mathcal{P}_i$ if and only if they are equivalent, where this equivalence is defined as the smallest relation for which the following rules hold:

R1 An incident (input or output) arc of a *place* is equivalent to all incident arcs of that place.
R2 An incident arc of an *invisible transition* is equivalent to all incident arcs of that transition.
R3 An incident arc of a *visible transition with a non-unique label* (that is, there exist other transitions that have the same label) is equivalent to all incident arcs of all transitions with that label.

As a result of these rules, any place, any invisible transition, and any visible transition with non-unique label will be part of a single submodel only, whereas visible transitions with unique labels may be split over different submodels. As a result, only these visible transitions with unique label interface between the different submodels. In [11], we have proved that this decomposition preserves perfect replay: The entire event log L can be replayed perfectly (that is, no costs and/or mismatches) on process model $\mathcal{P}$ if and only if every sublog L_i can be replayed perfectly on submodel $\mathcal{P}_i$. Moreover, a trace perfectly fits the overall model if and only if its projection fits each of the submodels. Hence, the fraction of fitting traces can be computed precisely using any decomposition.

3 Case Study Setting

For the case study, we will use four different process models (RepairExample, A32, Bpic2012A, and Bpic2012) with six corresponding different logs (RepairExample, A32f1n00, A32f1n10, A32f1n50, Bpic2012A, and Bpic2012). Table 1 shows the characteristics of these models, whereas Table 2 shows the characteristics of the corresponding logs.

The RepairExample model comes with a single event log, which is typically used for demonstration (or tutorial) purposes. The A32 model comes with three event logs, which contain a varying amount of noise. The first log, A32f1n00, contains no noise ('0%'), the second log, A32f1n10, contains some noise ('10%'), and the third log, A32f1n50, contains much noise ('50%'). This model and these logs were used to test genetic mining algorithms on their ability to recreate the original model from noisy event logs. The Bpic2012A model and event log stem from the BPI 2012 Challenge log and originate from [6]. Note that this log is

Table 2. Characteristics of event logs

Event log	Cases	Events	Event Classes
RepairExample	1104	11,855	12
A32f1n00	1000	24,510	32
A32f1n10	1000	24,120	32
A32f1n50	1000	22,794	32
Bpic2012A	13,087	60,849	10
Bpic2012	13,087	262,200	36

Table 3. Replay results of event logs

Event log	Running time (in seconds)	Costs
RepairExample	0.25	0.197
A32f1n00	11	0.000
A32f1n10	17	0.993
A32f1n50	32	4.521
Bpic2012A	0.59	1.293
Bpic2012	480	14.228

a real-life log, which was obtained from a company. The Bpic2012A event log is obtained by filtering out all events that start with either "O" or "W", that is, it contains only the events that start with "A". The corresponding model was hand-made based on results of running the Transition System Miner on this filtered log (cf. [13]). The Bpic2012 model and log also stem from this challenge log. In contrast with the Bpic2012A log, the Bpic2012 log contains all events from the Challenge event log, and some more. These extra events allowed us to mine this log using a passage-based technique, of which the Bpic2012 model is a result [2].

Table 3 shows the results of running the replayer on the six event logs, where running times have been rounded to the two most significant digits (this is done throughout the paper). For details on the replayer and other Prom 6 plug-ins used in this case study, we refer to [14]. Note that the absolute values for these running times are not that important for this paper, as we only want to *compare* them. In the next section, we will compare these results to the results of running the decomposed replayer on these logs. Our goal will be to check whether the decomposed replayer returns still-acceptable costs in better times for certain decompositions.

4 Case Study Results

The first decomposition we will check for the results, is a *near-maximal* decomposition. In Section 2, we have detailed a maximal decomposition based on an equivalence class between arcs. In this maximal decomposition, it is possible

Table 4. Decomposed replay results of event logs using near-maximal decomposition

Event log	**Submodels**	**Running time** (in seconds)	**Costs**
RepairExample	6	(171%) 0.43	(99%) 0.196
A32f1n00	30	(12%) 1.3	(100%) 0.000
A32f1n10	30	(7%) 1.1	(45%) 0.444
A32f1n50	30	(4%) 1.2	(48%) 2.155
Bpic2012A	8	(378%) 2.2	(49%) 0.629
Bpic2012	12	DNF	

to have a submodel $((P_i, T_i, F_i, l_i), M_{0,i}, M_{n,i})$ that subsumes some other submodel $((P_j, T_j, F_j, l_j), M_{0,j}, M_{n,j})$ when it comes down to the labels, that is, $\text{rng}(l_i) \subseteq \text{rng}(l_j)$. A typical submodel where this might happen is the submodel that contains the source (or sink) place in case of a workflow net [12]. As every label of the subsumed submodel is contained in the subsuming model, every event of the 'subsumed' event log will be contained in the 'subsuming' event log. As a result, we can replay (almost for free) the 'subsumed' log while replaying the 'subsuming' log. For this reason, we add the subsumed submodel to every subsuming submodel (there may be more subsuming submodels), and do not replay the subsumed submodel on its own. The resulting decomposition is called a near-maximal decomposition.

Table 4 shows the results of replaying the event logs on the submodels as obtained by the near-maximal decomposition. To enable easy comparison, Table 4 also shows the reduction percentages for both running times and costs (for example, 0.43 seconds is 171% of 0.25 seconds). These results show that the A32 logs are replayed much faster than with the regular replay (though the costs are significantly lower), but that the other replays got worse, with the Bpic2012 log as most negative example. For this log, the decomposed replay simply did not finish ("DNF"), as it eventually ran out of resources (4 GB of memory). From these results, we conclude that decomposing a replay problem need not improve the time required for replay.

Figure 1 shows the submodel that was the bottleneck for the decomposed replay of the Bpic2012 log. This submodel contains 12 invisible transitions (the entire net contains 22, so more than half of these invisible transitions ended up in this submodel), of which 3 loop back, 5 source transitions, and 5 sink transitions. As a result, the replay of the corresponding sublog on this specific submodel is very complex: In any reachable state, there are at least 5 enabled transitions, and in many states there will be some invisible transitions enabled as well). It might help the replay if we restricted the unlimited freedom of these source transitions in some way. For this reason, we introduced a place that effectively restricts the number of tokens, say n, that may be present in the submodel. Initially, this place p contains n tokens. Any transition t in the original submodel is changed in the following way:

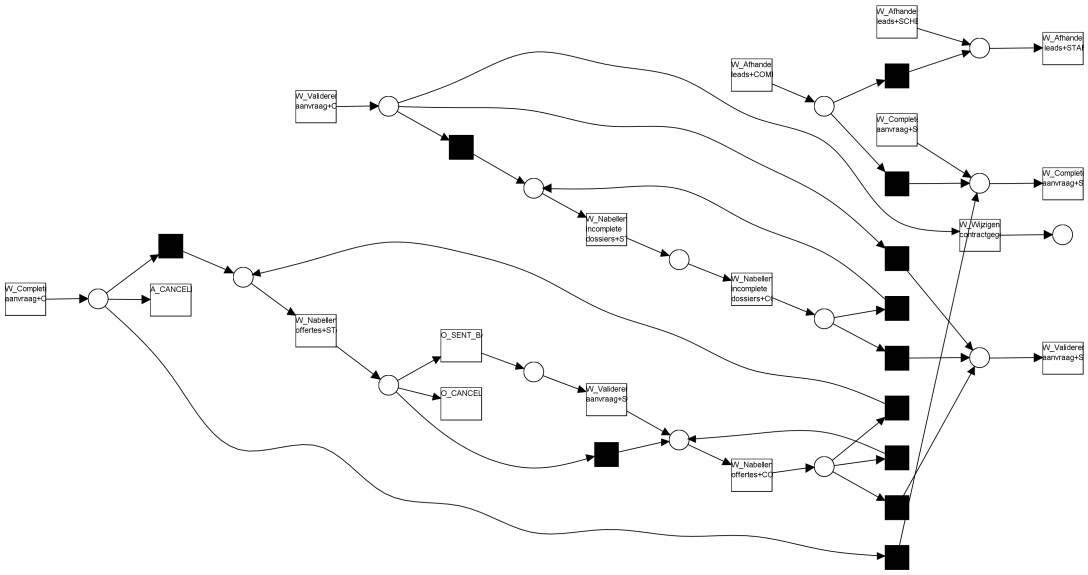


Fig. 1. Problematic submodel that results from near-maximal decomposition of BPIC2012

- If t has more incoming arcs than outgoing arcs, that is, if $|\bullet t| > |t\bullet|$, then x outgoing arcs from transition t to place p are added, where $x = |\bullet t| - |t\bullet|$.
- If t has more outgoing arcs then incoming arcs, that is, if $|t\bullet| > |\bullet t|$, then x incoming arcs from place p to transition t are added, where $x = |t\bullet| - |\bullet t|$.
- If t has as many incoming arcs as outgoing arcs, that is, if $|t\bullet| = |\bullet t|$, then nothing is added.

The exceptions to these rules are the arcs from source places and the arcs to sink places, as these arcs will not be counted. Reason for doing so is that otherwise all n tokens may end up in a sink place, which results in a dead submodel. As a result of applying these changes, every transition in the resulting net has as many incoming arcs as outgoing arcs (excluding source and sink places), and hence the number of tokens in every reachable marking (again excluding source and sink places) is constant, that is, equal to n. This amount of tokens n is determined in such a way that the execution of a source transition can be followed by the execution of a sink transition. Therefore, we will put a single token into the place for a source transition, and as many tokens as needed to fire (once) all other (non-source) transitions in the preset of this place. Figure 2 shows the resulting submodel, where the place in the middle (the one with the many incident arcs) is the place that restricts the otherwise unlimited behavior of the former source transitions. Given the structure of this submodel, this place contains initially a single token, which allows for only a single thread in this submodel during replay.

