

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

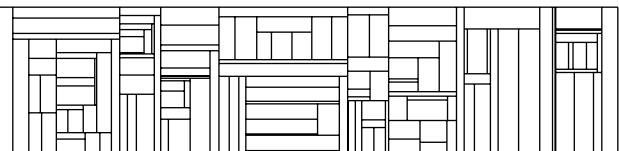
Aarhus, Denmark, October 22-24, 2007

Kurt Jensen (Ed.)

DAIMI PB - 584

October 2007

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



Preface

This booklet contains the proceedings of the Eighth Workshop on Practical Use of Coloured Petri Nets and the CPN Tools, October 22-24, 2007. The workshop is organised by the CPN group at Department of Computer Science, University of Aarhus, Denmark. The papers are also available in electronic form via the web pages: <http://www.daimi.au.dk/CPnets/workshop07/>

Coloured Petri Nets and the CPN Tools are now licensed to more than 5.600 users in 129 countries. The aim of the workshop is to bring together some of the users and in this way provide a forum for those who are interested in the practical use of Coloured Petri Nets and their tools.

The submitted papers were evaluated by a programme committee with the following members:

Wil van der Aalst, Netherlands
João Paulo Barros, Portugal
Jonathan Billington, Australia
Jörg Desel, Germany
Joao M. Fernandes, Portugal
Jorge de Figueiredo, Brazil
Monika Heiner, Germany
Thomas Hildebrandt, Denmark
Kurt Jensen, Denmark (chair)
Ekkart Kindler, Denmark
Lars M. Kristensen, Denmark
Johan Lilius, Finland
Daniel Moldt, Germany
Laure Petrucci, France
Rüdiger Valk, Germany
Lee Wagenhals, USA
Jianli Xu, Finland
Karsten Wolf, Germany

The programme committee has accepted 13 papers for presentation. Most of these deal with different projects in which Coloured Petri Nets and their tools have been put to practical use – often in an industrial setting. The remaining papers deal with different extensions of tools and methodology.

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

Kurt Jensen

Table of Contents

Invited Tutorials:

Lars M. Kristensen and Michael Westergaard

- The ASCoVeCo State Space Analysis Platform: Next Generation Tool
Support for State Space Analysis 1

Ekkart Kindler

- Component Tools: A Frontend for Formal Methods 7

Regular Papers:

Paul Fleischer and Lars M. Kristensen

- Towards Formal Specification and Validation of Secure Connection
Establishment in a Generic Access Network Scenario 9

E. Bacarin, W.M.P van der Aalst, E. Madeira, and C. B. Medeiros

- Towards Modeling and Simulating a Multi-party Negotiation Protocol with
Colored Petri Nets 29

Jonathan Billington and Amar Kumar Gupta

- Effectiveness of Coloured Petri nets for Modelling and Analysing the
Contract Net Protocol 49

Carmen Bratosin, Wil van der Aalst, and Natalia Sidorova

- Modeling Grid Workflows with Colored Petri Nets 67

Karolina Zurowska and Ralph Deters

- Overcoming Failures in Composite Web Services by Analysing Colored Petri
Nets 87

Nick Russell, Arthur H.M. ter Hofstede and Wil M.P. van der Aalst

- newYAWL: Specifying a Workflow Reference Language using Coloured
Petri Nets 107

Kristian Bisgaard Lassen and Simon Tjell

- Translating Colored Control Flow Nets into Readable Java via Annotated
Java Workflow Nets 127

Visar Januzaj

- CPNunf: A tool for McMillan's Unfolding of Coloured Petri Nets 147

Christine Choppy, Laure Petrucci, and Gianna Reggio

- Designing coloured Petri net models: a method 167

*R.S. Mans, W.M.P. van der Aalst, P.J.M. Bakker, A.J. Moleman, K.B. Lassen
and J.B. Jørgensen*

- From Requirements via Colored Workflow Nets to an Implementation in
Several Workflow Systems 187

Jóao M. Fernandes, Simon Tjell, and Jens Bæk Jørgensen

- Requirements Engineering for Reactive Systems with Coloured Petri Nets:
the Gas Pump Controller Example? 207

Óscar R. Ribeiro and Jóao M. Fernandes

- On the Use of Coloured Petri Nets for Visual Animation 223

Kristian L. Espensen, Mads K. Kjeldsen, and Lars M. Kristensen

- Towards Modelling and Validation of the DYMO Routing Protocol for
Mobile Ad-hoc Networks 243

The ASCoVeCo State Space Analysis Platform: Next Generation Tool Support for State Space Analysis*

Invited Tutorial

Lars M. Kristensen and Michael Westergaard

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

Extended Abstract

State space analysis is one of the main approaches to model-based verification of concurrent systems and is one of the most successfully applied analysis methods for Coloured Petri Nets (CP-nets or CPNs) [13, 16, 17]. The basic idea of state space exploration and analysis is to compute all reachable states and state changes of the concurrent system under consideration and represent these as a directed graph. From a constructed state space it is possible to verify and analyse a large class of behavioural properties by considering, e.g., the standard behavioural properties of CP-nets, traverse a constructed state space by means of user-defined queries, or conduct LTL and CTL model checking [4, 10]. Another main advantage of state space analysis is that it can be supported by computer tools in a highly automatic way, and it can provide counter examples demonstrating why the system does not have a certain property.

The main limitation of using state spaces to verify behavioural properties of systems is the *state explosion problem* [24], i.e., that state spaces of systems may have an astronomical number of reachable states which means that they are too large to be handled with the available computing power (CPU speed and memory). Methods for alleviating this inherent complexity problem is an active area of research and has led to the development of a large collection of *state space reduction methods*. These methods have significantly broadened the class of systems that can be verified and state spaces can now be used to verify systems of industrial size. Some of these methods [2, 15, 14, 25] have been developed in the context of the CPN modelling language. Other methods (e.g., [23, 22, 26, 11]) have been developed outside the context of the CPN modelling language, but state space reduction methods are generally independent of any concrete modelling language and hence also applicable for CP-nets.

A computer tool supporting state spaces must implement a wide range of state space reduction methods since no single reduction method works well on all systems. Both CPN Tools [5] and its predecessor Design/CPN [6] have supported state space analysis in its most basic form and experimental prototype libraries have been developed supporting state space reduction methods such as the symmetry method [15, 18], the equivalence method [14, 19], time condensed state spaces [3], and the sweep-line method [2, 21]. The software architectures of CPN Tools and Design/CPN have, however, made it difficult to support a collection of state space reduction methods in a coherent manner in these tools.

This tutorial presents the ASCoVeCo State Space Analysis Platform (ASAP) which is currently being developed in the context of the ASCoVeCo research project [1]. ASAP represents the next generation of tool support for state space exploration and analysis of CPN

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

models. The aim and vision of ASAP is to provide an open platform suited for research, educational, and industrial use of state space exploration. This means that the ASAP will support a wide collection of state space exploration methods and have an architecture that allows the research community to extend the set of supported methods. Furthermore, ASAP will be sufficiently mature to be used for educational purposes, including teaching of advanced state space methods, as well as sufficiently mature to be used in industrial projects as has been the case with CPN Tools and Design/CPN. ASAP is a stand-alone tool and is able to load models created with CPN Tools. ASAP will be available for Windows XP/Vista, Linux, and Mac OS X.

The ASAP platform consists of a graphical user interface (GUI) and a state space exploration engine (SSE engine). Figure 1(left) shows the software architecture of the graphical user interface which is implemented in Java based on the Eclipse Rich Client Platform (RCP) [9]. The software architecture of the SSE engine is shown in Fig. 1(right). It is based on Standard ML (SML) and relies on the CPN simulator used in CPN Tools. The SSE engine implements the state space **Exploration** and model checking algorithms supported by ASAP. The state space exploration and model checking algorithms implemented rely on a set of **Storage** and **Waiting Set** components for efficient storage and exploration of state spaces. Furthermore, the SSE engine implements the **Query Languages(s)** used for writing state space queries and to verify properties.

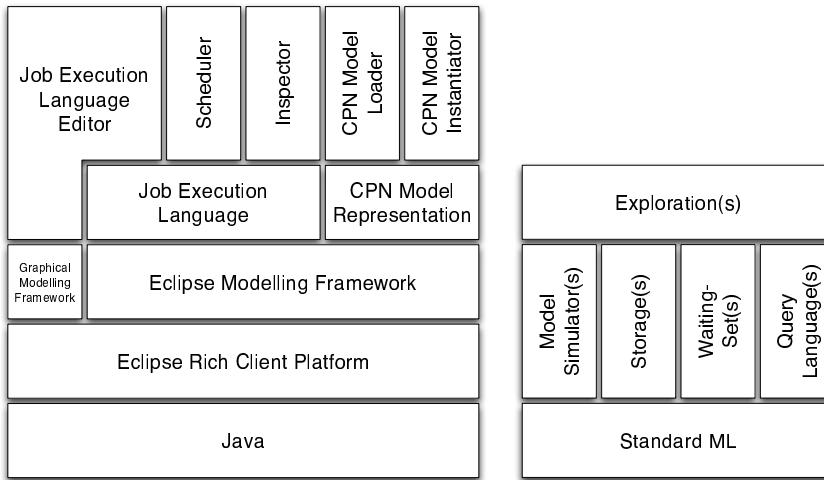


Fig. 1. ASAP platform architecture: GUI (left) and SSE engine (right).

The ASAP GUI makes it possible to create and manage *verification projects* consisting of a collection of *verification jobs*. The GUI has three different *perspectives* for working with verification projects:

- An *editing perspective* for creating and editing verification jobs.
- An *execution perspective* for controlling the execution of verification jobs.
- An *inspection perspective* for inspecting and interpreting analysis results.

Verification jobs are constructed and specified using the verification Job Execution Language (JEL) and the JEL Editor. JEL is a graphical language inspired by data-flow diagrams that

makes it possible to specify the CPN models, queries, state space explorations, and processing of analysis results that constitute a verification job. JEL and the JEL Editor are implemented using the Eclipse Modelling Framework (EMF) [8] and the Graphical Modelling Language [7] (GMF). The ASAP GUI additionally has a Model Loader component and a Model Instantiation component that can load and instantiate CPN models created with CPN Tools. Hence, models created using CPN Tools can be used directly with ASAP and since ASAP relies on the same underlying CPN simulator as CPN Tools, there is no CPN semantical gap between the two tools. It is worth noticing that it is only the CPN Model Loader, CPN Model Instantiator, and CPN Model Representation components that are specific to CPN models. The other components are independent of the CPN modelling formalism.

Figure 2 shows a snapshot of the graphical user interface in the editing perspective. The user interface consists of three main parts apart from the usual menus and tool-bars at the top.

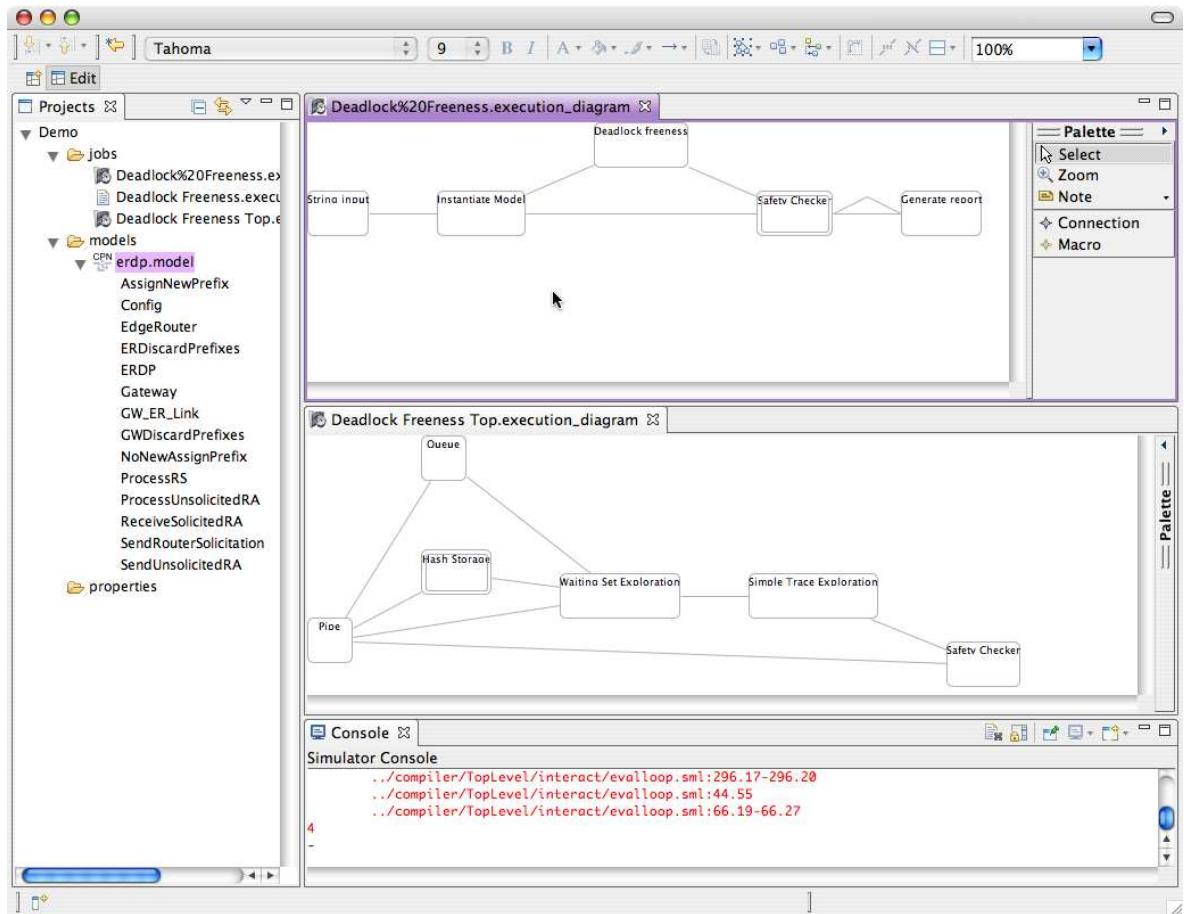


Fig. 2. Snapshot of the graphical user interface – editing perspective.

To the left is an overview of the verification projects loaded, in this case just a single project named **Demo** is loaded. A verification project consists of a number of verification jobs, models, and queries. In this case there are three verification job all concerned with checking

deadlock properties. A CPN model named `erdp` is loaded, which is a CPN model of an edge router discovery protocol [20]. At the bottom of the user interface is a console, which makes it possible to interact directly with the SSE engine using SML. This makes it easy to experiment with the tool and to issue queries that need not be stored as part of the verification project. The large area at the top-right is the editing area. The editing area can be used to edit queries and verification jobs. In this case two jobs are being edited and the two windows shows the graphical representation in JEL of the parts being edited.

ASAP has an interface to the CPN simulator providing a set of primitives that makes it easy to access the transition relation of the CPN model and thereby augment the platform with new state space exploration and model checking algorithms. The primitives available is specified by the `MODEL_SIMULATOR` SML signature (interface) listed in Fig. 3. The signature deals with `states` and `events` (ll. 3–4). It is possible to get the initial states of the model (l. 9). Each state is represented as the actual state and all enabled events in that state. In order to accommodate non-deterministic formalisms, we allow a number of initial states, so a list of states is returned instead of just a single state. From a state and an event, it is possible to get (all) successor states (l. 12), and from a state and a sequence (list) of events, it is possible to get (all) states that are the result of executing the sequence of events from the specified state (l. 16). If a user tries to execute an event that is not enabled, the `EventNotEnabled` exception (l. 6) is raised in both `nextStates` and `executeSequence`.

```

1  signature MODEL_SIMULATOR =
2  sig
3      eqtype state
4      eqtype event
5
6      exception EventNotEnabled
7
8      (* --- get the initial states and enabled events in each state --- *)
9      val getInitialStates : unit -> (state * event list) list
10
11     (* --- get the successor states and enabled events in each successor state --- *)
12     val nextStates : state * event -> (state * event list) list
13
14     (* --- execute an event sequence and return the list of resulting states --- *)
15     (* --- and enabled events --- *)
16     val executeSequence : state * event list -> (state * event list) list
17 end

```

Fig. 3. SML signature of the model simulator interface.

The SSE engine currently implements full state space exploration, bit-state hashing [11], hash compaction [26], the comeback method [25], state caching [12], and the equivalence method [14] based on canonicalisation of equivalent markings. All these methods have been implemented using the CPN simulator interface in Fig. 3 and the storage and waiting set components provided by the SSE engine. Currently only verification of safety properties is supported by ASAP. State space exploration can be done both breadth-first and depth-first. It is worth observing that the model simulator interface allows the state space exploration

and model checking algorithms to be implemented such that they are independent from a concrete modelling language.

As part of the development of ASAP, we have also developed a test and benchmarking tool containing a collection of small, medium, and large CPN models. This benchmarking suite makes it simple to profile the performance (e.g., time and space usage) of state space methods and their implementation. The benchmarking tool includes an SQL database where profiling data can be stored and a web-based GUI that makes it possible to query the database and display the results graphically.

During the CPN workshop we plan to make a first technology preview release of ASAP. For version 1.0 we plan to complete the implementation of the simulator interface in the SSE engine, and additionally implement the sweep-line method [2, 21], time condensed state spaces [3], and CTL and LTL model checking [10, 4]. Also, we plan to implement and provide the same set of state space query functions available in the state space tool of CPN Tools. In the user interface we will complete the implementation of the JEL Editor, implement an SML Query Editor, and implement the inspection perspective. Also, we will implement a functionality similar to the *state space report* and a simple interactive and automatic visualisation of state spaces as known from CPN Tools.

Acknowledgements. The authors wish to acknowledge the work of Surayya Urazimbetova and Mads K. Kjeldsen who are significantly contributing to the development of ASAP via their employment as student programmers in the ASCoVeCo project.

References

1. The ASCoVeCo Project. www.daimi.au.dk/~ascoveco.
2. S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proc. of TACAS'01*, volume 2031 of *LNCS*, pages 450–464. Springer-Verlag, 2001.
3. S. Christensen, L.M. Kristensen, and T. Mailund. Condensed State Spaces for Timed Petri Nets. In *Proc. of 22nd International Conference on Application and Theory of Petri Nets*, volume 2075 of *Lecture Notes in Computer Science*, pages 101–120. Springer-Verlag, 2001.
4. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.
5. CPN Tools. www.daimi.au.dk/CPNTools.
6. Design/CPN. www.daimi.au.dk/designCPN.
7. Eclipse Graphical Modelling Framework (GMF). www.eclipse.org/modeling/gmf/.
8. Eclipse Modelling Framework (EMF). www.eclipse.org/modeling/emf/.
9. Eclipse Rich Client Platform (RCP). www.eclipse.org/home/categories/rcp.php.
10. E. A. Emerson. *Temporal and Modal Logic*, volume B of *Handbook of Theoretical Computer Science*, chapter 16, pages 995–1072. Elsevier, 1990.
11. G.J. Holzmann. An Analysis of Bitstate Hashing. *Formal Methods in System Design*, 13:289–307, 1998.
12. C. Jard and T. Jeron. Bounded-memory Algorithms for Verification On-the-fly. In *Proc. of CAV'91*, volume 575 of *LNCS*, pages 192–202. Springer-Verlag, 1991.
13. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1:Basic Concepts*. Monographs in Theoretical Computer Science. Springer-Verlag, 1992.
14. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2: Analysis Methods*. Monographs in Theoretical Computer Science. Springer-Verlag, 1994.
15. K. Jensen. Condensed State Spaces for Symmetrical Coloured Petri Nets. *Formal Methods in System Design*, 9, 1996.
16. K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer-Verlag, In preparation.
17. K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. In *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3-4):213–254, 2007.

18. J.B. Jørgensen and L.M. Kristensen. Computer Aided Verification of Lamports Fast Mutual Exclusion Algorithm Using Coloured Petri Nets and Occurrence Graphs with Symmetries. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):714–732, 1999.
19. J.B. Jørgensen and L.M. Kristensen. Verification of coloured petri nets using state spaces with equivalence classes. In *Petri Net Approaches for Modelling and Validation*, volume 1 of *LINCOM Studies in Computer Science*, chapter 2, pages 17–34. Lincoln Europa, 2003.
20. L.M. Kristensen and K. Jensen. Specification and validation of an edge router discovery protocol for mobile ad-hoc networks. In *Proc. of Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 248–269. Springer-Verlag, 2004.
21. L.M. Kristensen and T. Mailund. A Generalised Sweep-Line Method for Safety Properties. In *Proc. of FME’02*, volume 2391 of *LNCS*, pages 549–567. Springer-Verlag, 2002.
22. D. Peled. All from One, One for All: On Model Checking Using Representatives. In *Proceedings of CAV’93*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1993.
23. A. Valmari. Error Detection by Reduced Reachability Graph Generation. In *Proceedings of the 9th European Workshop on Application and Theory of Petri Nets*, pages 95–112, 1988.
24. A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.
25. M. Westergaard, L.M. Kristensen, G.S. Brodal, and L.A. Arge. The ComBack Method – Extending Hash Compaction with Backtracking. In *Proc. of ICATPN’07*, volume 4546 of *Lecture Notes in Computer Science*, pages 445–464. Springer-Verlag, 2007.
26. P. Wolper and D. Leroy. Reliable Hashing without Collision Detection. In *Proc. of CAV’93*, volume 697 of *LNCS*, pages 59–70. Springer-Verlag, 1993.

Component Tools: A Frontend for Formal Methods

Ekkart Kindler, Technical University of Denmark, Copenhagen

Abstract

Petri Nets are a powerful technique for modelling, analysing, and validating all kinds of systems. In order to exploit the full power of Petri nets, different versions of Petri nets and tools need to be used, and, sometimes, we need to combine them with other techniques. This makes it necessary to model the same system in different Petri net formalisms or other notations over and over again. Moreover, systems will often be constructed from standard components so that the modeller would like to build his system from these components. This way, even users with no experience in Petri nets can model systems. In order to help this group of users, the analysis results obtained by Petri net tools or other formal methods must be presented and visualized independently from Petri nets. Component Tools is a platform designed for this purpose. Along with a 3D-visualisation of the behaviour of the system (PNVis), non-experts can build Petri net models and see their model running in (virtual) reality.

The tutorial will cover the concepts of ComponentTools as well as the concepts and modelling philosophy of the 3D-visualisation. Moreover, the tutorial will discuss the underlying software technology which makes it easy to implement this kind of tools. In addition to the existing concepts the tutorial will give an overview of ideas for future work and challenging research projects -- with the potential for many PhD theses.

References

- E. Kindler and F. Nillies: Petri Nets and the Real World.
Petri Net Newsletter 70, Cover Picture Story, pp. 3-8, April 2006.
<http://www.upb.de/cs/kindler/publications/copies/PRW-PNNL70.pdf>
- E. Kindler and C. Páles: 3D-Visualization of Petri Net Models: Concept and Realization.
In: J. Cortadella and W. Reisig (eds.): International Conference on Theory and Application of Petri Nets 2004, 25th International Conference, Bolgna, Italy. Springer, LNCS 3099: 464-473, June 2004.
- E. Kindler, V. Rubin, and R. Wagner: Component Tools: Application and Integration of Formal Methods.
In: Electronic proceedings of the Workshop Object Orientierte Software Entwicklung 2005 (OOSE '05), Satellite event of Net.ObjectDays 2005, Erfurt, Germany, September 2005.
<http://www.upb.de/cs/kindler/publications/copies/OOSE05.pdf>
- E. Kindler, V. Rubin, and R. Wagner: Component Tools: Integrating Petri nets with other formal methods.
International Conference on Theory and Application of Petri Nets 2006, 27th International Conference, Turku, Finland. June 2006. LNCS 4024, pp. 37-56.
- E. Kindler and R. Wagner: Triple Graph Grammars:
Concepts, Extensions, Implementations, and Application Scenarios.
Tech. Rep. tr-ri-07-284, Software Engineering Group, Department of Computer Science, University of Paderborn, June 2007.
<http://www.upb.de/cs/ag-schaefer/Veroeffentlichungen/Quellen/Papers/2007/tr-ri-07-284.pdf>

Towards Formal Specification and Validation of Secure Connection Establishment in a Generic Access Network Scenario*

Paul Fleischer and Lars M. Kristensen

Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark,
`{pf,kris}@daimi.au.dk`

Abstract. The Generic Access Network (GAN) architecture is defined by the 3rd Generation Partnership Project (3GPP), and allows telephone services, such as SMS and voice-calls, to be accessed via generic IP networks. The main usage of this is to allow mobile phones to use WiFi in addition to the usual GSM network. The GAN specification relies on the Internet Protocol Security layer (IPSec) and the Internet Key Exchange protocol version 2 (IKEv2) to provide encryption across IP networks, and thus avoid compromising the security of the telephone networks. The detailed usage of these two internet protocols (IPSec and IKEv2) is only roughly sketched in the GAN specification. As part of the process to develop solutions to support the GAN architecture, TietoEnator Denmark has developed a more detailed GAN scenario which describes how IPSec and IKEv2 are to be used during the connection establishment procedure. The contribution of this paper is to present a CPN model developed to formally specify and validate the detailed GAN scenario considered by TietoEnator.

1 Introduction

The Generic Access Network (GAN) [1] architecture as specified by the 3rd Generation Partnership Project (3GPP) [2] allows access to common telephone services such as SMS and voice-calls via generic Internet Protocol (IP) [3] networks. The operation of GAN is based on a *Mobile Station* (e.g., a cellular phone) opening an encrypted tunnel to a *Security Gateway* via an IP-network. A *GAN Controller* is responsible for relaying the commands sent via this tunnel to the telephone network, which in turn allows mobile stations to access the services on the telephone network. The Security Gateway and the GAN Controller can either reside on the same physical machine or on two separate machines communicating by IP. The encrypted tunnel is provided by the Encapsulating Security Payload (ESP) mode of the IP security layer (IPSec) [4]. In order to provide such an encrypted tunnel, both ends have to authenticate each other, and agree

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

on both encryption algorithm and keys. This is achieved using the Internet Key Exchange v2 (IKEv2) protocol [5]. The GAN specification [1] merely states that IKEv2 and IPSec are to be used, and in which operating modes. However, what that means for the message exchange is not specified, and is left to the IKEv2 and IPSec standards. As such, there is no clear specification of the IKEv2 message exchange and the states that the protocol entities are to be in when establishing a GAN connection.

TietoEnator Denmark [6] is working on providing solutions to support the GAN architecture. Prior to the implementation, a textual usage scenario was formulated [7] which constitutes a specific instantiation of the full GAN architecture. The purpose of this scenario was two-fold. First, it defines the scope of the software to be developed, i.e., which parts of the full GAN specification are supposed to be supported. Secondly, the scenario describes thoughts about the initial design of both the software and the usage of it. The scenario describes the details of how a mobile station is configured with an IP-address using DHCP [8] and then establishes an ESP-tunnel [9] to the Security Gateway using IKEv2 [5] and IPSec [4]. At this point, the mobile station is ready to communicate securely with the GAN Controller. The focus of the scenario is the establishment of the secure tunnel and initial GAN message exchanges which are the detailed parts omitted in the full GAN specification. Throughout this paper the term *GAN scenario* refers to the detailed scenario [7] described by TietoEnator, while *GAN architecture* refers to the generic architecture as specified in [1].

The contribution of this paper is to describe the results of a project at TietoEnator, where Coloured Petri Nets [10] were used as a supplement to a textual description of the GAN scenario to be implemented. The model has been constructed from the material available from TietoEnator [7], the GAN specification [1], and the IKEv2 specification [5]. The CPN model deals only with the connection establishing aspect of the GAN architecture, as this is the main focus of the TietoEnator project. As the scenario from TietoEnator deals with the aspect of configuring the mobile station with an initial IP-address, the model does not only cover the communication of the GAN protocol, but also of the DHCP messages and actual IP-packet flow. The CPN model includes a generic IP-stack model, which supports packet forwarding and ESP-tunnelling. This modelling approach was chosen to allow the model to be very close to the scenario description used by TietoEnator with the aim of easing the understanding of the model for TietoEnator engineers which will eventually implement the GAN scenario. The model was presented to the engineers at two meetings. Each meeting resulted in minor changes of the model. Even though the TietoEnator engineers have did not have any experience with Coloured Petri Nets, they quickly accepted the level of abstraction and agreed that the scenario was the same as they had described by text.

Coloured Petri Nets have been used to model various Internet protocols (e.g., [11][12]). The general approach of modelling Internet protocols is to abstract as much of the environment away, and only deal with the core of the protocol. The advantage of this approach is that the model gets simpler and the analysis

becomes easier due to restricted state space size. The approach presented in this paper also makes use of abstraction. However, the chosen abstraction level is based on a scenario, rather than a single protocol specification. This gives a complete picture of the scenario, rather than a single protocol. This is an advantage when working with design of actual protocol implementations as it gives an overview of the needed features and component interaction. The main drawback is that the model becomes larger and more difficult to analyse. Such a scenario model is not well suited for checking properties of a single protocol as done in [11] and [12].

The rest of this paper is organised as follows. Sect. 2 gives an introduction to the GAN scenario as defined by TietoEnator and presents the top-level of the constructed CPN model. Sect 3 and Sect. 4 present selected parts of the constructed CPN model including a discussion of the modelling choices made. In Sect. 5 we explain how the model of the GAN specification was validated using simulation and state space analysis. Finally, in Sect. 6 we sum up the conclusions and discuss future work. The reader is assumed to be familiar with the CPN modelling language [10] and CPN Tools [13].

2 The GAN Scenario

This section gives an introduction to the GAN scenario [7] as defined by TietoEnator and the constructed CPN model. Fig. 1 shows the top-level module of the constructed CPN model. The top-level module has been organised such that it reflects the network architecture of the GAN scenario. The six substitution transitions represent the six network nodes in the scenario and the four places with thick lines represent networks connected to the network nodes. The places with thin lines connected to the substitution transitions Provisioning Security Gateway, Default Security Gateway, Provisioning GAN Controller, and Default GAN Controller are used to provide configuration information to the corresponding network nodes. The module has been annotated with network and interface IP addresses to provide that information at the topmost level.

The substitution transition **MobileStation** represents the mobile station which is connecting to the telephone network via a generic IP-network. The place **Wireless Network** connected to **MobileStation** represents the wireless network which connects the mobile station to a wireless router represented by the substitution transition **WirelessRouter**. The wireless router is an arbitrary access point with routing functionality, and is connected to the **ProvisioningSecurityGateway**, through **Network B**. The provisioning security gateway is connected to the **ProvisioningGANController** via **Network C**. There is a second pair of security gateway (**DefaultSecurityGateway**) and GAN controller (**DefaultGANController**). The provisioning GAN controller is responsible for authenticating any connection, and redirecting them to another GAN controller in order to balance the load over a number of GAN controllers. In the GAN scenario, the default GAN controller represents the GAN controller which everything is redirected to. It is worth mentioning, that the generic GAN architecture sees the security gateway as a

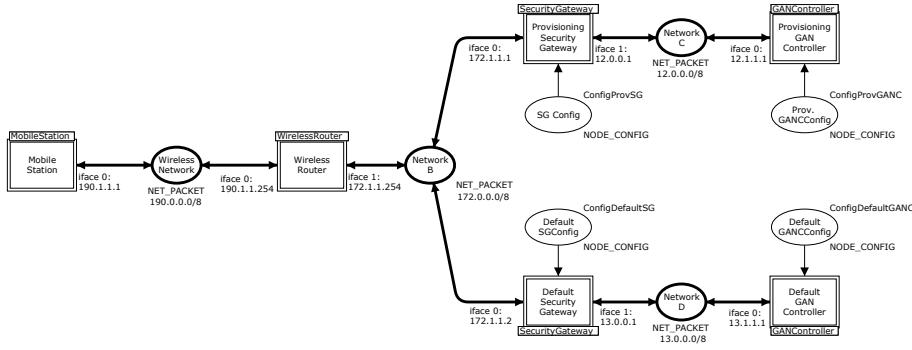


Fig. 1. Top-level module the GAN scenario CPN model.

component within the GAN controller. However, TietoEnator decided to separate the two which makes the message exchange more clear. Neither the Wireless Router nor Network B are required to be owned or operated by the telephone operator. However, all security gateways, GAN controllers, and Network C and Network D are assumed to be under the control of the telephone operator, as non-encrypted messages will be exchanged across them.

The basic exchange of messages in the GAN scenario consists of a number of steps as depicted in the Message Sequence Charts (MSCs) in Figs. 2-4. The MSCs have been generated from the constructed CPN model using the BritNeY animation tool [14].

The scenario assumes that the Mobile Station is completely off-line to begin with. It then goes through 5 steps: Acquire an IP address using DHCP, Create a secure tunnel to the provisioning security gateway, use the GAN protocol to acquire the addresses of the security gateway and GAN controller to use for further communication, create a secure tunnel to the new security gateway, and finally initiate a GAN connection with the new GAN controller. The last step is not modelled, and step 4 is similar to step 2 and will as such not be treated separately.

The first step is to create a connection to an IP-network which is connected to the Provisioning Security Gateway of the service provider. It is assumed that a physical connection to the network is present. This step is depicted in Fig. 2 where the Mobile Station sends a DHCP Request to the Wireless Router and re-

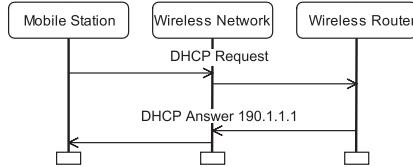


Fig. 2. MSC showing DHCP step of connection establishment.

ceives a DHCP Answer containing the IP address. The Mobile Station is assumed to be equipped with either the domain name or IP address of the provisioning security gateway and the provisioning GAN controller. In this paper, we assume that the IP address is known, as this allows us to not model the DNS server and name lookup.

Having obtained an IP-address via DHCP, the mobile station can now start negotiating the parameters for the secure tunnel with the provisioning security gateway using IKEv2. This is illustrated in the MSC shown in Fig. 3. This is done in 3 phases. The first phase is the initial IKEv2 exchange, where the two parties agree on the cryptographic algorithms to use, and exchange Diffie-Hellman values in order to established a shared key for the rest of the message exchanges. The second phase is the exchange of Extensible Authentication Protocol (EAP) messages. The idea of EAP is that it is possible to use any kind of authentication protocol with IKEv2. In this situation, a protocol called *EAP-SIM* is used. As can be seen in Fig. 3, the Provisioning Security Gateway initiates the

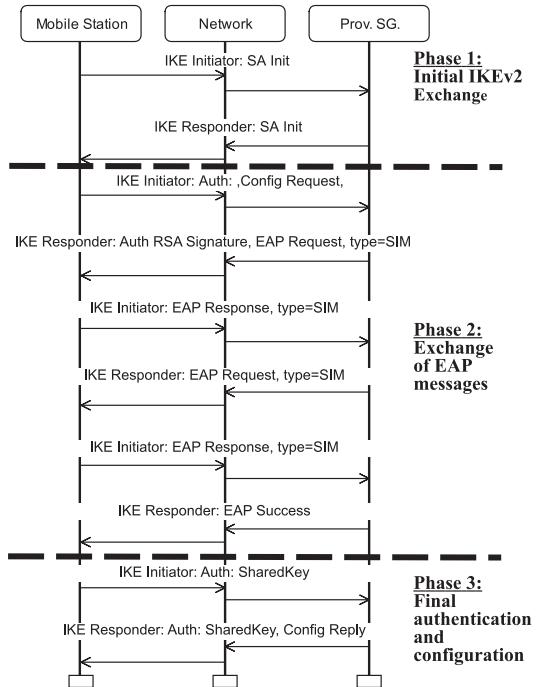


Fig. 3. MSC showing IKE step of connection establishment.

EAP message exchange by returning an *EAP Request* to the Mobile Station's authentication request. The actual EAP-SIM protocol exchanges 4 messages (2 requests and 2 responses) before it succeeds. As an result of the EAP-phase, the two parties have a shared secret key. In the third phase the Mobile Station

uses this shared key to perform final authentication. The last packet sent by the Provisioning Security Gateway contains the network configuration for the Mobile Station needed to establish a secure tunnel.

Having established the secure tunnel, the Mobile Station can open a secure connection to the Provisioning GAN Controller and register itself. This is shown in the MSC in Fig. 4. If the Provisioning GAN Controller accepts the Mobile Station, it sends a redirect message, stating a pair of security gateway and GAN controller to use for any further communication. The Mobile Station closes the connection to the Provisioning GAN Controller and the Provisioning Security Gateway. The final two steps of establishing a connection are to negotiate new IPSec tunnel parameters with the new security gateway, and establish a connection to the GAN controller. Having established the connection, the scenario ends. Fig. 4 only shows the registration with the Provisioning Security Gateway.

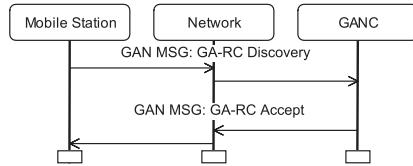


Fig. 4. MSC showing GAN step of connection establishment.

The scenario modelled involves multiple layers of the IP network stack. DHCP, used to configure the mobile station, is a layer 2 protocol, while IKE is a layer 4 protocol, and GAN is a layer 5 protocol. In order to accommodate all these layers in the model, a rather detailed representation of the IP components has been made. However, where simplifications were possible they have been made. For instance, the GAN protocol uses TCP, but TCP has not been modelled. Instead the network is currently lossless, but may reorder packets. This is not a problem, as GAN messages can be received out of order without messing things up. This is due to the fact, that the GAN Client only receives exactly the message it expects. The IP model contains routing which behaves similarly to the routing procedures implemented in real IP stacks. Some simplifications have been made to the routing algorithm, however, the end behaviour is the same. Each network node connected to an IP network has a routing table which contains information on how to deliver IP packets. In its simplest form, the entries in a routing table are pairs of destination network address and next hop, describing what the next hop is for IP packets matching the network address. In the IP model presented in this paper, IP addresses assigned to local interfaces have entries in the routing table as well, with a special next hop value. This is usually not the case for IP-stacks, but simplifies the routing model as ingoing routing can be performed without inspecting the list of interfaces. The ESP tunnel, which is used to secure the communication between the mobile station and the security gateway, is a part of the IPSec protocol suite, which is an

extension to the IP stack. Only enough of IPSec is modelled to support this tunnelling, and as such IPSec is not modelled. There are two components that are related to IPSec in the model: Routing and the Security Policy Database (SPD). The routing system ensures that packets are put into the ESP tunnel, and extracted again at the other end. The SPD describes what packets are allowed to be sent and received by the IP-stack, and is also responsible for identifying which packets are to be tunnelled. Each entry in the SPD contains the source and destination addresses to use for matching packets, and an action to perform. Modelled actions are *bypass*, which means allow, and *tunnel*, which means that the matched packet is to be sent through an ESP tunnel.

3 Modelling the GAN Network Nodes

The CPN module is organised in 31 modules, with the top-level module being depicted in Fig. 1. The top module has four submodules: MobileStation, WirelessRouter, SecurityGateway and GANController. Each of these modules has one or more protocol modules and an IP layer module. The modelling of the protocol entities and the IP layer will be discussed in Sect. 4.

In this section we present these modules in further detail. It can be seen that the provisioning and default GAN controller is represented using the same module and the same is the case with the provisioning and default security gateways.

3.1 Mobile Station

Fig. 5 shows the MobileStation module. The IP Layer substitution transition represents the IP layer of the mobile station and the Physical Layer substitution transition represents the interface to the underlying physical network. To the left are the three places which configure the IP layer module with a Security Policy Database, a Routing Table, and IP and MAC Addresses of the mobile station. These are also accessed in the DHCP, IKE, and GAN modules, as the configuration places are fusion places. This has been done to reduce the number of arcs in the module and since the security policy database, routing table, and addresses are globally accessible in the mobile station. The remaining substitution transitions model the steps that the mobile station goes through when establishing a GAN connection. The mobile station is initially in a Down state represented by a unit token on the place Down.

There are two tokens on the Addresses place representing the MAC and IP addresses assigned to the interfaces of the mobile station. The ADDR colour set is defined as follows:

```
colset IP_ADDR      = product INT * INT * INT * INT;
colset MAC_ADDR     = INT;
colset IFACE        = INT;
colset IFACExIP_ADDR = product IFACE * IP_ADDR;
```

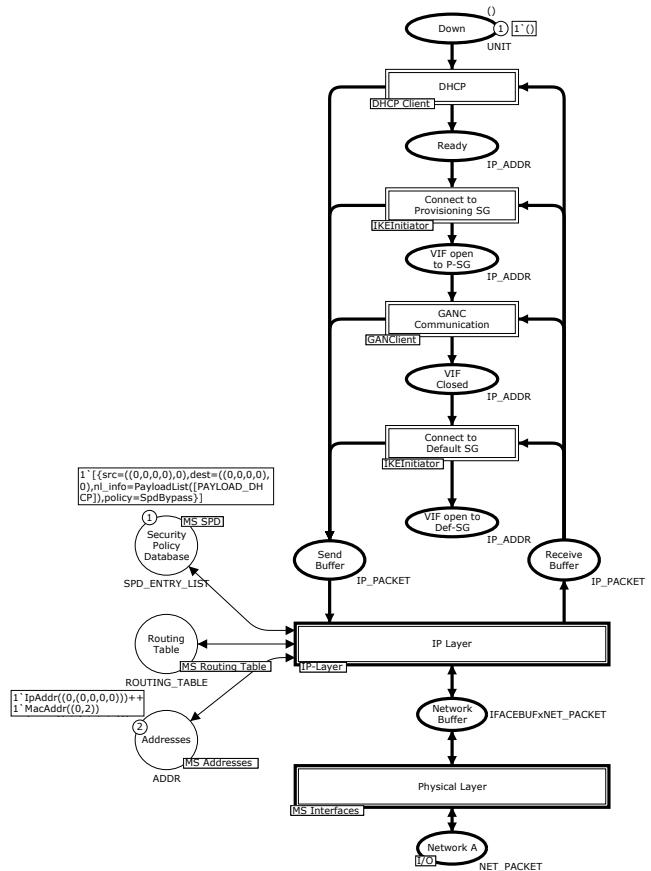


Fig. 5. The MobileStation module.

```

colset IFACEExMAC_ADDR = product IFACE * MAC_ADDR;
colset ADDR           = union IpAddr : IFACEExIP_ADDR +
                           MacAddr : IFACEExMAC_ADDR;

```

IP addresses are represented as a product of four integers, one for each octet of the address it represents. So, the IP address 192.0.0.1 becomes (192,0,0,1). MAC addresses and interfaces are represented as integers. On Fig. 5, `IpAddr((0,(0,0,0,0)))` means that interface 0 is assigned the all-zero IP address ((0,0,0,0)). `MacAddr((0,2))` means that interface 0 has the MAC address 2.

Initially, the SPD is configured to allow DHCP messages to pass in and out of the mobile station. The single token on the **Security Policy Database** place represents a single rule, matching packets with any source and any destination (`src=((0,0,0,0),0)` and `dest=((0,0,0,0),0)`) and DHCP payload (`nl_info=PayloadList([PAYLOAD_DCHP])`). The policy for this rule is *bypass*, meaning that packets matching this rule are to be allowed. As the routing table is initially empty (no IP configured), there are no tokens on the **Routing Table** place.

The first step is to perform DHCP configuration as was previously illustrated in Fig. 2. This is done in the submodule of the DHCP substitution transition in Fig. 5. The DHCP module accesses all three configuration places. After having configured the mobile station with DHCP, a token is placed on the **Ready** place, representing that the mobile station is now ready to start to communicate with the provisioning security gateway. The **Connect to provisioning SG** substitution transition takes care of establishing the ESP tunnel to the provisioning security gateway, as shown in the MSC on Fig. 3. After having connected to the provisioning security gateway, the **GAN Communication** transition is responsible for sending a GAN discovery message to the provisioning GAN controller and receiving the answer, which will be either reject or accept. In the first case, the mobile station keeps sending discovery messages until one is accepted. When an accept message is received, the ESP tunnel to the provisioning security gateway is closed, and the IP address of the security gateway in the accept packet is placed on the **VIF Closed** place. Finally, the **Connect to Default SG** transition is responsible for establishing a new ESP tunnel to the default security gateway (which was the one received with the GAN accept message).

3.2 Wireless Router

Fig. 6 shows the **WirelessRouter** module. The wireless router has an IP layer, a physical layer, a security policy database, a routing table, and a set of associated addresses similar to the mobile station. The SPD is setup to allow any packets to bypass it. The wireless router has 2 interfaces, the **Addresses** place assigns MAC address 1 and IP address 190.1.1.254 to interface 0, and MAC address 3 and IP address 172.1.1.254 to interface 1.

The routing table contains a single token of the colour set **ROUTING_TABLE** which represents the complete routing table. The definition of this colour set is as follows:

```

colset NETWORK_ADDR = product IP_ADDR * INT;
colset ROUTING_ENTRY = product NETWORK_ADDR * IP_NEXTHOP;
colset ROUTING_TABLE = list ROUTING_ENTRY;

```

The colour set is a list of ROUTING_ENTRY. The NETWORK_ADDR colour set represents a network address in an IP-network. It consists of an IP address and a prefix, which selects how much of the IP address to use for the network address. For instance, a prefix of 24 means to use the 24 first bits the IP address as the network address, which corresponds to using the first 3 octets ($3 * 8 = 24$). Usually, this is written as 192.2.0.0/24, but in our model it becomes ((192,2,0,0),24). The IP_NEXTHOP field is explained in detail in Sect. 4.4.

In the Wireless Router module, the routing table is set up such that packets to the host with IP address 190.1.1.1 are to be delivered directly via interface 0 (`Direct(0)`), packets to the network 172.1.1.0/24 are to be delivered directly through interface 1 (`Direct(1)`), and finally packets destined for 190.1.1.254 (the Wireless Router itself) are terminated at interface 0 (`Terminate(0)`).

The wireless router implements the DHCP Server which is used initially by the mobile station to obtain an IP address. It can be seen that the wireless router has a physical connection to both Network A and Network B.

3.3 Security Gateway

Fig. 7 shows the SecurityGateway module. The security gateway has an IP layer, a physical layer, a security policy database, routing table, and a set of associated addresses similar to the mobile station and the wireless router. The configuration places are initialised via the `Init` transition which obtains the configuration parameters from the input port place `Config`. The security gateway implements the IKE responder protocol module which communicates with the IKE Initiator of the mobile station as described by the MSC shown in Fig. 3.

The `Config` place is associated with the `SG Config` socket place of the top-level module (see Fig. 1) for the instance of the `SecurityGateway` module that corresponds to the Provisioning Security Gateway substitution transition (see Fig. 1). The initial marking of the `SG Config` configures the provisioning security gateway with two physical interfaces and a virtual interface for use with the ESP-tunnel. Interface 0 is configured with MAC address 4 and IP address 172.1.1.1, while interface 1 is configured with MAC address 5 and IP address 12.0.0.1. The third interface, interface 2, does not have any MAC address assigned to it, but only the IP address 80.0.0.1. The reason for this is that it is a virtual interface. The security policy database is set up such that packets sent via interface 2 are put into an ESP tunnel. The default security gateway is configured in a similar way.

3.4 GAN Controller

Fig. 8 shows the GAN Controller module which is the common submodule of the substitution transitions Provisioning GAN Controller and Default GAN Controller

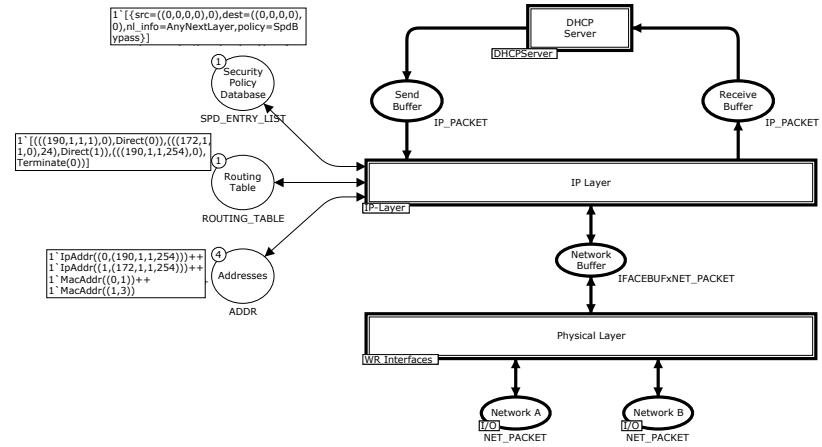


Fig. 6. The WirelessRouter module.

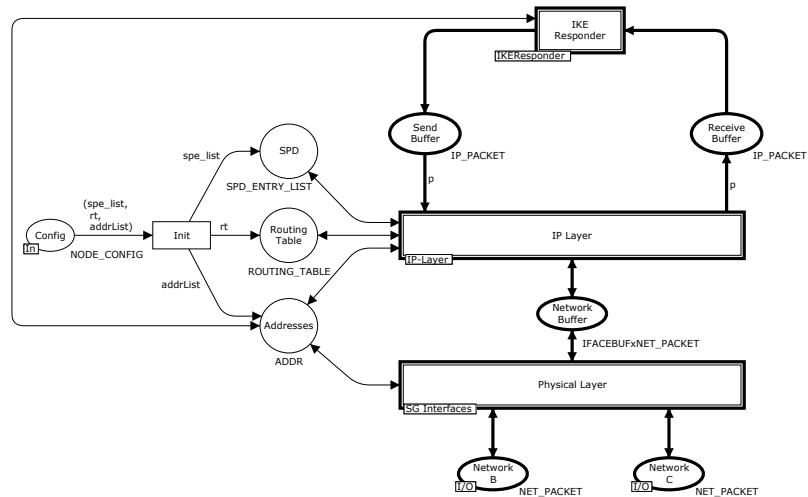


Fig. 7. The SecurityGateway module.

in Fig. 1. Besides the IP and physical network layers, the GAN Controller implements the **GANSERVER** module which will be presented in the next section. The GAN controllers are configured in a similar way as the security gateways.

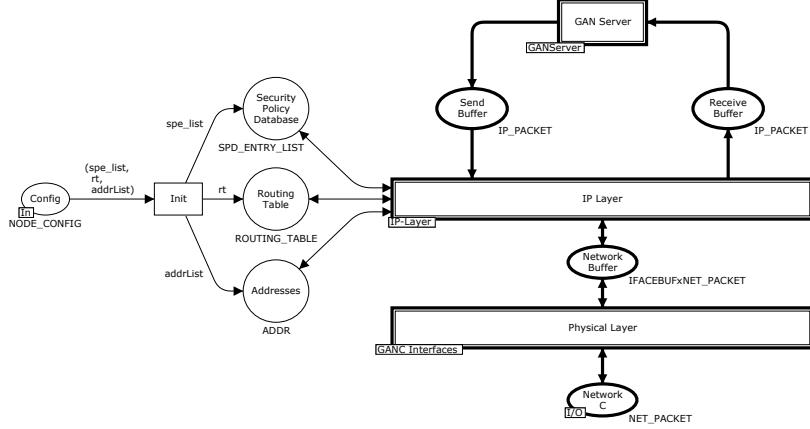


Fig. 8. The GAN Controller module.

4 Modelling the Protocol Entities

This section describes the modelling of the protocol entities present in the mobile station, wireless router, security gateways, and GAN controllers. The description of the protocol entities has been organised such that it reflects how we have decided to organise the protocol entities in the CPN model. This means that we always describe peer protocol entities, i.e., when describing the DHCP client of the mobile station we simultaneously describe its peer protocol entity which is the DHCP server of the wireless router.

4.1 Dynamic Host Configuration Protocol

Fig. 9(top) shows the DHCP Client module of the mobile station and Fig. 9(bottom) shows the DHCP Server module of the wireless router. The two modules model the part of GAN connection establishment which was shown in Fig. 2. The DHCP client starts by broadcasting a request for an IP address and then awaits a response from the DHCP server. When the DHCP server receives a request it selects one of its **FreeAddresses** and sends an answer back to the DHCP client. When the client receives the answer it configures itself with the IP address and updates its routing table and security policy database accordingly. Three entries are added to the routing table: `1'((#ip(da),32), Terminate(0))` means that the IP address received in the DHCP answer belongs to interface 0, while

`1' (calcNetwork(#ip(da), #netmask(da)), Direct(0))` specifies that the network from which the IP address has been assigned, is reachable via interface 0. Finally, a default route is installed with `1' (((0,0,0,0),0), Via(#default_gw(da)))`, such that packets which no other routing entry matches, are send to the default gateway specified in the DHCP answer. The SPD is modified so that the previous rule, which allowed all DHCP traffic is removed and replaced with a new rule, which states that all packets sent from the assigned IP address are allowed to pass out, and all packets sent to the assigned IP address are allowed to pass in.

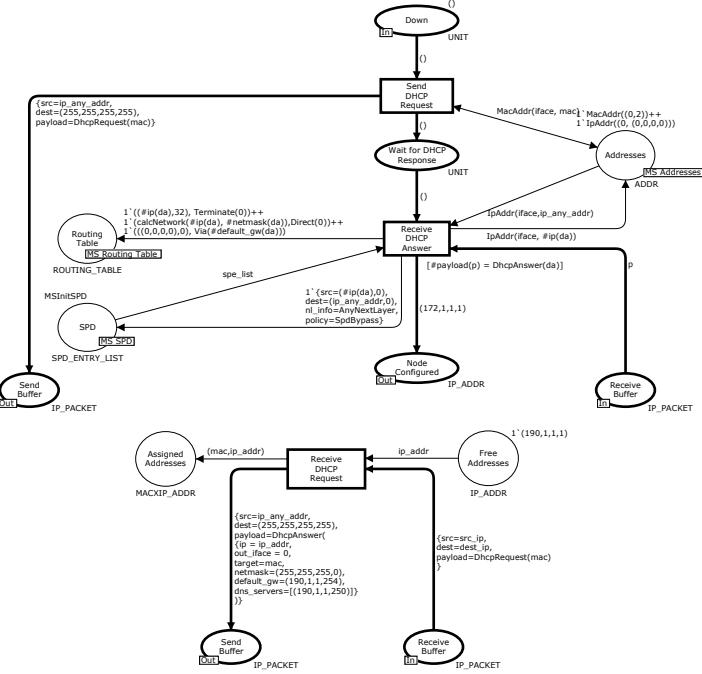


Fig. 9. DHCP client (top) and DHCP server (bottom) modules.

4.2 IKEv2 Modules

Fig. 10(left) shows the `IKEInitiator` module of the mobile station and Fig. 10(right) shows the `IKEResponder` module of the security gateways. The modules model second step of the GAN connection establishment that was illustrated in Fig. 3. Each module describes the states that the protocol entities go through during the IKE message exchange. The state changes are represented by substitution transitions and Fig. 11 shows the `Send IKE_SA_INIT Packet` and `Handle_SA_INIT_Request` modules.

The Send IKE_SA_INIT Packet transition on Fig. 11(top) takes the IKE Initiator from the state Ready to Await IKE_SA_INIT and sends an IKE message to the security gateway initialising the communication. The IP address of the security gateway is retrieved from the Ready place. Fig. 11(bottom) shows how the IKE_SA_INIT packet is handled by the IKE Responder module (which the security gateway implements). Upon receiving an IKE_SA_INIT packet it sends a response and moves the responder to the Wait for EAP Auth state. The submodules of the other substitution transition of the IKE modules are similar. Neither initiator nor responder will receive packets that are not expected. They remain in the network buffer forever.

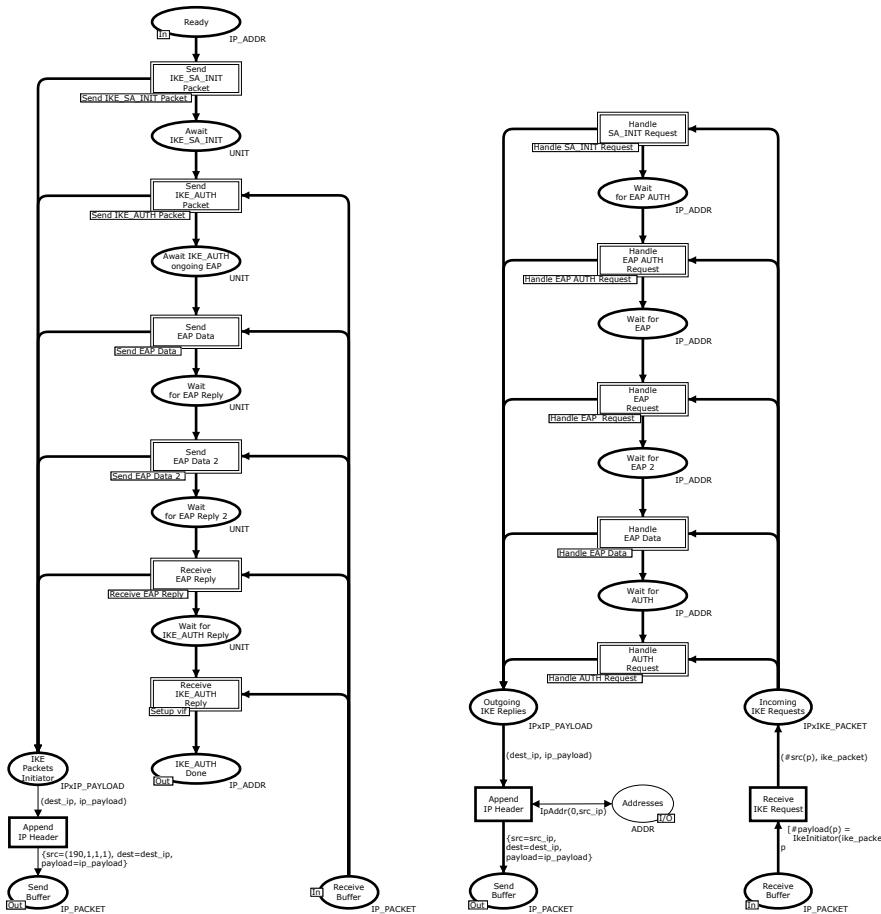


Fig. 10. IKE initiator (left) and IKE responder (right) modules.

4.3 GAN Modules

On Fig. 12(top) the **GANCClient** module of the mobile station is shown, while Fig. 12(bottom) shows the **GANSserver** module of the GAN controller. In the **Tunnel Configured** state, a secure tunnel to the security gateway has been established. The mobile station initiates the GAN communication by sending a **GA-RC** discovery message. The **Send GA RC Discovery Message** transition does just that, and places the mobile station in the **Wait for GA RC Response**, where the mobile station will wait until a response is received from the GAN controller. As can be seen in Fig. 12(right), the GAN controller can either answer a request with an accept or reject message. If the **GANCClient** receives a reject response, the **Handle GA RC Reject** transition will put the client back into the **Tunnel Configured** state, and another discovery message will be sent. This will continue until an accept message is received, in which case the **Handle GA RC Accept** transition puts the client in the **GA RC Accept Received** state, and closes the secure tunnel. This is done by removing the address associated with the tunnel from the **Addresses** place, and removing any entries in the SPD and routing table containing references to the interface.

4.4 IP Network Layer

Fig. 13 shows the **IPLayer** module which is used to model the IP network layer in the mobile station, wireless router, security gateways, and the GAN controllers. As mentioned in Sect. 2 many details of the IP stack have been modelled, such as the routing system and the security policy database. Transport protocols (e.g., TCP), are however, not modelled.

The access to the underlying physical network is modelled via the input/output port place **Network** which has the colour set **NET_PACKET** defined as:

```
colset NET_PACKET = product MAC_ADDR * MAC_ADDR * IP_PACKET;
```

The first component of the product is the source MAC address, while the second is the destination MAC address. The final component is the payload, which currently can only be an IP packet. MAC addresses are represented by integers. The actual physical network layer has been abstracted away, as we do not care whether the physical network is, e.g., an Ethernet or an ATM network.

The chosen representation of a network cannot lose packets but can reorder them, as tokens are picked non-deterministically from the **Network** place. A real network does occasionally drop packets, and considering losing packets is especially important when building fault-tolerant systems. Both IKE and TCP use timeouts and sequence numbers to retransmit packets, such that the protocols deal with lossy networks. Rather than modelling the retransmission schemes, we have chosen to have a lossless network.

Receiving packets from the network is modelled as a transition which consumes a **NET_PACKET**-token, with the destination MAC address corresponding to the MAC address of the network node. Sending packets is a matter of producing a **NET_PACKET**-token with the correct source and destination addresses and

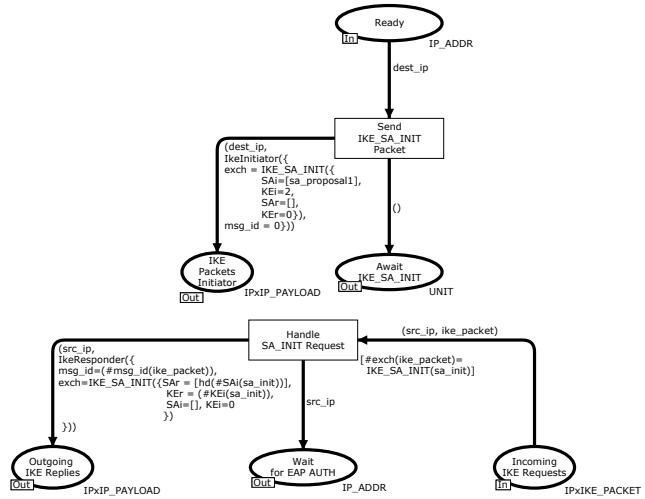


Fig. 11. Example of IKE initiator (top) and IKE responder (bottom) submodules.

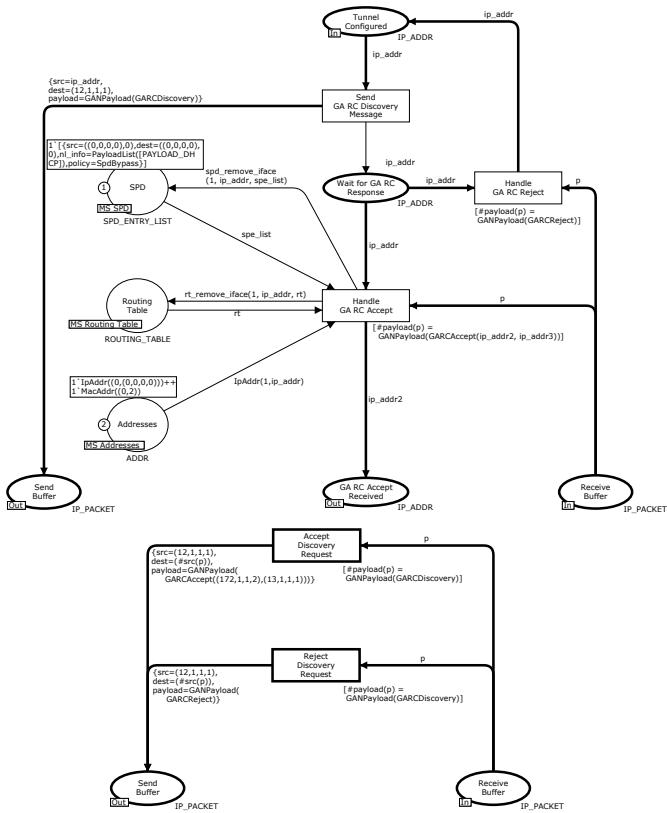


Fig. 12. GANC modules mobile station (top) and controller (bottom).

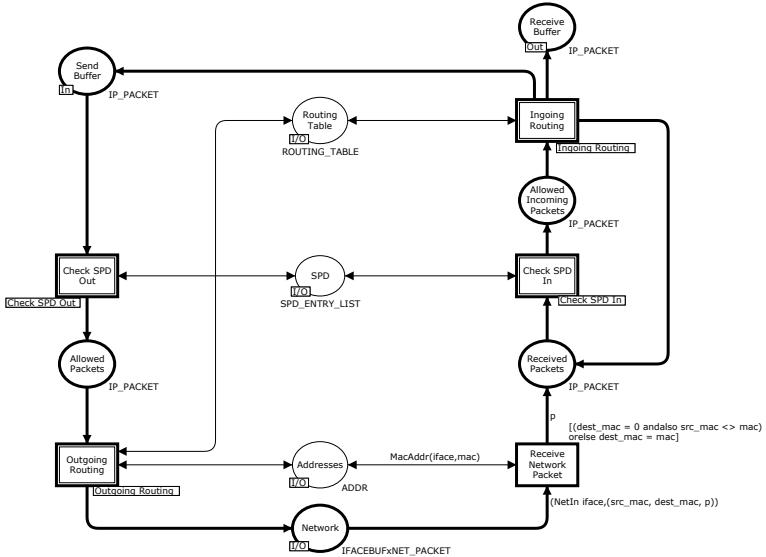


Fig. 13. The IP-Layer module. In-going packet flow (right) and outgoing packet flow (left).

the intended IP packet. The IP layer is completely generic and configured with 3 places for network addresses, routing table, and Security Policy Database (*SPD*). The *Routing Table* place corresponds to the routing table found in real IP implementations. It consists of a number of entries, each of a pair (*NETWORK_ADDR*, *IP_NEXTHOP*), where *IP_NEXTHOP* is as defined as:

```
colset IP_NEXTHOP = union Direct      : IFACE +
                           Via        : IP_ADDR +
                           Terminate : IFACE;
```

The colour set defines which action is to be taken for packets going to the specified IP network. There are three possible actions to take: *Direct*, the packet can be delivered directly via the local physical network via *IFACE*. *Via*, the packet can get closer to its destination by being delivered to the *IP_ADDR*. *Terminate*, the destination network address is a local interface address of *IFACE*.

The *SPD* place is the *SPD* database, and describes which packets are allowed to be sent and received by the IP layer. An entry in the *SPD* database can also specify that certain packets are to be tunneled through a secure tunnel. Finally, the *Addresses* place contains one token for each address known to the IP Layer. These addresses are both physical (MAC) addresses, and IP addresses. Each *ADDR* token contains both an interface number, and the address (MAC or IP).

Packets to be sent are put on the *Send Buffer* place by upper-layers. The first step done by the IP layer, is to check the *SPD* database, which is done by the *Check SPD Out* transition. The *Check SPD Out* module looks through the *SPD*

database to find an entry which matches the source and destination addresses of the packet. If no entry is found, the packet is dropped. If an entry is found, the entries action is applied. Either the action in *bypass*, meaning that the packet can be sent further down the IP-stack, or the action is *tunnel*, meaning that the packet is to be sent through a secure tunnel. In the latter case, a new IP-packet is constructed according to the tunnel information associated with the *tunnel* action. The new packet is placed on the Outgoing Packets place, and has to pass the outgoing SPD check as well. If the packet is allowed to be sent, it is put on the Allowed Packets place. In order to be able to construct a NET_PACKET token, the destination MAC needs to be found. This is done by the Outgoing Routing transition. As can be seen in Fig. 13, it needs to know both about the routing table and the addresses. If a next-hop has been found, a token is placed on the Network place with information about which interface to send on and the MAC address of the next-hop.

Ingoing packets are retrieved from the Network place by the Receive Network Packet transition. Destination MAC and interface number of the network packet have to match one of the MAC addresses configured in the Addresses place. The IP_PACKET inside the NET_PACKET is placed on the Received Packets place. Check SPD In performs incoming SPD check, while Ingoing Routing decides if the packet is to be delivered locally, forwarded, or is a tunneled packet which has to be de-tunneled. In the latter case, the packet is put on the Received Packets place, and goes through SPD checking and routing again. If the packet has its final destination at the local node, it is put in the Receive Buffer. The IP_PACKET colour set models IP packets and is defined as:

```
colset IP_PACKET = record dest    : IP_ADDR *
                     src     : IP_ADDR *
                     payload : IP_PAYLOAD;
```

It has a source and destination IP address and a payload. The IP_PAYLOAD colour set is a union of all possible payloads. The colour set is never used by any of the generic network components, and is as such defined accordingly to the rest of the model. In the GAN scenario, the IP_PAYLOAD colour set has constructors for DHCP, IKEv2, and GAN packets.

5 Validation of the GAN Scenario

During the construction of the model, simulation was used to check that the model behaviour was as desired. Even though simulation does not guarantee correct behaviour, it was very useful in finding modelling errors and make explicit where further specification of the message exchanges were required. For instance, simulation was a valuable tool in validating the packet forwarding capabilities of the IP-layer. By placing tokens that represent packets to be forwarded on a network nodes input buffer and starting the simulation, it is easy to see if the packet is being placed in the output buffer as expected. If not, then single-stepping through the simulation helps to understand the behaviour of the model,

and modify it so that it behaves as intended. Simulation was also heavily used to show engineers at TietoEnator, how the model and especially how the IP-packet flow worked and thereby enabling discussions of the specification. The advantage of simulation over state space verification is that the simulation has immediate visual feedback, and as such is much easier to understand.

A formal validation was performed on the state space generated from the model. The generated state space consists of 683 nodes, while the Strongly Connected Component (SCC) graph consists of 598 nodes of which 1 is non-trivial. There is a single home marking with number 683 which also is a dead marking. The most interesting property to check is that the mobile station can always end up being configured properly, i.e., that it has both gotten an IP address, has successfully communicated with the provisioning GAN controller, and received addresses of the default security gateway and GAN controller. For this, we checked that in marking 683 there is a token in the `VIF open to Def-SG` place of the mobile station (see Fig. 5). Furthermore, we checked that there were no tokens in any of the other places of the mobile station state machine. This would indicate an error in the model, as we do not want the mobile station to be able to be in two states at the same time. To do this we defined a predicate, which checks that only the `VIF open to Def-SG` contains tokens. Checking all nodes in the state space for this predicate, shows that it holds only for marking 683. It is also interesting to investigate whether the routing table and security policy database look as expected. Rather than defining predicates, we displayed the dead marking in the simulator tool of CPN Tools and inspected the configuration of the mobile station. This showed that both routing tables, address assignments, and security policy database were as expected. The state space report also showed that the transitions `RejectDiscoveryRequest` and `HandleGARCReject` (see Fig. 12) both were impartial. This means that if the system does not terminate in marking 683, then it is because the GAN controller keeps rejecting the connection.

6 Conclusion and Future Work

The overall goal of the project was to construct a CPN model and obtain a more complete specification of the GAN scenario to be implemented by TietoEnator. The act of constructing the CPN model helped to specify the details of the message exchanges that were not explicit in the textual scenario description. Including a detailed modelling of the IP stack was necessary in order to capture the details of sending GAN packets from the mobile station to the GAN controller. Furthermore, it was required in order to validate correctness of the the routing table contents, SPD entries, and IP address distribution. The CPN model has been discussed with TietoEnator engineers, and they were convinced of its correctness by using the simulation capabilities of CPN Tools. Using simulation and state space analysis has helped to give further confidence in the constructed model, but more importantly it has provided valuable information about the properties of the scenario.

The initial GAN model presented in this paper matches the GAN scenario as closely as possible, meaning that every entity in the scenario has been represented in the model, and every action in the scenario has a model counterpart. This allows much easier understanding of the model, when the scenario is known, as all components in the model are known in advance. Currently, we are working on an extended version with multiple mobile stations and GAN controllers.

In near future, TietoEnator is to implement the GAN controller. Based on the experience from the project presented in this paper, it has been decided that CP-Nets will be used to model the initial controller design. Besides the advantages of having a formal model which can be validated by means of state space analysis, the goal is to generate template code for the GAN controller directly from a CPN model of the controller. This will ease the work of the initial implementation and help ensure that the implementation is consistent with the design as specified by the CPN model.

References

1. 3GPP: Digital cellular telecommunications system (Phase 2+); Generic access to the A/Gb interface; Stage 2. 3GPP TS 43.318 version 6.9.0 Release 6 (2007)
2. 3GPP: Website of 3GPP. <http://www.3gpp.org> (2007)
3. Postel, J.e.a.: Internet Protocol DARPA Internet Program Protocol Specification. RFC791 (1981)
4. Kent, S., Seo, K.: Security Architecture for the Internet Protocol. RFC4301 (2005)
5. Kaufman, C.E.: Internet Key Exchange (IKEv2) Protocol. RFC4306 (2005)
6. TietoEnator: Website of TietoEnator Denmark. <http://www.tietoenator.dk> (2007)
7. Grimstrup, P.: Interworking Description for IKEv2 Library. Ericsson Internal. Document No. 155 10-FCP 101 4328 Uen (2006)
8. Droms, R.: Dynamic Host Configuration Protocol. RFC2131 (1997)
9. Kent, S.: IP Encapsulating Security Payload (ESP). RFC4303 (2005)
10. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. STTT (International Journal on Software Tools for Technology Transfer) (2007)
11. Villapol, M.E., Billington, J.: Analysing properties of the resource reservation protocol. In: Applications and Theory of Petri Nets 2003: 24th International Conference, ICATPN 2003, Eindhoven, The Netherlands, June 23-27, 2003. Proceedings. Volume Volume 2679/2003 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2003) 377–396
12. Kristensen, L.M., Jensen, K.: Specification and validation of an edge router discovery protocol for mobile ad hoc networks. In Ehrig, H., Damm, W., Desel, J., Groe-Rhode, M., Reif, W., Schnieder, E., Westkmper, E., eds.: Integration of Software Specification Techniques for Applications in Engineering. Volume Volume 3147/2004 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2004) 248–269
13. University of Aarhus: CPNTools. <http://www.daimi.au.dk/CPNTools> (2007)
14. Westergaard, M.: BRITNeY Suite. <http://wiki.daimi.au.dk/britney> (2007)

Towards Modeling and Simulating a Multi-party Negotiation Protocol with Colored Petri Nets

E. Bacarin¹, W.M.P van der Aalst², E. Madeira³, and C. B. Medeiros³

¹ Department of Computer Science - UEL - CP 6001 86051-990 Londrina, PR Brazil. bacarin@uel.br

² Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands. w.m.p.v.d.aalst@tue.nl

³ Institute of Computing - UNICAMP - CP 6176 13081-970 Campinas, SP Brazil.
{edmundo,cmbm}@ic.unicamp.br

Abstract. E-contracting, i.e., establishing and enacting electronic contracts, has become important because of technological advances (e.g., the availability of web services) and more open markets. However, the establishment of an e-contract is complicated and error prone. There are multiple negotiation styles ranging from auctions to bilateral bargaining. This paper provides an approach for modeling multi-party negotiation protocols in colored Petri nets. It is shown how different negotiation styles can be modeled in a unified and consistent way. Moreover, CPN Tools is used to analyze the resulting colored Petri nets. Simulation can be used for both validation and performance analysis, while state-space analysis can be used to discover anomalies in various multi-part negotiation protocols.

1 Introduction

Contracts are documents that states the mutual obligations and authorizations that reflect the agreements between two trading partners [1]. An *e-contract* is a contract modeled, specified and enacted by a software system [2].

E-contracting has become important because of technological advances, for instance, agent technology and web services have allowed for the establishment of virtual enterprises or management of supply chains. E-contracting is a means to regulate the relationship among partners of a virtual enterprise or a particular supply chain.

According to Angelov [3], automatic negotiation is not a trivial task. Most of the negotiation systems proposed so far have an overly limited scope: they run a particular kind of negotiation – typically a bargain or an auction – in a restricted market place. The produced contracts are signed by only two negotiators. Several bilateral contracts must be produced when the agreement comprises more than two partners. In a collaborative multi-partner settlement, this may cause semantic loss. A single contract signed by several partners can express a common goal shared by them by means of a rich set of complex relationships. Splitting this contract into a number of bilateral contracts can make unclear some of these relationships, and even destroy others. This paper aims at both automatic negotiation and multi-party contracts. It shows how the main parts of the SPICA negotiation protocol [4] can be modeled and simulated by means of Petri nets.

The main contributions are: a) the paper describes a negotiation protocol that combines different negotiation styles — namely: bargain, auction, ballot — to produce an e-contract that can be signed by several partners; b) it shows how such a protocol can be modeled as a colored Petri net (CPN) and simulated using CPN Tools [5]; c) it also shows how the capabilities of CPN Tools assisted in the identification of design flaws and how these were addressed.

The paper is organized as follows. Section 2 overviews some related work. Section 3 briefly introduces various negotiation protocols, presents a running example that is used throughout the paper, and illustrates the SPICA negotiation protocol through short examples. Section 4 shows how the main parts of the SPICA protocol were modeled in terms of colored Petri nets and presents some of its subnets (the total model is composed of 31 subnets). Section 5 describes how the model was adjusted to simulate the running example. Section 6 discusses what we have learnt by modeling and simulating the protocol using CPN Tools, which problems we have encountered, and how they were addressed. Finally, Section 7 concludes the paper.

2 Related Work

This section reviews related work on negotiation processes and the use of CPN Tools to model and validate protocols.

There are a number of mechanisms that guide the negotiation process. Bartolini [6] constructs negotiation templates that specify different negotiation parameters. Chiu [7] also uses contract templates as a reference document for negotiation. Similarly, [8] uses a contract template that describes the negotiation parameters, how they are interrelated, along with meta-level rules about the negotiation. In contrast, [9] uses a set of examples of good agreements and it is up to the negotiator to try to get as close as possible to one of the examples.

Like most of the related work, our negotiation process is guided by a contract model. Thus, broadly speaking, the negotiation process concerns determining values for unbound properties.

Our negotiation process is developed through the exchange of messages among the negotiators that comply with a specific protocol. This is a common approach, like the ones based on FIPA's standards [10], e.g., [11]. Governatori and others [12] use a different approach. They propose a negotiation process that uses Defeasible Logic.

Whatever the basis of the negotiation process, all negotiators must understand concepts and names used during the negotiation. The work of [11] combines the use of ontologies and agent technologies to help in solving naming problems in car assembly supply chain negotiations. Our approach also uses ontologies (see [4])

According to Angelov [3], automatic negotiation is a difficult task. Tools that help in the design and implementation of negotiation protocols are needed. Jensen and others [5] present CPN Tools. It builds on Coloured Petri Nets [13] and provides a rich set of resources for modeling, simulating and validating systems where concurrency, communication, and synchronisation play an important role.

CPN Tools has been used to analyse both experimental protocols and also well-established ones. Chen and others [14] use CPNs to model a multi-agent system for supply chain management based on agent negotiation. They describe their negotiation protocol and explain how they formally modeled the negotiation process by using CPNs. The negotiation protocol presented is similar to our auction and bargain negotiation styles. Liu and Billington [15] analysed the Capability Exchange Signalling (CES) protocol using CPN Tools and found it may fail in some specific circumstances.