With this change made to the decomposition of the entire net (note that only the behavior of the problematic submodel was restricted in the way as

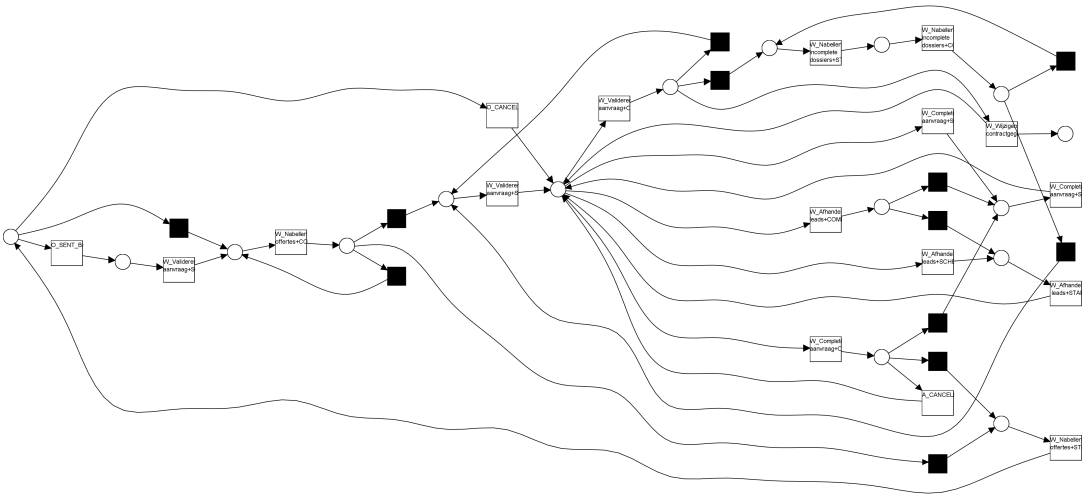


Fig. 2. Restricted problematic submodel

described, other submodels were unaffected), the decomposed replay finished in 470 seconds (98%), with costs of 8.722 (61%). The replay of the problematic submodel took 460 seconds (costs 5.210), which explains the better part of the replay time (please recall that for the decomposed replay we allowed 4 threads to be run simultaneously). Although the decomposed replay now finished, it finished in almost the same time as the regular replay with significant lower costs. Furthermore, by restricting the behavior of this submodel, we cannot claim anymore that the resulting costs are a lower bound for the actual costs. For this reason, we tried to reduce the number of invisible transitions instead.

An invisible transition with single input and single output can be reduced in such a way, that the behavior of the resulting model is not a restriction, but an extension of the behavior of the original model. In general, the single input place of the invisible transition will have input transitions and additional (not counting this invisible transition) output transitions. Likewise, its single output place will have additional input transitions (not counting this invisible transition) and output transitions. In general, a path from such an additional input transition to such an additional output transition will not be possible in the model, as the invisible transition works 'the wrong way'. However, all other paths are possible, like a path from an input transition to an output transition, and a path from an input transition to an additional output transition. Thus, if we merge the single input place and the single output place, and remove the invisible transition, than we only add behavior in the model. As a result, the replay costs might go down (which is safe for a lower bound), but cannot go up. Figure 3 shows the resulting submodel.

With this change made, the decomposed replay finished in 190 seconds (40%), with costs of 6.676 (47%). The replay of the problematic submodel took 140

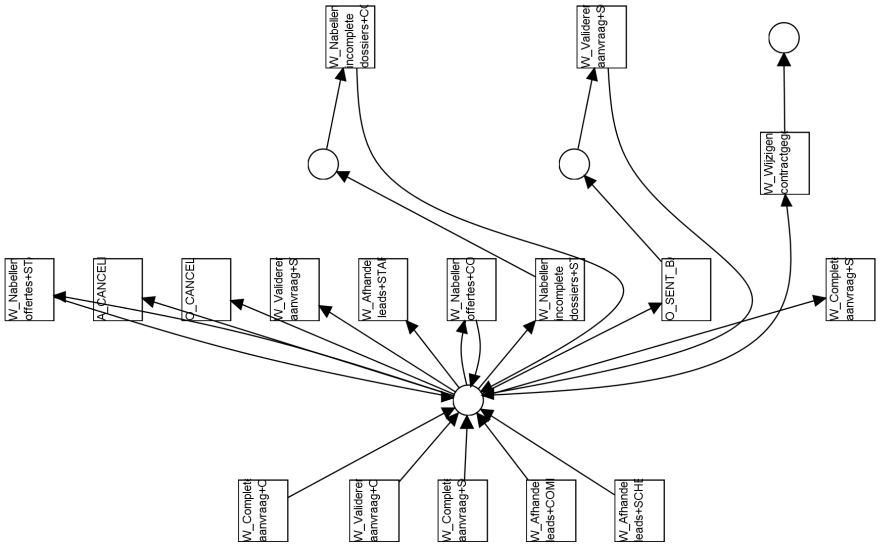


Fig. 3. Reduced problematic submodel

seconds (costs 3.163). The bottleneck submodel is now a different submodel, which requires a replay time of 180 seconds.

Obviously, the reduction of invisible transitions has led to a 60% decrease in running time (from 480 seconds to 190 seconds), but also to a 53% decrease in the computed costs (from 14.228 to 6.676). As we know that the decomposed replay returns a lower bound for the costs, this decrease in costs is okay, but perhaps we can do better by not reducing all invisible transitions. Possibly, there is some trade-off somewhere between the invisible transitions to reduce and the decrease in computed costs. For this reason, we again focus on the reduction of an invisible transition. Clearly, if (1) the single input place of an invisible transition has no additional output transitions and (2) the single output place of that invisible transition has no additional input transitions, then the reduction of this invisible transitions leads to no extra behavior. In contrast, if there would be 3 additional output transitions and 4 additional input places, then this reduction would lead to $3 \times 4 = 12$ possible new paths. If we want to retain as much of the original behavior in the reduced net, then we want to reduce invisible transitions for which the product of the number of additional output transitions and the number of additional input transitions is below some threshold. If this threshold equals 0, then no new paths are allowed, if it is sufficiently large (say, 100), then any reduction is allowed. To check the effect of this threshold, we have run the decomposed replay for a number of possible thresholds. Table 5 shows the results. Please note that the result of the reduction with some fixed threshold needs not be unique, the resulting net might be non-deterministic. We are aware of this, but want to check its effect anyway.

This table indicates that if we set the threshold to 32, that then all invisible transitions will be reduced (running time approx. the same and costs the same). Furthermore, this table indicates that, to get a good lower bound for the costs

Table 5. Effects of reducing invisible transitions on running time and computed costs

Threshold	**Running time** (in seconds)	**Costs**
32	(42%) 200	(47%) 6.676
16	(58%) 280	(50%) 7.091
8	(386%) 1900	(55%) 7.849

(if possible at all), we need to spend more running time than we would need for the original (not decomposed) replay problem: To obtain a lower bound for the costs that is slightly above half of the actual costs, we need to spend more than three times as much running time. As a result, we conclude that this reduction technique has its merits (it actually finishes with costs that are reasonable for the running time required), but that there is no significant gain in fine-tuning this technique.

A third option to cope with the problematic submodel could be to organize the submodels into submodels that are better suited for the replay at hand. Apparently, the replay techniques we are using have problems with certain submodels (many source transitions, many invisible transitions), so we should try to avoid generating such submodels. For this reason, we propose a slightly changed notion of equivalence, the only change being that the following rule is added:

R4 The i-th incoming arc of a *visible transition with unique label* is equivalent to the i-th outgoing arc of that transition, if both arcs exist.

Note that only the fourth rule on visible transitions with unique labels has been added. Also note that we now assume that some order exists among the incident arcs of such a transition, as we link the i-th incoming arc to the i-th outgoing arc, if possible. As a result, again, the result may be non-deterministic. Nevertheless, this approach may help in suppressing submodels with many source transitions, as the result of the fourth rule may be that a submodel with a visible sink transition t is merged with a submodel with a visible source transition t. We use the term *near-minimal* decomposition to refer to the decomposition that results from these adapted equivalence rules, as many submodels will be linked together using these new rules. Table 6 shows the results for this decomposition.

This table shows that we typically obtain good results (costs-wise) using the near-minimal decomposition, but that we sometimes use slightly more time. Furthermore, this table shows that for the BPIC2012A and BPIC2012 event logs the near-minimal decomposition turns out to be the minimal decomposition, as both resulted in only a single submodel. In both cases this is caused because every parallel construct includes invisible transitions (either as split or as join). To be able to split these constructs, we can relax two of the four equivalence rules in such a way that we also allow the decomposition to split invisible transitions in the same way as it splits visible transitions with unique labels:

R2' The i-th incoming arc of an invisible transition is equivalent to the i-th outgoing arc of that transition, if both arcs exist.

Table 6. Decomposed replay results of event logs using near-minimal decomposition

Event log	Submodels	Running time (in seconds)	Costs
RepairExample	2	(136%) 0.31	(100%) 0.197
A32f1n00	4	(18%) 1.9	(100%) 0.000
A32f1n10	4	(13%) 2.1	(94%) 0.929
A32f1n50	4	(10%) 3.1	(96%) 4.322
Bpic2012A	1	(125%) 0.74	(100%) 1.293
Bpic2012	1	(101%) 480	(100%) 14.228

Table 7. Decomposed replay results of event logs using near-minimal (with invisible transitions) decomposition

Event log	Submodels	Running time (in seconds)	Costs
Bpic2012A	3	(391%) 2.3	(98%) 1.272
Bpic2012	3	(83%) 400	(100%) 14.227

This relaxation maintains the property that any perfectly fitting trace in the overall model is a perfectly fitting trace in the submodels (as we are still possible to replay such a perfectly fitting trace in the submodels). However, it may break the property that a perfectly fitting in the submodels is a perfectly fitting trace in the overall model, as the invisible transitions on the border of the submodels may not agree. When replaying a trace on the submodels, such a disagreement will not be noticed, but when replaying it on the overall model, it will be noticed, which leads to a mismatch and extra costs. Table 7 shows the results of this decomposition for the two event logs. Both computed costs are quite good, and the running time of the Bpic2012 log is better than that of the regular replay, but the running time of the Bpic2012A log is worse.

Another way to split up the near-minimal decomposition a bit further is to define a set of (visible) transitions for which the relaxed equivalence does not hold, that is, that assume that all incident arcs are not equivalent. In this paper, we will use the term *milestone transitions* for such transitions. As a result, we then have the following changes in the rules for equivalence:

R2" The i-th incoming arc of an invisible *non-milestone* transition is equivalent to the i-th outgoing arc of that transition, if both arcs exist.