3 Negotiation: A Particular Protocol

3.1 Overview of Negotiation Protocols

Negotiation is a process by which autonomous entities communicate and compromise to reach an agreement on matters of mutual interest while maximizing their individuals utilities [16]. Automated negotiation (*e-negotiation*) is done by software rather than people and organizations, in general in the form of software agents, and their agreement is stated in *e-contracts*.

A negotiation process involves a number of issues. To begin with, the *number of negotiators* involved is typically one-to-one (*bargaining*), one-to-many (*bidding*), or many-to-many (*double-auction*). In bargains, one seller deals with one buyer. Bidding also has a broad range of flavors, which involve *request for proposals* and interactions among seller(s) and buyer(s). The so-called English or Dutch auctions have many buyers competing for a product sold by an auctioneer. In English auctions, the auctioneer defines a start price and the bidders increase continuously the item's value until the last bid cannot be beaten. Dutch auctions start with a high value and the auctioneer decreases this value until a buyer agrees to pay the proposed value. A many-to-many negotiation can occur through a so-called *double auction*. In this case, several sellers offer their products on a “shared space” and several buyers submit their bids. There is a mechanism that matches offers and bids.

A second issue that needs to be addressed by the negotiation process is related to the number of items. The negotiation process may be restricted to a single product or to a bundle of items. In the latter case, the buyer may be interested in buying all items or none of them [8].

A third issue refers to the negotiation strategy. According to [12], techniques for designing negotiation strategies can be classified into three categories: (i) game-theoretic, (ii) heuristic, and (iii) argumentation. The first approach models a negotiation situation as a game and attempts to find dominant strategies for each participant by applying game theory techniques. In heuristic-based approaches, a strategy consists of a family of tactics (i.e., a method for generating counter-offers), and a set of rules for selecting a particular tactic depending on the stage of the negotiation. Argumentation-based approaches extend heuristic ones by introducing communication patterns such as threats (e.g., “that is my last offer”), rewards, etc.

The fourth issue concerns how the negotiation process is finished: either by agreement or by withdrawal. In [6], agreement formation rules determine which proposals are matched. In [9], a negotiation is aborted if agreement is not reached in a predefined number of rounds.

Finally, the last issue concerns how to renegotiate an existing contract due to some external change. This includes discovering which contracts were affected by the change, which clauses should be modified, who should modify the contract, and so forth.

3.2 Running Example

This section presents the running example that will be used throughout the remainder of the paper.

Milkyway Dairy is a fictitious dairy industry that produces milk derivatives, especially lowfat yoghurt. The farms in its region mainly produce milk and fruits. Periodically, Milkyway Dairy negotiates contracts with nearby farmers for future delivery of milk, blueberries and peaches. Thus, they can plan their future production. Because of governmental regulations, the dairy and the farmers must agree upon maximum milk quota each one will be allowed to deliver. However, the dairy will seek the cheapest supplier for each fruit. The price of the milk is defined at delivery by an official price list, but Milkway negotiates with the farmers a minimum price that will be payed at delivery. In turn, the farmers guarantee that they will deliver at least 500 liters of milk each week. The prices and quantities of different fruits are established during the negotiation.

Milkyway has a predefined draft (or model) for this contract. The negotiation process consists of filling some gaps in this model. The gaps, denominated *properties*, are identified by names preceded by # (e.g., #MINPRICE, #PRICEBERRY). A simplified version of this contract model is shown in plain English in Figure 1. Thus, the parties will agree upon, for instance, a value for the minimum price (#MINPRICE in Clause 3) and which farm will provide berries (#Fa). Note that this contract is a *multi-party contract* signed by the dairy and by several farmers.

Clause 1: Milkyway Dairy, herein named Dairy, and farms Farm 1, Farm 2, Farm 3, Farm 4, herein respectively named F1, F2, F3 and F4, agree that the amount of milk Dairy may buy each week from any of the stated farms is at most #QMAX liters.

Clause 2: F1, F2, F3 and F4 will deliver altogether at least 500 liters of milk per week to Dairy.

Clause 3: The price of milk is defined by the Official Price List at the moment of delivery. However, the Dairy agrees to pay at least #MINPRICE per liter of delivered milk.

Clause 4: If farms F1, F2, F3 and F4 do not succeed to fulfill clause 2, they must pay Dairy a fine worth 10000*#MINPRICE.

Clause 5: Farm #Fa agrees to deliver #QBERRY Kg of blueberries at €#PRICEBERRY a kilogram weekly.

Clause 6: Farm #Fb agrees to deliver #QPEACH Kg of peaches at €#PRICEPEACH a kilogram weekly.

Fig. 1. Simplified multi-party contract with six properties that need to be negotiated

3.3 The SPICA Protocol

This section describes the negotiation process. In this paper, we focus on our SPICA protocol [4]. SPICA (SuPply chain Integration, Coordination, contracting and Auditing framework) is the Latin term for the ear of some grains (e.g., as in ear of corn, or wheat). It is the main star in the Virgo constellation. This constellation is associated to a few myths related to agriculture. In this protocol, the negotiation process is orchestrated by a leader negotiator and is guided by a contract model. The contract model is a predefined contract template which is filled in with values agreed upon by the negotiators. After a successful negotiation, a new contract is created from the model and the negotiated values. Subsequent sections detail the negotiation process itself.

Organization of the Negotiation. A negotiation process involves two or more negotiators. One of them is the *leader*. There is a *notary* responsible for bureaucratic chores (e.g., constructing the final contract) or acting as a trusted third-party (e.g., to control ballots).

These players exchange information within a negotiation process through asynchronous messages. The messages may be peer-to-peer or broadcasted. A contract negotiation has several phases, including leader election, objective and restriction announcement, property negotiation, commit attempt, and contract construction. These phases are described in [4], the core being property negotiation. This is the focus of this paper and is presented in the following sections.

Core Negotiation. There are three generic styles of negotiation: *ballots*, *auctions* and *bargains*. Ballots are used when the negotiators have to reach consensus on a property's value. In our example, most of the farms must agree upon a value for QMAX. Auctions are used when there is competition among different negotiators in order to bind a property to a value. This is the case of property PRICEBERRY. Bargains are used when there are two negotiators and they want to interact to reach a value that is convenient for both. For instance, in our example the dairy and Farm 4 interact to find a consensual value for PRICEPEACH. All these styles may be used to negotiate a single contract.

The negotiation styles are built on a few negotiation primitives, i.e., types of messages exchanged among the participants. These primitives rely on two basic mechanisms: *request for proposals* (RFP) and *offers*. The following paragraphs describe RFPs and offers. Subsequently, the negotiation primitives and the negotiation styles are shown in a few short examples.

An *RFP* is an invitation. A negotiator A sends an RFP to a negotiator B asking for a value for one or more properties. An RFP may prescribe some restrictions on the expected answer and may also bind the value of other properties. For instance, the dairy intends to buy 100kg of peaches from Farm 4. Thus, it sends an RFP to Farm 4 assigning the value 100 to QPEACH and asking a value for PRICEPEACH.

An *offer* is a promise. A negotiator who wants to assign a value to one or more properties sends an offer to another negotiator. The offer indicates the properties the first negotiator is interested in and the values it proposes for them. If the other negotiator accepts such offer, both negotiators are committed to the proposed values. A negotiator can answer to an RFP by sending back an offer that proposes values for the desired properties and that complies with the restrictions indicated in the RFP. If a negotiator is not interested in a RFP, it replies to the RFP by indicating that it will not send an offer. Conversely, a negotiator who receives an offer agrees upon it, disagrees, or proposes a counter-offer. Thus, two negotiators may engage in a cycle of counter-offers until they reach an agreement or give up.

To sum up, offers and RFPs have two distinguishing differences. First, all properties included in an offer must have their values defined in advance, the other party may only agree or disagree on those values. In contrast, an RFP may have properties with predefined values, but also, at least, one unbound property, whose value is subject to negotiation. Second, the party that issues an offer is committed to it, i.e., if the other party agrees upon, the issuer must honor it. In contrast, the party that issues an RFP is neither obliged to keep the RFP valid nor to accept any offer in response to it. That is, an RFP does not imply any level of commitment.

The following describes how RFPs and offers are exchanged by means of specific messages, and how such messages are organized into the three negotiation styles mentioned before (bargains,

ballots and auctions). In these examples, any text enclosed by triangles represents an RFP, and text enclosed by squares represents an offer. They do not show all the information they carry, but only the information relevant for the examples. Messages are numbered and show the sender (e.g., `dr:`) for convenience.

Figure 2 shows the dairy (`dr`) and Farm 4 (`f4`) bargaining the price of a certain amount of peaches. In the first message (1), the dairy sends an RFP to farm `f4` asking for a proposal for property `PRICEPEACH`, considering that property `QPEACH` has value 100. Next, (2) `f4` offers 108 for that property. However, the dairy does not agree upon such value and (3) sends a counter-offer proposing 92. Finally, (4) `f4` agrees upon the proposed value. This sequence of counter-offers may be arbitrarily long and may also be finished without agreement by sending a `proposal no_agree` message.

1. `dr: send f4 new_rfp <PRICEPEACH?; QPEACH=100>`
2. `f4: send dr new_offer □PRICEPEACH=108; QPEACH=100□`
3. `dr: send f4 new_offer □PRICEPEACH=92; QPEACH=100□`
4. `f4: send dr proposal agree □PRICEPEACH=92; QPEACH=100□`

Fig. 2. Bargain

The next two examples show how the notary cooperates in the negotiation process. Figure 3 presents the negotiation of property `QMAX`. This property requires that most of the farms agree on a value, thus a ballot is performed under control of the notary. First, the dairy asks the notary to conduct the ballot. It informs the issue to be voted and the possible votes. In the example, the issue is an offer proposing the value 187 to property `QMAX` and the alternatives are agree (`VOK`) or disagree (`VNOK`). Next, (2) the notary (`nt`) acknowledges to the leader (i.e., `dr`) that it will conduct that ballot. The notary (3) broadcasts the issue to be voted to the negotiators. Each negotiator (4-7) sends its vote to the notary. Finally, the notary counts the votes and (8) broadcasts the result informing that the issue was approved and the number of votes each alternative has received. There are other messages to inform that the issue was not approved or that a tie has happened.

1. `dr: send nt control_ballot □QMAX=187□ VOK,VNOK`
2. `nt: send dr will_conduct □QMAX=187□ VOK,VNOK`
3. `nt: broadcastvoting □QMAX=187□ VOK,VNOK`
4. `f1: send nt vote VOK`
5. `f2: send nt vote VOK`
6. `f3: send nt vote VNOK`
7. `f4: send nt vote VOK`
8. `nt: broadcastbal_res BAPPROVED VOK:3, VNOK:1`

Fig. 3. Ballot

The scenario shown in Figure 4 follows a variant of an English auction that aims at minimizing the value of `PRICEBERRY`. The maximum price is 180. Thus, (1) the dairy asks the notary to advertise an auction for an RFP and wants the notary to collect at most four answers within 30 seconds. The notary (2) broadcasts the RFP and waits as requested. The farms receive the RFP and (3-6) send offers to the notary in response. A negotiator may decline an RFP by returning a `no_offer` answer. The notary collects the answers and (7) sends them to the dairy. The labels O_1, \dots, O_4 stand for the offers from `f1, \dots, f4`, respectively. Now, the leader chooses the best offer (`PRICEBERRY=112`) and (8) asks the notary to start a new auction round, restricting the expected price. This process is repeated (9-12). Eventually none of the negotiators is interested in offering a lower price and all of them answer `no_offer`. Now, the dairy (13) agrees upon the best offer of

the previous round. The dairy also sends a proposal `no_agree` message to all defeated offers (not shown in the figure).

1. dr: send nt `first_answers` 4 30 <PRICEBERRY? \leq 180; QBERRY=100>
2. nt: broadcastnew_rfp <PRICEBERRY?<180; QBERRY=100>
3. f1: send nt `new_offer` □PRICEBERRY=160; QBERRY=100□
4. f2: send nt `new_offer` □PRICEBERRY=112; QBERRY=100□
5. f3: send nt `new_offer` □PRICEBERRY=130; QBERRY=100□
6. f4: send nt `new_offer` □PRICEBERRY=170; QBERRY=100□
7. nt: send dr `collected_answers` O_1, O_2, O_3, O_4
8. dr: send nt `first_answers` 4 30 <PRICEBERRY?<112; QBERRY=100>
9. nt: broadcastnew_rfp <PRICEBERRY?<112; QBERRY=100>
10. f2: send nt `no_offer`
11. f3: send nt `new_offer` □PRICEBERRY=98; QBERRY=100□
12. ...
13. dr: send f1 `proposal agree` □PRICEBERRY=47; QBERRY=100□

Fig. 4. Auction

The examples given in this section show how the properties of our running example (Figure 1) can be negotiated using different styles of negotiation. However, the style of negotiation has only been described in plain English and can be interpreted in different ways. Therefore, we formalize the different styles and primitives in terms of CPNs.

4 Modeling the SPICA Negotiation Protocol in CPNs

This section describes how the protocols presented in Section 3.3 can be modeled as Colored Petri Nets (CPNs) using CPN Tools. The Petri net model is composed of 31 subnets (19 if replicated instances are not counted). The top-level page is presented in Figure 5. In total there are about 300 places and 200 transitions. This model assumes the absence of exceptions (e.g., communication infrastructure is reliable) and the diligence and fairness of the negotiators. Thus, the negotiators will always receive and answer properly any message from the leader or the notary. The model also makes some simplifications. For instance, RPFs and Offers can be used to announce an auction, however, the model only accepts RFPs.

Section 4.1 presents the most important color sets used in the model. Section 4.2 shows a model for the overall negotiation process. Section 4.3 details the subnets for bargains. Finally, the models for ballots and auctions are only briefly described because of space limitations (Section 4.4).

4.1 Color Sets

This section presents the main color sets used in the model. The names of the properties in the contract model are represented by the `PropertyName` color set. The color set `NegId` is used to identify and to address negotiators: L stands for the leader and F1...F4 refer to the farms; `EVERYBODY` stands for all farms; `NO_BODY` is an invalid value for initialization purposes.

```
colset PropertyName = with QMAX | MINPRICE | QBERRY | PRICEBERRY | QPEACH | PRICEPEACH;
colset NegId = with L | F1 | F2 | F3 | F4 | EVERYBODY | NO_BODY;
```

The `PropertyName` color set represents the properties being negotiated. It consists of a property name (`PropertyName`), a status code that indicates if it was agreed upon (`PropAgreement`), the `Value` agreed upon (if any), and the list of the negotiators who had agreed upon this value `LstNegId`. `PropAgreement` is set to `AGREED` to indicate that the property was negotiated and the partners have agreed upon a value; `NOT AGREED` is the opposite, and `IN NEGOTIATION` indicates that the property was not negotiated.

```

colset LstNegId = listNegId;
colset PropAgreement = with AGREED | NOT AGREED | IN NEGOTIATION;
colset Value = INT;
colset Property = product PropertyName * PropAgreement * Value * LstNegId;

```

The leader builds a negotiation plan (`NegoPlan`) before it starts negotiating the contract's properties (Section 4.2). A negotiation plan is a table whose entries describe which properties will be negotiated together (`PropertyNameLst`) and the style of negotiation to be used (`TypeNego`). For instance, in our example, properties `QBERRY` and `PRICEBERRY` are negotiated together using an auction (`AUC`). Other styles of negotiation are ballots (`BLT`) and bargains (`BARG`).

```

colset TypeNego = with BLT | AUC | BARG;
colset PropertyNameLst = list PropertyName;
colset NegoPlanItem = product TypeNego * PropertyNameLst;
colset NegoPlan = list NegoPlanItem;

```

Negotiators exchange RFPs and offers within a negotiation. An RFP has a unique identifier (`RfpId`), and references to the originator (`From`) and recipient (`To`) of the request. The recipient may be a single negotiator (e.g., `F1`) or `EVERYBODY` which causes the RFP to be broadcasted.

```

colset RfpId = MessageId;
colset From = NegId;
colset To = NegId;
colset Operator = with GE | LE | EQ | OM | NEQ;
colset Restriction = product PropertyName * Operator * Value;
colset LstRestriction = list Restriction;
colset Rfp = product RfpId * From * To * LstRestriction * AuctId;

```

The RFP also has a list of restrictions (`LstRestriction`) on the properties values. For instance, the list of restrictions for the RFP of the initial message in Figure 4 should be:

```
[ (QBERRY,EQ,100), (PRICEBERRY,LE,180), (PRICEBERRY,OM,0) ]
```

Note that `OM` stands for “offer me”, and the value after it is meaningless.⁴ Thus the negotiator is expecting an offer for `PRICEBERRY`, provided that it is less or equal (`LE`) than 180. The other operators are great or equal (`GE`), equal (`EQ`), and not equal (`NEQ`). The value for `QBERRY` is already bound by the RFP and the recipient must agree on it. An RFP may be used in an auction. If so, the auction identifier (`AuctId`) is also set.

An offer may be an answer to a previous RFP or to another previous offer. Thus, `ParentId` relates the offer to the previous RFP. The answer to an RFP can be a *No Offer*. This means that the negotiator is not interested in sending an offer for such RFP. In this case, `NoOffer` is set. The offer identifies its originator and recipient to be an RFP (`From` and `To`). When a negotiator receives an offer, it evaluates it. If the negotiator agrees upon the received offer, it assigns the value `OK` to field `Eval`; otherwise, it assigns the value `X` to it. In an auction, the best offer of the current round is marked with `NR` (New Round), indicating that a new auction round should start. When an offer is created, the `Eval` field is set to `NE` (Not Evaluated). `PropertyLst` has all the properties being negotiated. This list must contain all properties referenced by a previous RFP (if any). The negotiator that issued the offer may accept a subsequent counter-offer by setting `CounterOfferAllowed` to `true`. Note that in an auction (Section 4.4) it is always set to `false`. `ParentIsRfp` is set to `true` whenever the offer answers an immediate previous RFP. Finally, an offer has a unique identifier (`OfferId`).

```

colset ParentId = MessageId;
colset Eval = with OK | X | NE | NR;
colset NoOffer = int with 1..0;
colset PropertyLst = listProperty;
colset CounterOfferAllowed = BOOL;
colset Offer = product ParentId * NoOffer * From * To * Eval * PropertyLst * CounterOfferAllowed
* ParentIsRfp * OfferId;

```

⁴ Note that the union type could be used here. However, for reasons of simplicity we did not do so.

Other colors used in the model that are specific to different negotiation styles will be described when needed.

4.2 The Overall Negotiation Process

Figure 5 shows the main net of our model. The negotiation process aims at assigning values to properties. The properties to be negotiated are in the place **ToNegotiate**. Broadly speaking, the leader builds a negotiation plan (see transition **PlanHowToNegotiate**). Then, the leader follows this plan and coordinates the negotiation. At the end, properties are grouped based on whether they were agreed on or not (see places **Agreed** and **NotAgreed**). Some properties may fail to be negotiated, e.g., when no negotiator is interested in an auction. Such properties are put in the place **NotNegotiated**. If all properties were successfully negotiated, a new contract can be built. This last step is out of the scope of this paper.

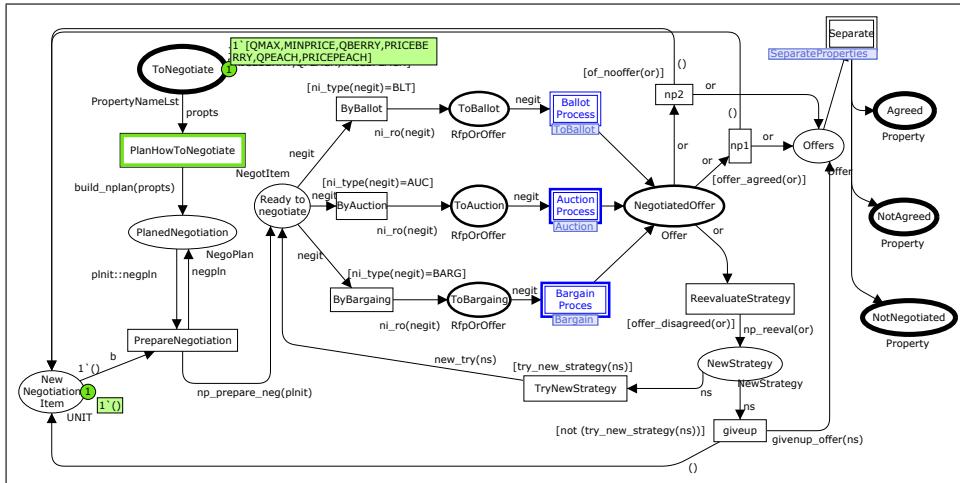


Fig. 5. Negotiation Process (*NegotiateProperty* page)

The negotiation plan for our example is shown in Figure 6. It shows that the leader will first negotiate property **QMAX** using a ballot, then **MINPRICE**. In the third step, it will negotiate properties **QBERRY** and **PRICEBERRY** by an auction. Finally, **QPEACH** and **PRICEPEACH** will be negotiated through a bargain. Note that each step uses one negotiation item and starts after the previous one has ended (notice place **NewNegotiationItem**).

Each negotiation style (bargain, ballot, or auction) is started by sending the first RFP or offer to the corresponding subnets (substitution transitions **BallotProcess**, **AuctionProcess** and **BargainProcess**). To enable reuse, a new color set (**RfpOrOffer**) is introduced and used in places where tokens represent RFPs or offers. The negotiation is carried out by the three subnets shown in Figure 5. These subnets deliver back the offers that were agreed upon and those that were not. In the latter case, the leader may change the negotiation strategy (transition **ReevaluateStrategy**), even changing the negotiation style, and submit them again to be negotiated.

Style	Properties
1. BLT	[QMAX]
2. BLT	[MINPRICE]
3. AUC	[QBERRY,PRICEBERRY]
4. BARG	[QPEACH,PRICEPEACH]

Fig. 6. Negotiation plan for the running example

Figure 7, shows the tree view of all subpages. Figure 7.a depicts the topmost net and the three main subnets. Figures 7.b to 7.d detail these subnets. The nodes are named after the page names, and a number in front of the name denotes the number of replications of this page (if any). For instance, there are four replications of **ProcessRfp** within page **AuctionNegotiators** (Figure 7.b). Each node is annotated with a pair P:X T:Y. The first stands for the number (X) of places (P) in that page and the latter, the number (Y) of transitions (T), e.g., the page **BargainNegotiators** has 10 places and 6 transitions ((Figure 7.d)). The paper only shows the underlined pages.

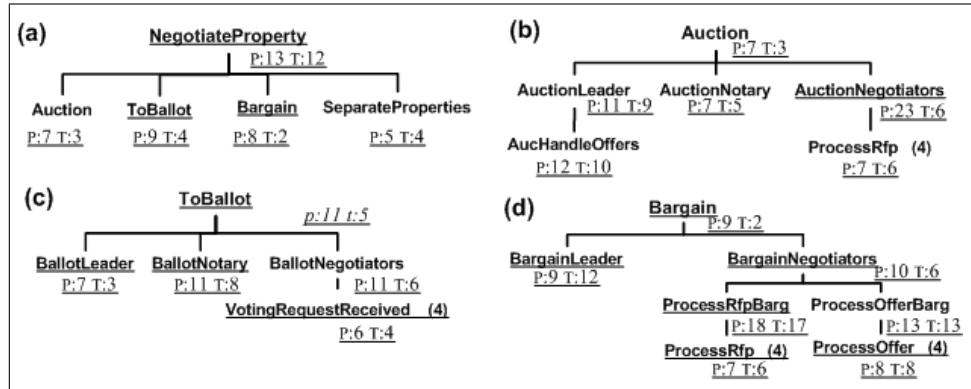


Fig. 7. Tree view: (a) topmost page and its main subpages; (b)-(d) subpages are detailed

4.3 Modeling Bargains

A bargain is characterized by a cycle of offers and counter-offers. The negotiators end up agreeing upon an offer or they give up. Figure 8 shows the subnet that models the bargaining process. The RFP or offer to be bargained enters the subnet via place **ToBargain**. The leader directs this RFP (or offer) to the other negotiators by putting it in place **RfpToNegotiators** (or **OfferToNegotiators**). The negotiators can make counter-offers by putting offers in the place **OfferToLeader**. Leader and negotiators may engage in a cycle of counter-offers until one of them agrees upon the offer or gives up the bargain. In both cases, the partner places an *acceptance notification* in the place **NotificationToNegotiators** or **NotificationToLeader**. An acceptance notification is an offer with special settings, which is denoted by the **Eval** field. At the end, the leader puts the negotiated offers in the place **OfferNeg**.

Figures 9 and 10 detail the substitution transitions **Leader** and **Negotiators**, respectively. Figure 9 shows how the leader develops the bargain and Figure 10 shows how the negotiators react to it. The latter figure also shows how RFPs and offers are processed in a bargain. Figure 9 shows a few design directives used in the model. First of all, the messages were modeled as transitions. The firing of such transitions means that a message is sent, broadcasted, or received. The transitions are named after the message. Prefixes are used to identify whether the data was sent (**s_**), broadcasted (**b_**) or received (**r_**). For instance, the transition named **b_new_rfp** (Figure 9) means that the message **new_rfp** was broadcasted to the negotiators. There are transitions not related to exchange of messages. The second design decision that should be noted is that data conveyed by messages are modeled as tokens, e.g., the place **ToBarg** (Figure 9) stores the RFP or offer that starts the bargaining process.

When there is an RFP to be negotiated in place **ToBarg** (Figure 9), the leader sends or broadcasts a **new_rfp** message that conveys such RFP. This is represented by transitions **s_new_rfp** and **b_new_rfp**, respectively. After a few firings, such an RFP will be available to the other negotiators in the place **RFP** (Figure 10). Similarly, when there is an offer to be negotiated, the leader sends or broadcasts it. It will be available to the negotiators via place **OFFER_fr LEADER** (Figure 10).

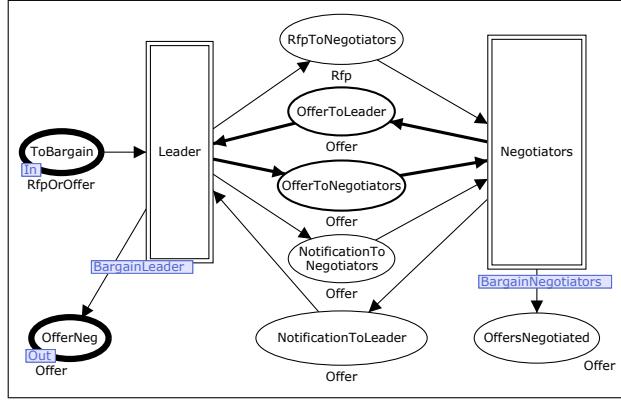


Fig. 8. Bargain subnet. Cycle of counter-offers is emphasized.

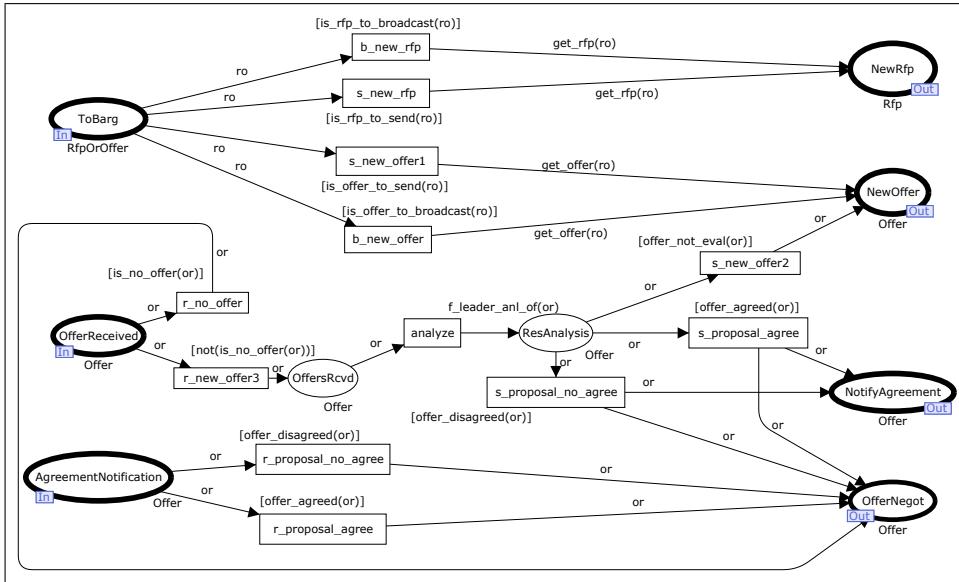


Fig. 9. BargainLeader subnet

When the other negotiator(s) receive(s) an RFP (in the place RFP, Figure 10), the answer to the leader is put in the out-port OFFER_to LEADER. The negotiator(s) receive(s) offers from the leader through the in-port OFFER_fr LEADER. Such an offer may be a response to a directly preceding RFP. In this case, the offer passes through the Process RFP (Bargain) subnet, but is handled by the Process Offer (Bargain) subnet. The arriving offer should also be a new offer or a counter-offer from the leader. In both cases, the offer goes directly to the Process Offer (Bargain) subnet.

An agreement notification from the leader arrives via the port AgrNotffFrLeader and may follow two paths: (a) it is directed to Process RFP (Bargain) subnet, when the leader immediately agreed upon the first offer sent by the negotiator, or (b) the notification is directed to the Process Offer (Bargain) subnet.

Figure 10 shows neither individual negotiators nor makes a distinction whether the messages were sent or broadcasted. These details are presented in Figure 11. An RFP is directed to the corresponding receiver or broadcasted to all negotiators. Note that the answers of each negotiator are joined in one respective single place. The subnet for Process Offer (Bargain) is designed in a similar way.

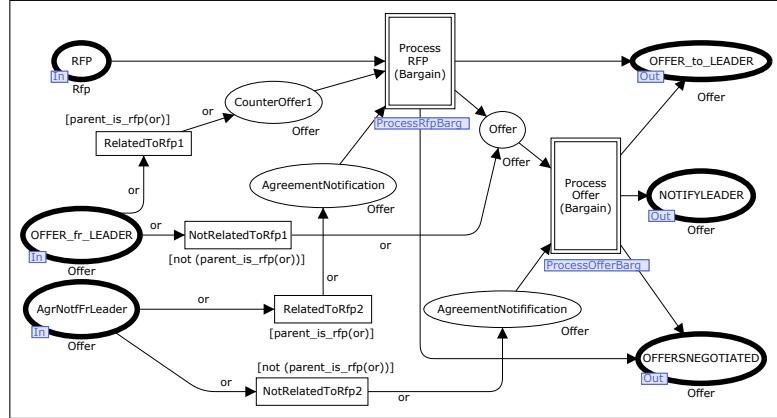


Fig. 10. *BargainNegotiators subnet*

Figure 12 presents the elementary **ProcessRFP**. It shows the messages that are received and sent by the negotiator and the function it uses to assess the received RFP (**f_analyze_rfp**). Recall that this subnet is used by several negotiators and each negotiator evaluates the RFP differently. Thus, the **f_analyze_rfp** function actually dispatches the RFP to specific functions. For this reason, the token in RFP place is of color **RfpXNegId** and contains the RFP and the identification of the negotiator it is meant for.

Finally, Figure 13 shows the elementary **ProcessOffer**. New offers arrive in the place **Offer**. Function **f_analyze_offer** dispatches it to the correct negotiator. The (dis)agreed offers are made available to the partner negotiator through the place **AgreeDisagree**. They are collected in the **OffersNegotiated** place. Conversely, the negotiator decides to make a counter-offer, which is placed in **CounterOffer**. A **No Offer** message is just collected (transition **r_no_offer**). Agreement notifications related to previous offers are received in through port **AgreementNotification** and are also collected.

Note that the same patterns of **ProcessRfp** and **ProcessOffer** were applied to subpages at different levels. For instance, Figures 10 and 11 have the same structure of Figure 12.

4.4 Modeling Ballots and Auctions

Figure 14 details the **BallotProcess** subnet referred to in Figure 5. It shows the leader, the notary and the negotiators interacting in a ballot process. The leader role is detailed in Figure 15. Note that the leader asks the notary to conduct the ballot (transition **s_control_ballot**), receives the notary's acknowledgment (transition **r_will_conduct**) and, eventually, receives the result of the ballot (**r_bal_res**).

The notary role is detailed in Figure 16. First, the notary receives a message from the leader to conduct a ballot (transition **r_control_ballot**). It accepts the job (transition **s_will_conduct**) and broadcasts the issue to the negotiators (**b_voting**). The notary receives votes (**r_vote**) or vetoes (**r_veto**). In case it does not receive any veto, the notary counts the votes and broadcasts the result (**b_bal_res**). Otherwise, the notary informs all participants that a veto has occurred. However, none of the farms in the running example has veto power. Thus, the veto scenario does not occur.

The **BallotNegotiators** subnet (not shown) directs the issue to be voted to each negotiator, collects their votes and, eventually, sends them the ballot result. Figure 17 shows how a negotiator reacts when it receives a vote request. Basically, it analyzes the issue and decides either to vote or to veto (see function **vrr_analyze**). Its choice is sent to the notary (**s_vote** or **s_veto**). Eventually, the negotiator receives the ballot result (**r_bal_res**).

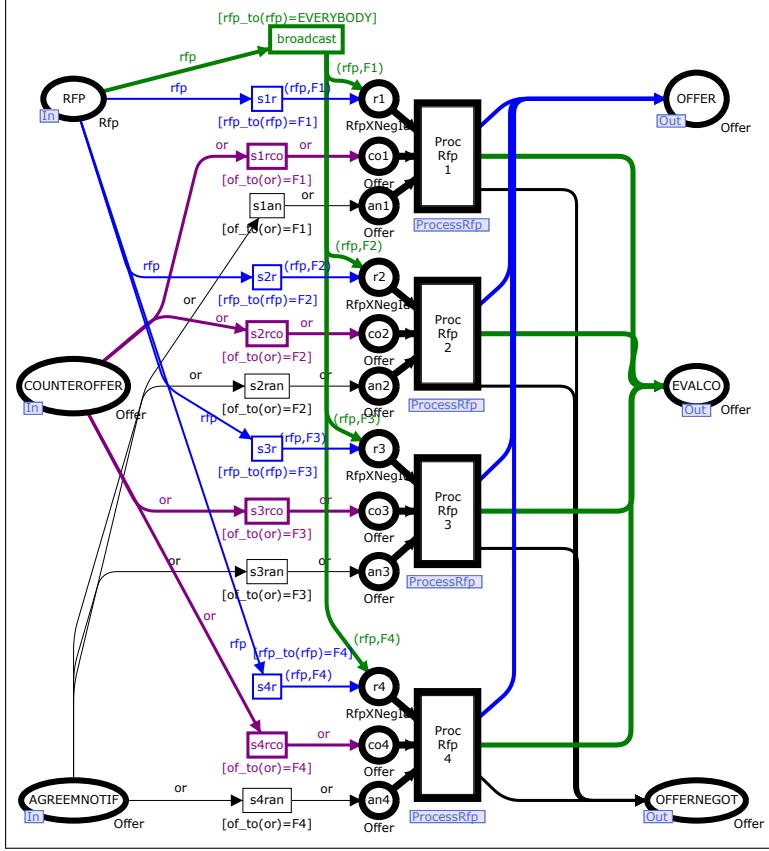


Fig. 11. *ProcessRfpBarg* subnet

The notary also collaborates in an auction. Figure 18 shows an auction step being directed to negotiators. Each negotiator uses *ProcessRfp* subnet to react to a negotiation step. Note that this is the same subnet used in bargains. The other subnets are not presented for the sake of brevity.

5 Implementing the Negotiator's Strategies

To simulate the model, we configured the CPN for the scenario presented in Section 3.2. For each property, each negotiator (including the dairy) has a value (or a range of values) it considers a “good deal” and for which it will always agree upon. There are also values the negotiator will never agree upon. Values in-between will be accepted with specific probabilities: the nearer to the expected value, the higher the acceptance probability. Each farm has also distinctive characteristics, mainly its production capability for each product (Figure 19). These characteristics influence its decision. For instance, Farm 1 weekly produces a large amount of milk (400 liters). Thus, it tends to agree upon high values for QMAX. However, Farm 2 has a low production capacity (50 liters). It will not agree upon high values for QMAX for fear that the week quota should be fulfilled by the other farms.

According to the model presented in Figure 5, the leader may have several strategies to negotiate the value of a property. When a negotiation fails, the leader can try again negotiating that property using a different strategy. The simulation uses this capability to negotiate properties QMAX and MINPRICE. For instance, the leader has a lower and an upper limit to QMAX (*QMAX_min* and *QMAX_max*). It starts in the middle point between these values and submits increasing values to successive ballots. If the proposed values reach unsuccessfully the upper limit, the leader submits successive decreasing values starting from the middle point.

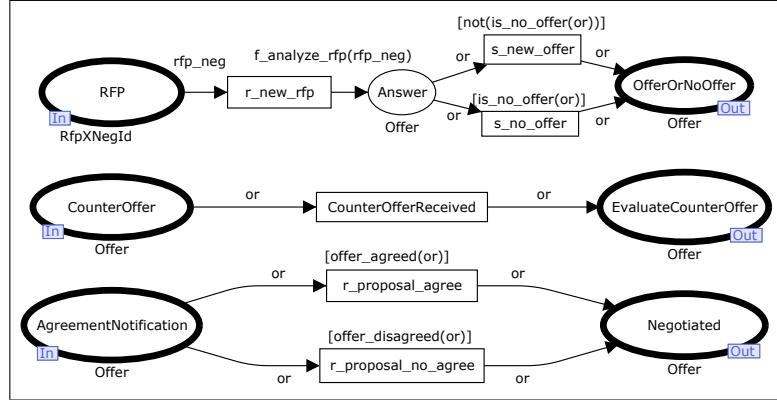


Fig. 12. Elementary Process RFP (ProcessRfp subnet)

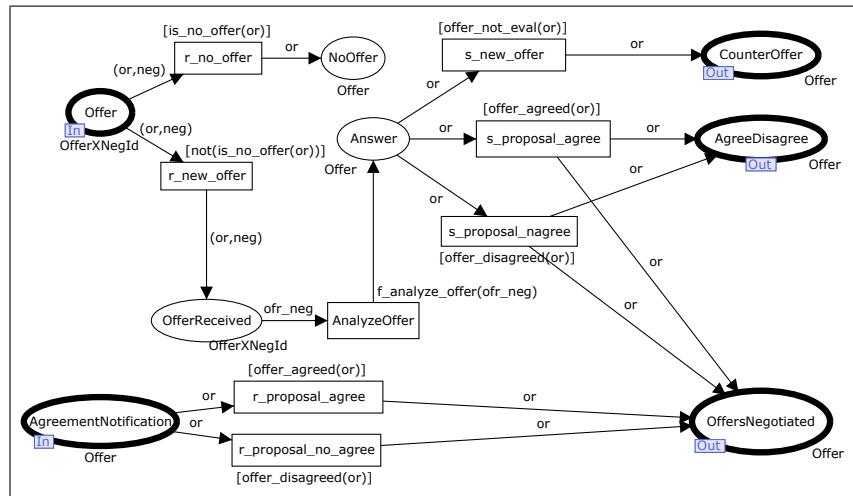


Fig. 13. Elementary Process Offer (ProcessOffer subnet)

The dairy is interested in buying 100kg of peaches weekly. Only Farm 4 can afford such amount. Thus, they engage in a bargain that comprises two properties: QPEACH (bound by the leader) and PRICEPEACH. The dairy has two parameters that guide the negotiation. PEACH_PRICE_GD_LD is a threshold value, i.e., values lesser or equal to it are considered a good deal (cheap enough) and are always accepted. PEACH_PMAX_LD is the opposite, values higher than it are refused.

The dairy starts the bargain by sending an RFP to Farm 4 assigning the value of 100Kg to QPEACH and requesting a value for PRICEPEACH. Eventually, the dairy receives an offer from Farm 4. The closer the offered price is to the “good deal”, the higher the probability of being accepted. If not accepted, the dairy will choose (by chance) between refusing the offer or making a counter-offer. If it chooses to make a counter-offer, the value for a future counter-offer uses the value of PEACH_PRICE_GD_LD (e.g., 10) as a reference point: if the proposed value (e.g., 14) is at a “distance” d (i.e. 4), beyond the “good deal”, the counter-offer will be exactly at the same “distance” before the “good deal” (i.e., 6).

Similarly, Farm 4 has two parameters to negotiate PRICEPEACH. The value it considers a good deal is PEACH_PRICE_GD_F4. Values higher than it are always accepted. PEACH_PMIN_F4 is the opposite (too cheap), values lower than it are always rejected. The diagram in Figure 20 depicts Farm 4’s decision table. GD stands for PEACH_PRICE_GD_F4; MP, for PEACH_PMIN_F4; numbers above the line are fractions (in percentage) of GD; and numbers below the line are acceptance probabili-

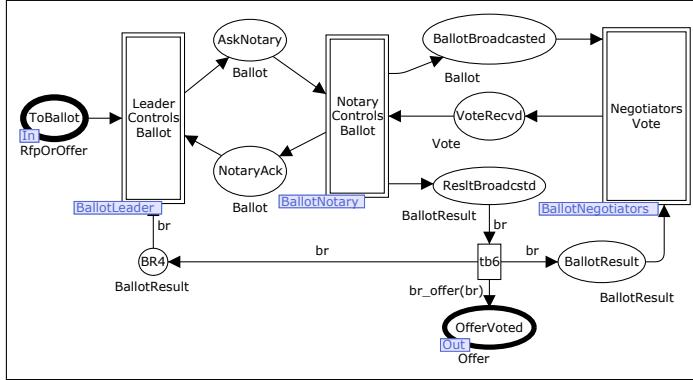


Fig. 14. *BallotProcess* subnet

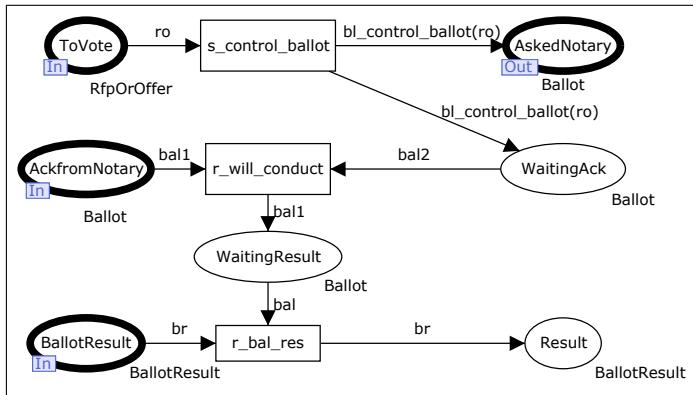


Fig. 15. *BallotLeader* subnet

ties. For instance, if Farm 4 receives an offer whose value is at least 90% of GD, but no bigger than GD, it will accept the offer with probability 0.8.

Other negotiators use similar approaches, but with different parameter values. They are not described here for brevity.

Recall Figure 5. One execution of the model should produce as a result of a negotiation the markings shown in Figure 21. Note the agreed values for properties MINPRICE, QMAX and PRICEBERRY. There was no agreement on the value for PRICEPEACH.

6 Discussion

This section discusses what we have learnt from the modeling process and some results obtained from simulations and state space analysis.

The construction of the model led us to new insights into the modeling process and into our negotiation protocol. We followed a top-down approach to model the protocol. At the end, we noticed that there were different subnets to handle RFPs in auctions and in bargains. This also happened to offers. Thus, we updated our model using a bottom-up approach: we focused on the handling of RFPs and offers and how to combine them to allow different styles of negotiation (ballots, auctions, and bargains), e.g., compare Figures 11 and 18. This gave us a better understanding about the role of RFPs and offers within our protocol. This allowed us to replicate elementary structures in higher subnets. For instance, compare Figures 10, 11 and 12. Note that they have identical interfaces, and the flow of information is the same.

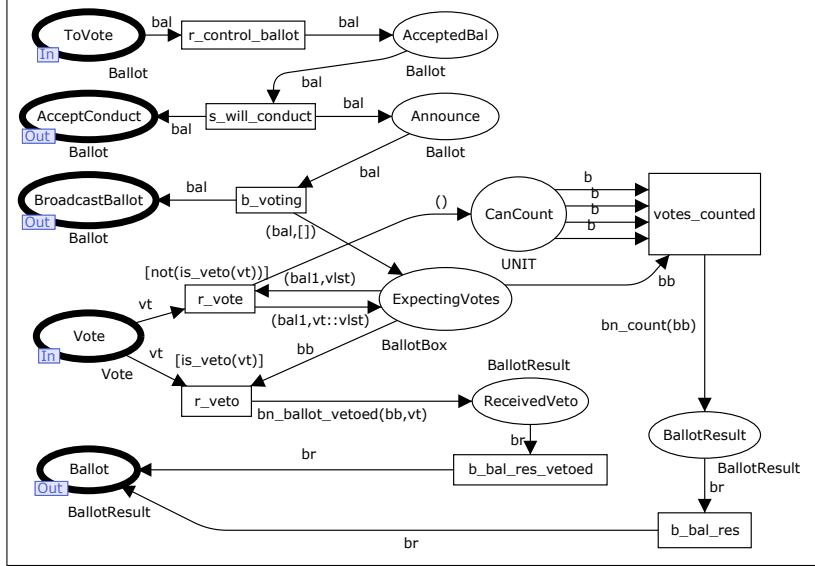


Fig. 16. *BallotNotary subnet*

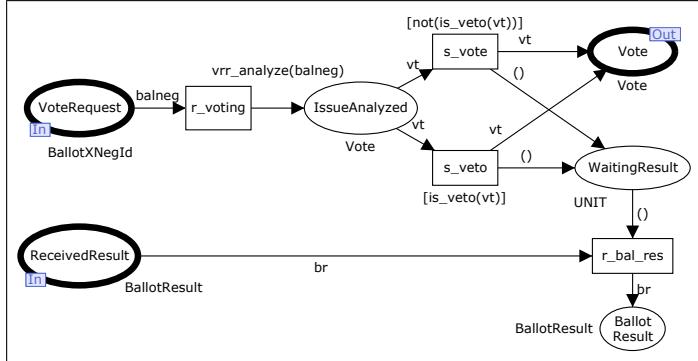


Fig. 17. *VotingRequestReceive subnet*

The modeling process also helped us to improve the SPICA protocol. It showed that the protocol lacked negotiation plans and also that it should allow alternative strategies to negotiate a property.

We used CPN Tools to build the model. CPN Tools helped us in several ways. First, calculating the model's state space was a hard job. Several simplifications and processing hours would be needed to produce a full state space. This made clear that the protocol's implementation will demand extreme care. Second, it helped us to assess the correctness of the model, especially by indicating some unexpected dead markings. Third, it allowed to simulate the running example presented in Section 3.2.

State space analysis. The calculation of the state space was a non-trivial task for several reasons. The model's first version aimed at producing actual values for the properties, e.g., a negotiation should come out with the value 187 for property QMAX in our running example. This easily produces a huge state space. We tried several alternatives to calculate a full state space. First, we created a new model version with some simplifications. The allowed values for properties were restricted (e.g., $-1..+1$) and the decisions were not based on the offered values, but taken by chance. However, these restrictions did not make possible the calculation of a full state space, even spending several processing hours of a Pentium 4 (3GHz, 1Gb). Then, we split the model

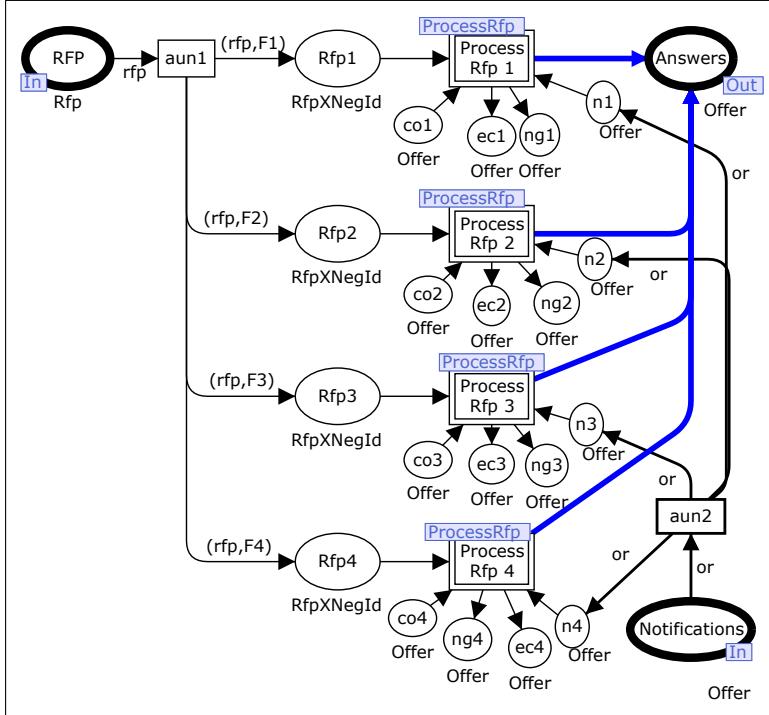


Fig. 18. AuctionNegotiators subnet

Production Capacity	F1	F2	F3	F4
Milk (liters)	400	50	200	150
Blueberry (kg)	100	200	150	300
Peach (kg)	50	30	80	100

Fig. 19. Production capacity

into three distinct models – one for auctions, another for ballots and another for bargains – and deleted unneeded places and transitions of the main net (Figure 5).

For the bargain sub-model, we managed to calculate the full state space. However, this was not possible for the ballot sub-model. Thus, we tried to reduce the number of negotiators by dropping two negotiators, i.e., only the other two remaining negotiators took part in ballots. We succeeded in obtaining a full state space for the reduced ballot sub-model (Figure 22). This analysis showed a few dead transitions, which are caused by two factors: (a) the model's reduction and (b) some possibilities allowed by the model were not used by the running example. The dead transitions `SeparateProperties'not_negotiated` and `SeparateProperties'sp3` are examples for the first factor. A property can be *not negotiated* only when a negotiator is not interested in an RFP and sends back a *No Offer* answer. This cannot happen in a ballot. The other transitions are related to the veto power a negotiator may have in a ballot. The farms in our running example do not have such a power.

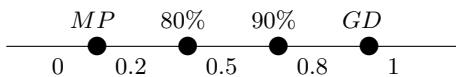


Fig. 20. F4's PRICEPEACH decision table

Agreed	$1'(\text{QMAX}, \text{AGREED}, 202, []) ++ 1'(\text{MINPRICE}, \text{AGREED}, 15, []) ++ 1'(\text{QBERRY}, \text{AGREED}, 100, [\text{F3}]) ++ 1'(\text{PRICEBERRY}, \text{AGREED}, 39, [\text{F3}])$
NotAgreed	$1'(\text{QPEACH}, \text{NOT_AGREED}, 100, [\text{L}] ++ 1'(\text{PRICEPEACH}, \text{NOT_AGREED}, 57, [\text{L}])$

Fig. 21. A negotiation result

Statistics

State Space	Nodes: 6387	Arcs: 22816	Secs: 26	Status: Full
Scc Graph	Nodes: 6387	Arcs: 22816	Secs: 2	

Boundedness Properties
(Omitted)

Home Properties

Home Markings	None
---------------	------

Liveness Properties

Dead Markings	8 [6387, 6386, 6385, 6380, 6372, ...]												
Dead Transition Instances	<table> <tr> <td>BallotNotary'b_bal_res_vetoed</td> <td>1</td> <td>BallotNotary'r_veto</td> <td>1</td> </tr> <tr> <td>SeparateProperties'not_negotiated</td> <td>1</td> <td>SeparateProperties'sp3</td> <td>1</td> </tr> <tr> <td>VotingRequestReceived's_veto</td> <td>1</td> <td>VotingRequestReceived's_veto</td> <td>2</td> </tr> </table>	BallotNotary'b_bal_res_vetoed	1	BallotNotary'r_veto	1	SeparateProperties'not_negotiated	1	SeparateProperties'sp3	1	VotingRequestReceived's_veto	1	VotingRequestReceived's_veto	2
BallotNotary'b_bal_res_vetoed	1	BallotNotary'r_veto	1										
SeparateProperties'not_negotiated	1	SeparateProperties'sp3	1										
VotingRequestReceived's_veto	1	VotingRequestReceived's_veto	2										
Live Transition Instances	None												

Fig. 22. State space report for the ballot sub-model (excerpt).

For the auction sub-model, even making such reductions, it was not possible to produce a full state space.

Statistics

State Space	Nodes: 241569	Arcs: 1134868	Secs: 43000	Status: Partial
Scc Graph	Nodes: 241569	Arcs: 1134868	Secs: 190	

Fig. 23. State space report for the auction sub-model (excerpt).

Model correctness. CPN tools helped us to assess the correctness of the model. It showed a few expected dead markings (e.g., in those states that collect negotiated offers), but also unexpected ones. For instance, one configuration did not replace the token in `NewNegotiationItem` (Figure 5) hindering the execution of the next negotiation round. CPN Tools also highlighted another synchronization problem. It may happen when a negotiation fails and the convenience of a new strategy is evaluated. The solution found is based in specific characteristics of the running example and cannot be generalized. Thus, such synchronization issues must be better addressed in future versions of the model.

Simulation. We used CPN tools to run simulations and to produce performance reports. The generated performance report helped us to assess the correctness of the model. We wanted that: (a) new strategies were tried, (2) distinct simulation produced different outputs, (3) all properties reached the final places `Agreed`, `NotAgreed` or `NotNegotiated` (Figure 5), (4) eventually, some auction failed (this guarantees that a property is put in the `NotNegotiated` place). Figure 24 shows a performance report. Recall Figure 5. Regarding the first concern, a new

strategy is tried when transition TryNewStrategy fires. According to the performance report, this happens, in average, 1.74 times in each simulation run (see line count_iid of monitor Count_trans_occur_NegotiateProperty'TryNewStrategy_1). To assess the second concern, three marking size monitors were assigned to the final places (see columns Min and Max of monitors Marking_size_NegotiateProperty'Agreed_1, Marking_size_NegotiateProperty'NotAgreed_1 and Marking_size_NegotiateProperty'NotNegotiated_1). Note that the maximum numbers of tokens observed in each place is greater than zero. This means that eventually a token reached such places. Note also that the minimum number of tokens differs from the maximum. It means that the outputs can vary in each simulation. To assess the third concern, we grouped the final places and assigned a monitor (Final_places) that counted the total number of tokens in these places. For the running example, the number of tokens in the final places at the end of a simulation must be ever 6. Note that the measures Min and Max for max_iid are both 6. This does not guarantee the correctness of the model, however the opposite would show that the model was incorrect. Finally, a transition occurrence monitor was assigned to np2 transition (Figure 5) in order to verify the fourth concern. If an auction fails, this transition fires. The performance report showed that this transition eventually fires.

Number of replications: 100							
Statistics							
Name	Avg	90% Half Length	95% Half Length	99% Half Length	StD	Min	Max
Count_trans_occur_NegotiateProperty'TryNewStrategy_1							
count_iid	1.740000	0.276611	0.330936	0.439088	1.661325	0	7
max_iid	0.700000	0.076684	0.091745	0.121728	0.460566	0	1
min_iid	0.700000	0.076684	0.091745	0.121728	0.460566	0	1
Count_trans_occur_NegotiateProperty'np2_1							
count_iid	0.080000	0.045398	0.054314	0.072064	0.272660	0	1
max_iid	0.080000	0.045398	0.054314	0.072064	0.272660	0	1
min_iid	0.080000	0.045398	0.054314	0.072064	0.272660	0	1
Final_places							
count_iid	345.070000	10.611371	12.695406	16.844357	63.731960	230	569
max_iid	6.000000	0.000000	0.000000	0.000000	0.000000	6	6
min_iid	0.000000	0.000000	0.000000	0.000000	0.000000	0	0
sum_iid	6.000000	0.000000	0.000000	0.000000	0.000000	6	6
avrg_iid	0.017982	0.000557	0.000667	0.000884	0.003346	0.010545	0.026087
Marking_size_NegotiateProperty'Agreed_1							
count_iid	345.070000	10.611371	12.695406	16.844357	63.731960	230	569
max_iid	5.260000	0.161584	0.193318	0.256496	0.970473	3	6
min_iid	0.000000	0.000000	0.000000	0.000000	0.000000	0	0
Marking_size_NegotiateProperty'NotAgreed_1							
count_iid	345.070000	10.611371	12.695406	16.844357	63.731960	230	569
max_iid	0.580000	0.148130	0.177223	0.235140	0.889671	0	3
min_iid	0.000000	0.000000	0.000000	0.000000	0.000000	0	0
Marking_size_NegotiateProperty'NotNegotiated_1							
count_iid	345.070000	10.611371	12.695406	16.844357	63.731960	230	569
max_iid	0.160000	0.090796	0.108628	0.144128	0.545320	0	2
min_iid	0.000000	0.000000	0.000000	0.000000	0.000000	0	0

Fig. 24. Performance report

7 Conclusion

The paper presented the initial steps aiming at the implementation of our negotiation protocol. First, we modeled the core part of this negotiation protocol using CPN Tools. The goal was to simulate a negotiation for the running example presented in Section 3.2. We used the simulation capabilities of CPN Tools to assess the correctness and the adequacy of the model. Although such simulations cannot prove the correctness of the model, they can, at least, show flaws in the model. In fact, the simulations helped us to identify problems that could be repaired. Some weaknesses were also identified and will be taken into consideration in the actual implementation of the model. The simulation also helped us in ascertaining that the protocol is suitable to the purposes it is intended for. Although each negotiator used a simple strategy, the negotiation process provided by/observed during the simulation corresponded to our expectations, and reasonable values were produced.

We also desired to have a stronger confidence about the model's correctness. Thus, we tried to perform state space analysis. We had to simplify the model and even to split it into three distinct sub-models. We managed to obtain full state spaces for the bargain and ballot sub-models, but not for the auction sub-model. Although we did not fully succeed, this process also helped us to identify flaws in the model. We tried to correct them both in the original model and in the sub-models.

Acknowledgment

This paper is partially supported by CAPES/Brazil under grant 4635/06-0, CNPq WebMaps II Project, and FAPESP.

References

1. Weigand, H., Heuvel, W.: Cross-organizational workflow integration using contracts. *Decision Support Systems* **33**(3) (July 2002) 247–265
2. Krishna, R., Karlapalem, K., Chiu, D.: An ER^{ec} framework for e-contract modeling, enactment and monitoring. *Data & Knowledge Engineering* **51**(1) (oct 2004) 31–58
3. Angelov, S.: Foundations of B2B Electronic Contracting. PhD thesis, Technische Universiteit Eindhoven (2005)
4. Bacarin, E., Madeira, E., Medeiros, C.: Contract e-negotiation in agricultural supply chains. *Intl. Journal of Electronic Commerce* (2007) <http://www.gvsu.edu/business/ijec> (to appear).
5. Jesen, K., Kristensen, L., Wells, L.: Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *Intl. J. on Software Tools for Technology Transfer* **9**(3-4) (2007) 213–254
6. Bartolini, C., Preist, C., Jennings, N.: A software framework for automated negotiation. In: SELMAS. (2004) 213–235
7. Chiu, D., Cheung, S., Hung, P., Chiu, S., Chung, A.: Developing e-negotiation support with a meta-modelling approach in a web services environment. *Decision Support Systems* **40**(1) (July 2005) 51–69
8. Reeves, D., Wellman, M., Grosof, B.: Automated negotiation from declarative contract descriptions. In: Proc. of the 5th International Conference on Autonomous Agents, Canada, ACM Press (2001) 51–58
9. Henderson, P., Crouch, S., Walters, R., Ni, Q.: Comparison of some negotiation algorithms using a tournament-based approach. In: Agent Technologies, Infrastructure, Tools and Applications for E-Services. Volume 2592 of Lecture Notes in Artificial Intelligence. Springer (Jan 2003) 137–150
10. FIPA: Fipa abstract architecture specification. Available at www.fipa.org (2000)
11. Malucelli, A., Palzer, D., Oliveira, E.: Ontology-based services to help solving the heterogeneity problem in e-commerce negotiations. *Electronic Commerce Research and Applications* **5**(1) (2006) 29–43
12. Governatori, G., Dumas, M., ter Hofstede, A., Oaks, P.: A formal approach to protocols and strategies for (legal) negotiation. In: ICAIL. (2001) 168–177
13. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Springer-Verlag (1997)

14. Chen, Y., Peng, Y., Finin, T., Labrou, Y., Cost, S.: A negotiation-based multi-agent system for supply chain management. In Working Notes of the Agents '99 Workshop on Agents for Electronic Commerce and Managing the Internet-Enabled Supply Chain., Seattle, WA, April 1999. (1999)
15. Liu, L., J, B.: Enhancing the CES Protocol and its Verification. Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (2005)
16. Dang, J., Huhns, M.: Concurrent Multiple-Issue Negotiation for Internet-Based Services. IEEE Internet Computing **10**(6) (2006) 42–49

Effectiveness of Coloured Petri nets for Modelling and Analysing the Contract Net Protocol

Jonathan Billington and Amar Kumar Gupta

Computer Systems Engineering Centre
School of Electrical and Information Engineering
University of South Australia
Mawson Lakes Campus, SA 5095, AUSTRALIA

Email: jonathan.billington@unisa.edu.au, gupay003@students.unisa.edu.au

Abstract. The Contract Net Protocol was developed to facilitate contract negotiation in Multi-Agent Systems, between an auctioneer and many bidders. It is therefore important to analyse the protocol to ensure that it terminates correctly and satisfies other important properties. There have been few attempts to model and analyse this protocol in the literature. The main paper on its verification, published in 2004, suggests that Coloured Petri nets are inadequate for this task. This seems to be due to a misunderstanding of what Coloured Petri Nets are. The main aim of this paper is therefore to show how Coloured Petri nets can be used effectively to model and analyse this protocol. We present a model of the protocol implemented in CPN Tools. The level of abstraction excludes details of the messages communicated between the agents and processing that does not affect the operation of the protocol. We analyse the protocol and show that it terminates correctly (there are no deadlocks or livelocks, and the terminal states show that the auctioneer and bidders have consistent states) and that there is no dead code (all procedures are executable). We also demonstrate that the channel bounds and number of terminal states depend linearly on the number of bidder agents participating in the negotiations for up to 6 bidders and conjecture that this is true in general.

1 Introduction

1.1 Background

Planning and task allocation are important activities for most organizations. Task allocation may involve the distribution of tasks among different subcontracting companies. The design and use of protocols to aid the above process is still of great interest to both business and government organizations. In this paper, we attempt to analyse one such protocol, called the Contract Net Protocol [17], which is used in Multi-Agent Systems (MAS) [4] to facilitate agent interaction. This work was motivated by discussions with researchers at the Australian Defence Science and Technology Organisation (DSTO), who are working on the application of multi-agent systems to transport logistics [16].

The Contract Net Protocol involves contracting between agents comprising an auctioneer (the service requesting agent) and a set of bidders (service providing agents). The auctioneer initiates the negotiation process by sending a Task Announcement to a number of bidders, who bid to provide the service or respond with a refusal. After all bids are received, the auctioneer selects the most suitable one (if present) and rejects all the others.

The first specification of the Contract Net Protocol was developed by Smith [17]. The protocol has also been adopted by the Foundation for Intelligent Physical Agents (FIPA) [8] and specified using the Unified Modelling Language (UML) [3], but with a minor modification to the one proposed by Smith [17]. The use of the Foundation for Intelligent Physical Agents - Agent Communication Language (FIPA-ACL) [7] for message interaction is quite complicated. To overcome this, Perugini [16] presents a new protocol representation, called the Protocol Flow Diagram, and uses it to describe the Contract Net Protocol.

We believe it is important to verify the properties of a protocol before implementation [2]. This is very important for the Contract Net Protocol as it is being used as the basis for developing more complex negotiation protocols [16]. Thus we see this work as a necessary first step for analysing more complex agent negotiation protocols.

1.2 Related Work

A lot of work has been reported regarding the application of the Contract Net Protocol in different domains (see for example [10, 18]), but there is little work on verifying the properties of the protocol. Nowostawski et al [13] advocate the use of Coloured Petri nets (CPNs) [11, 12] as a modelling technique for agent interaction protocols, using the Contract Net Protocol as an example. Unfortunately, they present an incomplete ‘Coloured Petri Net model for 3 bidders/contractors (their Fig. 7), which is not clear because no net annotations nor declarations are included, so that the CPN just looks like a Petri Net model. Hence it does not specify the behaviour of the protocol. This model reflects similar behaviour for three different contractors by repeating net structure, which does not take advantage of the power of coloured Petri nets in modelling similar behaviour by using the same net structure with different tokens. Further, there is no discussion of analysis, let alone verification of properties. Paurobally et al [15] represent the Contract Net Protocol (when its messages are expressed in FIPA-ACL) formally in extended Propositional Dynamic Logic and graphically in extended statecharts and prove properties like termination and consistency in beliefs. Based on the (Coloured) Petri net model of [13], Paurobally et al [15] argue that Petri nets are not suited to modelling and analyzing the Contract Net Protocol. The inadequacy of Petri nets for modelling agent interactions has also been discussed in [14]. We believe this is due to a misunderstanding of Petri nets and CPNs in particular, as demonstrated in Chapter 5 of [14]. This has further stimulated us to show that the Contract Net Protocol can be modelled and analysed using Coloured Petri Nets.

1.3 Contribution

In this paper, we present a Coloured Petri Net model of the Contract Net Protocol, based on the Protocol Flow Diagram representation of it in Chapter 4 of [16], and analyse it using state spaces.

This paper has a threefold contribution. Firstly, we present, for the first time, an abstract model of the Contract Net Protocol using Coloured Petri Nets that is suited to analysis. Secondly, we analyse the Contract Net Protocol and confirm the results in [15]. Further, we prove some additional properties of the protocol, including absence of livelocks, and obtain expressions for the number of terminal states and channel bounds, as a function of the maximum number of bidders. We thus demonstrate that Coloured Petri Nets are suited to the modelling and analysis of interaction protocols contrary to the claims made by Paurobally et al [15].

1.4 Organization

The rest of the paper is organized as follows. Section 2 discusses the main features of the Contract Net Protocol [16], leading to the development of our model in Section 3, which further explains the operation of the protocol. We analyse this model in Section 4. Lastly, Section 5 presents conclusions along with a discussion of future work.

2 Contract Net Protocol

2.1 Time Sequence Diagram Representation

The Contract Net Protocol is illustrated by the three time sequence diagrams shown in Figs. 1 to 3. These diagrams just represent the interactions between the auctioneer and one bidder. The time sequence diagrams use the terminology of Perugini [16] which is different from FIPA’s representation [9]. For instance, the **auctioneer** replaces the initiator, **bidder** replaces the participant, and **Task Announcement** the cfp (call for proposals). The agents (auctioneer and bidder agents) involved in the negotiations are represented by two vertical lines, where

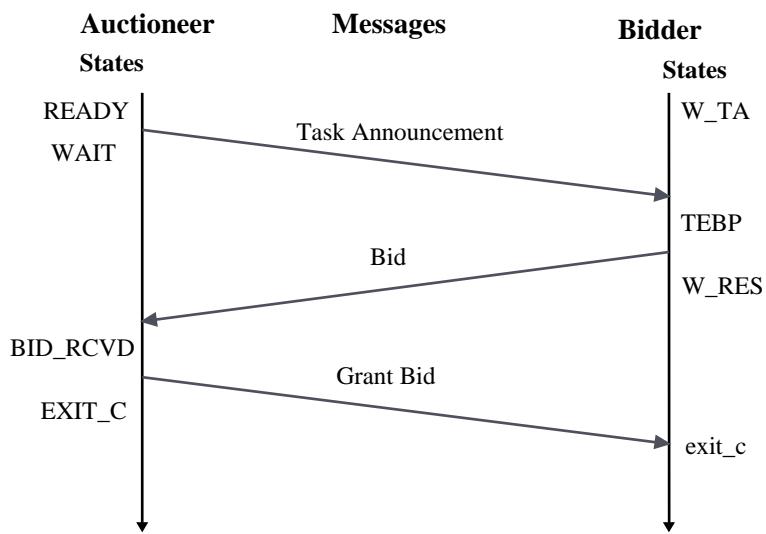


Fig. 1. Bidding Process ending in a Contract.

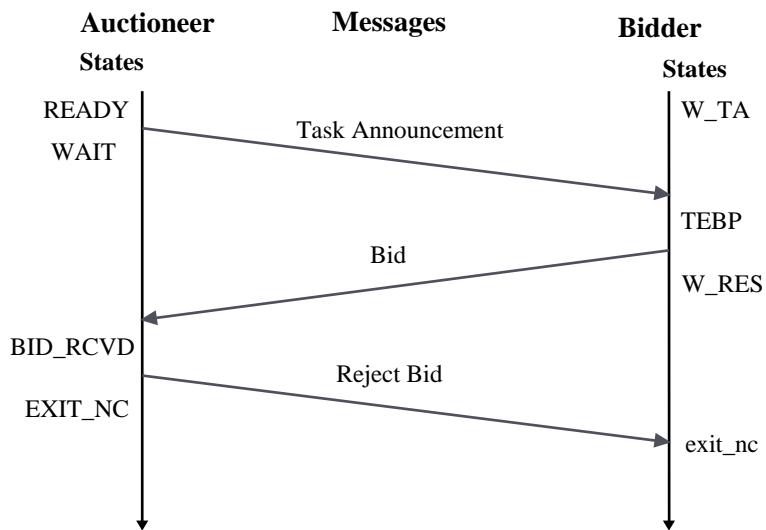


Fig. 2. Bidding Process ending Without a Contract.

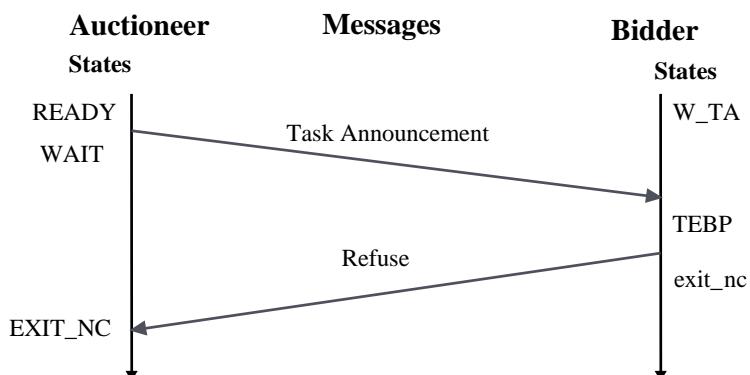


Fig. 3. Refusing Process.

time increases downwards. The so called ‘speech acts’ (messages communicated between the agents) are represented by arrows between the two vertical lines.

The states for the auctioneer and the bidders are given in Table 1. There are five possible states for the auctioneer and for each bidder. The table also defines the meaning of each state. READY and W_TA correspond to initial states, while (EXIT_NC, EXIT_C) and (exit_nc, exit_c) represent the set of terminal states of the auctioneer and bidders respectively. Terminal states of the auctioneer are represented by uppercase letters, whereas the bidders use lowercase. The terminology for terminal states is that used by Perugini [16].

Table 1. Representation of States.

Auctioneer	Bidders
READY (READY to send a Task Announcement) WAIT (WAITing for bids) BID_RCVD (BID ReCeivEd) EXIT_NC (EXIT with No Contract) EXIT_C (EXIT with Contract)	W_TA (Waiting for a Task Announcement) TEBP (Task Evaluation and Bid Preparation) W_RES (Waiting for REsult) exit_nc (exit with no contract) exit_c (exit with contract)

2.2 Operation

The auctioneer initiates the negotiation process by broadcasting a Task Announcement to a number of bidders. The Task Announcement contains the details of the tasks to be performed and the deadline for the submission of the bids. On receipt of the Task Announcement, the bidders evaluate the task and then decide whether to bid or refuse. Each bidder must either Bid or Refuse (to bid). The Bid may contain information concerning the price that must be paid to carry out the task, the time required to complete the task and any other aspect that the auctioneer might have asked for in the Task Announcement. The Refuse signifies that the bidder is not willing to perform the task. As the auctioneer is dealing with a number of bidders, it may receive both Bid and Refuse messages but can only receive one of either from each bidder. In case of a Refuse, the bidder exits the process without a contract, which the auctioneer records. Once a message (Bid or Refuse) is received from all the bidders, the auctioneer starts processing the bids (if any). After the selection of a bid, it sends a Grant Bid message to the concerned bidder and sends a Reject Bid message to the rest of the bidders. The auctioneer may also reject all the bids by sending a Reject Bid to those that have bid. The bidders exit the process with or without a contract on receipt of the Grant Bid or Reject Bid respectively.

The three possible scenarios of the negotiation process are illustrated by the time sequence diagrams in Figs. 1 to 3. The figures also show the sequence of messages exchanged between the auctioneer and the bidder and the state resulting from their receipt. For instance, the auctioneer changes state from READY to WAIT after broadcasting a Task Announcement, while a bidder that is waiting for a Task Announcement in the state W_TA, would change state to Task Evaluation and Bid Preparation (TEBP) on receipt of the Task Announcement. Figure 1 corresponds to the case in which the bidder sends a Bid in response to the Task Announcement. After considering all the bids (not shown in Fig. 1), the auctioneer grants this bid (and no others) by sending a Grant Bid message, resulting in a contract at the end of the negotiation process. This is also reflected by the terminal states, EXIT_C (for auctioneer) and exit_c (for bidder). Figure 2 represents a similar scenario where the auctioneer rejects the bid by sending a Reject Bid message and changes state to EXIT_NC. At the other end, the bidder in the state W_RES changes state to exit_nc on receipt of the message. The negotiation ends without a contract as is evident from the terminal states. The third scenario depicted by

Fig. 3 is the case when a bidder refuses to bid for the task by sending a Refuse message in response to the Task Announcement sent by the auctioneer. Without communication of any further messages, the negotiation ends without a contract.

Since the auctioneer is dealing with a number of bidders simultaneously, the auctioneer may be in several different states (but with respect to different bidders) during the negotiation. Even though the negotiation process with some bidders may come to an end (on submission of a Refuse in response to the Task Announcement), the process would still continue with respect to other bidders.

3 CPN Model of the Contract Net Protocol

In this section we firstly detail the assumptions made when creating the model and then define its data structures, before discussing the model's structure. Then using an equivalent flat model we describe all the procedures and the operation of the protocol. This allows visualisation of the message flows between the auctioneer and the bidders.

3.1 Assumptions

1. All the bidders are known to the auctioneer before the negotiation takes place. This is appropriate for software agents, or a list of preferred suppliers.
2. All the speech acts (Task Announcement, Bid, Refuse, Grant Bid and Reject Bid) are just represented by their names, as other information contained in these messages does not affect the protocol's actions.
3. All the bids are received before the process of selecting a bid occurs. This means that we do not have to model a deadline, and is reasonable for software agents and preferred suppliers.
4. The communication channel is reliable i.e. messages do not get lost but the receipt of messages is concurrent. This could correspond to a Task Announcement being placed on a website, and accessed by the bidders independently.
5. We only consider task allocation, and not any further interaction that may occur after the task has been allocated [16].

3.2 Declarations

Figure 4, taken directly from CPN Tools, shows the declarations for the CPN model of the Contract Net Protocol. The protocol facilitates interaction between a single auctioneer and multiple bidders. The identity of the bidders is represented by the colour set BDR as can be seen in the declarations (Fig. 4). Its value ranges from 1 to the maximum number of bidders (MaxBdrs). Hence MaxBdrs is a parameter of the model. The effect of varying this parameter on the analysis of the model is presented in Section 4.

The states of the auctioneer and the bidders are defined by the colour sets STauc and STbdr respectively according to Table 1. The state READY (for auctioneer) and W_TA (for bidders) corresponds to the initial state of the negotiation process in which the auctioneer is ready to broadcast a Task Announcement and the bidders are waiting for it. EXIT_NC and EXIT_C represent terminal states of the negotiation process for the auctioneer. EXIT_NC indicates that the negotiation has ended without a contract and EXIT_C that a contract has been awarded. The same applies for the bidders where the set of terminal states are represented by exit_nc and exit_c.

The colour sets MESauc and MESbdr define the messages communicated from the auctioneer to the bidders (TA, GB, RB) and from the bidders to the auctioneer (BID, REFUSE) respectively. They are defined in Table 2.

```

▼ Declarations
  ► Standard declarations
  ▼ (*-----BIDDERS-----*)
    ▼ val MaxBdrs=2;
    ▼ colset BDR=index B with 1..MaxBdrs;
    ▼ var bdr:BDR;
  ▼ (*-----STATES-----*)
    ▼ colset STAuc=with READY|WAIT|BID_RCVD|EXIT_NC|EXIT_C;
    ▼ colset STbdr=with W_TA|TEBP|W_RES|exit_nc|exit_c;
    ▼ colset BDR_STauc=product BDR*STAuc;
    ▼ colset BDR_STbdr=product BDR*STbdr;
  ▼ (*-----MESSAGES-----*)
    ▼ colset MESauc=with TA|GB|RB;
    ▼ colset MESbdr=with BID|REFUSE;
    ▼ colset BDR_MESauc=product BDR*MESauc;
    ▼ colset BDR_MESbdr=product BDR*MESbdr;
  ▼ (*-----RESPONSES-----*)
    ▼ colset RESP=int;
    ▼ var resp:RESP;
  ▼ (*-----GRANT-----*)
    ▼ colset GR1=with gr1;

```

Fig. 4. Declarations for the CPN model of the Contract Net Protocol.

Table 2. Messages and their representation in the CPN Model.

Message	Value in Colour Set MESauc or MESbdr
Task Announcement	TA
Bid	BID
Refuse	REFUSE
Grant Bid	GB
Reject Bid	RB

The auctioneer negotiates with a number of bidders simultaneously. Thus we need to associate the auctioneer's states and messages with the bidder's identity. This has been achieved by the declaration of the colour sets BDR_STauc and BDR_MESauc. Because all the bidders are stored in one place, their states and messages are also associated with the bidder's identity as defined by colour sets BDR_STbdr and BDR_MESbdr respectively.

The colour set RESP (RESPonse) is defined as an integer which keeps track of the number of responses received. This is needed to detect when all the responses from the bidders have been received. Lastly, the singleton colour set GR1 (GRant only 1) is defined to ensure that only one bidder is granted the contract.