R4" The i-th incoming arc of a visible *non-milestone* transition with unique label is equivalent to the i-th outgoing arc of that transition, if both arcs exist.

Note that the equivalence of milestone transitions is derived from these rules. Initially, incident arcs of milestone transitions are not equivalent, but they may become equivalent if this equivalence is the result of any combination of the four

rules mentioned above. As such, it is still possible that multiple incident arcs for a milestone transition are equivalent.

As an example of this *milestone* decomposition, we have selected six such *milestone* transitions in the Bpic2012 model:

1. t14402
2. t16523
3. W_Beoordelen fraude+SCHEDULE
4. W_Nabellen incomplete dossiers+SCHEDULE
5. W_Valideren aanvraag+COMPLETE
6. W_Valideren aanvraag+START

Please note that the first two milestone transitions are invisible transitions, which have no counterpart in the event log, whereas the remaining four are visible. Together, the first four transitions form a sparsest cut in the model, where the middle two transitions link a submodel containing parallelism to a submodel containing an invisible transitions that allows the former submodel to be started over and over again during replay. Using this decomposition, the decomposed replay takes only 100 seconds (22%) and results in a costs of 11.722 (82%), which is quite acceptable. The left submodel (the large one) took 100 seconds to replay (costs 7.102), the top-right submodel took 97 seconds (costs 2.806), the middle-right submodel took 66 seconds (costs 0.951), and the bottom-right submodel (the parallel one) took 2.8 seconds to replay (costs 0.863). Obviously, the fact that we could run the decomposed replay on 4 different threads helped a lot in reducing the running time, as the total running time would have taken not 100 but $100 + 96 + 66 + 2.8 \approx 260$ seconds.

5 Tool implementation

The decomposed replay has been implemented in the ProM 6[1] *Passage* package, which depends on the *PNetReplayer* package for the replayer. For sake of completeness, we mention that for this case study we have used version 6.1.122 of the former and version 6.1.160 of the latter package. The *Passage* package contains a number of plug-ins that are relevant for the decomposed replay.

Create Decomposed Replay Problem Parameters Using this plug-in, the user can set the parameters which are to be used for both decomposing a model and a log into submodels and sublogs, and the decomposed replay that may follow this decomposition. Both an automated and a manual version exist for this plug-in. The automated version takes a model (a Petri net) and generates default parameters for this model. The manual version takes either a model with default parameters or existing parameters, and allows the user to change these parameters through a dialog. Figure 4 shows this dialog. For additional details on the implementation, we refer to [14].

[1] A nightly build version of ProM 6 containing the required packages can be downloaded from http://www.promtools.org/prom6/nightly.

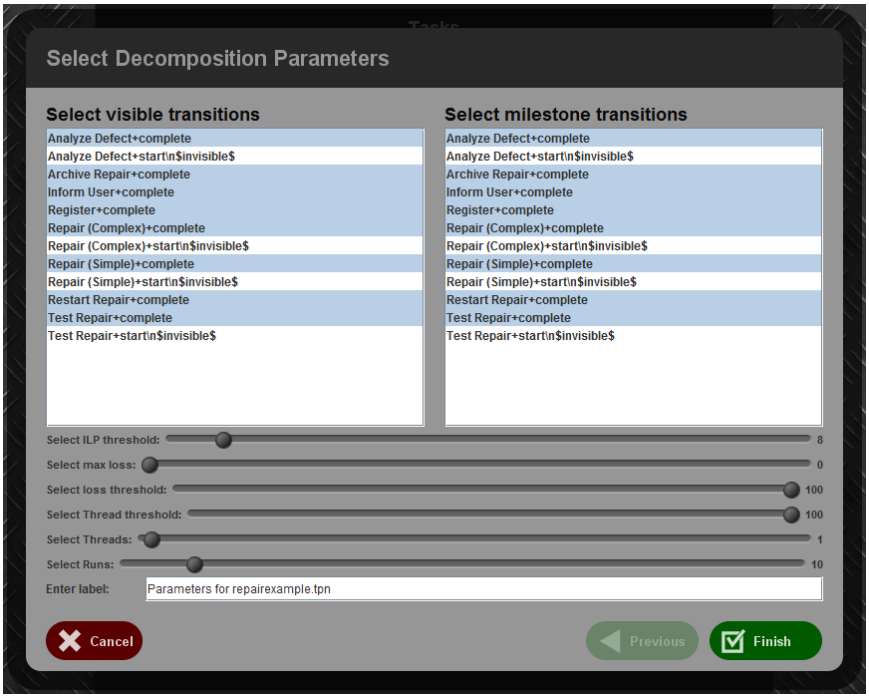


Fig. 4. Dialog for setting decomposed replay parameters

Select visible transitions Using this list, the user can select the transitions that are allowed to be present in multiple submodels. Only selected transitions can appear on the interface between different submodels. By default, the visible transitions will be selected.

Select milestone transitions Using this list, the user can select the milestone transitions. When determining the equivalence classes between the arcs, only the non-selected transitions will be taken into account. By default, the visible transitions will be selected.

Select ILP threshold Using this slider, the user can influence for which submodels the ILP-based replay will be used. This replayer will be used for all submodels for which the number of transitions exceeds this threshold. The non-ILP-based replayer will be used otherwise. By default, this threshold is set to 8.

Select max loss Using this slider, the user can influence to what extend singe-input-single-output invisible transitions are to be reduced. Such a transition will be reduced if the product of its number of additional inputs and its number of additional outputs (that is, the number of new paths) is below this threshold. By default, this threshold is set to 0.

Select loss threshold Using this slider, the user can influence for which submodels single-input-single-output invisible transitions are to be reduced. These transitions are only reduced if the number of transitions in the submodel exceeds this threshold. By default, this threshold is set to

100 (the maximal value), which means that these transitions will only be reduced for very large submodels.

Select Thread threshold Using this slider, the user can influence for which submodels the behavior of the source transitions will be restricted by adding a place. The submodel will only be restricted if the number of transitions in the submodel exceeds this threshold. By default, this threshold is set to 100 (the maximal value), which means that this restricting place will only be added for very large submodels.

Select Threads Using this slider, the user can influence the number of tokens in the place restricting the behavior of (otherwise) source transitions. The number of tokens equals the number set by this slider plus the number of outgoing arcs from this place to non-source transitions. By default, this number is set to 1.

Select Runs With this slider, the user can influence how many times the regular replay and decomposed replay are to be run. The regular replay and decomposed replay will be run as many times as indicated by this slider.

Enter label With this textbox, the user can provide a meaningful name to the selected parameter setting. This name will be used by ProM 6 to identify this parameters setting. By default, this name equals the name of the process model prefixed by "Parameters for ".

Show Replay Costs Using this plug-in, the user can run the regular replay and the decomposed replay as many times as indicated by the parameters that are required as input. Additional inputs are the process model (a Petri net) and the event log. This plug-in will result in an overview that contains the parameter settings (for sake of reference) and the results (both running times and costs for both the regular replay and the decomposed replay, including the running times and costs for the replay of every sublog on the corresponding submodel).

Create Decomposed Replay Problem Using this problem, the user can construct a decomposed replay problem. Required inputs are the parameters, the process model (a Petri net), and an event log. Please note that this plug-in only constructs the decomposed replay problem, it does not solve it by actually replaying it. To actually do this replay, the user can select the "Visualize Replay" visualization, which first performs the replay and then shows the results.

6 Concluding Remarks

In this paper, we have applied our decomposed replay techniques on six event logs and four corresponding process models. For two out of six event logs any decomposed replay we tried took longer than the regular replay. However, for exactly these two event logs the regular replay takes less than a second. The four remaining event logs all take longer than second to replay, where the actual time needed varies from 11 seconds to 480 seconds. For all of these four event logs,

Table 8. Result of (decomposed) replay for event logs that take longer than a second to replay

Event log	**Submodels**	**Running time** (in seconds)	**Costs**
A32F1N00	1	11	0.000
	4	(18%) 1.9	(100%) 0.000
A32F1N10	1	17	0.993
	4	(13%) 2.1	(94%) 0.929
A32F1N50	1	32	4.521
	4	(10%) 3.1	(96%) 4.322
BPIC2012	1	480	14.228
	4	(22%) 100	(82%) 11.722

we could achieve better running times at acceptable decreases in costs. Table 8 shows the result of the decomposed replay for these event logs, and compares these results to the results of the regular replay (where there is only a single submodel).

Although this shows that we can use decomposed replay to achieve better running times at acceptable costs, there is an issue with the actual decomposition. Especially the BPIC2012 event log has been a hard nut to crack, as the replay of this event log turns out to be very sensitive to the actual decomposition. The replay simply fails to finish if we decompose the process model and event log in as many submodels and sublogs as possible. Especially the replay of one of the sublogs on its corresponding submodel caused the replay to not finish.

In the paper, we have shown that we can take several approaches to this replay problem. First of all, we can restrict the possible behavior of the problematic submodel. This leads to a replay that finishes, but not to a costs that is by definition a lower bound for the actual costs, which is what we need. Second, we can reduce the single-input single-output invisible transitions such that none remains. As a result of this reduction, we possibly introduce new paths (new behaviors) in the resulting submodel, but this is safe as this only lowers the costs. We also tried to fine-tune this reduction of invisible transitions, but that did not help much: The costs did not improve more than the running time of the replay did increase. Third, we can aim for submodels that suit the actual replay techniques we are using in some way. Doing so leads to a decomposed replay that finishes, although it might decompose the model in a single submodel.

To be able to use the decomposed replay technique effectively in practice, we need to be able to come up with a good decomposition that decreases running times while keeping the estimated costs at an acceptable level. We have shown that even for the hard BPIC2012 log this was possible, but we haven't shown that we can do this for any log. For this, we need more research on what these good decompositions are, how they look like, and how we can derive them from a given model (possibly, given the corresponding event log). As shown, for some

logs, the default (*near-maximal*) decomposition works fine, but other logs require more sophisticated decompositions.

References

1. W.M.P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer-Verlag, 2011.
2. W.M.P. van der Aalst. Decomposing Process Mining Problems Using Passages. In S. Haddad and L. Pomello, editors, *Applications and Theory of Petri Nets 2012*, volume 7347 of *Lecture Notes in Computer Science*, pages 72–91. Springer-Verlag, 2012.
3. W.M.P. van der Aalst, B.F. van Dongen, C.W. Günther, R.S. Mans, A.K. Alves de Medeiros, A. Rozinat, V. Rubin, M. Song, H.M.W. Verbeek, and A.J.M.M. Weijters. ProM 4.0: Comprehensive Support for Real Process Analysis. In J. Kleijn and A. Yakovlev, editors, *Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2007)*, volume 4546 of *Lecture Notes in Computer Science*, pages 484–494. Springer-Verlag, 2007.
4. A. Adriansyah, B. van Dongen, and W.M.P. van der Aalst. Conformance Checking using Cost-Based Fitness Analysis. In C.H. Chi and P. Johnson, editors, *IEEE International Enterprise Computing Conference (EDOC 2011)*, pages 55–64. IEEE Computer Society, 2011.
5. J. Carmona, J. Cortadella, and M. Kishinevsky. Divide-and-Conquer Strategies for Process Mining. In U. Dayal, J. Eder, J. Koehler, and H. Reijers, editors, *Business Process Management (BPM 2009)*, volume 5701 of *Lecture Notes in Computer Science*, pages 327–343. Springer-Verlag, 2009.
6. B. F. van Dongen. BPI challenge 2012. Dataset. http://dx.doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f, 2012.
7. J. Munoz-Gama, J. Carmon, and W. M. P. van der Aalst. Hierarchical conformance checking of process models based on event logs. In *ATPN 2013*, LNCS. Springer, 2013. Accepted for publication.
8. T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
9. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
10. A. Rozinat and W.M.P. van der Aalst. Conformance Checking of Processes Based on Monitoring Real Behavior. *Information Systems*, 33(1):64–95, 2008.
11. W. M. P. van der Aalst. Decomposing Petri Nets for Process Mining: A Generic Approach. Technical Report BPM-12-20, BPMcenter.org, 2012.
12. Wil van der Aalst, Arya Adriansyah, and Boudewijn van Dongen. Replaying history on process models for conformance checking and performance analysis. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(2):182–192, 2012.
13. H. M. W. Verbeek. BPI Challenge 2012: The Transition System Case. In M. La Rosa and P. Soffer, editors, *BPM 2012 Workshops*, volume 132 of *LNBIP*, pages 225–226. Springer, 2013.
14. H. M. W Verbeek and W. M. P. van der Aalst. Decomposing replay problems: A case study. Technical Report BPM-13-09, BPMcenter.org, 2013.

Part IV

PNSE'13: Short Papers

Building Petri nets tools around Neco compiler

Łukasz Fronc and Franck Pommereau
{fronc,pommereau}@ibisc.univ-evry.fr

IBISC, Université d'Évry/Paris-Saclay
IBGBI, 23 boulevard de France
91037 Évry Cedex, France

Abstract. This paper presents Neco that is a Petri net compiler: it takes a Petri net as its input and produces as its output an optimised library to efficiently explore the state space of this Petri net. Neco is also able to work with LTL formulae and to perform model-checking by using SPOT library. We describe the components of Neco, and in particular the exploration libraries it produces, with the aim that one can use Neco in one's own projects in order to speedup Petri nets executions.

Keywords: Petri nets compilation, optimised transition firing, tools development, explicit state space exploration

1 Introduction

Neco is a Petri net compiler: it takes a Petri net as its input and produces as its output a library allowing to explore the Petri net state space. Neco operates on a very general variant of high-level Petri nets based on the Python language (*i.e.*, the values, expressions, etc., decorating a net are expressed in Python) and including various extensions such as inhibitor-, read- and reset-arcs. It can be seen as coloured Petri nets [1] but annotated with the Python language instead of the dialect of ML as it is traditional for coloured Petri nets.

Firing transitions of such a high-level Petri net can be accelerated by resorting to a compilation step: this allows to remove most of the data structures that represent the Petri net by inlining the results of querying such structures directly into the generated code. Moreover, instead of relying on generic data structures and algorithms, specialisations can be performed on a per-transition and per-place basis. In particular, Neco can exploit various properties of Petri nets in order to further optimise the representation of a marking (both for execution time and memory consumption), as well as transitions firing algorithms. Finally, Neco is able to type most of the Python code embedded in a Petri net thanks to the typing information on places. This allows to generate efficient C++ code instead of relying on the interpreted Python code.

All this yields a substantial speedup in transition firing (and consequently in state-space exploration and explicit model-checking) that was evaluated in [2]. This was also confirmed by the participation of Neco to the *model-checking*

contest (satellite event of the Petri nets 2012 conference) that showed that Neco was able to compete with state-of-the-art tools [3].

The goal of this paper is to introduce the main concepts of Neco and its usage in order to enable tool developers for efficiently using the detailed online documentation and exploit Neco in their own projects. This may concern most tools that perform explicit exploration of Petri nets states spaces and take advantage of the speedup that Neco can offer.

Neco is free software released under the Gnu lgpl and it can be downloaded from `http://code.google.com/p/neco-net-compiler` where its documentation is also available, including a tutorial as well as the precise api of libraries generated by Neco and concrete examples.

2 General architecture and usage guidelines

Neco is a collection of two compilers, one exploration tool and one model-checker:

- `neco-compile` is the main compiler that produces an exploration engine of a Petri net (a library);
- `neco-explore` is a simple exploration tool that computes state spaces using the engine produced by `neco-compile`;
- `neco-check` is compiler for LTL formulae that produces a library to handle these formulae;
- `neco-spot` is a LTL model-checker that uses outputs of tools `neco-compile` and `neco-check`, as well as SPOT library for model-checking algorithms [4].

As a compiler, Neco has two backends: the *Python backend* allows to generate Python code while the *Cython backend* generates annotated Python [5] that can be compiled to C++. Each tool composing Neco is dedicated to a specific task. Here we *focus on compilation* but we will also say a few words about the rest. The detailed compilation workflow is shown in Figure 1. In this section we assume that we use the *Cython backend* which is the most efficient one. First we present how the exploration engine is built and how to use it to build state-spaces, this part remains globally valid for the *Python backend*. Next we present how to perform LTL model-checking within Neco, and this part is currently *not supported* by the Python backend. However, there are also features that are currently only available in the Python backend, like *reductions by symmetries* [6], thus not yet available for LTL model-checking.

2.1 Exploration engine builder and state-space construction

The first step using Neco is to create a module that provides exploration primitives: a marking structure, successor functions specific to transitions, and a global successor function that calls the transition specific ones [2]. As shown in Figure 2, this exploration engine can be used by a client program (*e.g.*, a model-checker or a simulator) to perform its task. The generated library directly

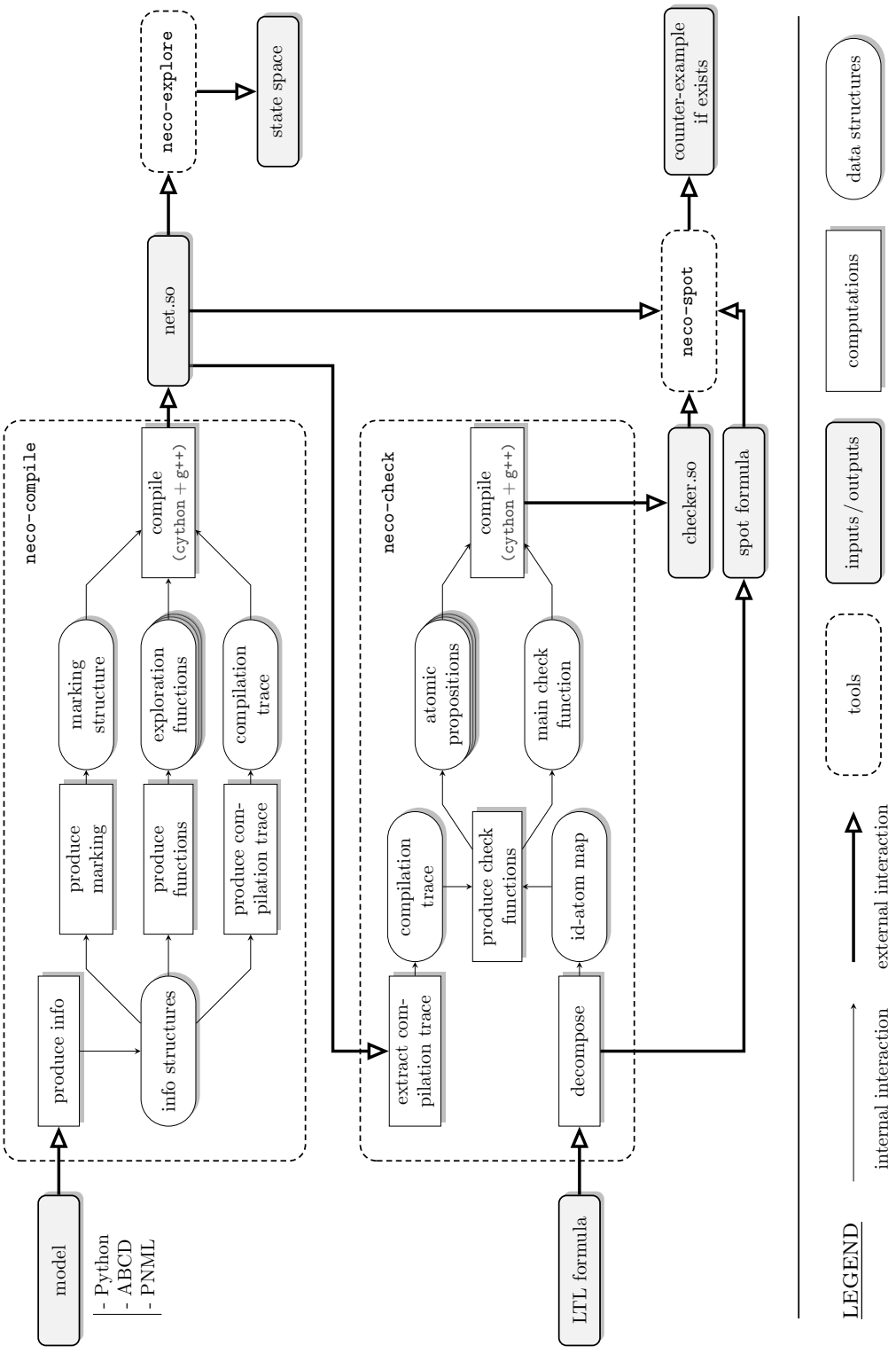


Fig. 1. Compilation pipeline and exploration tools within Neco (Cython backend).

embeds code from the model (*i.e.*, Petri net annotations) but also relies on existing data structures (in particular, sets and multisets) forming core libraries, and accesses them through normalized interfaces. Model code itself has very few constraints and may use existing libraries. This is detailed in [2].