3.3 Model Structure and CPN Diagrams

Figure 5 shows the main page of the CPN model of the Contract Net Protocol. It contains 5 places and 4 substitution transitions and provides the main structure for the protocol. The auctioneer's and the bidders' states are modelled by the places Auctioneer's State and Bidders State respectively. They communicate with each other via the abstract communication channel represented by the places AUCTIONEER 2 BIDDERS and BIDDERS 2 AUCTIONEER as seen in the middle of Fig. 5. The place Responses records the number of responses received from the bidders, so that the auctioneer knows when all bids are in. The main actions of the protocol are modelled by the 4 substitution transitions, Send Messages, Receive Messages, Bidding OR Refusing and Receive Responses.

The auctioneer's procedures for sending messages to the bidders are hidden by the substitution transition Send Messages which is expanded in Fig. 6. The transitions Broadcasting TAs,

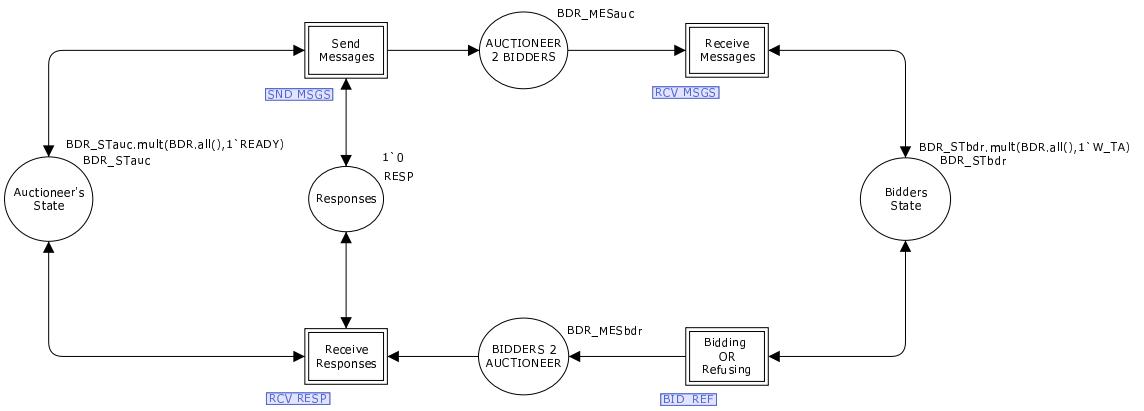


Fig. 5. Main Page of the CPN Model

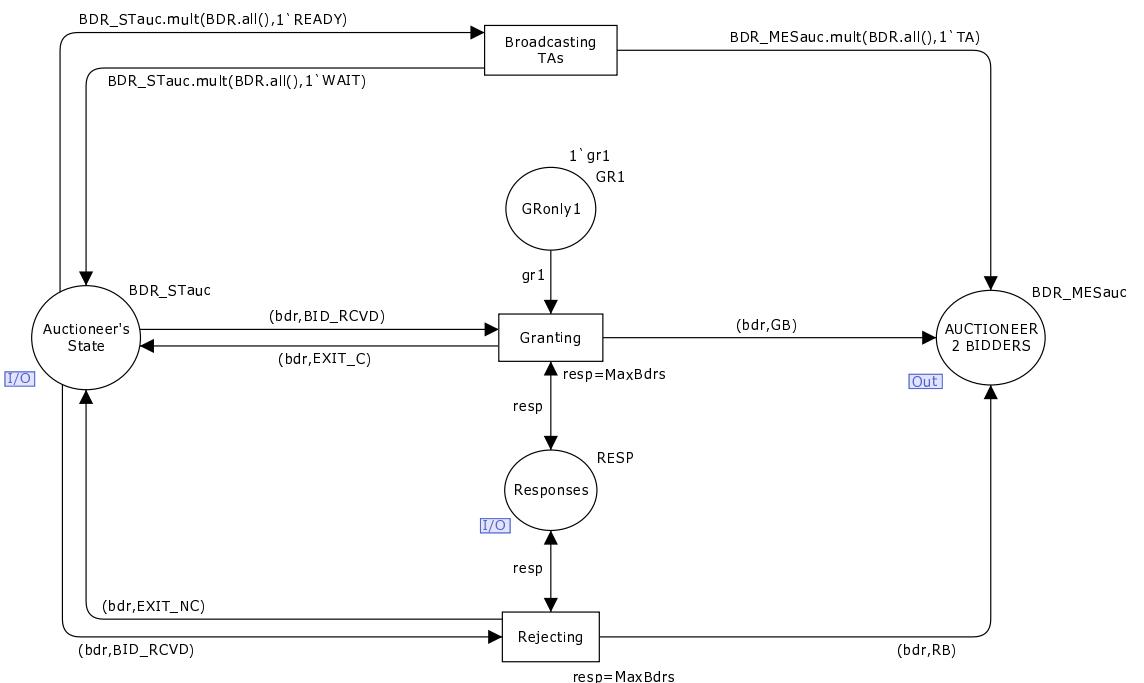


Fig. 6. Send Messages

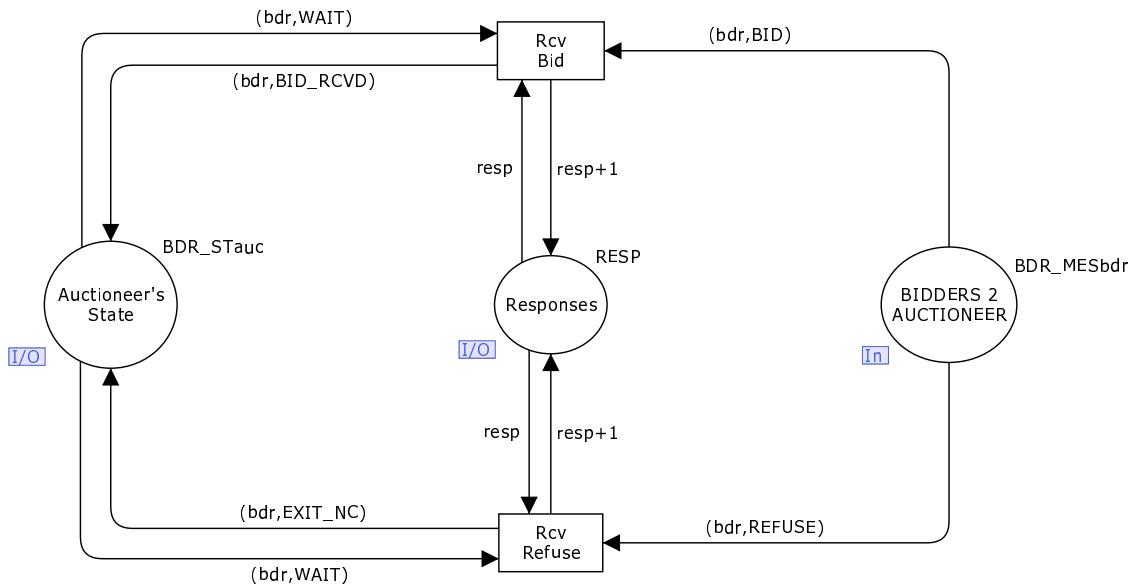


Fig. 7. Receive Responses

Granting and Rejecting define the rules for broadcasting a Task Announcement, and Grant and Reject Bid speech acts respectively. Receive Responses, shown in Fig. 7, details the process of receiving Bid and Refuse messages from the bidders in response to the Task Announcement, modelled by the transitions Rcv Bid and Rcv Refuse respectively. Similarly, Fig. 8 models the actions taken by the bidders on reception of messages from the auctioneer and Fig. 9 details their bidding and refusing procedures.

This structure allows the individual procedures used by the different parties to be defined in separate blocks, which makes each individual diagram easier to read. However, the disadvantage is that it is no longer readily apparent how the bidders and auctioneer interact, which is important for understanding how the protocol works. Now that we have introduced the main procedures we would like to describe the model in detail with respect to a flat model that is equivalent to combining Figs. 6 to 9. This will make it easier to understand the operation of the protocol as described in Section 3.4.

Figure 10 shows the CPN diagram for the Contract Net Protocol in a single page which makes it easier to visualise message flow while relating it to the protocol's procedures. With reference to Fig. 10, the auctioneer is modelled on the left, the bidders on the right, and they communicate via two places, AUCTIONEER 2 BIDDERS and BIDDERS 2 AUCTIONEER, which represent a reliable (but not ordered) channel for each direction of communication. The auctioneer is modelled with 3 places, which represent its state (Auctioneer's State), the number of responses received (Responses) and if a bid has been granted (GRonly1). In contrast, the bidders are modelled with only a single place, (Bidders State), representing the states of each of the bidders. The significance of each place is very briefly discussed below.

Auctioneer's State: This place stores the states of the auctioneer with respect to all the bidders. It is typed by the colour set BDR_STauc (product of BDR and STauc) to identify the state of the auctioneer in relation to a particular bidder. The initial marking of this place is: BDR_STauc.mult(BDR.all(),1'READY).

BDR.all is a predefined function that returns a multi-set comprising the set of all bidders. BDR_STauc.mult is another predefined function that returns a product based on the multi-sets specified in its argument, i.e. one appearance of all the bidders and 1'READY. For this argument, the function returns a multi-set with one appearance of all the bidders in the READY

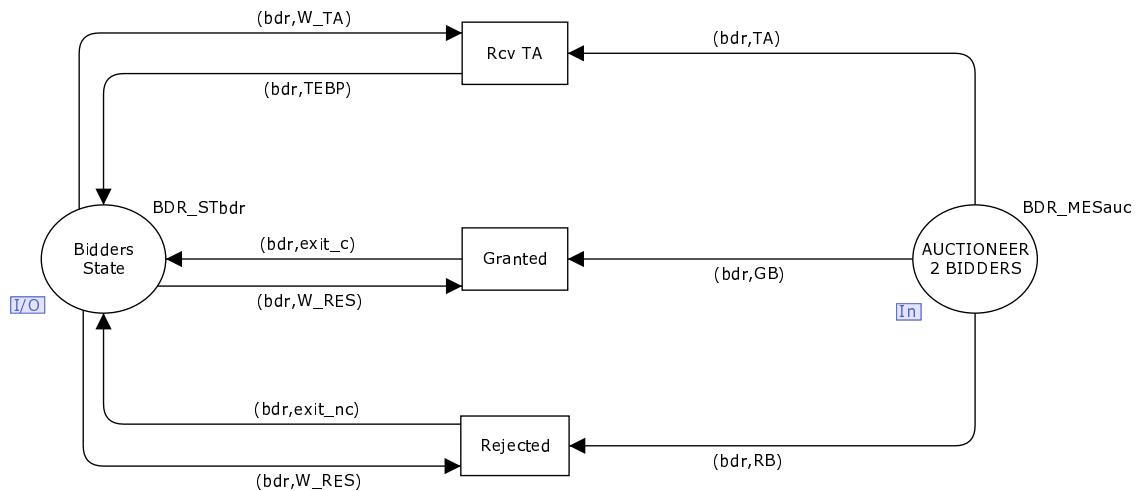


Fig. 8. Receive Messages

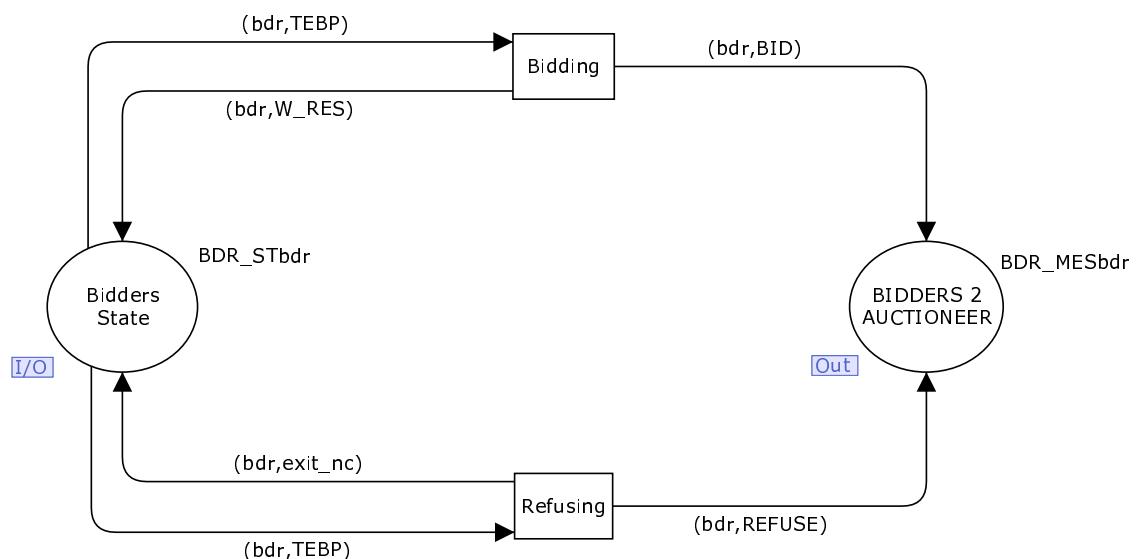


Fig. 9. Bidding OR Refusing

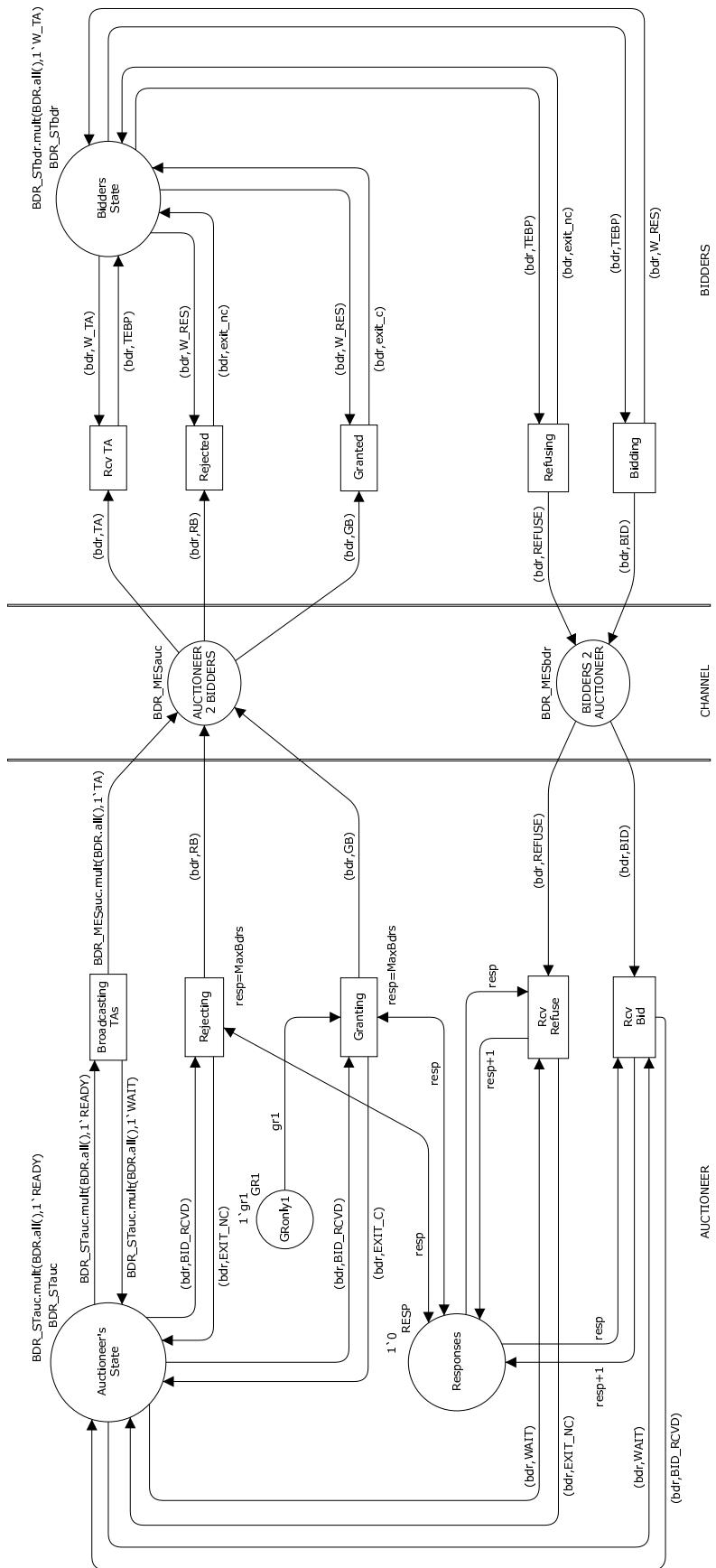


Fig. 10. CPN Diagram of the Contract Net Protocol.

state. For example for $\text{MaxBdrs}=3$, the function would evaluate to: $1'(\text{B}(1),\text{READY})++1'(\text{B}(2),\text{READY})++1'(\text{B}(3),\text{READY})$.

Bidders State: This place stores the states of all the bidders. Typed by the colour set BDR_STbdr (product of BDR and STbdr), the initial marking of this place corresponds to the multi-set evaluated by the function: $\text{BDR_STbdr}.\text{mult}(\text{BDR.all}(), 1'W_TA)$, which returns a multi-set with one appearance of all the bidders in the state W_TA .

AUCTIONEER 2 BIDDERS: This place represents an abstract communication channel used by the auctioneer to broadcast messages to the bidders. Only the messages of the auctioneer (TA , GB and RB) would be available at this place along with the identity of the bidder to which it is destined and hence it is typed by the colour set BDR_MESauc . The place is initially empty.

BIDDERS 2 AUCTIONEER: Similar to **AUCTIONEER 2 BIDDERS**, this place contains only the bidder's messages (BID , $REFUSE$) along with their identity as defined by BDR_MESbdr . This place is also empty initially.

Responses: This place records the number of messages received (whether BID or $REFUSE$) from the bidders, so that the auctioneer knows when all responses to the Task Announcement have been received. It thus contains the token 0 at the start of the negotiation process.

GRonly1: (GRant only 1) This place ensures that a maximum of one bid is granted.

3.4 Operation of the model

Initially, the auctioneer is ready to broadcast a Task Announcement to all bidders (in the state $READY$ with respect to all bidders) and all the bidders are in the state W_TA (Waiting for Task Announcement). The auctioneer initiates the negotiation process by sending a Task Announcement to each of the bidders via the occurrence of the transition **Broadcasting TAs** (Broadcasting Task Announcements). After **Broadcasting TAs** fires, the auctioneer is in the state **WAIT** for each of the bidders.

At this point, the transition **Rcv TA** (Receive Task Announcement) is enabled as the bidders are in the state W_TA and the TA has arrived. As the bidders are all different entities and in general geographically distributed, we model all the activities of the bidders to be concurrent. When the transition **Rcv TA** fires, a bidder removes its Task Announcement from the channel and changes state to **TEBP** (Task Evaluation and Bid Preparation).

Once any bidder is in the state **TEBP**, it first evaluates the task (by considering its resources and any other criteria needed for the execution of the task). After the evaluation of the task, the bidder decides whether to bid or refuse. If the bidder opts to bid, then it prepares the bid. The bid contains details such as the price to be paid, the time it would take to complete the task and any other details that the auctioneer might have asked for. We neither consider the details of the Task Announcement nor the details of the bid in the model. The decision to bid or refuse is modelled as a non-deterministic choice, by transitions **Bidding** and **Refusing**. The occurrence of the transition **Bidding** changes the state of the bidder to W_RES (a state where it would be waiting for a decision on its submitted bid), while the occurrence of the transition **Refusing** changes the state of the bidder to $exit_nc$ (a state of the bidder where the negotiation has ended without any contract).

After the message from the bidder gets through the communication channel and reaches the auctioneer (in the state **WAIT** for that bidder), the transition **Rcv Bid** or **Rcv Refuse** is enabled depending on the message BID and $REFUSE$ respectively. The occurrence of **Rcv Bid** causes the auctioneer to change state (with respect to that particular bidder) from **WAIT** to **BID_RCVD** while the occurrence of **Rcv Refuse** causes the auctioneer to change state to **EXIT_NC** (the negotiation process is complete and no contract is formed). In addition, the number of responses received is increased by one each time **Rcv Bid** or **Rcv Refuse** occurs. The purpose of this mechanism is to keep track of the number of bidders who have replied to the

Task Announcement to determine when all replies have been received, which would be equal to the value of MaxBdrs. When this happens, the transitions **Granting** and **Rejecting** are enabled provided the auctioneer receives at least one bid in response to the Task Announcement. If all bidders refuse to bid, the protocol terminates with the auctioneer in the EXIT_NC state and all the bidders in exit_nc.

Given that there was at least one bid received, the auctioneer could grant any one of the bids or reject all the bids (in case none of the bids received are suitable). We model this process of selection or rejection of a bid non-deterministically. When all the responses are received, the guard on transitions **Granting** and **Rejecting** evaluates to true. When the auctioneer receives at least one bid, the transitions **Granting** and **Rejecting** would both be enabled. The place **GRonly1** ensures that only a single bid is granted (if at all). The absence of a token at the place **GRonly1** signifies that a bid has been granted and the auctioneer's objective achieved.

On occurrence of **Granting**, the auctioneer changes state from BID_RCVD to EXIT_C with respect to the chosen bidder and sends a GB (Grant Bid) message to that bidder. All the other bids received must then be sent a RB (Reject Bid) message via **Rejecting**. It may also be possible that the auctioneer chooses to reject all the bids in which case the transition **Granting** would not occur. The transition **Rejecting** operates in a similar way except that it causes the auctioneer to change its state from BID_RCVD to EXIT_NC (with respect to the bid that is being rejected) and sends a RB (Reject Bid) to the bidder of the rejected bid, when fired.

Once the messages reach the bidders, the transition **Granted** or **Rejected** is enabled depending on the message GB and RB respectively. The occurrence of **Granted** causes the bidder to change its state from W_RES to exit_c while the occurrence of **Rejected** causes the transition in state of the bidder from W_RES to exit_nc. This process continues until all the bidders reach a terminal state (exit_c or exit_nc). We expect that when the negotiations have terminated, the bidder that has received the contract (in exit_c) will be the same as the bidder that the auctioneer believes has the contract (i.e. is in state EXIT_C).

4 State Space Analysis Results

The state space analysis results are presented in Table 3. It shows the properties of the state space as a result of varying the parameter MaxBdrs from 1 to 6.

Table 3. State space analysis results as a function of the parameter MaxBdrs.

Properties/MaxBdrs	1	2	3	4	5	6
State Space Nodes	10	54	290	1578	8798	50238
State Space Arcs	10	93	721	5185	36181	248833
Time (hh:mm:ss)	00:00:00	00:00:00	00:00:00	00:00:03	00:02:02	01:08:11
Scc Graph Nodes	10	54	290	1578	8798	50238
Scc Graph Arcs	10	93	721	5185	36181	248833
Dead Markings	2	3	4	5	6	7
Home Space (Dead Markings)	true	true	true	true	true	true
Dead Transition Instances	none	none	none	none	none	none
Live Transition Instances	none	none	none	none	none	none
Channel Bound	1	2	3	4	5	6

4.1 Absence of Deadlocks and Consistency in Beliefs

The state space is illustrated for MaxBdrs = 1 in Fig. 11. We can observe from the table that the number of dead markings is one more than MaxBdrs:

$$\text{No. of Dead Markings} = \text{MaxBdrs} + 1.$$

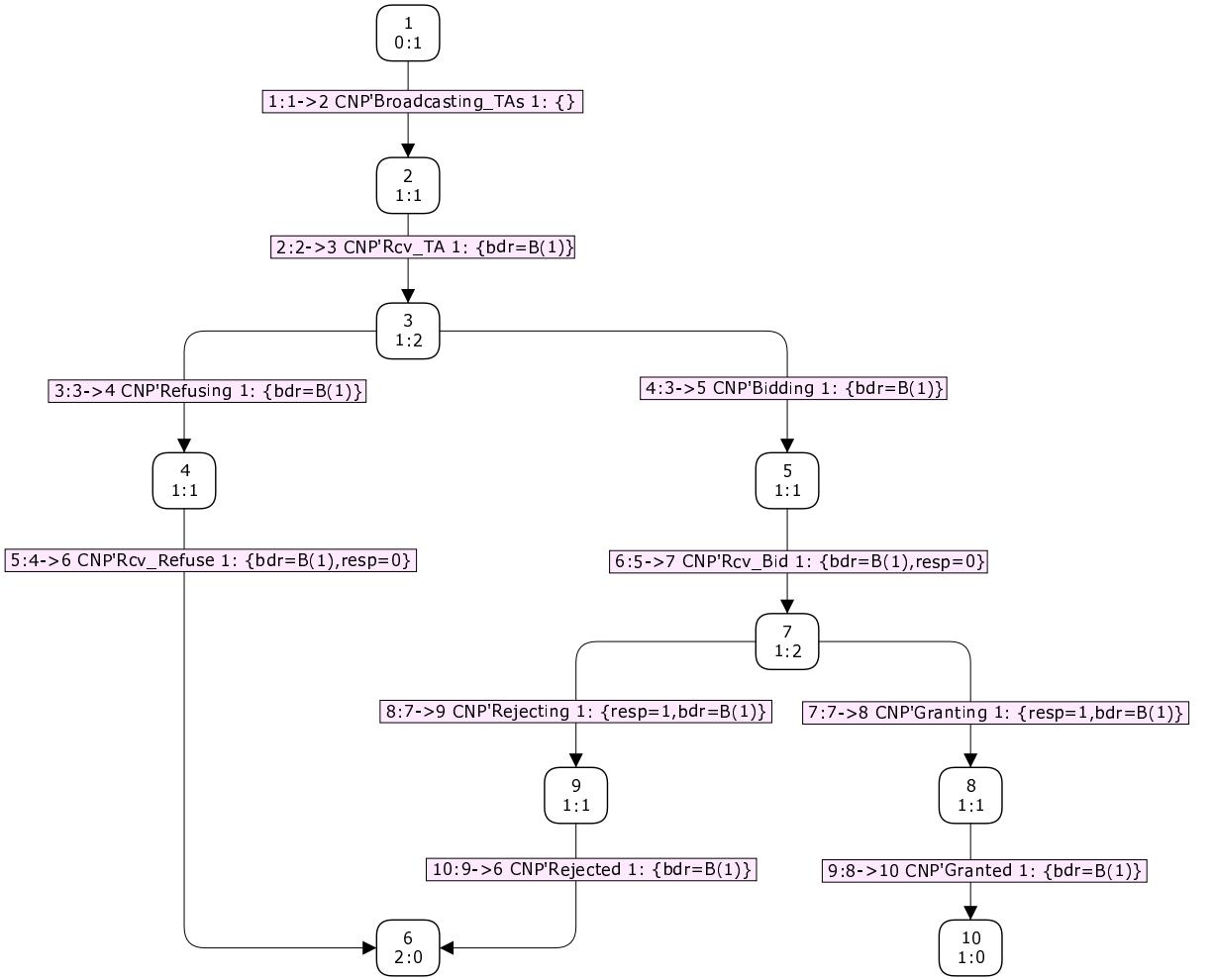


Fig. 11. Reachability Graph ($\text{MaxBdrs}=1$)

For any value of MaxBdrs , one of the dead markings corresponds to a contract not being established at the end of the negotiations. This marking consists of all the bidders in the state `exit_nc` and the auctioneer in the state `EXIT_NC` with respect to all the bidders. This is marking 6 in Fig. 11. This situation can be reached in two ways. Firstly, all the bidders could have submitted a `REFUSE` message (not willing to bid for the task) in response to the Task Announcement and changed state to `exit_nc`. The auctioneer on receipt of the `REFUSE` messages would change state to `EXIT_NC`. When the auctioneer receives all the messages (all `REFUSE`), the auctioneer would be in the state `EXIT_NC` with respect to all the bidders. The negotiations thus terminate with no contract being awarded. This corresponds to path 3,4,6 in Fig. 11. The second way in which the above scenario could occur would be after the receipt of all the replies from the bidders, with the number of bids ranging from at least one to a maximum of MaxBdrs . The auctioneer rejects all the bids by sending a `RB` message to each of the bidders that bid and changes state to `exit_nc` in each case. It would already be in this state for those bidders that submitted a `REFUSE` message. At the other end, those bidders waiting for a response to their bid (state `W_RES`) on receipt of the `RB` message, would change state to `exit_nc`. Other bidders who had submitted a `REFUSE` message would already be in this state. This corresponds to path 3,5,7,9,6 in Fig. 11. The places (`AUCTIONEER` 2 `BIDDERS` and `BIDDERS` 2 `AUCTIONEER`) representing the communication channel are empty, signifying

no unprocessed messages from either the auctioneer or any of the bidders. The value of the token on the place **Responses** is equal to MaxBdrs, since the auctioneer has received all the responses. The place **GRonly1** contains the gr1 token indicating that the transition **Granting** wasn't fired, and hence no bid had been granted. Though the protocol would end without a contract being formed, this particular dead marking is acceptable, because the auctioneer may not receive any bids or may reject all the bids if none of the received bids are acceptable.

The additional MaxBdr dead markings comprise for each of the MaxBdrs bidders, one bidder in the state **exit_c** with the rest of the bidders in the state **exit_nc** and the auctioneer in state **EXIT_C** with respect to the same bidder (who is in the state **exit_c**), and **EXIT_NC** with respect to the rest of the bidders. This signifies that a contract has been formed at the end of the negotiations with one bidder while the negotiations with the rest of the bidders have ended without a contract. The places representing the communication channel are empty signifying that all the messages of the auctioneer and the bidders have been processed. The place **Responses** contains a token whose value equals MaxBdrs indicating that all the responses have been received. Lastly, the place **GRonly1** is empty, signifying the grant of a bid which is also corroborated by the auctioneer and one of the bidders only, being in the state **EXIT_C** and **exit_c** respectively. These dead markings are desired terminal states of the protocol. When **MaxBdrs=1**, there is only one of these markings, corresponding to marking 10 in Fig. 11.

<pre> 210: CNP'Auctioneer's_State 1: 1` (B(1),EXIT_NC)++ 1` (B(2),EXIT_NC)++ 1` (B(3),EXIT_NC) CNP'AUCTIONEER_2_BIDDERS 1: empty CNP'Bidders_State 1: 1` (B(1),exit_nc)++ 1` (B(2),exit_nc)++ 1` (B(3),exit_nc) CNP'BIDDERS_2_AUCTIONEER 1: empty CNP'Responses 1: 1` 3 CNP'GRonly1 1: 1` gr1 val it = () : unit </pre>	<pre> 242: CNP'Auctioneer's_State 1: 1` (B(1),EXIT_NC)++ 1` (B(2),EXIT_NC)++ 1` (B(3),EXIT_C) CNP'AUCTIONEER_2_BIDDERS 1: empty CNP'Bidders_State 1: 1` (B(1),exit_nc)++ 1` (B(2),exit_nc)++ 1` (B(3),exit_c) CNP'BIDDERS_2_AUCTIONEER 1: empty CNP'Responses 1: 1` 3 CNP'GRonly1 1: empty val it = () : unit </pre>
<pre> 243: CNP'Auctioneer's_State 1: 1` (B(1),EXIT_NC)++ 1` (B(2),EXIT_C)++ 1` (B(3),EXIT_NC) CNP'AUCTIONEER_2_BIDDERS 1: empty CNP'Bidders_State 1: 1` (B(1),exit_nc)++ 1` (B(2),exit_c)++ 1` (B(3),exit_nc) CNP'BIDDERS_2_AUCTIONEER 1: empty CNP'Responses 1: 1` 3 CNP'GRonly1 1: empty val it = () : unit </pre>	<pre> 249: CNP'Auctioneer's_State 1: 1` (B(1),EXIT_C)++ 1` (B(2),EXIT_NC)++ 1` (B(3),EXIT_NC) CNP'AUCTIONEER_2_BIDDERS 1: empty CNP'Bidders_State 1: 1` (B(1),exit_c)++ 1` (B(2),exit_nc)++ 1` (B(3),exit_nc) CNP'BIDDERS_2_AUCTIONEER 1: empty CNP'Responses 1: 1` 3 CNP'GRonly1 1: empty val it = () : unit </pre>

Fig. 12. Node Descriptors for the dead markings (**MaxBdrs=3**)

This is illustrated further with the help of Fig. 12 from CPN Tools, which shows the node descriptors for the dead markings for three bidders. Node 210 corresponds to the dead marking where no contract is formed at the end of the negotiations. The multi-set of tokens on the place **Auctioneer's_State** shows that the auctioneer is in the state **EXIT_NC** with respect to all the three bidders. The place **Bidders_State** shows all three bidders are in the state **exit_nc**. The channel places are empty which is also true for the rest of the dead markings (242, 243 and 249). The place **Responses** contains a single 3 token indicating that the responses from all three bidders have been received, which is also the case for all the dead markings. The place **GRonly1** contains one gr1 token indicating that no bid has been granted. Node Descriptors of the remaining three dead markings (242, 243 and 249) in Fig. 12 reveal that a contract is formed between the auctioneer and the bidders B(3), B(2) and B(1) respectively. This is

consistent with the absence of a token at the place GRonly1 in these three cases. The total number of dead markings is 4 ($3 + 1$) as given by the generalized expression, of which one case corresponds to no contract being formed and the remaining three cases correspond to a contract being formed as described above.

The above discussions also justify the consistency in beliefs between the auctioneer and the bidders. It is clear from Fig. 12 that the bidder that has been awarded the contract (in state exit_c) is the same as the bidder that the auctioneer believes has the contract (in state EXIT_C with respect to that bidder). Similarly, the bidders that have not been awarded a contract (in state exit_nc) are the same as the bidders that the auctioneer believes haven't received the contract (in state EXIT_NC with respect to those bidders). This belief is consistent as can be seen in each of the dead markings in Fig. 12 and would hold in general.

4.2 Absence of Livelocks and Proper Termination

As can be seen from Table 3, the size of the state space increases exponentially with the number of bidders. Also, the number of nodes and arcs in the Scc Graph always remain the same as that of the State Space for all values of MaxBdrs that we have examined. This leads us to the conclusion that there is no cyclic behavior in the system, which is expected. This indicates that there are no livelocks, further corroborated by the fact that all the dead markings ($\text{MaxBdrs} + 1$) form a home space for all the values of MaxBdrs specified in Table 3. We conjecture that this would be true for any value of MaxBdrs. Since all the dead markings are desirable and they form a home space (for the cases examined), this implies that the system will always terminate correctly. The presence of dead markings excludes the possibility of live transition instances, as is confirmed in Table 3.

4.3 Absence of Dead Code

There are no instances of dead transitions for any value of MaxBdrs examined, revealing that all transitions are required (no dead code). This means that all the specified actions are executed.

4.4 Channel Bound

Table 3 also shows the bounds (upper integer bounds) on the communication channel which is the same for both **AUCTIONEER 2 BIDDERS** and **BIDDERS 2 AUCTIONEER**. As can be seen, this limit is equal to MaxBdrs for all the cases considered.

5 Conclusions and Future Work

In this paper, we have presented the first complete CPN model of the Contract Net Protocol. We claim the level of abstraction of the model is appropriate for analysing important properties of the protocol. For the values of MaxBdrs considered (1 to 6), we have proved a number of properties using state space analysis. Firstly, if the protocol terminates, then the protocol terminates correctly (partial correctness). Secondly, the fact that all the dead markings form a home space implies the absence of livelocks in the system. This ensures that the protocol will always terminate in all possible behaviours (progress). The number of terminal states is given by $\text{MaxBdrs}+1$, corresponding to a contract with each of the bidders or no contract being granted. Further we show that both channel bounds are limited to MaxBdrs.

We therefore believe that the model and results in this paper refute the claims made by Paurobally [14, 15] that Coloured Petri Nets are not suited to the modelling and analysis of interaction protocols, by illustrating their ability to successfully model and analyse the Contract Net Protocol. We believe our CPN model is complete, non-ambiguous and compact, being able

to be drawn on a single page. The CPN models the multithreaded nature of the auctioneer, as it deals with many bidders concurrently. Further it is parametric, modelling the Contract Net Protocol for any number of bidders using the same structure and inscriptions, contrary to the model presented in Fig. 7 of Nowostawski et al [13], which is structurally different, incomplete and ambiguous. Further, the CPN diagram and its declarations provide a semantics for the Protocol Flow Diagram representation of Perugini [16].

In addition to proving correct termination and consistency in beliefs (that the protocol terminates with a common belief between the auctioneer and each of the bidders) as was done in [15], we have also proved the absence of livelocks. Further we have shown how the number of terminal states and channel bounds are related to the parameter MaxBdrs.

This work can be extended in many directions. We would like to prove these properties for all values of MaxBdrs and then consider relaxing Assumptions 3 and 5. This would introduce deadlines and allow further interaction between the auctioneer and the successful bidder as the task proceeds. One could then consider relaxing Assumption 1 to extend the model to open multi-agent systems, where the identities of all the bidders are not necessarily known a priori. We believe that Assumption 4 is realistic and that the abstractions made in Assumption 2 do not affect the properties we are trying to prove, which do not concern data. We would also like to extend this work to more elaborate negotiation protocols such as the Extended Contract Net Protocol (ECNP) [5, 6], the Contract Net Protocol extension (CNP-ext) [1] and the Provisional Agreement Protocol (PAP) [16]. Each of these protocols provides greater flexibility with planning and task allocation than its preceding ones and is considerably more complex.

Acknowledgements

The authors would like to acknowledge fruitful discussions with their colleagues, Guy Gallasch and Nimrod Lilith, regarding this work. We are also very grateful to the anonymous reviewers for their constructive comments that helped us to improve the paper.

References

1. S. Aknine, S. Pinson, and M. F. Shakun. An Extended Multi-Agent Negotiation Protocol. *Autonomous Agents and Multi-Agent Systems*, 8(1):5–45, 2004.
2. J. Billington, G. E. Gallasch, and B. Han. A Coloured Petri Net Approach to Protocol Verification. In *Lectures on Concurrency and Petri Nets, Lecture Notes in Computer Science*, volume 3098, pages 210–290. Springer-Verlag, 2004.
3. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 2nd edition, 2005.
4. J. Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison Wesley Longman, 1999.
5. K. Fischer and N. Kuhn. A DAI Approach to Modelling the Transportation Domain, DFKI Research Report RR-93-25. German Research Centre for Artificial Intelligence (DFKI), Saarbrücken, 1993.
6. K. Fischer, J. P. Müller, I. Heimig, and A. W. Scheer. Intelligent Agents in Virtual Enterprises. In *Proceedings of the 1st International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology, London, UK*, pages 205–223, 1996.
7. Foundation for Intelligent Physical Agents-Agent Communication Language specification (FIPA-ACL). <http://www.fipa.org>.
8. Foundation for Intelligent Physical Agents (FIPA). <http://www.fipa.org>.
9. Foundation for Intelligent Physical Agents (FIPA). <http://www.fipa.org/specs/fipa00029/SC00029>.
10. F. S. Hsieh. Modelling and Analysis of Contract Net Protocol. In *Lecture Notes in Computer Science*, volume 3140, pages 142–146. Springer-Verlag Berlin Heidelberg, 2004.
11. K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volumes 1 to 3, Basic Concepts, Analysis Methods and Practical Use*. Monographs in Theoretical Computer Science. Springer-Verlag, 2nd edition, 1997.
12. 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, Springer-Verlag, 2007.