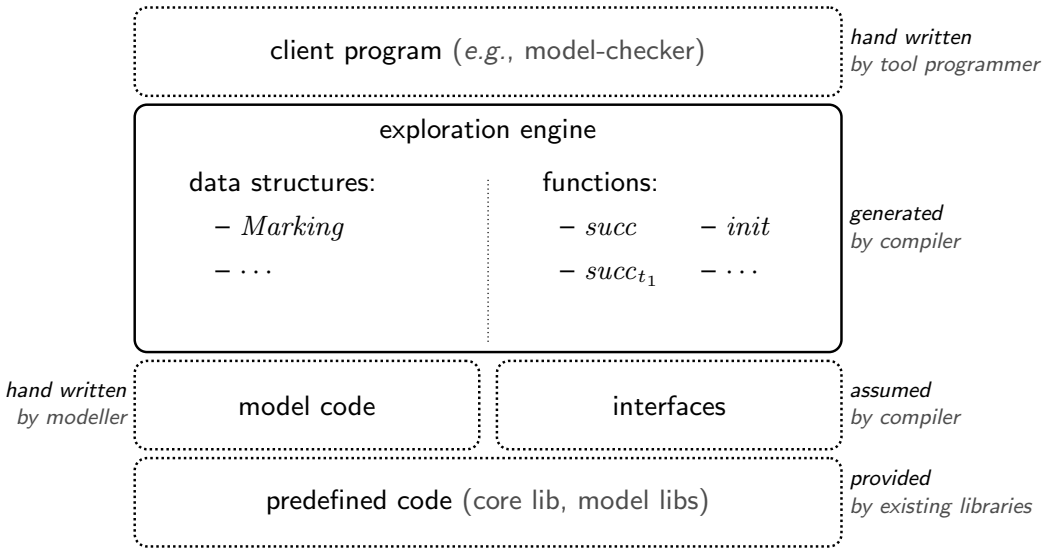


Fig. 2. The exploration engine (plain-line box) and its context (dotted boxes). [2]

This module is built using command `neco-compile`. To do so, Neco takes a Petri net model as input which can be described programmatically in Python using the *SNAKES* toolkit [7], or using the *ABCD* formalism [8], or specified in *PNML* [9]. Once the model is loaded, some types are *inferred* allowing to statically type Python code later, which is an important feature because Cython language can produce optimized C++ code from annotated Python code [5]. However, because we allow a high degree of expressivity, all source code cannot be typed and Neco falls back to calling the Python interpreter in such cases. Basically, if a net contains only black tokens, integers or Boolean values, and static strings, as well as collections (tuples, lists, sets, dictionaries) of such values, it will be fully translated into C++.

The next step is to produce a *marking structure* to represent Petri net states. It is optimized based on previously discovered types. This allows to use native types or to generate per-place specialised implementations. Then, we can produce *exploration functions* specific to the model (mainly an initial marking function and successor functions), this allows to efficiently produce state spaces [2].

An additional step is to produce a *compilation trace* which contains information about the marking structure and the model. This metadata is essential for consistency preservation among tools, and it prevents the user from having to call each tool with exactly the same options which is error-prone.

The last step is to compile generated code producing a *native Python module* that is a *shared library* which can be used from C++ as a regular library as well as from Python as a regular module. This is actually done with *Cython compiler* and a C++ compiler.

State spaces can be built using `neco-explore` tool. This tool builds sets of reachable states, and reachability graphs using a simple exploration algorithm that aggregates discovered states by repeatedly calling successor functions.

2.2 LTL model checking

LTL model checking is performed using *SPOT library* [4], however, SPOT cannot directly handle atomic propositions appearing in LTL formulae which are specific to the used formalism. Moreover, because our marking structures are model specific, we also need to generate an atomic proposition checker module for each compiled net. This is made by `neco-check` compiler.

This tool takes two inputs, a LTL formula in Neco compatible LTL syntax [10], and compilation metadata extracted from an exploration module (previously created with `neco-compile`).

The first step is to decompose the formula, extract atomic propositions and map them to unique identifiers ("id-atom map" on Figure 1). A simplified formula where all atomic propositions have been replaced by these identifiers is stored as a file. This way, atomic propositions can be abstracted away leading to a simple interface with the checking module. Basically, the interface is a function `check` that takes a state and an atomic proposition identifier, and returns the truth value of the atomic proposition at the provided state.

The next step is the creation of one check function for each atomic proposition, plus the generic check function exposed to users. During this step, using the compilation trace is essential because we need to create functions that are compatible with the optimized marking structure, and thus be aware of used types and memory layout. Finally the generated code is compiled using *Cython compiler* and a C++ compiler.

The checker module finalized, it can be used together with the formula file by `neco-spot` tool and it will output a counter-example if one exists, *i.e.*, if the formula is not satisfied.

3 Perspectives

Several new features are already planned for Neco. First, a method to reduce symmetries based on [6] has been already prototyped in the Python backend. We would like to implement it in the Cython backend also to achieve better performance. Next, Neco will be adapted to compute unfoldings *à la* McMillan using the approach described in [11, chap.6]. This should be feasible by reusing most of the code that Neco already generates to discover bindings. Finally, we would like to implement fast simulation in Neco, which could be a variant of the current exploration algorithm that would compute only one successor for a

state instead of all its successors. However, for better performance, we would like to experiment with a co-routine based implementation of Python [12] in order to define a highly concurrent architecture while avoiding the overhead of using threads.

Neco will also participate to the 2013 edition of the model-checking contest. As a side effect, this will lead us to develop new case studies for Neco (*i.e.*, those models that are included in the contest), which will be extended later with more case studies.

Based on case studies, we would like to perform extensive benchmarks of the Cython backend by comparing it to a combination of the Python backend with various Python compilers (in particular [13] and [14]) as well as with PyPy implementation of Python that features efficient just-in-time compilation [15]. This should allow either to drop Cython backend if it happens that it is outperformed by other approaches, or, more probably, to define typical situations where Cython should not be used. In particular, we expect PyPy to be more efficient on Petri nets that embed a lot of Python objects that cannot be converted to efficient C++ code.

Finally, we are working on an additional Java backend, allowing to compile Petri nets and LTL formulae to Java code. This will require some internal reorganisation of Neco so its core will become language-agnostic while only the backends will have to deal with language-specific aspects. Thanks to this work, we expect that more backends will be implemented in the future to handle Petri nets annotated with a wider variety of languages.

References

1. Jensen, K., Kristensen, L.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer (2009)
2. Fronc, Ł., Pommereau, F.: Optimizing the compilation of Petri nets models. In: Proc. of SUMo'11. Volume 726., CEUR (2011)
3. Kordon, F., Fronc, Ł., Pommereau, F., et al.: Raw Report on the Model Checking Contest at Petri Nets 2012. Technical report (2012)
4. Duret-Lutz, A.: LTL translation improvements in Spot. In: Proceedings of the 5th International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS'11). Electronic Workshops in Computing, Tunis, Tunisia, British Computer Society (2011)
5. Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D., Smith, K.: Cython: The best of both worlds. Computing in Science Engineering **13** (2011) 31 –39
6. Fronc, Ł.: Effective Marking Equivalence Checking in Systems with Dynamic Process Creation. In: Infinity'12. Electronic Proceedings in Theoretical Computer Science, Paris (2012)
7. Pommereau, F.: Quickly prototyping Petri nets tools with SNAKES. Petri net newsletter (2008)
8. Pommereau, F.: Algebras of coloured Petri nets. LAP LAMBERT Academic Publishing (2010)
9. Hillah, L., Kindler, E., Kordon, F., Petrucci, L., Trèves, N.: A primer on the Petri Net Markup Language and ISO/IEC 15909-2. In: 10th International workshop on Practical Use of Colored Petri Nets and the CPN Tools (CPN'09). (2009)

10. Fronc, Ł.: Neco net compiler wiki. `goo.gl/CXrry` (2012)
11. Khomenko, V.: Model checking based on Petri net unfolding prefixes. PhD thesis, PhD thesis, School of Computer Science, University of Newcastle upon Tyne (2002)
12. Tismer, C.: Continuations and stackless Python. In: Proceedings of the 8th International Python Conference. (2000)
13. Dufour, M.: Shed skin. `http://code.google.com/p/shedskin` (2012)
14. Hayen, K.: Nuitka. `http://nuitka.net` (2012)
15. Bolz, C.F., Cuni, A., Fijalkowski, M., Rigo, A.: Tracing the meta-level: PyPy's tracing JIT compiler. In: Proc. ICOOOLPS '09, ACM (2009)

RT-Studio: A tool for modular design and analysis of realtime systems using Interpreted Time Petri Nets

Rachid Hadjidj and Hanifa Boucheneb

Abstract. RT-Studio (Real Time Studio) is an integrated environment for modeling, simulation and automatic verification of realtime systems modeled as networks of Interpreted Time Petri Nets (ITPNs). The tool allows to construct several abstractions for ITPNs suitable to verify reachability, linear and branching properties, in addition to a TCTL model checker. In this paper we describe the ITPN model as a proposed extension to the classical TPN model and describe the modular modeling capabilities of RT-Studio. The classical railroad crossing model is used to illustrate these capabilities.

Keywords: Formal verification, Time Petri Nets, timed properties, model checking, simulation

1 Introduction

Interpreted Time Petri Nets (ITPNs) are Time Petri Nets (TPNs) [15, 8, 11] we extended with bounded data variables to increase their modeling power and expressiveness. TPNs are Petri nets extended with temporal constraints in the form of time intervals associated with their transitions [15]. A transition can fire if it is enabled and if the time elapsed since it became enabled most recently is within its time interval, but must be fired or disabled if this elapsed time reaches the upper bound of its time interval. With this extension, TPNs are powerful enough to model realtime systems. However, their analysis is much more complicated than simple Petri nets. In fact, the state space of the TPN model is in general infinite due to the continuous aspect of time. Furthermore boundedness is undecidable for this model [5, 4]. Hopefully, a subclass of TPNs called bounded TPNs, for which reachability is decidable, allows to model most useful systems. The analysis and verification of the TPN model is generally performed using model checking techniques. Model checking requires first to represent the behavior of a system as a finite *state transition system*[1], then specify properties of interest in a temporal logic (LTL, CTL, CTL*,...), and finally explore the state transition system to determine whether these properties hold or not [8, 11–13].

In the field of realtime systems modeling and verification, UPPAAL is one of the best tools available [3]. UPPAAL allows for a modular design of realtime systems using an extension of the Timed Automata (TA) model [2] instead of the TPN model. In this extension, variables can be defined, and both guards and actions can be associated with transitions. In addition to its modeling capabilities, UPPAAL integrates an efficient on the fly model checker (a key success component) for a subset of TCTL [1].

[1] Also called *state graph* or *state space*.

For the TPN model, many tools exist for modeling, analysis and verification. Some well known tools are *Roméo* [10] and *TINA* [6]. But to our knowledge, there is no tool for the TPN model that compares to UPPAAL in terms of modular design, extension with variables and verification performance. TAPAAL2 [9], as an inspiration from UPPAAL, is an interesting reascent addition to the pool of tools dedicated to the modeling and verification of real time systems using a timed extension of Petri Nets. In this extension clocks are associated with tokens and time intervals with arcs. A token cannot pass trough an arc unless its clock is within the time interval of the arc. Time invariants associated with places allow to model urgency. The tool, has good verification performance and a nice graphical user interface capable of component based design. Even if it is a great step forward toward tightening the gap with UPPAAL in terms of providing a similar user experience and modeling power, UPPAAL still has a more powerful modular design capability in addition to the use of variables which TAPAAL2 lacks. RT-Studio presented here, attempts to tighten the gap with UPPAAL even more, by allowing for a similar modular design capability based on the ITPN model. RT-Studio also allows to simulate these models and analyze them using several means. In the ITPN model, each transition has, in addition to a time interval, a guard and a set of update operations on user defined data variables and places markings. An enabled transition can be fired if it is enabled and the guard associated with it is true in the current state of the model. When a transition is fired, its updates are applied.

2 Interpreted Time Petri Nets

To increase the modeling power and expressiveness of TPNs , we extended the TPN model with bounded integer variables and associate guards and update actions with transitions. The resulting model is called Interpreted TPN (ITPN). As a consequence, in addition to the normal transition firing requirements, a transition requires also that its guard is true in the current ITPN state. A guard is a condition on the ITPN state defined on its marking, firing intervals of transitions, and the state of associated variables. After a transition t is fired, associated updates are performed. Updates can target variables associated with the ITPN, but also its marking. An update can only target places not attached to t for not conflicting with the usual operational semantics of transition firing. With the new extension, inhibitor arcs and priority on transitions firing can be modeled. Note that if variables associated with an ITPN are always positive then they can be implemented as normal places.

Let $\mathbb{Q}^+$, $\mathbb{R}^+$ and $\mathbb{Z}$ be respectively the set of positive rational numbers, the set of positive real numbers and the set of integers. Let $\mathbb{Q}^+_{[\,]}$ be the set of non empty intervals of $\mathbb{R}^+$ which bounds are respectively in $\mathbb{Q}^+$ and $\mathbb{Q}^+ \cup \{\infty\}$. For an interval $I \in \mathbb{Q}^+_{[\,]}$, $\downarrow I$ and $\uparrow I$ denote respectively its lower and upper bounds.

Definition 1. ***Interpreted Time Petri Net (ITPN)***
An ITPN $\mathcal{P}$ is a tuple $(P, T, V, Pre, Post, m_0, v_0, Is, G, U)$ where:

- *P is a finite set of places,*
- *T is a finite set of transitions, with $P \cap T = \emptyset$,*
- *V is a finite set of integer variables, with $(P \cup T) \cap V = \emptyset$,*

- *Pre and $Post$ are respectively the backward and forward incidence functions: $P \times T \to \mathbb{N}$, where $\mathbb{N}$ is the set of nonnegative integers,*
- *$m_0 : P \to \mathbb{N}$, is the initial marking,*
- *$v_0 : V \to \mathbb{Z}$, is the initial valuation on $\mathcal{P}$ variables.*
- *$Is : T \to \mathbb{Q}^+_{[\,]}$ associates with each transition t an interval $[\downarrow Is(t), \uparrow Is(t)]$ called its* static firing interval. *The bounds $\downarrow Is(t)$ and $\uparrow Is(t)$ are called the* minimal *and* maximal static firing delays *of t.*
- *$G : T \to Bool(\mathcal{P})$, associate with each transition a guard from the set $Bool(\mathcal{P})$. $Bool(\mathcal{P})$ is the set of boolean functions on the set $\mathcal{S}_\mathcal{P}$ of states of $\mathcal{P}$ (The ITPN state is defined next).*
- *$U : T \to Update(\mathcal{P})$, where $Update(\mathcal{P})$ is the set of integer functions on the set $\mathcal{S}_\mathcal{P} \times (P \cup V)$, associates with each transition t an update function on places and variables associated with $\mathcal{P}$.*

Let M be the set of all markings of $\mathcal{P}$. Let $m \in M$ be a marking, and $t \in T$ a transition of $\mathcal{P}$. t is said to be *enabled* in m, iff all tokens required for its firing are present in m, i.e.: $\forall p \in P, m(p) \geq Pre(p,t)$. We denote by $En(m)$ the set of all transitions enabled in m. If m results from firing transition t_f from another marking, $New(m, t_f)$ denotes the set of all newly enabled transitions in m, i.e.: $New(m, t_f) = \{t \in En(m) | \exists p, m(p) - Post(p, t_f) < Pre(p, t)\}$.

Definition 2. ***ITPN state***
The state of ITPN $\mathcal{P}$ is a couple (m, v, I), where m is a marking, v is a valuation on the variables of $\mathcal{P}$, and I is an interval function $I : En(m) \to \mathbb{Q}^+_{[\,]}$ [5].

For a state $s = (m, v, I)$, and $t \in En(m)$, $I(t)$ is called the *firing interval* of t. It is the interval of time where t can fire. The initial state of $\mathcal{P}$ is $s_0 = (m_0, v_0, I_0)$, where $I_0(t) = Is(t)$, for all $t \in En(m_0)$. The state of $\mathcal{P}$ evolves either by time progression or by firing transitions. When a transition t becomes enabled, its firing interval is set to its static firing interval $Is(t)$. The bounds of this interval decrease synchronously with time, until t is fired or disabled by another firing. t can fire, if the lower bound $\downarrow I(t)$ of its firing interval reaches 0 and its guard $G(t)$ evaluates to true in the current state of $\mathcal{P}$, but must be fired, without any additional delay if the upper bound $\uparrow I(t)$ of its firing interval reaches 0 while its guard evaluates true. The firing of a transition takes no time.

Let $s = (m, v, I)$ and $s' = (m', v', I')$ be two states of $\mathcal{P}$. We write $s \xrightarrow{\theta} s'$, iff state s' is reachable from state s after a time progression of θ time units (s' is also denoted $s + \theta$), i.e.:

$$\begin{cases} \exists \theta \in \mathbb{R}^+, \ \bigwedge_{t \in En(m)} \theta \leq \uparrow I(t), \\ m' = m, v' = v, \\ \forall t' \in En(m'), \ I'(t') = [max(\downarrow I(t') - \theta, 0), \uparrow I(t') - \theta]. \end{cases}$$

We write $s \xrightarrow{t} s'$ iff state s' is immediately reachable from state s by firing transition t. i.e.:

$$\begin{cases} t \in En(m), \\ \downarrow I(t) = 0 \wedge G(t,s) = true, \\ \forall p \in P \begin{cases} m'(p) = m(p) - Pre(p,t) + Post(p,t) \; if \; Pre(p,t) \neq 0 \vee Post(p,t) \neq 0, \\ m'(p) = U(t,s,p) \qquad\qquad\qquad\qquad otherwise, \end{cases} \\ \forall x \in V, \; v'(x) = U(t,s,x), \\ \forall t' \in En(m') \begin{cases} I'(t') = Is(t') \; if \; t' \in New(m',t), \\ I'(t') = I(t) \quad otherwise. \end{cases} \end{cases}$$

Definition 3. ***ITPN state space***
The state space of an ITPN *model $\mathcal{P}$ is the structure $(\mathcal{S}, \rightarrow, s_0)$, where:*

- *$s_0 = (m_0, v_0, I_0)$ is the initial state of $\mathcal{P}$,*
- *$s \rightarrow s'$ iff either $s \xrightarrow{\theta} s'$ for some $\theta \in \mathbb{R}^+$ or $s \xrightarrow{t} s'$ for some $t \in T$,*
- *$\mathcal{S} = \{s | s_0 \xrightarrow{*} s\}$, where $\xrightarrow{*}$ is the reflexive and transitive closure of $\rightarrow$, is the set of reachable states of $\mathcal{P}$.*

3 Modular design of realtime system

RT-Studio uses the concept of programming project[2] to allow for a modular design of a realtime system. A project is basically a collection of ITPN components, and a *system description file*. An ITPN component is actually a template with possible parameters. The system description file is where the system configuration is specified. It consists of a list shared variable declarations and instantiations of ITPN components.

Instances of ITPN components within the same project can communicate and synchronize using data variables defined in the system description file, and using shared transitions and shared places. A transition or a place is shared between several ITPNs if it has the same name and set as external (not local) in each one of them. When synchronizing ITPN components instances, shared places with the same name are merged in one single place. The marking of that place is the maximum marking of synchronized places. For transitions, the synchronization is little bit more demanding. A transition meant for synchronization is implicitly associated with a signaling channel having the same name. Transitions which names end with and exclamation mark '!' represent send actions on associated channels. Those with names ending with an interrogation mark '?' represent receiving actions on associated channels. When instances of ITPN components are synchronized, transitions representing complimentary actions are synchronized by merging each sending transition with a copy of a receiving transition having the same name. A copy of a transition is created by duplicating the transition and all its ingoing and outgoing arcs.