13. M. Nowostawski, M. Purvis, and S. Cranefield. A Layered Approach for Modelling Agent Conversations. In *Proceedings of the 2nd International Workshop on Infrastructure for Agents, MAS, and Scalable MAS, a satellite workshop of the 5th International Conference on Autonomous Agents, Montreal*, pages 163–170, 2001.
14. S. Paurobally. *Rational Agents and the Processes and States of Negotiation*. PhD thesis, Imperial College, London, UK, 2002.
15. S. Paurobally, J. Cunningham, and N. R. Jennings. Verifying the Contract Net Protocol: A Case Study in Interaction Protocol and Agent Communication Language Semantics. In *Proceedings of 2nd International Workshop on Logic and Communication in Multi-Agent Systems, Nancy, France*, pages 98–117, 2004.
16. D. Perugini. *Agents for Logistics: A Provisional Agreement Approach*. PhD thesis, The University of Melbourne, Victoria, Australia, 2006.
17. R.G. Smith. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions On Computers*, C-29(12):1104–1113, 1980.
18. P. Ulam, Y. Endo, A. Wagner, and R. Arkin. Integrated Mission Specification and Task Allocation for Robot Teams-Design and Implementation. In *Proceedings of the IEEE International Conference on Robotics and Automation, Roma, Italy*, pages 4428–4435, 2007.

Modeling Grid Workflows with Colored Petri Nets^{*}

Carmen Bratosin, Wil van der Aalst, and Natalia Sidorova

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
c.c.bratosin@tue.nl, w.m.p.v.d.aalst@tue.nl, n.sidorova@tue.nl

Abstract. Grid computing refers to the deployment of a widely distributed architecture for the execution of computationally challenging tasks. The grid provides a set of distributed resources which can be used for “computing on demand” or for constructing a “virtual supercomputer”. Recently, several researchers started to look at the relation between workflow management and grid computing. The flow of work through a grid can be seen as a classical “workflow”. However, as opposed to the classical workflows, the resources are not humans and are not managed by some centralized client-server architecture. Instead, the grid is highly distributed and the resources are computing power, memory, etc. Currently, there is no conceptual framework for grid computing and the role of workflows in grids is unclear. This paper provides initial steps towards a conceptual framework expressed in terms of Colored Petri Nets. CPN Tools is used to model grids while focusing on the workflow aspects. The resulting model can be analyzed to detect deadlocks, etc. The framework is illustrated using process mining as an application.

Keywords: Colored Petri nets; grid computing; modeling.

1 Introduction

Grid computing [21] is concerned with the development and advancement of technologies that provide seamless and scalable access to wide-area distributed resources. Currently, many researchers and practitioners are developing software to support grid computing. A well-known example is the *Globus Toolkit* which provides an open source software toolkit for building grids [17]. Within the grid community several research groups have made attempts to adopt ideas from workflow management and apply them in a grid context [15, 18, 19, 23, 25]. For many grid applications the workflow-paradigm is quite natural, e.g., complex scientific computations can be modeled as workflows. However, unlike classical workflows the control is decentralized and resources are computing power, memory, etc. rather than people.

* This research is supported by the GLANCE NWO project “Workflow Management for Large Parallel and Distributed Applications”.

Despite the current interest in grids and workflow, *a good conceptual model of grids is missing* and most researchers are focusing on the practical realization of grids. Terms like “resource”, “job”, and “workflow” are subject to multiple interpretations. Therefore, in this paper, we model the basic grid concepts in terms of *Colored Petri Nets* (CPNs, [20]). The main purpose is to clarify the basic concepts. Moreover, we also show that the mapping of grids onto CPNs allows for all kinds of analysis. Given the fact that the allocation and deallocation of resources in grids is done in a distributed manner and that multiple resources may be involved in some task, a grid workflow may easily deadlock. Therefore, this paper will focus on the use of state-space analysis to discover deadlocks.

Grids are often used in areas where there is a need for a lot of (preferably inexpensive) computing power. Examples can be found in scientific computing, e.g., SETI@home searches for possible evidence of radio transmissions from extraterrestrial intelligence using data from radio telescopes. SETI@home uses CPU-scavenging for this, i.e., a grid of unused desktop computers is exploited to analyze the radio transmissions. This particular form of grid use is also called “voluntary computing” because the resources are made available without a clear economic motive for the participants. Another interesting application domain is the use of grids for data mining to analyse the large volumes of data generated today (cf. the DataMiningGrid project [2]). In this paper, we will focus on a particular application: the utilization of grid computing for *process mining*. The goal of process mining is to extract models (e.g. Petri nets) from event logs [9]. This is possible because many systems ranging from enterprise information systems and web applications to embedded and high-tech systems are collecting enormous volumes of audit trails. To deal with these large amounts of data and computationally expensive process mining algorithms, grids are particularly useful. High-level process mining tasks can easily be described as workflows where the activities correspond to the execution of particular process mining algorithms. Therefore, process mining is an interesting application domain for grid computing.

The remainder of the paper is organized as follows. In Section 2 we present a running example and motivate the utilization of grid computing for process mining. Section 3 introduces the basic grid concepts which are mapped onto CPNs in Section 4. Related work is discussed in Section 6 and Section 7 concludes the paper.

2 Running Example: Applying Grid Technology to Process Mining

As indicated in the introduction, in this paper we focus on using grid technology for large process mining tasks. In recent years, process mining has emerged as a way to analyze systems and their actual use based on the event logs they produce [11]. Note that, unlike classical data mining, the focus of process mining is on concurrent processes and not on static or mainly sequential structures. A

classical example is the α -algorithm [11] which automatically constructs a Petri net based on a set of observed system traces.

Process mining is applicable to a wide range of systems. These systems may be pure information systems (e.g., ERP systems) or systems where the hardware plays a more prominent role (e.g., embedded systems). The only requirement is that the system produces *event logs*, thus recording (parts of) the actual behavior. Information systems such as classical workflow management systems (e.g. Staffware) case handling systems (e.g. FLOWer), PDM systems (e.g. Windchill), middleware (e.g., IBM's WebSphere), hospital information systems (e.g., Chipsoft) record detailed information about the activities that have been executed. Other systems recording events are medical systems (e.g., X-ray machines), production systems (e.g., wafer steppers), copiers, sensor networks, etc. An example is the "CUSTOMerCARE Remote Services Network" of Philips Medical Systems (PMS). This is a worldwide internet-based private network that links PMS equipment to remote service centers. An event that occurs within an X-ray machine (e.g., moving the table, setting the deflector, etc.) is recorded and analyzed. The logging capabilities of the machines of PMS illustrate that event logs are widely available.

The goal of process mining is to extract information (e.g., process models) from these logs, i.e., process mining describes a family of *a-posteriori* analysis techniques exploiting the information recorded in the *event logs*.

Simple algorithms such as the α -algorithm [11] are linear in the size of the event log. However, such algorithms do not perform well on real-life data and their simplicity is misleading for two reasons: (1) more advanced process mining algorithms are needed that require lots of computing power and parameter tuning and (2) the "process of process mining" consists of additional pre- and post-processing steps (filtering, cleaning, merging, conformance checking, etc.). It should be noted that event logs may be huge, e.g., there may be thousands of different cases and there may be thousands of events per case. Logs such as the ones produced by the machines of PMS illustrate the computational challenges. Moreover, some process mining techniques require lots of computing power. Consider for example the genetic process mining algorithms described in [24]. All of the more advanced algorithms have lots of parameters that need to be set. Typically, the algorithms are run with different parameter settings to achieve acceptable results. Hence, different process mining experiments are run iteratively or in parallel. Besides running the core process mining algorithms several pre- and post-processing steps need to be conducted.

The main goal of grid computing is to offer wide distributed computing and storage facilities for complex applications. From the observations just made, process mining process can require challenging computational executions, and also has to deal with a large amount of data. Therefore, *process mining is an interesting application domain for grid computing*. On the one hand, there are clear computational challenges that can be addressed through grid computing. On the other hand, the "process of process mining" can be seen as a workflow

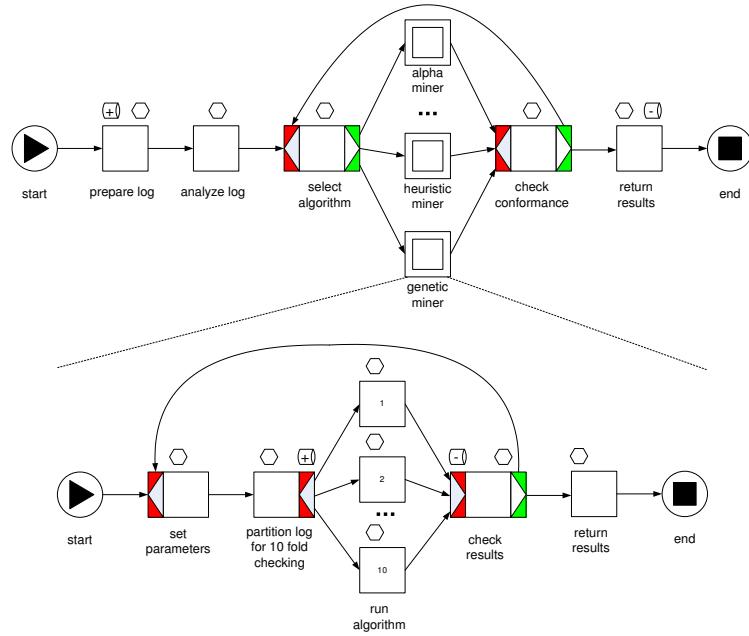


Fig. 1. The process mining workflow expressed in terms of YAWL [6]

consisting of activities ranging from data preparation and filtering to discovery and conformance checking.

To illustrate the application of grid computing to process mining, we use a rather simple and abstract example as shown in Figure 1. It is kept simple so that it is understandable by non-process mining experts and to allow for understandable CPN models later in this paper.

Figure 1 describes a typical process mining scenario in terms of the workflow language YAWL [6]. YAWL extends Petri nets with notations useful for representing workflows. The workflow at the top of Figure 1 shows that a high-level process mining job starts with the preparation of the log (task *prepare log*). It includes the scanning of the log for inconsistencies (e.g., descending timestamps, missing event types, etc.) and the addition of dummy start and end events if needed. Then the log is analyzed (task *analyze log*) and several characteristics are collected, e.g., size, completeness, number of event types, distribution of activities, etc. Based on this task *select algorithm* chooses a particular algorithm that is expected to perform well given the characteristics of the log. If characteristics indicate that the log is highly structured and has no noise, the α -algorithm may be selected. If it contains some noise and is large, the heuristic miner [22] may be selected. The genetic miner may be selected if the structure of the model is complicated, there is noise, and the log is not too large. After running one of the process mining algorithms, the quality of the result is checked (task *check conformance*). If the quality is acceptable or there are no more alternatives,

the results are returned. Otherwise, another algorithm is selected and the process is repeated. Each of the process mining algorithms corresponds to a YAWL subprocess. In Figure 1 only the subprocess *genetic miner* is described. The genetic miner starts by setting the parameters. Genetic algorithms typically have many different parameters that one can experiment with. The genetic miner has parameters such as population size, number of generations, seed, elitism, mutation rate, fitness function, crossover type, etc. Although not shown in Figure 1 different instances of the same algorithm could run in parallel with different parameters to improve response time. After task *set parameters*, the log is split and replicated for 10-fold checking. k -fold cross validation divides the data set into k subsets. Each time, one of the k subsets is used as the test set and the other $k - 1$ subsets are put together to form a training set. Then the average error across all k trials is computed. In this workflow the cases in the logs are split over 10 sets and each of the 10 parallel branches in Figure 1 takes 9 of these 10 sets to construct a process model based on the genetic algorithm. After applying the algorithm the result is evaluated using the remaining test set. Task *check results* collects these results and decides whether a new experiment is needed, i.e., the subprocess returns to task *set parameters* or ends with task *return results*.

Figure 1 also shows some annotations describing the use of resources. For this simplified example, we assume that there are only two types of required properties for task execution: CPU and disk space. Disk space is denoted by the small tube and CPU power is denoted by a small hexagon. Disk space is typically allocated for multiple subsequent tasks while CPUs are typically released after each task. Task *prepare log* claims space for storing the entire log and the overall results. This space is only returned at the end of the workflow. Since the genetic algorithm is a more complex process, the algorithm has its own private data space.

In the remainder of this paper, we will use process mining and in particular the example shown in Figure 1 to illustrate our approach.

3 Grid Workflows

This section introduces the basic grid concepts relevant for the remainder of this paper. As indicated, we will model grids in terms of CPNs and emphasize the workflow aspect of grid computing.

The standard grid architecture [16] is composed of several layers: (1) the *infrastructure layer* composed of resources (e.g. databases, cluster computers), (2) the *application layer*, where the grid user describes the processes to be submitted to the grid, and (3) the *middleware layer*, which is in charge of finding a resource for the user requirements and other management issues (e.g. monitoring, fault recovery).

Infrastructure layer The grid infrastructure is a widely distributed infrastructure, composed of different resources, linked via the Internet. The resources allow for the execution of different tasks. Examples of typical resources in a grid infrastructure are *computing elements* (e.g. cluster computers) and *storage elements*

(e.g databases) [3]. A computing element is usually described in terms of its computing power and software available, e.g. number of CPU's, installed software packages, main memory size, operating system. A storage element is a resource that allows grid users to store and manage files together with the space assigned to them. The typical characteristics of a storage element are the software used to manage the device, the allocated space, and, an identifier of the data contained. A storage element typically contains multiple *storage areas*.

We define the resource *capacity* as a set of characteristics that we will refer to as *properties*. Examples are the number of CPU's and HDD size. The capacity of a resource can be described as a multiset of properties, e.g., two CPU's and one disk of 1GB.

Because a resource may host applications according to the available capacity, we split the capacity in *free capacity* (i.e., available computing power or storage to be used by a grid job) and *busy capacity* (i.e., capacity that is already allocated/reserved for the performance of certain jobs). We refer to the set of resources composing a grid infrastructure as the *resource pool*. In the model presented in this paper, we assume that the resource pool is fixed and that the resources are reliable, i.e. if an application was allocated on a resource, then the resource will eventually perform it.

Application layer The upper level of a grid architecture is composed of user applications. Such applications define the jobs to be executed using the grid infrastructure. Since jobs may causally depend on one another, the application level needs to specify the “flow of work”. Therefore, we use the term *grid workflow* to refer to the processes specified at the application level. Note that there may be different grid workflows using the same infrastructure and that there may be multiple *instances* of the same grid workflow (referred to as process instances). For each process instance, a partially ordered set of jobs needs to be executed. The grid workflow defines the dependencies between jobs and the properties required per job. In a grid workflow one can find the classical workflow patterns [7] but also patterns focusing on resource allocation, e.g., allocating multiple resources to the same job.

Middleware layer The linking between user jobs and resources is done by a matchmaker (or broker). In this paper we restrict ourselves to middleware working according to a “just-in-time” strategy, i.e., at the moment job instance must be executed, the matchmaker searches for an available resource matching the job, and if it exists the job is allocated to that resource. After the allocation, the free capacity of the resource and the busy capacity are updated according to the job requirements.

In the next section we map the concepts mentioned onto CPNs with two goals in mind: (1) to clarify the basic grid concepts and (2) to show that Petri-net based analysis is useful and feasible in a grid context. Figure 2 illustrates the *grid model* we aim to represent in terms of CPNs. The model is composed of the grid workflows (i.e., application layer) submitted to the grid and a common resource pool, containing all the infrastructure resources with their capacity (i.e.,

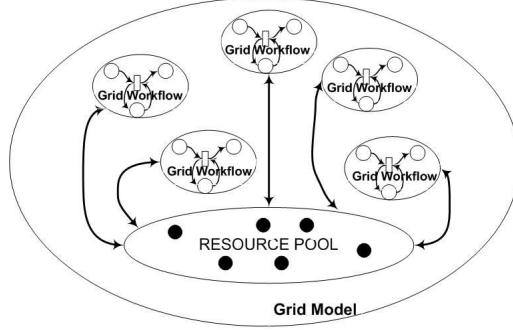


Fig. 2. Grid Model

infrastructure layer). The grid model assumes a very simple middleware layer and will be represented by the allocation/deallocation of the jobs instances only.

4 Modeling Grids in Terms of CPNs

In the previous section, we have presented the main components of a grid model. In this section, we describe how to model a grid using Colored Petri Nets (CPN) [20] and present some basic design patterns [7] that support the modeling of dependencies between grid jobs. We conclude the section by providing a CPN model for the running example presented in Section 2.

4.1 Mapping the grid model onto CPNs

As we discussed in the previous section, a grid model is composed of a set of grid workflows, a pool of resources, and allocation/deallocation mechanisms. In our examples, we typically focus on a single grid workflow, however the same approach can be used to model multiple grid workflows sharing a grid infrastructure.

We model the grid infrastructure in terms of a *resource pool place*. This place contains tokens corresponding to the resources. Each resource has a unique id modeled by the color set *ResID*. The capacity of a resource is expressed in terms of available (*free*) capacity and allocated (*busy*) capacity. Both types of capacity are modeled as a multiset of properties. Recall that a property refers to a single resource characteristic, e.g., a capability like storage space, computing power, bandwidth, etc. The color set *Prop* is used to model properties (e.g. CPU, storage area), and color set *Props* represents a multiset of properties. Note that *Props* is defined as a list of *Prop* elements to model multisets. The tokens of the *resource pool* place are of the color set *Res*. This color set incorporates the resource id, the available capacity, and allocated capacity.

The grid workflows are modeled as an extension of the classical workflow nets [4] and there are also clear relations with the so-called colored workflow

```

▼ Multisets
  ▼ fun m_size(m) = length(m);
  ▼ fun m_elt(e,[]) = false | m_elt(e1,e2::m) = (e1=e2) orelse m_elt(e1,m);
  ▼ fun m_ins(e,m) = e::m;
  ▼ fun m_del(e1,e2::m) = if e1=e2 then m else e2::m_m_del(e1,m) | m_del(e1,[]) = [];
  ▼ fun m_leq([],m2) = true | m_leq(e::m1,m2) = m_elt(e,m2) andalso m_leq(m1,m_m_del(e,m2));
  ▼ fun m_add(m1,m2) = m1^m2;
  ▼ fun m_min(m1,[]) = m1 | m_min(m1,e::m2) = m_min(m1,m_m_min(e,m1),m2);

▼ griddefs
  ▼ colset PInst=int;
  ▼ colset ResID = string;
  ▼ colset Prop = string;
  ▼ colset Props = list Prop;
  ▼ colset Res = product ResID * Props * Props;
  ▼ colset Job = product PInst*ResID;
  ▼ fun en((rid,free,busy),req) = m_leq(req,free);
  ▼ fun take((rid,free,busy),req) = (rid,m_min(free,req),m_add(busy,req));
  ▼ fun return((rid,free,busy),req) = (rid,m_add(free,req),m_min(busy,req));

```

Fig. 3. Color sets for a grid workflow

nets [8]. Like in a workflow net there is a single input *start* place and a single output *end* place, and every node of the grid workflow is on a path from the *start* place to the *end* place. However, we distinguish between two types of places: *job places* and *control places*. Job places correspond to the execution of jobs while using resources from the grid and control places are merely added for the routing of process instances. Job places are mirrored by *requirement* places indicating the resource requirements in terms of a multiset of properties. Another difference with classical workflow nets is that transitions do not represent tasks but correspond to the allocation or deallocation of resources, i.e., we are forced to model the workflow at a finer level of granularity. Initially, all job places and control places are empty and only the requirement places contain tokens. Moreover, for each process instance a token is added to the *start* place.

Process instances are referred to using the color set *PInst*. All control places, including the *start* and *end* place, are of type *PInst*. Job places are of type *Job*. This color set is defined as the product of a process instance id (color set *PInst*) and a resource id (color set *ResID*).¹ Each requirement place contains one token of type *Props*, i.e., the token holds a multiset of properties denoting the resource requirements of the corresponding job place.

In the CPN model we assume a very simple middleware layer, therefore the binding between the grid infrastructure and the grid workflows can be realized through *allocation* and *deallocation* transitions. When a job is created, i.e., a token is put on a job place, an allocation transition fires. If a job completes, i.e., a token is removed from a job place, a deallocation transition fires.

Figure 3 presents all the basic color sets defined for the grid model. In the definitions, we define also the basic operations for multisets. These operations will be used for modeling with allocating and deallocation capacity.

Figure 4 shows a very simple example of a grid model which uses the color sets mentioned before. There is one start and one end place and these are the only two control places. There is just one job place *j* of type *Job*. The corresponding requirements place is named *r* and is of type *Props*. The resource

¹ Note that this color set assumes that a job cannot use multiple resources. Later we will relax this requirement.

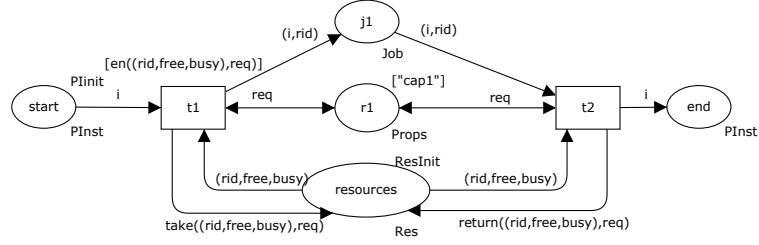


Fig. 4. A simple job example

pool is modeled by the place *resources*. An allocation transition *t1* precedes the job place in Figure 4. The guard of this transition is given by the function $en((rid,free,busy),req)$. The transition is enabled if there exists at least one resource (*rid*) such that the token in *r* place (*req*) is a subset of the multiset *free* (i.e. free resource capacity).

By the firing of transition *t1*, a token containing the process id (*i*) and the allocated resource id (*rid*) is created for the job place *j*. At the same time, the allocated resource characteristics are retrieved from the resource pool. The token of the resource pool is modified by function $take((rid,free,busy),req)$. The function modifies the capacity occupied by the job as busy capacity.

When the job is finished, the deallocation transition fires (transition *t2* in Figure 4). The function $return((rid,free,busy),req)$ modifies the token of the resource that the job releases, by updating the free capacity of the resource (i.e. the new free capacity is the reunion of the free capacity with the capacity equal with the job requirements).

4.2 Basic patterns

In the previous subsection we modeled a grid model containing just one grid workflow and this grid workflow consisted of only one job. However, it is obvious how these types and naming conventions can be used to represent larger grid models. To illustrate this we define some basic patterns to help a grid user to define his process. These are inspired by the workflow patterns in [7]. However, we also provide a pattern dedicated to multiple resource allocation.

Atomic job In the previous section, we have presented a simple job example (see Figure 4). The quadruple *job* place, *requirement* place, *allocation* transition, and *deallocation* transition is the most simple pattern that we use in our model. All the other patterns are composed of this pattern. Therefore, for simplicity we define a subpage containing this pattern. The user can use this subpage in different locations of his grid workflow, adapting the marking of the *req* place according to the job description. Note that for each job type a different subpage can be defined that is reused multiple times.

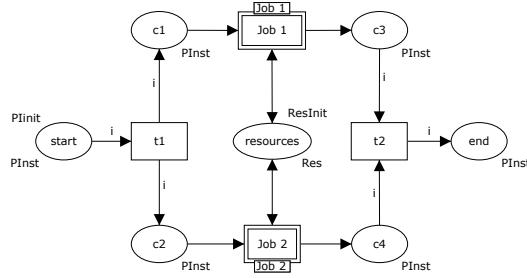


Fig. 5. Parallel pattern

Parallel pattern A common pattern in grid computing is the execution of different jobs in parallel. Figure 5 presents the parallel pattern. Since the matchmaker assumes all the jobs to be independent, two tokens are created so that the matchmaker allocates each job when it finds a suitable resource for this job. The jobs can in principle be different, therefore they are mapped to two different *Job* sub-pages. The process instance will wait till both jobs finish, and then the transition *t2* fires, which terminates the execution of the pattern.

Multiple resource allocation A typical scenario in grid computing is that multiple resources are needed for the execution of a job. For example, a job requires access to a storage area, that contains some data and, after some computation, the result is written back to the same storage area. The storage area has to be reserved till the computation is finished. Figure 6 illustrates one of the possible patterns to realize this. This pattern creates a job to reserve one of the resources, and then looking for a second resource. The first resource remains locked till the second resource finishes the computation.

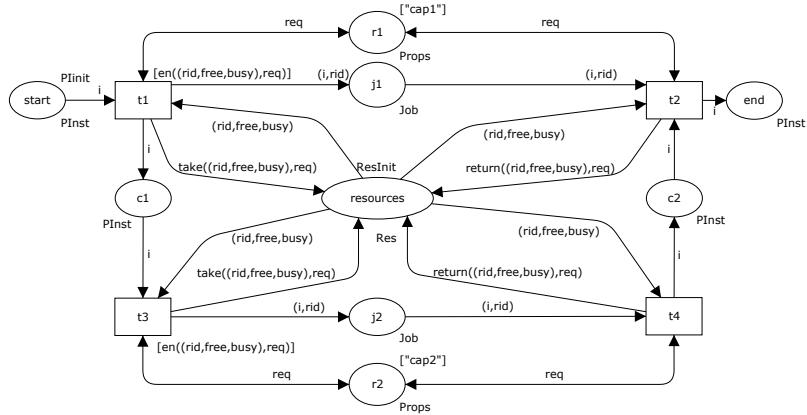


Fig. 6. Multiple resource allocation: Pattern 1

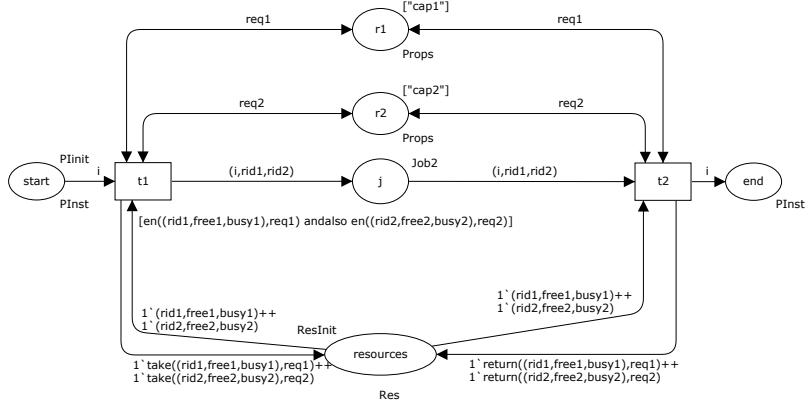


Fig. 7. Multiple resource allocation: Pattern 2

The disadvantage of the pattern illustrated by Figure 6 is that it can lead to deadlocks when a second resource is not available in the resource pool because it is already locked by other jobs.

Figure 7 presents another pattern to realize the allocation of multiple resources. For each of the required resources, the user creates a requirement place: r_1 and r_2 . From the resource pool the allocation transition retrieves two resources, each one corresponding to one of the requirements places. In this case the token of the job place contains one process instance id and two resources ids. The deallocation transition will release both resources. In this second pattern, there may be multiple resources involved in the same job. This makes the model less simple and may lead to modeling errors such as releasing the wrong resource.

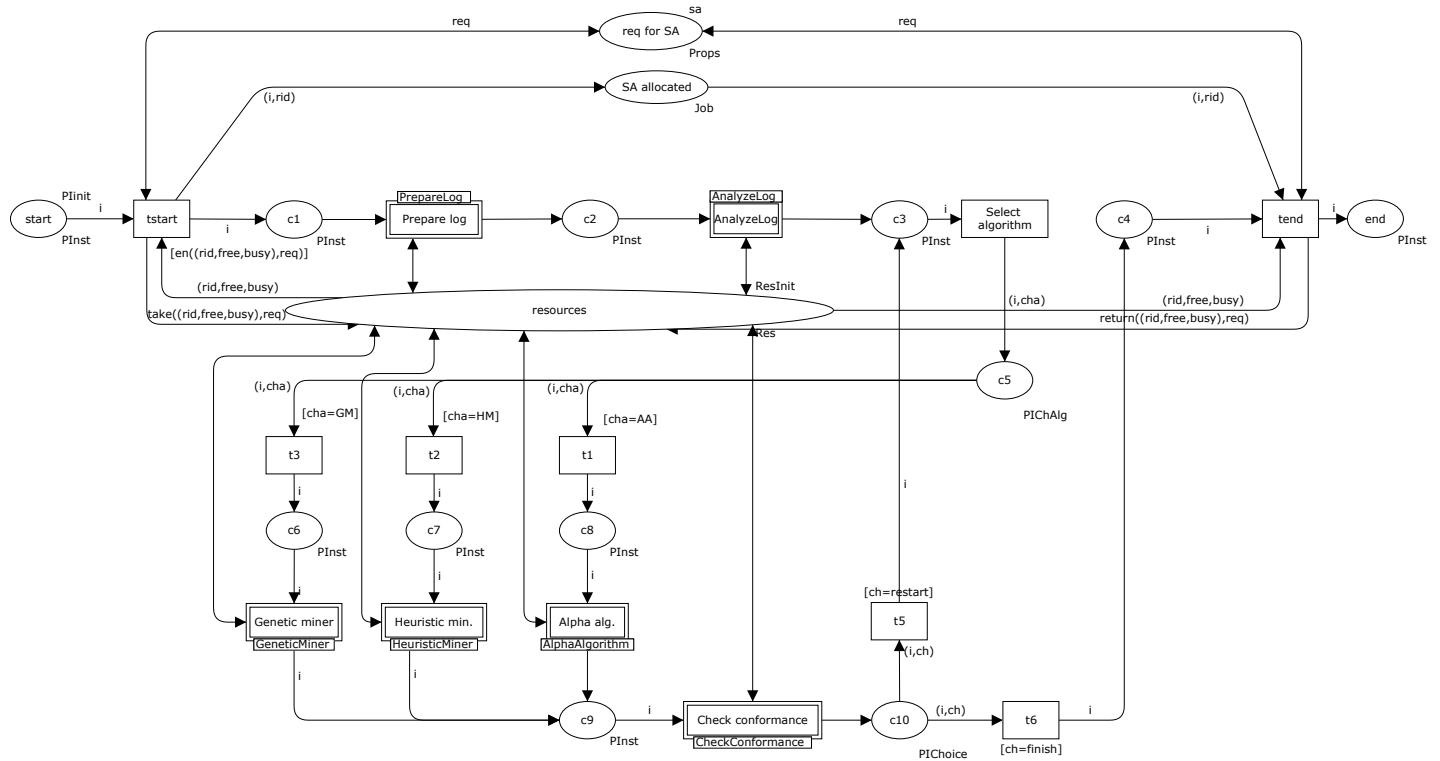
4.3 Process mining example mapped to CPN

Figure 8 shows the CPN model of the process mining example already described in Section 2. In Figure 1 this grid model was introduced using a YAWL diagram. Here we focus on the mapping of Figure 1 into Figure 8 using the patterns defined before.

First, a storage area has to be allocated (places $SA_{allocated}$ and $req\ for\ SA$). The storage area will be used to store and to retrieve the results of all the executed computations. The allocated storage area will be released only when all the other jobs of the same instance have finished. We choose to use the multiple resource pattern shown in Figure 6 to model this behavior.

The sequence of jobs *prepare log* and *analyze log* follows the sequence pattern. After, the *analyze log*, a choice between different mining algorithms should be made. To model this choice we introduce a new color set $ChAlg$ that can take three values: AA , HM , and GM . Each of the three values corresponds to the choice of one of the process mining tools: AA for the plug-in using the α

Fig. 8. CPN model of the process mining workflow



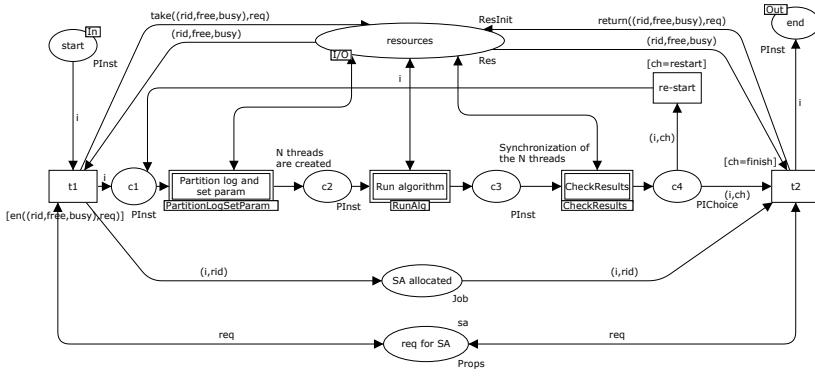


Fig. 9. Genetic miner

algorithm, *HM* for heuristic miner, and *GM* for genetic miner. Note that place *c5* has *PIChAlg* as color set. This color set is the product of *PInst* and *ChAlg*.

After the selected algorithm is executed, job *check conformance* is submitted to the grid. According to the results one of the algorithms (re-)starts or the process ends. To make this choice we use a variable from the color set *Choice*, that can take two values: *restart* to (re-)do one of the algorithms and *finish* to end the process. Note that place *c10* has *PIChoice* as color set. This color set is the product of *PInst* and *Choice*.

For the execution of the α -algorithm and heuristic miner, a resource with free *CPU* capacity is needed. On the other hand, the use of genetic algorithms requires a more complex process (cf. Figure 9). A new storage area is allocated in order to store/retrieve intermediate results of the genetic miner. Then a computing element is used to partition the log and to set the parameters. N different threads are created inside subpage *PartitionLogSetParam* (not shown here), and on each thread the genetic miner algorithm is performed (a grid job requiring a computing element). When all threads have finished, the results of the threads are combined and analyzed (job *check results*). After the job *check results* ends, all the process re-runs or it stops (for this choice an element of color set *Choice* is used).

The example CPN model shows that it is indeed possible to model complex computations using a grid infrastructure. Using CPN Tools we were able to clarify the basic terms used in grid computing. Moreover, the allocation and deallocation of resources may lead to deadlock problems as will be shown in the next section. Because of this, the CPN model is also useful from an analysis point of view.

5 Analysis of Grid Workflows

The analysis of grid workflows may include *validation* (i.e. checks whether the workflow behaves as expected), *verification* (i.e. workflow correctness) and *per-*

formance analysis (i.e. evaluating whether time or cost requirements are fulfilled). In this section we focus on a specific type of analysis: verification. The verification of grid workflow is related to the correctness properties such as absence of deadlocks, livelocks and resource conflicts (cf. [14]). To verify these properties, we use the state space analysis functionality provided by CPN Tools. The analysis is conducted in two steps. First, we perform a *soundness* check. For this purpose, we will extend the soundness property [10] for traditional workflow such that it takes into account the grid workflow characteristics. In the second step, we verify whether there are any resource conflicts between different jobs originating from different grid workflows and their instances. We also show a more efficient deadlock analysis technique which allows us to look at one instance in isolation.

5.1 Soundness check

Since we are interested in soundness, we assume in this subsection that the resource pool has an “infinite” amount of resources (i.e. whenever a job claims resources, available resources exist in the resource pool).

The correctness property that we want to establish is *soundness*. A traditional workflow is sound if it satisfies the following property: if we put a token in the *start* place, there is a possible path to reach the *end* place with just one token from each reachable state, and if the *end* place is reached no “garbage” tokens are left behind (i.e. all the places except the *end* place are unmarked). In our case, because the grid workflow contains also *requirements* places and a *resource* place, we have to extend the soundness property with the following two conditions: (1) the marking of the requirements places remains the same for all the reachable markings and (2) the resource pool place marking in the final state has the same value as in the initial state.

The necessary and sufficient conditions that ensure that a grid workflow is sound using the state space report of CPN Tools are the following: (1) there exists only one dead marking, this marking should also be a home marking (i.e. a marking reachable from all other markings) and there are no other home markings, moreover this dead marking is indeed the desired final marking (i.e. the marking meets the soundness conditions) and (2) the lower and upper bounds of a requirement place marking are equal.

For Figure 8, CPN Tools generated a full state space of 91 nodes and 139 arcs in less than one second for an initial state with just one process instance and a resource pool tokens²: `1'("CE", ["CPU", "CPU", "CPU"], [])++ 1'("SE", ["SA", "SA", "SA"], [])`.

The state space report provided information about boundedness properties, home properties and liveness properties as below³:

² Note that we did not add infinitely many resources to the initial marking. However, it is easy to check whether sufficient resources have been added by checking the multi set bounds in the state space report.

³ We present a partial state space report that contains only the necessary information to establish grid workflow soundness.

Boundedness Properties			Best Lower Multi-set Bounds		
<hr/>					
Best Integer Bounds			AlphaAlgorithm'r 1	1'["CPU"]	
Upper	Lower		AnalyzeLog'r 1	1'["CPU"]	
AlphaAlgorithm'r 1	1	1	CheckConformance'r 1		1'["CPU"]
AnalyzeLog'r 1	1	1	CheckResults'r 1	1'["CPU"]	
CheckConformance'r 1	1	1	GeneticMiner'req_for_SA 1		1'["SA"]
CheckResults'r 1	1	1	HeuristicMiner'r 1	1'["CPU"]	
GeneticMiner'req_for_SA 1	1	1	PM'req_for_SA 1	1'["SA"]	
HeuristicMiner'r 1	1	1	PartitionLogSetParam'r 1		1'["CPU"]
PM'req_for_SA 1	1	1	PrepareLog'r 1	1'["CPU"]	
PartitionLogSetParam'r 1	1	1	RunAlg'r 1	1'["CPU"]	
PrepareLog'r 1	1	1			
RunAlg'r 1	1	1			
Best Upper Multi-set Bounds			Home Properties		
AlphaAlgorithm'r 1	1'["CPU"]		<hr/>		
AnalyzeLog'r 1	1'["CPU"]		Home Markings		
CheckConformance'r 1			28		
1'["CPU"]			<hr/>		
CheckResults'r 1	1'["CPU"]		Liveness Properties		
GeneticMiner'req_for_SA 1			<hr/>		
1'["SA"]			Dead Markings		
HeuristicMiner'r 1	1'["CPU"]		28		
PM'req_for_SA 1	1'["SA"]		<hr/>		
PartitionLogSetParam'r 1			Dead Transition Instances		
1'["CPU"]			None		
PrepareLog'r 1	1'["CPU"]		<hr/>		
RunAlg'r 1	1'["CPU"]		Live Transition Instances		
			None		

From the boundedness properties, we observe that each of the requirement places (i.e. r places) has a lower bound equal to the upper bound. Marking 28 is a home marking and a dead marking and it corresponds to the desired final marking. Therefore, we conclude that the process mining grid workflow is sound.

5.2 Resource verification

The main problem in resource sharing is the potential of deadlocks when multiple instances run in parallel and compete for the same resources step-by-step. Therefore, we now look at the analysis of a grid model with *multiple instances* and a *limited set of resources*. Using CPNTools we want to verify whether soundness is jeopardized. We do this by looking for deadlocks.

Let us consider the process mining grid workflow again. For two process instances and a resource pool containing the following tokens:

```
1'("CE1", ["CPU", "CPU", "CPU"], [])++1'("SE1", ["SA"], [])++1'("SE2", ["SA"], [])
```

CPN Tools generates a full state space with 1140 nodes and 2176 arcs in less than 2 seconds. The state space report contains three dead markings and no home markings.

Home Properties	Liveness Properties
Home Markings	Dead Markings
None	[420, 456, 952]

Dead marking 952 is the desired final marking (i.e. the grid model proper terminates), and the other two (420,456) refer to instances where both of process instances deploy the genetic miner concurrently. The deadlocks occur because each of the instances needs an additional storage area, but the resource pool depleted all available storage areas. Even if we increase the number of available resources, the system will deadlock when the number of process instances running in parallel is also increased.