Each ITPN component element (place, transition, arc), including the ITPN itself has an extendable list of attributes (properties). When an ITPN component or an ITPN component element is created, a default list of attributes is associated with it. These attributes describe its different features, like the number of tokens for a place, the guard and update actions for a transition, the list of parameters for the ITPN component,

[2] A collection of files.

including the visual appearance of ITPN elements like the color, the shape, the border size, etc. The *Attribute viewer* is a panel for editing, modify and adding new attributes for the currently selected ITPN element. Added attributes can be referred to in guards and updates actions of transitions. Three attribute types are supported in the current version of RT-Studio: number (real value), boolean and string.

To analyze a system designed as a project, its system description file need to be compiled. The compilation consists in synchronizing all ITPN components instances in one single ITPN. In the case of any error, error messages are displayed in a message pane at the bottom of the main window, otherwise the synchronized ITPN is generated and displayed in a separate panel and ready to be analyzed if it is self contained[3]. Note that a self contained ITPN components can be analyzed without compilation.

4 Functionality

RT-Studio's verification and analysis capabilities can be grouped in three categories: abstract state spaces construction, model checking and simulation. Both known characterizations of the TPN state (interval and clock characterizations) [11] can be used to construct abstractions for ITPN models. For the interval characterization, computed abstractions are: the classical State Class Graph (SCG), the Strong State Class Graph (SSCG) and the Atomic State Class Graph (ASCG) [7]. Both the SCG and the SSCG preserve linear properties, but the SCG is a better alternative for linear properties as it is smaller and faster to compute. On the other hand, the SSCG is used as a starting point in a refinement process to generate the ASCG which preserves branching properties (CTL*). During the refinement, state classes are split by linear constraints so that each state captured in a sub class has a successor in each one of the following classes. Such sub classes are said to be *atomic*, and atomicity of all classes ensures preservation of CTL* properties [8]. For the clock characterization of states, computed abstractions are: the Concrete State Zone Graph (CSZG) for linear properties [11], and its atomic version, the ACSZG [11] for CTL* properties. Compared with the ASCG, the ACSZG is in general smaller and faster to compute. For reachability properties, RT-Studio implements three contraction techniques (by inclusion, by convex combination and by convex hull) to rapidly generate contracted versions of the SCG, SSCG and CSZG which are suitable to verify reachability properties [11, 14]. It also implements several post contraction operations to further contract abstractions after they are constructed, and a minimizer under bisimulation. After computing any abstraction, RT-Studio allow the user to explore and edit it graphically. It also generates some statistics about the abstraction like its size (the number of nodes and edges) and the generation time. To verify properties, RT-Studio implements two model-checkers: a classical CTL model-checker and an innovative TCTL model-checker based on the forward on the fly verification technique described in [12, 13]. Finally, the simulator implemented in RT-studio allows for an assisted and interactive generation of any abstraction. Using the simulator, the user can intervene during the generation process, guide the generator to explore any branch of the state space graph being constructed, and even alter the model

[3] A self contained ITPN has no parameters and does not need to be synchronized with other ITPNs.

during the simulation if needed. RT-Studio stores a project in a single XML file. It also allows to import or export self contained ITPNs to a text file in a simple format accepted by the TINA toolbox [6]. Generated state spaces can also be exported in a text format supported by many model checkers.

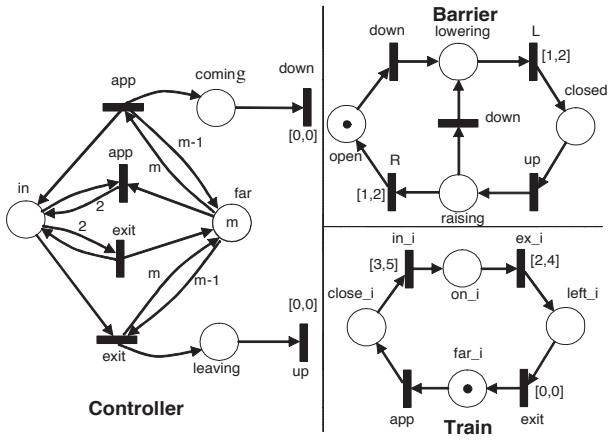


Fig. 1. The level crossing TPN model

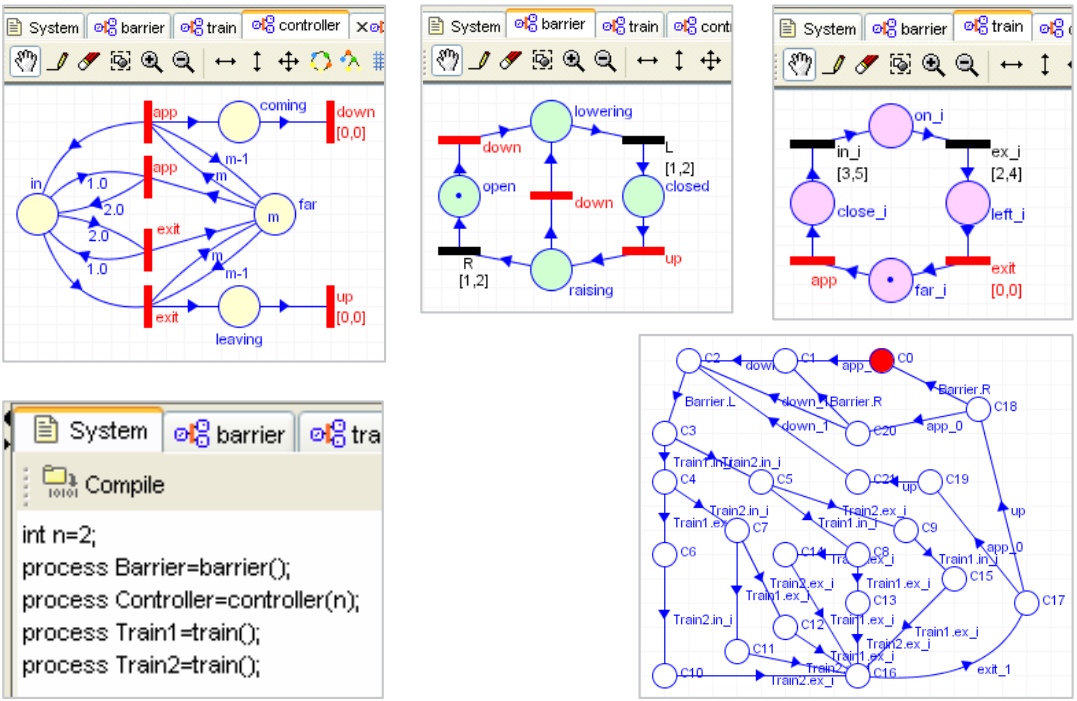


Fig. 2. Screen shots from the project "level crossing model": the controller, the barrier, the train. The picture at the right bottom is the SCG of the model

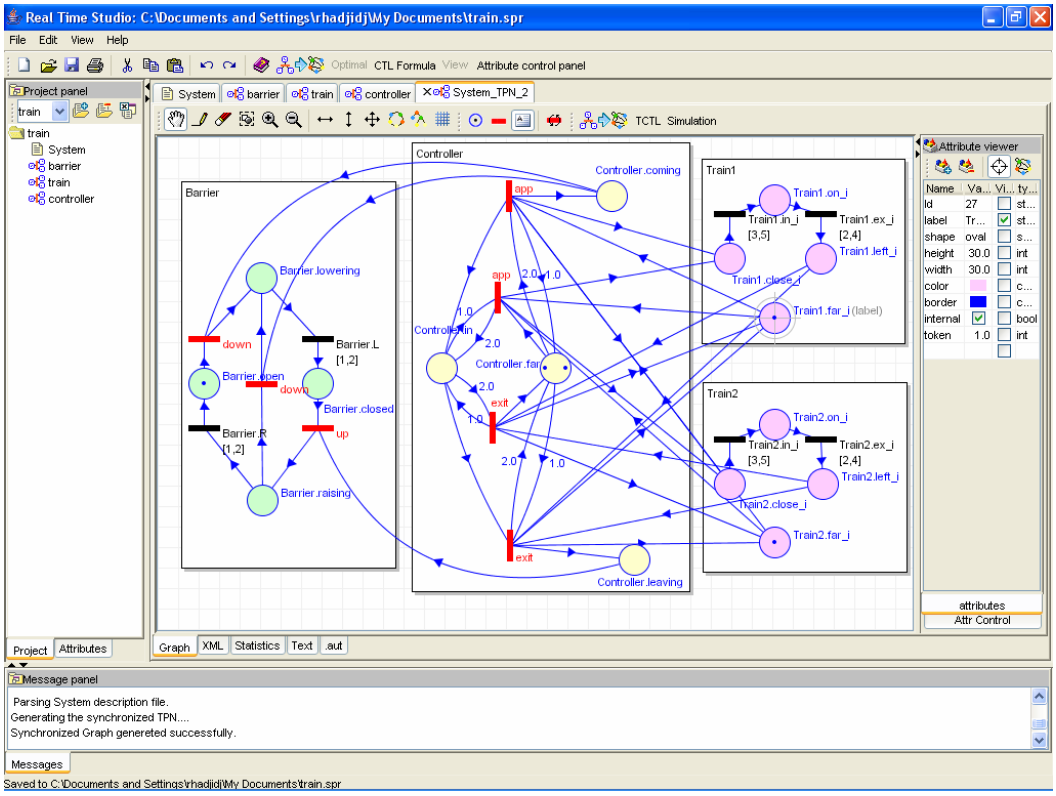


Fig. 3. RT-Studio interface

5 An illustrative example

As an illustrative example, we consider the classical *Railroad Crossing* model. Components of this system are shown in Figure 1 and also in Figure 2 as screen shots from RT-Studio. The ITPN model of n trains crossing concurrently the road is obtained by synchronously composing the controller model with its parameter m[4] set to n, the barrier model, and n instances of the train model. In Figure 2, we can see the system description file for the railroad crossing project where two instances of the train model (Train1 and Train2) are synchronized with one instance of the barrier model (Barrier) and one instance of controller model (Controller). After compiling the project, we obtain the synchronized ITPN of the whole system shown in Figure 3. Note that for clarity, in Figure 3 each transition *app* is actually the superposition of two *app* transitions; one synchronized with first train instance, the other one with the second train instance. At the right bottom of Figure 2, we can see the SCG of the synchronized ITPN computed by RT-studio.

[4] m is a number of tokens.

6 Installation

RT-Studio can be downloaded from *http://faculty.qu.edu.qa/rhadjidj/rt-studio.aspx*. The tool comes in a Zip file that includes three files : *RT-studio.jar*, *realtime.exe*, *readme.txt*, and a directory for examples. *RT-studio.jar* is a Java executable representing the graphical interface of the tool. *realtime.exe* is the engine written in C++ for performance. Installing the tool consists in simply unziping the zip file in chosen directory. Double clicking on the file *RT-studio.jar* launches the tool.

On the download web page there are video tutorials that explain how to install and use the tool to model, simulate and verify properties.

References

1. R. Alur, C. Courcoubetis, and D. Dill. Model checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.
2. R. Alur and D. Dill. Automata for modelling real-time systems. In *In Proc. Of ICALP'90*, volume 443 of LNCS, pages 322–335. Springer-Verlag, 1990.
3. G. Behrmann, J. Bengtsson, A. David, K. G. Larsen, P. Pettersson, and W. Yi. Uppaal implementation secrets. In *In Proc. of FTRTFT-02*, pages 3–22, 2002.
4. B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Trans. on Software Eng*, 17(3):259–273, 1991.
5. B. Berthomieu and M. Menasche. An enumerative approach for analyzing time petri nets. In *In Proc. of IFIP83*, volume 9, pages 41–46, September 1983.
6. B. Berthomieu, P.O. Ribet, and F. Vernadat. The tool tina – construction of abstract state spaces for petri nets and time petri nets. In *International Journal of Production Research*, 2004.
7. B. Berthomieu and F. Vernadat. State class constructions for branching analysis of time petri nets. In *In Proc. of TACAS'03*, volume 2619 of LNCS, pages 442–457. Springer-Verlag, 2003.
8. H. Boucheneb and R. Hadjidj. Ctl* model checking for time petri nets. *Theoretical Computer Science*, TCS 353(1-3):208–227, 2006.
9. Alexandre David, Lasse Jacobsen, Morten Jacobsen, Kenneth Yrke Jørgensen, Mikael H. Møller, and Jirí Srba. Tapaal 2.0: Integrated development environment for timed-arc petri nets. In *TACAS*, pages 492–497, 2012.
10. D. Lime G. Gardey, M. Magnin, and O.H. Roux. Roméo: A tool for analyzing time petri nets. In *In 17th International Conference on Computer Aided Verification (CAV'05)*, 2005.
11. R. Hadjidj. Analyse et validation formelle des systèmes temps réel. *Ph.D. Theses,University of Montreal (Ecole plytechnique)*, 2006.
12. R. Hadjidj and H. Boucheneb. On the fly TCTL model checking for time petri nets using the state class method. In *In Proc of ACSD'06*, pages 111–120. IEEE Computer Society Press, 2006.
13. R. Hadjidj and H. Boucheneb. On the fly TCTL model checking for time petri nets. *Theoretical Computer Science*, TCS 410(42):4241–4261, 2009.
14. R. Hadjidj and H. Boucheneb. Efficient reachability analysis for time petri nets. *IEEE Transaction on Computers*, 60(8):1085–1099, 2011.
15. P. Merlin and D. J. Farber. Recoverability of communication protocols - implication of a theoretical study. *IEEE Trans. on Communications*, 24(9):1036–1043, 1976.

Part V

PNSE'13: Poster Abstracts

A Tool to Synthesize Intelligible State Machine Models from Choreography using Petri Nets*

Toshiyuki Miyamoto[1] and Hiroyuki Oimura[1]

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. We have proposed a method to synthesize hierarchical state machine models for the choreography realization problem in the last PNSE. In this paper, we present a prototypical tool for the method.

In recent years, the internationalization of activities and information technology in the enterprise has intensified competition between companies. Under such circumstances, service-oriented architecture (SOA)[6] 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 SOA, the problem to synthesize the concrete model from an abstract specification is known as the choreography realization problem[5]. 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[4] 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.

* This work was supported by KAKENHI (23500045).

Table 1. Number of simple states(NSS), transitions(NT), and guards(NG) of synthesized state machines.

ex	service	message	projection			CSCB		
			NSS	NT	NG	NSS	NT	NG
1	4	10	32	52	0	11	32	4
2	3	4	7	10	0	7	10	0
3	4	6	15	23	0	8	15	0
4	4	7	17	25	0	7	20	0
5	5	7	15	22	0	10	18	0
6	6	12	48	88	0	16	36	1
7	5	10	23	32	0	16	27	4

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 [3], we have proposed a method to synthesize hierarchical state machines by using Petri nets from a choreography defined by single communication diagram, where the method is called the CSCB method. So far, we have developed a prototypical tool of the CSCB method and the projection method in [2], and evaluated the CSCB method on several examples. Table 1 shows the results, and the CSCB method is better than the projection method in term of several metrics for the intelligibility by Antonio et al. We, however, found some points to be improved in the algorithm and the intelligibility.

We are going to import the prototypical tool into an UML modeling tool as a plug-in and extend the CSCB method so as to synthesize more intelligible 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. Miyamoto, T., Hasegawa, Y.: A Petri Net Approach to Synthesize Intelligible State Machine Models from Choreography. In: International Workshop on Petri Nets and Software Engineering 2012. pp. 222–236 (Jun 2012)
4. OMG: Unified modeling language, http://www.uml.org/
5. 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)
6. Thomas, E.: Service-Oriented Architecture. Prentice Hall (2004)

Transforming Platform Independent CPN Models into Code for the TinyOS Platform: A Case Study of the RPL Protocol

Vegard Veiset and Lars Michael Kristensen

Department of Computing, Bergen University College
Email: vegard.veiset@stud.hib.no,lmkr@hib.no

Abstract. TinyOS is a widely used platform for the development of networked embedded systems offering a programming model targeting resource constrained devices. We present a software engineering approach where Coloured Petri Net (CPNs) models are used as a starting point for developing protocol software for the TinyOS platform. The approach consists of five refinement steps where a platform-independent CPN model is gradually transformed into a platform-specific model that enables automatic code generation. To evaluate our approach, we use it to obtain an implementation of the IETF RPL routing protocol for sensor networks.

Introduction. Model-based software engineering and verification have several attractive properties in the development of flexible and reliable software systems. In order to fully leverage modelling investments, it is desireable to use the constructed models also for the implementation of the software on the platform under consideration. Coloured Petri Nets [2] (and Petri Nets in general) constitute a general purpose modelling language supporting platform-independent modelling of concurrent systems. Hence, in most cases, such models are too abstract to be used directly to implement software. In order to bridge the gap between abstract and platform independent CPN models and the implementation of software to be deployed, the concept of *pragmatics* was introduced in [4]. Pragmatics are syntactical annotations that can be added to a CPN model and used to direct code generation for a specific platform. The contribution of this paper is an approach [5] that exploits pragmatics in combination with a five step refinement methodology to enable code generation for the TinyOS platform. Applications for TinyOS [3] are implemented using the nesC programming language (a dialect of C) providing an event-based split-phase programming model. An application written in nesC is organised into a wired set of modules each providing an interface consisting of commands and events.

Refinement Steps. The model refinement starts from a platform independent CPN model constructed typically with the aim of specifying the protocol operation and performing model checking of the protocol design. Each step consists of a transformation that uses the constructs of the CPN modelling language to add details to the model. Furthermore, in each step pragmatics are added that direct the code generation performed after the fifth step:

Step 1: Component Architecture consists of annotating CPN submodules and substitution transitions corresponding to TinyOS components, and make explicit the interfaces used and provided by components.

Step 2: Resolving Interface Conflicts resolves interface conflicts allowing components to use multiple instances of an interface. This is done by annotating CPN arcs with information providing locally unique names.

Step 3: Component and Interface Signature adds type signatures to components and interfaces by creating explicit submodules for command and events, and by refining colour sets to reflect the interface signatures.

Step 4: Component Classification further refines the components by classifying them into four main types: timed, external, boot, and generic.

Step 5: Internal Component Behaviour consists of refining the modelling of the individual commands and events such that control flow and data manipulation become explicit and organised into atomic statement blocks.

After the fifth refinement step has been performed, the CPN model includes sufficient detail to be used as a basis for automated code generation.

The RPL Protocol and Code Generation. To evaluate our approach on an industrial-sized example, we have conducted a case study based on the RPL routing protocol [1] developed by the Internet Engineering Task Force. The RPL protocol allows a set of sensor nodes to construct a destination-oriented directed acyclic graph which can be used for multi-hop communication between sensor nodes. To support the automatic code generation for TinyOS, we have developed a software prototype in Java that performs a template-based model-to-text transformation on the models resulting from the fifth refinement step. The software prototype relies on the Access/CPN framework [6] to load CPN models created with CPN Tools. The code generator performs a top-down traversal of the CPN model where code templates are selected according to the pragmatic annotations on the CPN model elements encountered.

References

1. T. Winter et. al. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550, 2012. Internet Engineering Task Force.
2. K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, 2007.
3. P. Levis. *TinyOS Programming.* Cambridge University Press, 2009.
4. K. Simonsen, L.M. Kristensen, and E. Kindler. Code Generation for Protocols from CPN Models Annotated with Pragmatics. In *Proc. of NWPT'12*, volume 403 of *Report in Informatics*, pages 46–48. University of Bergen, 2012.
5. V. Veiset. An Approach to Semi-Automatic Code Generation for the TinyOS Platform using Coloured Petri Nets. Master's thesis, Bergen University College, 2013.
6. M. Westergaard. Access/CPN 2.0: A High-Level Interface to CPN Models. In *Proc. of ICATPN'11*, volume 6709 of *LNCS*, pages 328–337. Springer, 2011.