In the following subsection, we propose a method to correct such a grid workflow in order to ensure its proper termination independently from the number/type of available resources.

5.3 Correcting a resource constraint grid workflow

In [26], we studied resource-constraint processes with homogeneous resources and we have shown that a necessary and sufficient condition that ensures proper termination (i.e. absence of deadlocks violating e.g. the soundness property) is that any path resulting in the claim of one or more resources should have a successor path resulting in the release of some resources.

We verify the necessary and sufficient condition for each property type. Therefore, we construct an automaton modeling the behavior of the system just from the point of view of claiming and releasing of resources, abstracting from all the other possible events (i.e. those events are considered as silent steps). Figure 10 presents the automaton for the process mining workflow. In state s_1 , the system needs to reserve a computing element with a free CPU capacity, and in state s_3 a new storage area if the genetic miner algorithm is selected. The two claims are made without being preceded by releases of resources providing the same type of property. Therefore, the workflow does not satisfy the necessary

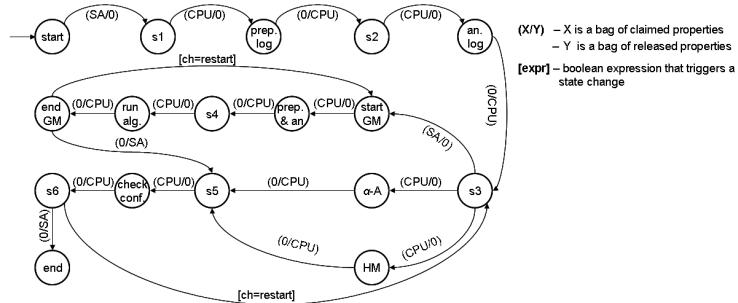


Fig. 10. Process mining automaton

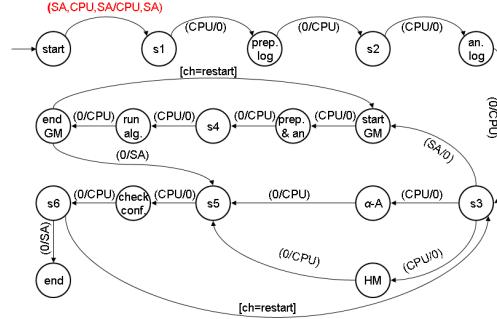


Fig. 11. Modified process mining automaton

and sufficient condition (presented in [26]) neither for *CPU* property, neither for *SA* property.

To avoid deadlocks the model shown in Figure 8 needs to be corrected. The correction of the grid workflow is made by checking if there are enough resources to execute all the jobs composing the workflow (cf. [26]). The resources are just claimed and released, ensuring that the necessary and sufficient condition is satisfied. The algorithm presented in [26] is applied for each property type.

For the running example, the corrected automaton, by joining the results for each property type, is shown in Figure 11. The only modification made is to claim and to release in state *start* two additional resources with properties *SA* and *CPU*. For the corrected automaton, we observe that the necessary and sufficient condition is fulfilled. We map the solution from the automaton to the

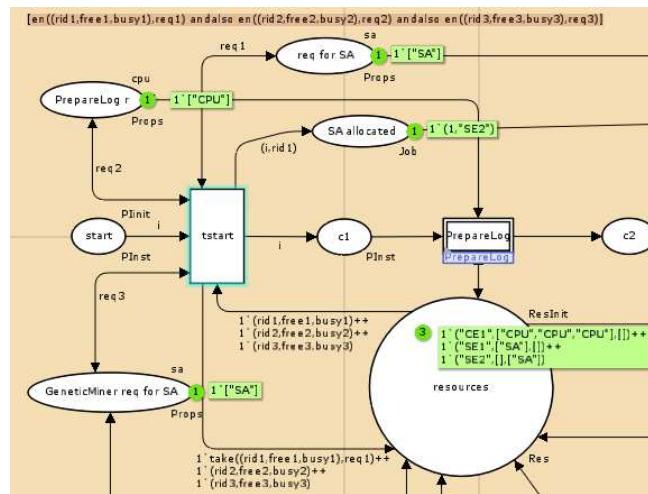


Fig. 12. The changed process mining workflow

CPN model, by changing the guard of the allocating transition of the first storage area as shown in Figure 12. From the *resource pool*, three resources are required (*rid1*, *rid2* and *rid3*). Each of this resources must fulfill one of the *requirements* of the jobs contained by the workflow. Just one resource is kept (*rid1*, as in the original model), and the other two are released without any modification.

For the modified model, CPN Tools generates a full state space with 492 nodes and 664 arcs in less than one second. Just one marking is reported as both a home marking and dead marking and it corresponds to the desired final marking. Hence the grid model is sound.

6 Related Work

The idea of using grid workflows to model and enact complex scientific processes and distributed resources such as computing elements and data has emerged in the grid community in recent years. Grid workflow systems [1, 15, 25] have been developed, but most of them support only directed acyclic graphs (DAGs) as a modeling language. The disadvantage of using a DAG is that it does not support loop patterns and choice patterns. Therefore, in the last years, Petri Nets were introduced [12, 19] as a more powerful and suitable language to model grid workflows. However, most of the work is focused on the practical realization of grids, and less attention has been paid to providing a conceptual framework to model grid workflows.

Many researchers have applied Petri nets to workflow management [4, 27]. Note that these papers do not necessarily focus on grid workflows. Moreover, these papers typically focus on a single aspect, e.g. control flow verification, while ignoring the interplay between resources and workflows. However, there have been some papers using CPNs to address other aspects of workflows, e.g. [8]. To address the deadlock problem in grids we propose to use a variant of the technique proposed in [26].

The running example comes from the process mining domain. In [13], the authors proposed a process mining workflow that can call process mining algorithms using a process engine. More details related to process mining concepts and algorithms can be found in [11, 22, 24].

7 Conclusion

In this paper, we propose a conceptual framework to use CPN for modeling and verifying of grid workflows. Grid concepts such as “job”, “grid workflow” and “resource” have been explained in order to map them on CPNs.

We proposed some basic patterns in order to model common grid behavior such as parallel execution and multiple resources allocation. Combining these patterns, a complex grid workflow was build up.

Moreover, we showed that CPN Tools can be used to conduct soundness and resource verification to find potential deadlocks. The state space analysis from

CPN Tools is able to discover deadlocks when multiple instances run in parallel while incrementally claiming similar resources. Based on our previous work [26], we have corrected the model such that all instances terminate properly.

This paper is mainly conceptual. However, it is good to mention that we are in the process of connecting YAWL, Globus, and ProM. YAWL [6] is used as a workflow engine taking care of the orchestration. Globus [17] is an open source software toolkit used for building Grid systems and applications but is not process-aware. ProM [5] is used to execute the actual process mining activities. ProM provides a pluggable architecture with dozens of process mining algorithms and a lot of functionalities related to filtering, conversion, conformance checking, etc. Currently, ProM [5] aims at interactive mining. However, as shown in [13], ProM can be adapted such that its functionality can be invoked by a workflow engine.

References

1. DAGMan (Directed Acyclic Graph manager): Condor meta-scheduler. <http://www.cs.wisc.edu/condor/dagman/>.
2. DataMiningGrid Project. <http://www.datamininggrid.org/>.
3. GLUE Schema V1.3. <http://forge.gridforum.org/sf/go/doc14185?nav=1>.
4. 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.
5. W.M.P. van der Aalst, B.F. van Dongen, C. Günther, R. Mans, A.K. Alves de Medeiros, A. Rozinat, V. Rubin, M. Song, H. Verbeek, and A. Weijters. ProM 4.0: Comprehensive Support for Real Process Analysis. In J. Kleijn and A. Yakovlev, editors, *ICATPN 2007*, volume 4546 of *LNCS*, pages 484–494. Springer-Verlag, Berlin, 2007.
6. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.
7. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
8. W.M.P. van der Aalst, J. Jørgensen, and K. Lassen. Let’s Go All the Way: From Requirements via Colored Workflow Nets to a BPEL Implementation of a New Bank System Paper. In R. Meersman and Z. T. et al., editors, *OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005*, volume 3760 of *LNCS*, pages 22–39. Springer-Verlag, Berlin, 2005.
9. W.M.P. van der Aalst, B. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003.
10. W.M.P. van der Aalst and K. M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.
11. W.M.P. van der Aalst, A. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.
12. M. Alt, S. Gorlatch, A. Hoheisel, and H.-W. Pohl. A Grid Workflow Language using High-Level Petri Nets. Technical Report TR-0032, Institute on Grid Information and Monitoring Services, CoreGRID - Network of Excellence, March 2006.

13. L. Cabac and N. Knaak. Process mining in Petri net-based agent-oriented software development. In D. Moldt, F. Kordon, K. van Hee, J.-M. Colom, and R. Bastide, editors, *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'07)*, pages 7–21, Siedlce, Poland, June 2007. Akademia Podlaska.
14. J. Chen and Y. Yang. Key research issues in grid workflow verification and validation. In *ACSW Frontiers '06: Proceedings of the 2006 Australasian workshops on Grid computing and e-research*, pages 97–104, Darlinghurst, Australia, 2006. Australian Computer Society, Inc.
15. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflows onto the Grid. *Grid Computing: LNCS*, 3165:11–20, 2004.
16. I. Foster. The anatomy of the grid: Enabling scalable virtual organizations. *Proceedings. First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 6–7, 2001.
17. I. Foster. Globus toolkit version 4: Software for service-oriented systems. In H. Jin, D. A. Reed, and W. Jiang, editors, *NPC*, volume 3779 of *LNCS*, pages 2–13. Springer, 2005.
18. G. C. Fox and D. Gannon. Workflow in Grid Systems. *Concurrency and Computation: Practice and Experience*, 18(10):1009–1019, 2006.
19. A. Hoheisel. User tools and languages for graph-based Grid workflows. *Concurrency and Computation: Practice and Experience*, 18(10):1101–1113, 2006.
20. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1*. EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1997.
21. C. Kesselman and I. Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.
22. L. Maruster, A. Weijters, W.M.P. van der Aalst, and A. van den Bosch. A Rule-Based Approach for Process Discovery: Dealing with Noise and Imbalance in Process Logs. *Data Mining and Knowledge Discovery*, 13(1):67–87, 2006.
23. A. S. McGough, W. Lee, and J. Darlington. Workflow deployment in ICENI II. In *International Conference on Computational Science (3)*, pages 964–971, 2006.
24. A. Medeiros, A. Weijters, and W.M.P. van der Aalst. Genetic Process Mining: A Basic Approach and its Challenges. In C. Bussler et al., editor, *BPM 2005 Workshops (Workshop on Business Process Intelligence)*, volume 3812 of *LNCS*, pages 203–215. Springer-Verlag, Berlin, 2006.
25. T. M. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, R. M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.
26. K. M. van Hee, A. Serebrenik, N. Sidorova, M. Voorhoeve, and J. van der Wal. Scheduling-free resource management. *Data and Knowledge Engineering*, 61(1):59–75, 2007.
27. Y. Wang, C. Lin, Y. Yang, and Y. Qu. Grid service workflow models and their equivalent simplification methods. In *GCC Workshops*, pages 302–307. IEEE Computer Society, 2006.

Overcoming Failures in Composite Web Services by Analysing Colored Petri Nets

Karolina Zurowska, Ralph Deters

Department of Computer Science
University of Saskatchewan
Saskatoon, CANADA
`zurowska.kar@usask.ca, deters@cs.usask.ca`

Abstract

Web Services (WS) is a middleware approach aimed at improving interoperability and flexible aggregation of new and existing software components. Unlike other middleware approaches (e.g. CORBA, RPC) web services are based on already established standards (e.g. HTTP, SOAP and XML), which in turn have enabled it to become a widely accepted technology for system integration. One of the most important elements of WS is the simplicity with which existing WS can be aggregated into new composite services. But the flexibility, offered by the WS, also poses new problems. One of them is how to ensure that composite web services deal efficiently with faults and failures of their underlying services and components.

This paper focuses on the issue of faults and failures in composite web services (CWS). After a review of existing approaches the paper presents a novel model-driven approach that is based on the use of colored Petri Nets. It shows how they are used to model CWS and interactions with other web services, and how the model is analyzed to make CWS more robust.

1 Introduction

A web service (WS) is an interface that exposes software functionality over a network in a declarative manner, and thus enables clients dynamic binding [21].

By using XML to describe operations and exchanged messages, and well established standards like HTTP and SOAP, web services support interoperability and aggregation of software components and existing WS. The first feature enables service providers and service consumers to overcome organizational and platform borders. And due to the ability to aggregate existing WS into new composite web services (CWS), it becomes possible to reuse existing services and then develop layers of services. The layers of services can support modeling complex business processes, thus allowing the realization of applications that spread across many providers.

WS expose existing software that may contain faults [7], which can lead to failures. Consequently every WS will inherit the faults and failures of its underlying components and may also introduce additional ones due to its implementation. It is also important to note that due to the networked nature of web services they can also encounter failures that are characteristic to distributed systems. These failures are mostly consequences of servers unavailability or problems with network connections. To avoid propagation of faults and failures, composite web service must handle them locally.

Propositions to overcome faults and failures in WS and CWS exist in the standards specifications (WSDL [34] and BPEL [26]) and in the literature. The standards define special types of messages that

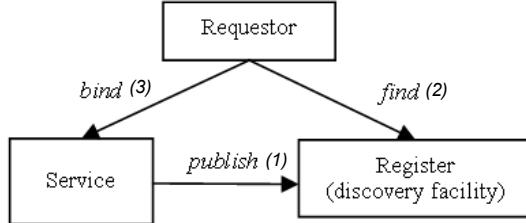


Figure 1: The concept of SOA [36].

indicate exceptional situations, so during the implementation of a CWS, special procedures (compensation or fault handlers) to handle them should be designed. In the literature [3, 4, 9, 10, 13, 14, 15, 22, 25] there are works that address the problem of how to implement replication techniques in web services, thus how to make web services more reliable. The number of methods dedicated to the specific challenges of CWS is however rather small and often based on assumptions (e.g. reduced autonomy) that are not consistent with the goals of composite web services.

The work presented in this paper aims to preserve the autonomy of web services used in CWS, which means that only the model of the CWS and the results obtained during its executions is taken into account. The presented work is based on the concept of analyzing the workflow of the CWS in order to determine alternatives, which enable reliable execution even if used components encounter failures. Searching for those alternatives is performed before the execution of the CWS, and the choice of an appropriate alternative to execute is made before each subsequent execution of the CWS.

In this paper colored Petri Nets are used, which offer a modeling language for design, specification and the simulation of systems [23]. The colored Petri Nets have a graphical representation as well as formal mathematical foundations, and they offer many formal verification methods [23]. One of them, namely an analysis of an occurrence graph [24], is used in this work. It is the basis for inferring the reachability of a success state of the CWS from states that represent failures. This approach doesn't require knowledge or control over interacting WS, thus it ensures that their autonomy is not limited in CWS.

The remainder of the paper is organized as follows. Section 2 presents the definitions of web services, which is followed by a review of solutions for the fault and failure management in web services. Section 3 describes the proposed solution, and section 4 and 5 present modeling and analysis details. The execution is described in section 6. The paper ends with conclusions and an outlook on future work.

2 Background and related work

In this section we present the definitions of web services, as well as ways in which we can compose them. Then we define faults and failures and we review the approaches proposed in the literature to deal with them in web services.

2.1 Atomic and composite WS

WS are currently the most popular implementation of the Service Oriented Architecture (SOA) design and integration approach. The basic principle that underlies SOA is presented in Figure 1. According to this an abstract description of a service, understood as software functionality, is published in a registry (discovery facility). There it can be found by a requestor that wants to use it. The requestor, after finding the description, binds to the service, so it obtains enough information to connect over a network to it [36].

The broad idea of the Service Oriented Architecture is adapted by the WS. Therefore the notion of WS refers to systems that are built from several networked modules, however it also refers to a set of standards, which supports an implementation of such applications [38]. In the first approach a web service can be described as an interface that collects operations accessible over the network, and this access is possible by sending standard XML messages [21]. In the second approach the web service is treated as the WSDL (Web Service Description Language [34]) description of a group of operations, which are invoked over a network using SOAP (Simple Object Access Protocol [33]) messages. These operations can be published with UDDI (Universal Description, Discovery and Integration [28]) in a register.

The main advantage of the web services is their interoperability, which means that they allow communication and cooperation between software components, which are implemented in different programming languages or deployed on different platforms. The web services technology achieves this by using the already mentioned set of standards as well as by relying on XML-based artifacts for describing, publishing and invoking activities. Choosing XML as the underlying language is an important element of ensuring the interoperability, because XML is machine and platform independent.

The next important attribute of web services is their compositability (aggregation). Web services can be aggregated (orchestrated or choreographed) to work together, in order to provide more complex functionality. In most of the cases the goal of such composition is to model a business process, like supply-production chains or planning services. There are two aspects in which a composition of web services can represent a business process: orchestration and choreography [29]. Orchestration is an executable business process that interacts with other web services, and is controlled by one party. Choreography is more collaborative, and it allows involved parties to define their role in interaction, so it tracks sequences of exchanged messages. For both types of modeling the BPEL (Business Process Execution Language) specification, which supports both abstract and executable business processes [26], can be used. Other specifications that serve the same purpose are Business Process Management Language (BPML) together with Web Services Choreography Interface (WSCF) [29]. There are thus different ways that web services can be aggregated and can build recursively other web services.

2.2 Faults and failures

A software failure is a result returned by a program that is not expected and deviates from specified requirements [27]. A failure occurs, when an application is executed in particular conditions, and is caused by a fault, which is understood as a defect in an implementation [27]. To deal with faults, there are two general strategies: fault prevention and fault tolerance [2]. In the first strategy efforts are made to avoid or to remove faults existing in an implementation by testing and analyzing a code. It is based on the assumption that it is possible to predict all use cases of a program and all conditions of its execution. Such assumption is however invalid if a software system consists of a larger number of components or modules. Then the way to deal with faults is fault tolerance, which assumes that faults are present in software. This inherent presence of faults in applications may result in failures, so the responsibility of fault tolerant systems is to provide means to cope with encountered failures [2].

If components that are able to perform computations to achieve a goal of a software system are spread over a network, the system is distributed. These components may be called processes [19] or servers providing services [12]. The services are simply collections of operations, which are executed after receiving defined inputs. There are many methods to specify the processes or services, however they must at least identify a set of inputs and corresponding outputs. If after receiving an input, a server's output or state differs from a service's specification, a failure occurs. We can distinguish four main categories of failures in distributed systems [12]:

- *omission failure* - if a server omits a response,
- *timing failure* - if a response comes outside of a specified time interval (if a response is received after this interval then this failure is called a *performance failure*),

- *response failure* - if a response comes on time, but its value is wrong or a server's state after the response is incorrect,
- *crash failure* - if a server fails to produce outputs for several subsequent inputs and it must be restarted.

2.3 Overcoming faults and failures in web services

In the following sections we present the solutions to deal with faults and failures in web services. First we show the methods used in standards (like WSDL or BPEL), and then other solutions present in the literature.

2.3.1 Standards

The current set of standards, which provides means for describing, executing and composing web services, to overcome failures proposes fault tolerance that bases on the exception handling concept.

The WSDL standard [34], which specifies a web service's description, allows declaring, for each operation available in the web service, a special fault message with its name and content. If the web service's transport protocol is SOAP [33], then at the execution time, the description of the fault message is transformed to a special type of message, namely a SOAP fault. It contains information about a general type of a fault, its name and details specified in a WSDL description. SOAP fault can be mapped to an exception in a programming language (for example in Java) and handled there [33].

The BPEL language, which is considered the standard language for implementing compositions, offers two mechanisms that are used to deal with failures: compensation and fault handlers [26]. Compensation handlers are used to perform backward error recovery, so they allow defining procedures that can be invoked to undo changes, which have been made prior to a failure's occurrence. Fault handlers are used to forward error recovery, so to successfully overcome failures. Both of the mentioned structures, compensation and fault handlers, are defined for a scope, which is BPEL's unit of processing [26].

2.3.2 Other solutions

Solutions for the problem of dealing with failures in web services that are not composed (atomic WS) are based in most cases more or less on replication techniques. Deron et al. [13] propose to deploy the same web service on many hosts. A description of the web service is then enhanced with the additional element indicating locations of primary and backup servers. Each request to the web service is logged and stored, thus in case of a failure (detected after a request), the next host is chosen to take over a primary server role, and response to the request. Liu et al. [25] give a similar solution, but make the fault tolerance transparent to requests. They split a web service into two layers: one is responsible for delivering its functionality and the other for providing fault tolerance (by replication management, logging services and recovery from logs). The replication mechanism is also used by Dobson [15], but he investigates the possibility of using BPEL language to implement a fault tolerant web service. In his proposition BPEL serves as an additional layer (previously realized by SOAP engines) for choosing atomic web service, according to their accessibility status. When one host with a web service crashes then another is invoked, which is implemented in compensation handler of the web service.

The concepts proposed to manage servers failures presented above are not suitable for composite web services, because they are not considering the hierarchical structure of such web services. This issue is addressed in the following approaches. Dialani et al. [14] designed an architecture that consists of an application layer with a general fault manager and of the service layer that is enhanced with messaging capabilities. Such distinction makes it possible to introduce local and global recovery mechanisms; the first one is responsible for recovery of an individual web service, whereas the second one is used by the entire application. Dialani et al. propose backward error recovery, so in case of a failure, the system goes back to

Authors	Type of WS	Type of failures	Description
BPEL, WSDL [26, 34]	Composite	Declared exceptions	Exception handling, which is based on faulty messages.
Deron et al. [13]	Atomic	Crashes	Replication of web services, with an additional element in a WSDL description for specifying locations of primary and backup servers.
Liu et al. [25]	Atomic	Crashes	Replication of web services that is transparent for clients, and uses two layers of WS.
Dobson [15]	Atomic	Omission failures, crashes	Using BPEL’s compensation handlers to manage replication of WS.
Dialani et al. [14]	Composite	Crashes	Informing cooperating web services about failures, assumes control over atomic web services.
Ardissone et al. [3, 4]	Composite	Exceptions	Reasoning about possible cause of an exception, assumes knowledge about atomic web services.
Issarny et al. [22]	Composite	Exceptions thrown in parallel execution	Concept of Coordinated Atomic (CA) actions adapted to web services (gathering all cooperating web services and perform recovery procedures).
Chafle et al. [10, 9]	Decentralized composition	Exceptions	Inserting handlers into each part of a distributed composition.

Table 1: The summary of solutions for overcoming faults and failures in web services.

the previous valid state, and all services that might be affected are notified. This solution assumes that an application that composes atomic web services can control them, unfortunately in most CWS it is invalid.

The work of Ardissono et al. [3, 4] does not assume any control over composed web services. Their framework reasons about possible causes of an exception and chooses the best fault handler for it. Their main concept is to use local diagnosers with each web service, so in case of an exception the diagnoser produce local hypotheses about its cause. They are verified by the global diagnoser, which has knowledge about the whole composition and can generate global diagnostic hypotheses. Ardissono et al. use a model-based diagnosis, which allows modeling a composition as a set of components, which store exchanged variables. Although in this work the control over web services is not assumed, this framework still requires a substantial amount of knowledge about the behavior of individual web services to build the diagnostic model.

The next two solutions consider slightly different problems and explore specific approaches to the composition of web services. Issarny et al. [22] solve the problem of how to perform forward error recovery if individual web services are invoked in a composition in parallel. They adapt the concept of Coordinated Atomic (CA) actions used in decentralized systems. The CA actions are used to control cooperative concurrency and exception handling, by gathering all interacting threads and synchronizing their initial and final states [30]. In the web services domain this works similarly, and web services are participants of a CA action. In case of a failure there are procedures that deal with global results of concurrently executed services [22]. Chafle et al. [10] propose decentralized mechanism of composing web services in opposition to centralized one defined in BPEL. Although decentralization may improve performance, it makes the fault handling more complex, because all parts of decentralized composition can throw exceptions. To overcome this handlers are inserted into each part and then, according to the type of the part, appropriate data are gathered and sent to related nodes [9].

All solutions presented above are gathered and compared in Table 1. It can be concluded from this

summary that solutions for composite web services are based on the assumption of having control or knowledge about used WS. Such assumption is in conflict with the interoperability paradigm, because it requires additional dependency between components.

3 The algorithm for overcoming failures in CWS

In this paper we present a model-driven approach to overcome failures encountered by CWS during interactions with other web services. We focus on omission failures of one or more components, thus on the case if they are not responding. However, since a missing response from a web service may result in a fault message, we treat such message (if it is not a part of the WS description) as an indication of problems with the web service.

To deal with these problems we use a model of a composite web service and check if it is possible to execute it successfully without one or more external web services. The underlying idea is to model a composite web service, then analyze it and prepare alternatives, one of which is chosen according to states of used web services and then executed. Hence there are two general phases: the analysis and the execution. The first phase requires representing the CWS in order to analyze whether it is possible to "skip" any interactions with external web services. In the second phase a version of CWS must be chosen, the basis for this choice are current states of used external web services.

There are many approaches how to model or represent CWS and with which formalism. Here are some of them:

- state transitions models that are used to capture protocols for conversations between web services [6],
- finite state guarded automata with queues for incoming messages [18],
- process algebras [17],
- Petri Nets [20] and hierarchical colored Petri Nets [37].

All these proposition use models for the purpose of CWS verification and, except for the process algebra approach, transform already existing compositions. In the approach presented here the primary goal is to provide means to indicate alternative execution paths. So the concepts presented above are not very well suited to meet such requirements. Although we use one of the already mentioned formalism, the colored Petri Nets, it is used in a different way.

We chose the colored Petri Nets modeling language [23] to model CWS for three reasons. First, colored Petri Nets can model hierarchies, which are very useful to represent interactions with other web services, because they encapsulate these interactions in separate routines. Second, colored Petri Nets introduce the notion of types of variables, so types of messages can be represented. Finally, colored Petri Nets have a graphical representation, and they are also based on solid mathematical foundations. Besides these this modeling language offers different formal verification methods like occurrence graphs [24], used in our approach to analyze reachability of successful CWS execution.

The general algorithm for overcoming failures is presented in Figure 2. In the analysis phase CWS are modeled with hierarchical colored Petri Nets and their occurrence graphs are constructed. From the graphs, by means of the reachability analysis, it is inferred whether the CWS can be executed if one or more web services are not working. Then versions of CWS are implemented: one which invokes all external WS and, if the successful execution of CWS is possible, without one or more interactions. Hence additional versions just skip one or more invocations to the external WS. The analysis phase is performed once and its complexity is proportional to the complexity of the modeled CWS. The result from this phase is a set of implemented versions of the CWS, each containing also the condition when each version should be use. In the execution phase states of the external web services are checked, and the appropriate for the set of currently working external WS alternative of the CWS is chosen and executed. This step is performed

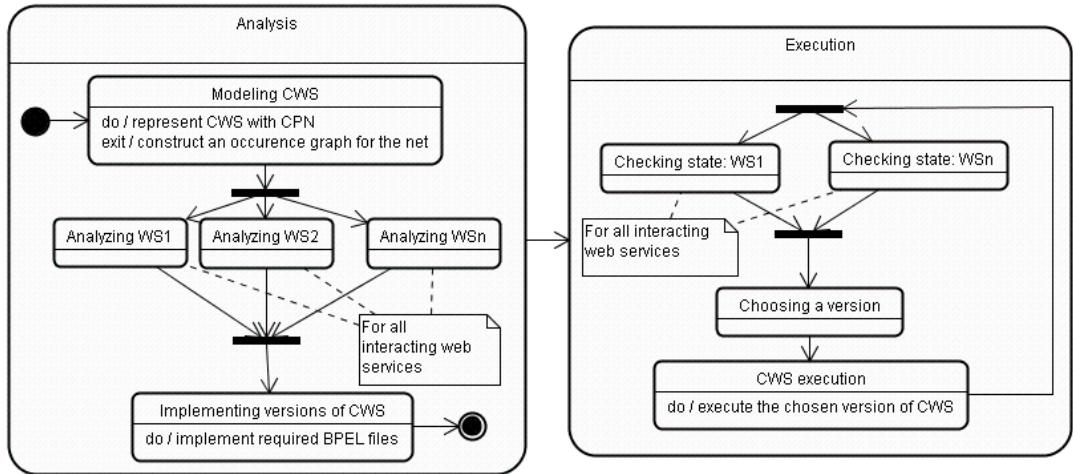


Figure 2: The algorithm for overcoming failures in CWS.

every time there is a new request to the CWS, and its complexity is proportional to the number of used WS (to check states and find the appropriate version) and to the complexity of the implemented CWS.

What is not shown on Figure 2 is what is the source of information about states of external WS and how its is maintained. In our approach we assume that:

- there is an additional WS (for each CWS), which stores the information about states of external WS and it also performs the choice of the version to execute (it acts as a proxy for requests, thus the version selection is not visible to clients of the CWS),
- each execution of the CWS is the source of data about the current state of external WS: if while executing the CWS an external WS does not respond its state is "not working",
- each "not working" state expires after a predefined period of time, and then the external WS is invoked again.

The simplicity of the model for states of external WS used in our approach is sufficient to illustrate the idea, but it can be inaccurate (especially if states are changing frequently). The algorithm in Figure 2 is only a general approach, so it is also possible to use it with other methods, which can more accurately anticipate states of remotely deployed components.

The idea described in this section has several advantages:

- only information from a model of a CWS is used,
- it doesn't require any knowledge or control over used components,
- interactions with not working components, which may be slow, are avoided (however the successful execution of the CWS may be possible even if they are invoked),
- failures are not propagated in the hierarchy of WS,
- overloaded servers with web services do not receive any new requests, thus they might recover faster.

In the following sections we describe how to model CWS with colored Petri Nets and then how to prepare occurrence graphs for them. After that we present an example with more details about the execution phase of CWS.

4 The model of CWS

This section presents how in our approach we model a CWS with the colored Petri Nets modeling language. The modeling of CWS is supported only by CPN Tools [11], so there are not used any specific tools to perform the steps presented below: modeling interaction with other WS and modeling CWS.

4.1 Interactions with other web services

During its execution a CWS can interact with external web services, which means that it sends an input data and gets in turn required information. Thus we can represent each such interaction as a function $output = externalWS(query)$. In colored Petri Nets this function can be modeled as a transition which has an input arc with a color from the query color set and has an output arc with a color from the output color set. But, according to Section 2, to invoke external WS and to get results we need XML messages, and because of the distributed nature of WS we need also to deal for example with omission failures. That is why the simple transition is enough only at the high abstraction level, to make interactions with external WS more realistic we need to take into account also above details.

One of the features of colored Petri Nets is the possibility to model hierarchies of nets [23]. In the context of modeling composite web services the hierarchies allow us to define in a separate routine details of interactions with other web services. So each call to an external web service is represented in a separate net by substituting an appropriate transition and its surrounding arcs. The set of all nets that model CWS is a set of pages [23], whereas in each page there is a non-hierarchical net. Hence there is a main page with a general model of CWS and additional pages for each interaction with an external web service.

In this work we assume that the details about external web services used by CWS are based on information from WSDL descriptions [34]. Each such description contains definitions of operations gathered by a web service, its bindings and protocols [34]. We don't need implementation details (bindings, protocols) in the model of a CWS, thus we limit the WSDL description to a definition of a single operation. We consider then an operation as an external web service. Hence in order to model a web service we need to provide the following data:

- the name of a web service,
- contents of the XML message sent to the external WS (types and names of arguments),
- contents of the response XML message from the external WS (types and names of arguments),
- exceptions for the web service.

All of the above pieces of information are specified in a net that describes an interaction with WS.

To invoke a web service and to get a result we intercept XML messages, which contain names and values of input parameters or responses. To model these XML messages in colored Petri Nets appropriate color sets have to be declared. We do it with a record type, because it enables mapping names and values as defined in a WSDL description of messages. An example of how a WSDL message's description is mapped to a color set is shown in Figure 3. The figure also presents an example of an initial value (a token with a color) for the declared color set. Color sets of the records type can also use other record types as fields, so we are able to map more complex and compound types of messages. The mapping between types in XML Schema [35] and default color sets is rather straightforward, so it is omitted here.

Each correct, synchronous interaction with an external web service returns a response message: either described in a WSDL message with a result or a fault message (defining in WSDL fault messages for operations is optional) [34]. Moreover, because a CWS interacts with remotely deployed components, it is possible that there is no response at all (omission failures described in the previous points) or there is fault message different than described in a WSDL. All these possibilities are taken into account while modeling an interaction. So there are 3 possible types of output from an external web service:

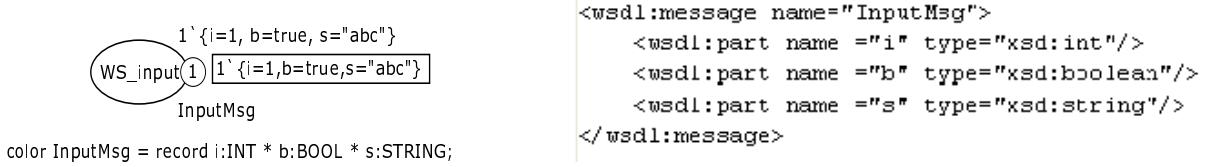


Figure 3: A color set in CPN (with a place and its initial values) and an appropriate WSDL description of a message.

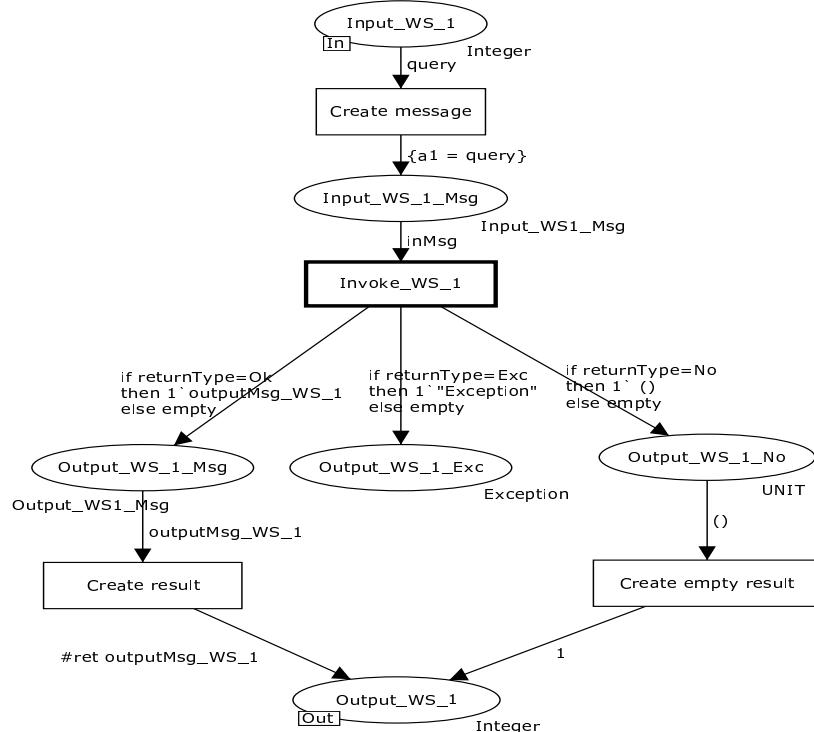


Figure 4: An example net that specifies details of an interaction with an external web service.

- a response message with a result as specified in a WSDL description of a web service,
- an optional fault message also specified in the description,
- no response message or fault message other than specified - it indicates failure of a web service.

For the last two types of output we either define a routine to specify how we deal with them, or we do not define anything if nothing can be done. If in a WSDL description there are no fault messages for the modeled operation there are only 2 types of output : correct value and no response.

Figure 4 presents the Petri Net's page that models an interaction with a web service. Places with inscriptions "In" and "Out", which refer to port nodes, are the part of a CWS that invokes this web service. In the example the CWS uses an integer value for the input and output, thus a conversion from and to format of web service's messages must be made. The input message has only 1 parameter and is

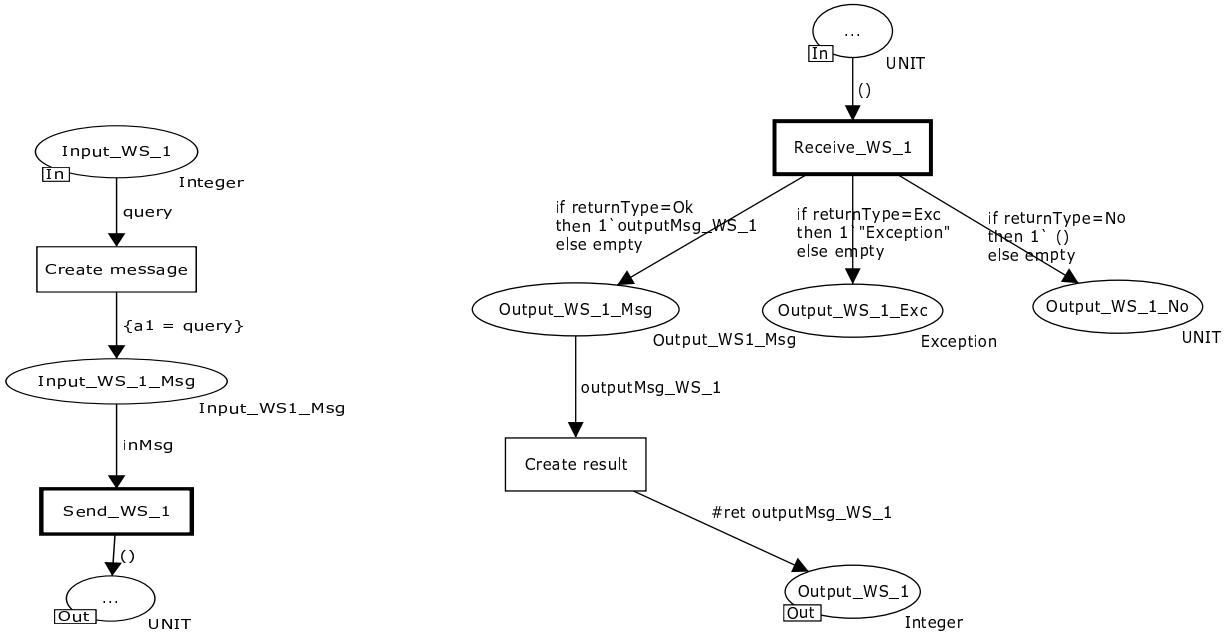


Figure 5: Example nets that represent an asynchronous type of an interaction with WS. On the left there is a send operation, and on the right a receive operation.

modeled by a record type with only one field "a1". The transition "Create message" creates new variable of that type and assigns to it data from the "main" CWS. Then a web service is invoked and three types of possible output are shown, with a variable "returnType" to represent them. After the call the output in format required by the CWS is created: it is the contents of the return message, or value 1 if there is no response (it also represents an unknown fault message). If an exception from the web service's description is a response, nothing is specified afterward, which means that there is no routine to deal with it.

The interaction presented in Figure 4 assumes that the call is synchronous. However it is also possible to invoke a web service's operation in an asynchronous mode: first send a message and then request a response. Moreover it is also possible that only a message is sent, without requesting a response, thus an external WS is only notified. To model this kind of interaction we define 2 additional operations: send and receive. The first one is to create a message and send it to a web service, the second is to get a response and transform its contents to a format required by a CWS. In the latter operation all three possible types of output (correct response, declared fault message if any, no response) must be considered. An example of two nets that represent an asynchronous call are shown in Figure 5. They are similar to the interaction from Figure 4, however there is no routine to deal with the no response type of output.

The approach and examples presented in this section are a proposition of how to model with the colored Petri Nets language interactions of CWS with external web services. We define 3 types of interactions: invoke, send and receive. In the colored Petri nets they are modeled as transitions, thus we have 3 subsets of all transitions to represent those operations: $T_{invokeWS}$, T_{sendWS} and $T_{receiveWS}$. A transition t that represents an invoke operation can be defined:

$$t \in T_{invokeWS} \text{ iff } (t \in T) \wedge (\text{size}(In(t)) = 1) \wedge (\text{size}(Out(t)) \geq 2) \wedge \\ (\exists p \in In(t) : C(p) \approx inMsg) \wedge (\exists p1 \in Out(t) : C(p1) \approx outMsg) \wedge \\ (\exists p2 \in Out(t) : C(p2) = UNIT)$$

where:

- T is a set of all transitions in a net,
- In and Out are functions that map a node to its input and output nodes, respectively,
- $size$ is a size of a set,
- C maps a place into its color set,
- \approx maps WSDL messages (XMLSchema) into record types,
- $inMsg$ and $outMsg$ represent accordingly all input and all output messages (XMLSchema) defined in a WSDL description for a web service.

According to this definition we require that a transition that models an invoke operation has one input place with the color set that is mapped from a WSDL input message, and it has at least two output places: one with the color set that is mapped from a WSDL output message and one with the unit color set (it represents "no response" type of output). The size of the set of output places can be bigger than 2, because we can have fault messages in WSDL description, each of which is modeled as an output place.

A send operation is a transition which:

$$t \in T_{sendWS} \text{ iff } (t \in T) \wedge (size(In(t)) = 1) \wedge (size(Out(t)) = 1) \wedge (\exists p \in In(t) : C(p) \approx inMsg) \wedge (\exists p1 \in Out(t) : C(p1) = UNIT)$$

Here the difference is that we do not have different output types but only the unit color set. Finally a receive operation is a transition which:

$$t \in T_{receiveWS} \text{ iff } (t \in T) \wedge (size(In(t)) = 1) \wedge (size(Out(t)) \geq 2) \wedge (\exists p \in In(t) : C(p) = UNIT) \wedge (\exists p1 \in Out(t) : C(p1) \approx outMsg) \wedge (\exists p2 \in Out(t) : C(p2) = UNIT)$$

The difference between this definition and the invoke is that for input there is the unit color set, so we do not model an input message. We can also define the set of all interactions for CWS:

$$T_{WS} = T_{invokeWS} \cup T_{sendWS} \cup T_{receiveWS}$$

4.2 Modeling CWS

Representing details of interactions with external web services on separate nets is only one part of modeling them. These interactions (without any details) are also a part of a more abstract CWS specification, which is modeled on a main page. Two examples of such pages are presented in Figure 6. They are main pages of CWS which model only interactions presented in Figure 4 and Figure 5, and they call external web services in synchronous and asynchronous mode, accordingly. The transitions that model interactions with external WS (named "Invoke_WS_1", "Send_WS_1" and "Receive_WS_1") are substituted with previously shown nets. The hierarchy of pages makes it possible to model CWS in abstract way on the main page and to deal with details of interactions on additional nets. In turn the general model of CWS, which is on the main page, operates only on variables and colors required by it without transformations to record types (for the net from Figure 6 it is color set "Integer" and variables "query" and "result").

Interactions with external web services are not the only operations that a CWS can contain. We also model operations on data, various control operations (for example if statements, while loops) or others. All of these can be represented in colored Petri Nets. The examples of different possible structures can be found in workflow patterns [31]. It is also possible that a CWS and its colored Petri Nets model can be used to represent complex workflows, the details of this kind of modeling can be found in [32]

Although we can model many different CWS, we need to add some restrictions on the main page of CWS:

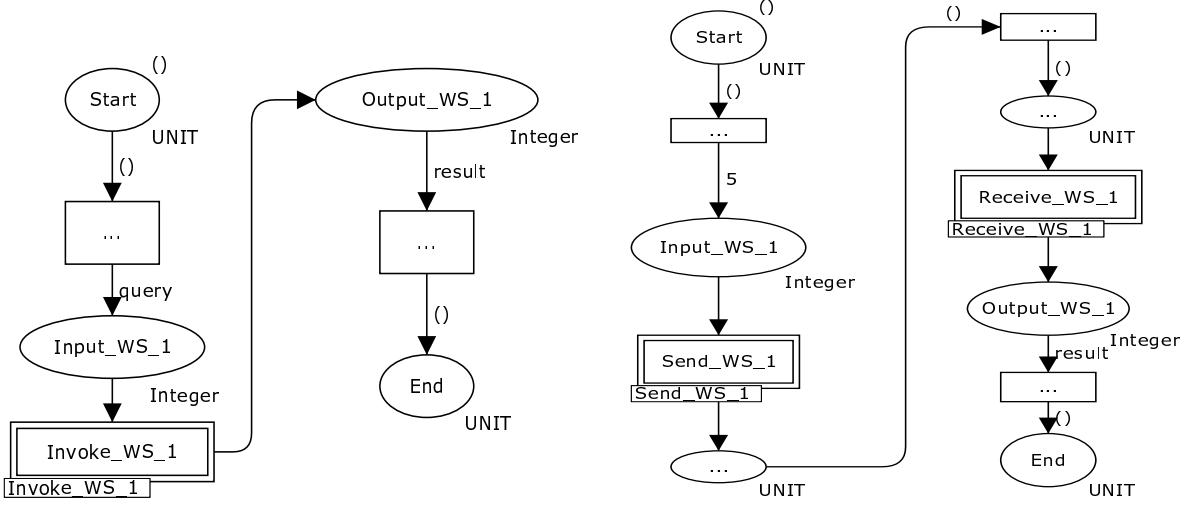


Figure 6: Example nets that present the main pages of CWS. On the left there is CWS with invoke operation and on the right there is CWS with send and receive operations.

- the place that represents the beginning of CWS should not contain input arcs, and the color of this place determines what are input parameters to this CWS,
- the place that represents the end of CWS should not contain output arcs, and the color of this place determines output from this CWS,
- transitions that represent the interactions with external WS have 1 input and 1 output arc (the interactions are functions),
- to allow faster identification of markings, which is used in the analysis of occurrence graphs, we should name the end place with "End" and the transitions for the interactions with the name of used WS.

5 Analysis of CWS

One of the Petri Nets analysis methods are occurrence graphs [24]. In this work we use them to analyze composite web services to find out how failures of required WS may influence the overall CWS execution. An occurrence graph is a graph, which has a node for each reachable marking (a distribution of tokens between places) and an arc for a transition and its binding (called binding elements) [24]. This graph is the basis for checking whether CWS can be executed successfully even if one or more used web services do not respond, which as described in Section 4 is modeled as a "No response" type of output and in the colored Petri Nets as an output place of an interaction with the unit color set. To perform such checking we must infer the reachability of a marking that represents a success of CWS execution from markings that represent different outputs from external web services.

Occurrence graphs are a very useful tool for our purpose, the only problem is that even for simple CWS they may become very large. This happens because we model interactions with external components and we do not know what is the actual result of this interaction. For example for interaction from Figure 4 there can be as many different results as there are elements in the color set "Integer". If this interaction is the part of the CWS modeled in Figure 6 and the Integer color set consists of 100 elements, the occurrence

graph has 404 nodes and 30401 arcs. These values become greater if we have more parameters or more complex results from external web services. At some point the analysis may even be intractable.

To overcome the problem of very big occurrence graphs we use occurrence graphs with equivalence classes (OE-graphs) [24], which can reduce the number of nodes and make the state space analysis more tractable. An equivalence specification is a pair $(\approx_M, \approx_{BE})$, where \approx_M is equivalence relation on markings and \approx_{BE} is an equivalence relation on binding elements [24]. In the context of modeling CWS we define those equivalence relations on results from used web services, according to how the results are used in CWS. Hence we can assume that all the results from the web service are equal, or we can define several classes of them. The equivalence for markings is:

$$M_1 \approx_M M_2 \Rightarrow \forall p \in P : (M_1(p) = M_2(p) \vee (p \in (X(T_{WS}) \cup PN) \wedge M_1(p) \approx_{result} M_2(p)))$$

where:

- P is a set of all places,
- PN is a set of port nodes (places between pages in a hierarchy),
- T_{WS} is a transition that represents call to an external web service (as defined in the previous point),
- X is function that maps a node to a set of its surrounding nodes,
- \approx_{result} is an equivalence relation on results from the web service.

From the above definition two markings are the same if the only places they differ are the input or output from an interaction page or places surrounding a transition for an interaction with an external WS. Additionally colors for those places are equal as we defined for the WS's results. The binding elements equivalence is:

$$BE_1 \approx_{BE} BE_2 \Rightarrow (t(BE_1) = t(BE_2) \wedge (\forall v \in Var(t(BE_1)) (b(BE_1)(v) \approx_{result} b(BE_2)(v))))$$

where:

- t maps BE to its transition,
- b maps BE to its binding,
- $Var(t)$ is set of variables for transition t ,
- \approx_{result} is the same as previously.

From the definition two binding elements are equivalent if they are for the same transition and the bindings for variables are equivalent according to the relation defined for the WS's results.

For the CWS from Figure 4 and Figure 6 we can assume that all results from the web service are equal. Figure 7 presents an occurrence graph with equivalence classes for this CWS. Each node is specified with a name of place that is not empty in a marking, binding elements (arcs) are omitted, because they are straightforward for this example. By the specification of equivalence relationships we reduced the number of nodes from 404 to 8, and the number of arcs from 30401 also to 8.

To use an OE-graph to check an influence of failures of other web services on CWS we need to check reachability of successful execution of CWS. A marking that represents this state is the one that contains token element only in a place named "End", thus m is $M_{success}$ iff:

$$((m \in M) \wedge (m(p_{End}) \neq empty) \wedge (\forall p \neq p_{End} m(p) = empty))$$

where:

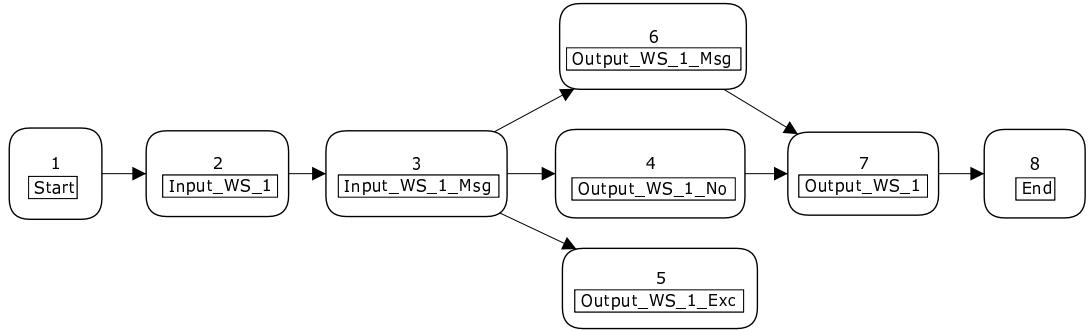


Figure 7: An example of an occurrence graph with equivalence classes (nodes are identified by names of non-empty places).

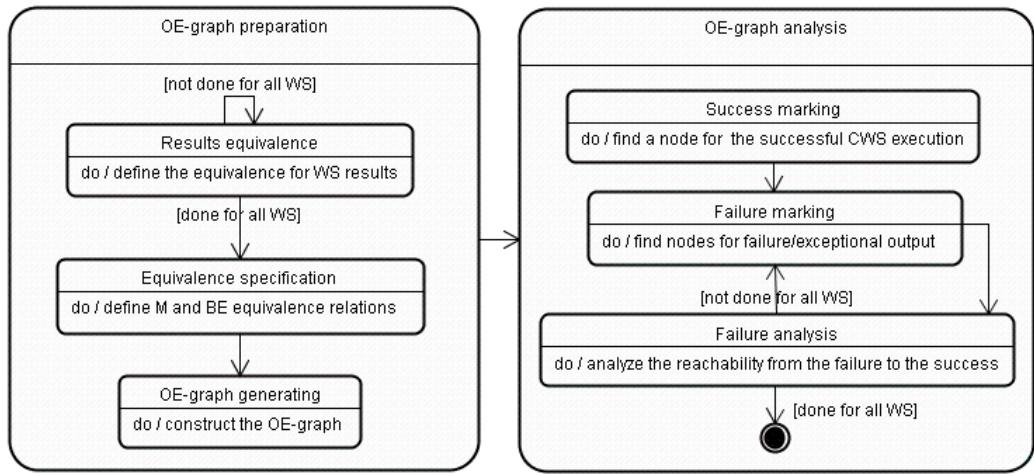


Figure 8: The preparation and analysis of an occurrence graph.

- M is a set of all markings in an OE-graph,
- p_{End} is a place named "End".

Analogously we can identify nodes and markings in an OE-graph that represent exceptional or no response types of output for each used external web service. Then we check the reachability of $M_{success}$ from all those states. If the success is reachable we can execute CWS even if there is an exception or no response, otherwise in case of a failure of a component we cannot execute CWS successfully. In the OE-graph in Figure 7 the successful marking is represented by a node 8. It can be concluded that even if the external web services is not responding (node 4) we can use it, but we cannot overcome exceptional messages (node 5).

The summary of the analysis of occurrence graphs in colored Petri Nets used in our approach is shown in Figure 8. The analysis consists of 2 main steps: preparation of OE-graphs and their usage. In the first step appropriate classes of equivalence are defined and then an OE-graph is constructed. In the second phase we first find a node in the graph that represents the successful state of execution and then nodes

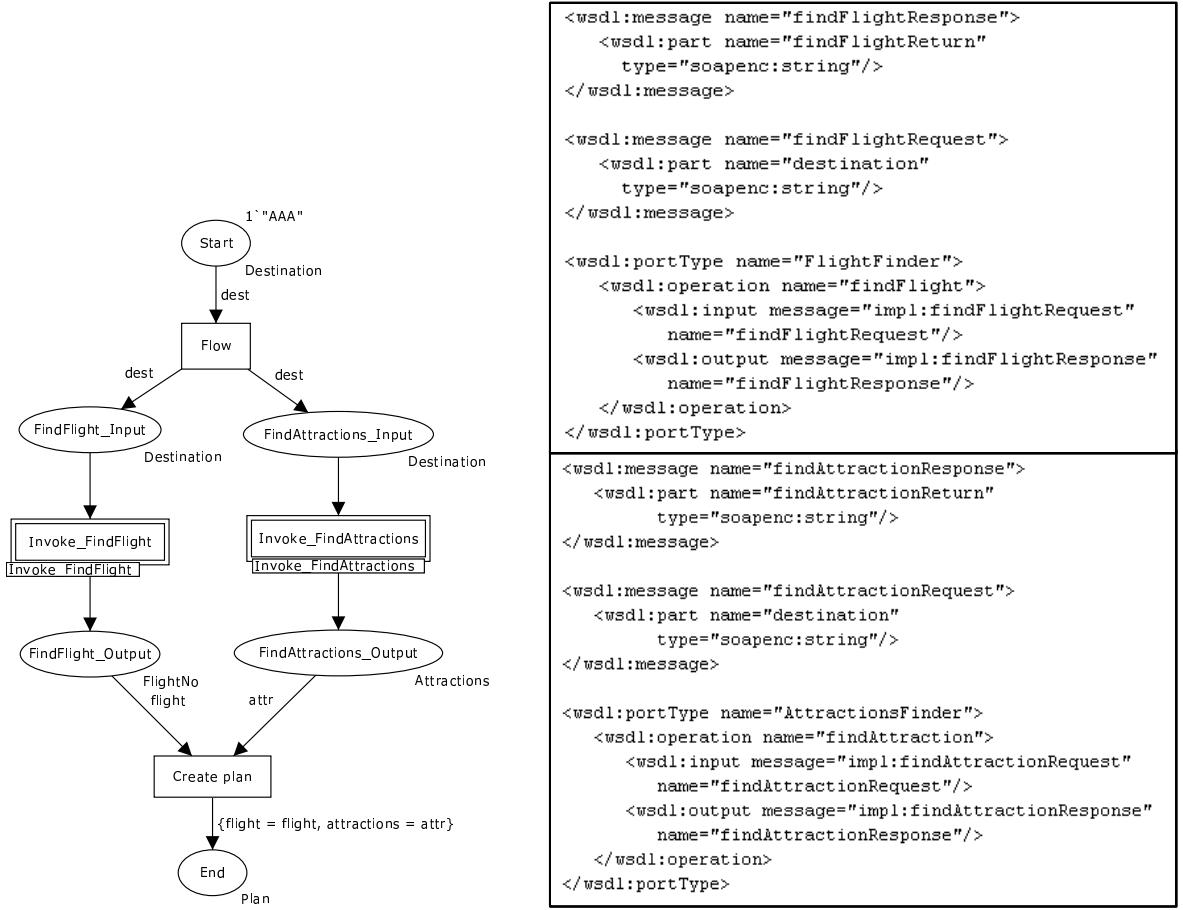


Figure 9: An example of CWS and WSDL descriptions for used operations

that represent each used web service's failure or exception. We perform on those nodes the reachability analysis. The result of the analysis is the answer to question whether CWS can be executed if one or more external WS fail.

6 An execution of CWS

The result of the analysis phase is used to prepare different versions of CWS and then choose an appropriate version according to states of external WS. In our approach a source of information about those states are subsequent executions of the CWS. Thus if a web service returns a failure (fault message) we assume that it is not working and it is not used in the next execution of CWS. The main gain of this solution is that CWS avoids, if it is possible, usage of WS that are not responding.

To illustrate this idea we present a CWS that prepares a simplified plan for a vacation. Given a destination name it should find a flight and attractions that are available at the destination. The main page of this CWS, as well as a part of WSDL descriptions are shown in Figure 9. Both of the external web services do not define exceptional messages, and input and output messages contain only string values. The response from such a CWS is a token that will be in the place End after execution. Pages with detailed interactions are not shown, however it is assumed that if the FindAttractions web service does not answer

”From” marking	”To” marking	Reachable
Output_FindAttractions_No + Output_FindFlight_Msg	End	Yes
Output_FindAttractions_Msg + Output_FindFlight_No	End	No

Table 2: The results of the analysis of the Vacation planner composite web service.

then a default data is returned.

According to Figure 8 an analysis starts with the definition of equivalence classes for results from external WS. In this example both WS return only 1 type of the correct response, thus all markings and binding elements that represent results from the FindFlight or the FindAttractions are equal. It is enough to create an OE-graph. This kind of graphs are not supported by the CPN Tools [11], so to construct them we just limit all input and results color sets to the sets with one color each (or more, according to the defined equality for WS results). Using the CPN Tools and a model of the CWS with color sets limited to 1 color, we constructed the OE-graph, and in the case of the Vacation planner CWS there are 27 nodes (we omitted here its graphical representation). We now use the graph to analyze the reachability of the successful Vacation planner execution. There are 2 used WS, without any defined exception so only two possible paths must be examined: one from the marking that represents the failure of the FindFlight web service and one from the FindAttractions failure. The results of this analysis are shown in the Table 2. The first line describes the marking that has a token in a place in which the FindAttractions returns a failure and the FindFlight returns a correct message, the second line is the opposite. Because the success (a marking with a token in the End place) is reachable from a marking that represents the FindAttractions failure, we can execute the Vacation planner even if this WS is not working properly or is not available. Unfortunately if we cannot find a flight the whole CWS cannot be successfully executed.

The above results are used to prepare 2 versions of the Vacation planner: with and without a call to the FindAttractions. Both of them are implemented in BPEL [26], and the implementation of the version without the FindAttractions just omits the call to it. We need also an additional web service that stores the results of calls to the FindFlight and the FindAttractions and in case of a failure switches between the versions. In order to store the results the additional (“reasoning”) WS offers operations used in the Vacation planner to inform about exceptions or failures that are encountered during calls. The communication between components used for the deployment is shown in Figure 10. First the Vacation planner decides which version invoke, then appropriate WS are called. If during execution of the CWS a failure of one of used components is encountered, then the general CWS is invoked and informed about the failure. If it is the FindAttractions web service, then the next execution will use the VacationPlanner_v2 (the version without the FindAttractions).

To test the Vacation planner CWS (presented in Figure 10) all components were deployed as shown in Figure 11. Both versions of the CWS are executed on an ActiveBPEL 2.1 [1] engine and both atomic WS (the FindFlight and the FindAttractions) as well as the additional CWS (Java implementation) on an Axis 1.4 [5] engine. For this configuration the FindAttractions web service was returning a fault message with a server exception, which simulates this web service’s failure. Because for this model it is possible to successfully execute the CWS without the FindAttraction, the successful execution was possible, however for the next execution the VacationPlanner_v2 was chosen, thus interaction with the faulty component was avoided. This is important because such interaction may take longer time, so the whole CWS is executed longer. The other advantage of avoiding interactions with not working web services is that servers can be overloaded, so when requests are not sent, then server will recover faster.

7 Conclusions and future work

The solution presented in this paper is the model-driven approach to deal with failures encountered during an execution of CWS. Our proposition focuses on omission failures and uses the colored Petri Nets modeling

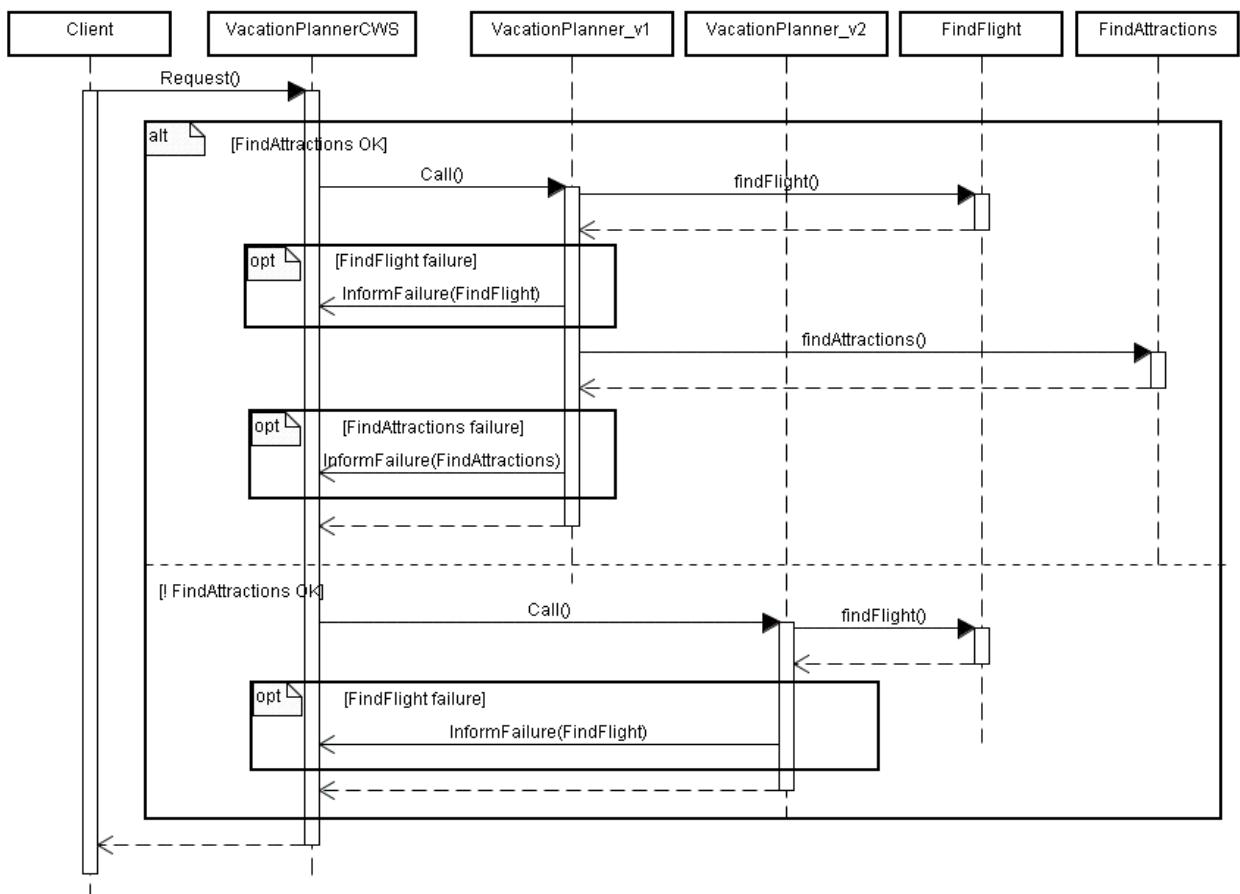


Figure 10: Communication between components in the Vacation planner CWS.

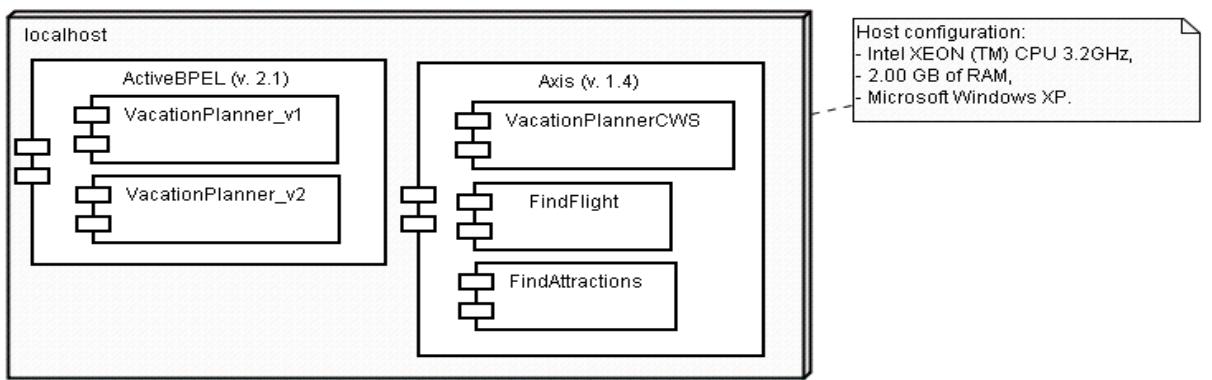


Figure 11: The deployment of the Vacation planner components.

language to infer about alternative paths to execute CWS. The main advantage of our solution is that we avoid interactions with components that are not working. This goal is achieved without reducing the autonomy of used components, which is very important in CWS and makes our proposition different from the others.

The main drawback of this concept is that alternatives must be prepared before execution can start, and the number of them may become high. For n used WS we can have as many as 2^n versions, because we may need a version for working and not working case of every WS. Because each version must be translated into BPEL and deployed on server, complex CWS may require substantial computation power and memory. To overcome this disadvantage we propose to use the Petri Nets model as an execution basis and perform the reachability analysis during execution. This was (partially) done by implementing an engine to execute CWS, based on the colored Petri Nets models. The implementation is a plugin to the BRITNeY tool [8], which allows us to add to the nets the code that invokes external WS (they are deployed on Axis 1.4 server) and which uses a simulation as the basis for executing CWS. Within this engine we can perform dynamically the reachability analysis using OE-graphs, which are built for every loaded CPN model of CWS. In this way we can avoid storing many versions of CWS and decide whether external WS should be invoked at the time of executing CWS.

The other problem of our proposition is the very limited number of solutions to overcome failures of used WS. We use only one way to do it: not to invoke the faulty WS. The result is that in case when the faulty web service is compulsory to execute successfully CWS (like FindFlight in the example) we are unable to overcome it. To make it possible, at the level of CWS, we can only try to find other web service that can perform requested operation. This can be done by maintaining a register of equivalent web services or if we use a dynamic discovery of the same WS. Another approach may be to use the semantic web services [16], thus find services that are semantically compatible.

The solution for overcoming failures of external web services presented above is based on the knowledge that the composite web services have during their definition and execution. It was shown that this solution is feasible to implement in the current standards (BPEL [26] and [34]), however we are still investigating possibilities of improving it.

References

- [1] ActiveEndpoints. ActiveBPEL open source BPEL engine. <http://www.active-endpoints.com/active-bpel-engine-overview.htm>.
- [2] Tom Anderson. *Fault tolerance, principles and practice*. Prentice Hall International, Englewood Cliffs, N.J., 1981.
- [3] L. Ardissono, L. Console, A. Goy, G. Petrone, C. Picardi, M. Segnan, and D. T. Dupre. Advanced fault analysis in web service composition. In *WWW '05: Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 1090–1091, New York, NY, USA, 2005. ACM Press.
- [4] L. Ardissono, R. Furnari, A. Goy, G. Petrone, and M. Segnan. Fault tolerant web service orchestration by means of diagnosis. In *Software Architecture. Third European Workshop, EWSA*, volume 4344 of *Lecture Notes in Computer Science*, pages 2–16, Nantes, France, 2006. Springer-Verlag.
- [5] Apache Project Axis v1.4. <http://ws.apache.org/axis/>.
- [6] B. Benatallah, F. Casati, and F. Toumani. Analysis and management of web service protocols. In *Conceptual Modeling - ER 2004. 23rd International Conference on Conceptual Modeling. Proceedings, 8-12 Nov. 2004*, pages 524–41, Shanghai, China, 2004. Springer-Verlag.
- [7] K. P. Birman. *Reliable Distributed Systems*. Springer-Verlag, New York, 2005.

- [8] Basic Real-time Interactive Tool for Net-based animation BRITNeY Suite. http://wiki.daimi.au.dk/britney_britney.wiki.
- [9] G. Chafle, S. Chandra, P. Kankar, and V. Mann. Handling faults in decentralized orchestration of composite web services. In *Service-Oriented Computing - ICSOC 2005. Third International Conference. Proceedings, 12-15 Dec. 2005*, volume 3826 of *Lecture Notes in Computer Science*, pages 410–23, Amsterdam, Netherlands, 2005. Springer-Verlag.
- [10] G. Chafle, S. Chandra, V. Mann, and M. Nanda. Decentralized orchestration of composite web services. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 134–143, New York, NY, USA, 2004. ACM Press.
- [11] CPN Tools. CPN Group University of Aarhus. Denmark. <http://wiki.daimi.au.dk/cpn-tools/cpntools.wiki>.
- [12] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 02 1991.
- [13] L. Deron, F. Chen-Liang, C. Chyouhwa, and L. Fengyi. Fault tolerant web service. In *Tenth Asia-Pacific Software Engineering Conference*, 2003.
- [14] V. Dialani, S. Miles, L. Moreau, D. De Roure, and M. Luck. Transparent fault tolerance for web services based architectures. In *Euro-Par 2002 Parallel Processing. 8th International Euro-Par Conference. Proceedings, 27-30 Aug. 2002*, volume 2400 of *Lecture Notes in Computer Science*, pages 889–98, Paderborn, Germany, 2002. Springer-Verlag.
- [15] G. Dobson. Using ws-bpel to implement software fault tolerance for web services. In *Proceedings. 32nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 29 Aug.-1 Sept. 2006*, page 8. IEEE Comput. Soc, 2006.
- [16] D. Fensel, H. Lausen, A. Polleres, J. de Bruijn, M. Stollberg, D. Roman, and J. Domingue. *Enabling Semantic Web Services. The Web Service Modeling Ontology*. Berlin, Springer-Verlag, 2007.
- [17] A. Ferrara. Web services: a process algebra approach. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251, New York, NY, USA, 2004. ACM Press.
- [18] X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 621–630, New York, NY, USA, 2004. ACM Press.
- [19] R. Guerraoui and R. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag, 2006.
- [20] R. Hamadi and B. Benatallah. A petri net-based model for web service composition. In *ADC '03: Proceedings of the fourteenth Australasian database conference*, pages 191–200, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.
- [21] Definition of WS. IBM. <http://www-128.ibm.com/developerworks/webservices/newto/websvc.html>.
- [22] V. Issarny, F. Tartanoglu, A. Romanovsky, and N. Levy. Coordinated forward error recovery for composite web services. In *Proceedings 22nd International Symposium on Reliable Distributed Systems, 6-18 Oct. 2003*, pages 167–76, Florence, Italy, 2003. IEEE Comput. Soc.

- [23] K. Jensen. *Coloured Petri nets :basic concepts, analysis methods, and practical use, v.1*, volume 1. Springer-Verlag, Berlin, c1992.
- [24] K. Jensen. *Coloured Petri nets :basic concepts, analysis methods, and practical use, v.2*, volume 2. Springer-Verlag, Berlin, c1992.
- [25] L. Liu, Y. Meng, B. Zhou, and Q. Wu. A fault-tolerant web services architecture. In *Advanced Web and Network Technologies, and Applications. APWeb 2006 International Workshops: XRA, IWSN, MEGA, and ICSE. Proceedings, 16-18 Jan. 2006*, volume 3842, pages 664–71, Harbin, China, 2006. Springer-Verlag.
- [26] Business process execution language BPEL v.1.1. Microsoft BEA IBM. <http://www-128.ibm.com/developerworks/library/specification/ws-bpel/>.
- [27] John D. Musa. *Software reliability :measurement, prediction, application*. McGraw-Hill, New York, 1990.
- [28] Universal Description Discovery & Integration (UDDI) OASIS. <http://www.uddi.org/>.
- [29] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 10/ 2003.
- [30] B. Randell. Fault tolerance in decentralized systems. In *Autonomous Decentralized Systems, 1999. Integration of Heterogeneous Systems. Proceedings. The Fourth International Symposium on*, pages 174–179, 21-23 March 1999.
- [31] W.M.P. Van Der Aalst, A.H.M. Ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5 – 51, 2003/07.
- [32] W.M.P. Van Der Aalst and K. M. Van Hee. *Workflow management: models, methods, and systems*. MIT Press, Cambridge, Mass., 2004.
- [33] Simple object access protocol (SOAP) 1.2 W3C. <http://www.w3.org/TR/soap12-part1/>.
- [34] Web Services Description Language (WSDL) v. 1.1 W3C. <http://www.w3.org/TR/wsdl>.
- [35] XML Schema W3C Recommendation W3C. <http://www.w3.org/XML/Schema>.
- [36] S. Weerawarana. *Web Services Platform Architecture SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Upper Saddle River, NJ: Prentice Hall PTR,, 2005.
- [37] Y. Yang, Q. Tan, and Y. Xiao. Verifying web services composition based on hierarchical colored petri nets. In *IHIS '05: Proceedings of the first international workshop on Interoperability of heterogeneous information systems*, pages 47–54, New York, NY, USA, 2005. ACM Press.
- [38] L. Zhang. Challenges and opportunities for web services research. *International Journal of Web Services Research*, 1(1):vii–xiii, 2004/01 2004.

new YAWL: Specifying a Workflow Reference Language using Coloured Petri Nets^{*}

Nick Russell¹, Arthur H.M. ter Hofstede² and Wil M.P. van der Aalst^{1,2}

¹Eindhoven University of Technology,
PO Box 513, 5600MB, Eindhoven, The Netherlands
{n.c.russell,w.m.p.v.d.aalst}@tue.nl

²Queensland University of Technology,
PO Box 2434, QLD, 4001, Australia
a.terhofstede@qut.edu.au

Abstract. *new* YAWL is a business process modelling language founded on the workflow patterns. It radically extends the YAWL offering to provide holistic support for the control-flow, data and resource perspectives and allows business processes to be captured in sufficient detail that they can be directly enacted. In order to ensure that business processes are executed in a deterministic way, *new* YAWL is based on formal foundations. This paper describes the approach taken to specifying the operational semantics for *new* YAWL based on *Coloured Petri Nets*. It discusses the development of the semantic model for *new* YAWL, which was undertaken using *CPN Tools*, and the experiences associated with developing a complete operational design for an offering of this scale using formal techniques.

1 Introduction

Over recent years the concept of the business process has garnered increasing interest as organisations seek to better understand what they do and how they can do it more efficiently. Business processes are increasingly viewed as corporate assets which companies must manage and maintain if they are to continue to operate effectively. In response to the rising demand for flexible means of automating parts of (or even entire) business processes, the field of workflow technology underwent explosive growth during the 1980s and 1990s as organisations sought configurable forms of process support.

As with many early stage technologies, individual workflow offerings provided a distinct approach to modelling the processes that they sought to automate, thus obviating any potential for standardising the representation and enactment of processes or integrating processes based on distinct offerings. Attempts by industry bodies such as the Workflow Management Coalition (WfMC) (e.g. [Wor95]) to resolve this impasse have only been marginally successful in establishing broadly adopted modelling formalisms. Consequently the majority of guidance in this area has come from

* This research is conducted in the context of the *Patterns for Process-Aware Information Systems (P4PAIS)* project which is supported by the Netherlands Organisation for Scientific Research (NWO). It is also receives partial support from the Australian Research Council under the Discovery Grant *Expressiveness Comparison and Interchange Facilitation between Business Process Execution Languages*.

one of two sources: (1) individual workflow offerings or models, such as MOBILE, WIDE and XPDL 2.0, that support a comprehensive range of concepts that generalise well to other workflow initiatives and (2) the enterprise modelling field, which includes techniques such as the Zachmann framework, EKD, IDEF, CIMOSA and ARIS, that seek to characterise the range of concepts that are relevant to modelling an organisation and its constituent processes and provide integrated approaches to capturing this information. Whilst there is general agreement across most of these offerings that a comprehensive business process model should include consideration of (at least) the control-flow, data and resource (or organisational) aspects of a business process, there is a wide variation in the range of concepts that individual techniques support for each perspective. Moreover, none of the popular modelling notations are based on a formalised model, thus leaving open the potential for ambiguity when capturing and enacting a business process.

An alternate approach to identifying the range of constructs that should be supported in a business process modelling language can be pursued which is based on *patterns*. By definition, patterns identify meaningful constructs that exist in a given problem domain. The *Workflow Patterns Initiative*¹ has established a catalogue of patterns that are relevant to the domain of business process modelling and enactment through a comprehensive evaluation of workflow and case handling systems, business process modelling and execution languages and web service composition standards. Proposed by van der Aalst et. al [AHKB03] in an effort to characterise the desirable properties of workflow languages, this research initially focussed on the control-flow perspective and identified 20 patterns which described “generic, recurring constructs” [RZ96]. The ubiquity of the patterns was soon recognised and catalogues of patterns have also been developed for the data [RHEA05] and resource [RAHE05] perspectives.

In this paper we propose *newYAWL*, a comprehensive workflow reference language based on formal foundations. *newYAWL* is founded on the workflow patterns ensuring that it recognises current practice in the process technology field and supports the capture and enactment of a wide range of workflow constructs in a deterministic way. The formalisation of *newYAWL* is based on Coloured Petri Nets [Jen97] thus providing a precise definition of the operational semantics of the *newYAWL* language that can be directly executed. The complete formalisation of a comprehensive workflow language encompassing multiple perspectives is a complex activity as demonstrated by the resultant size of the semantic model for *newYAWL* which incorporates 55 distinct pages of CPN diagrams and encompasses 480 places, 138 transitions and in excess of 1500 lines of ML code. Nevertheless, the development of the semantic model effectively demonstrates that with the correct specification tools, it is possible to formally define languages of this scale.

The paper proceeds as follows: Section 2 introduces the various constructs that make up the *newYAWL* workflow language. Section 3 describes the content of the *newYAWL* business process language and the manner in which it is captured and operationalised. Section 4 overviews related work. Section 5 discusses the experiences associated with the formalisation of *newYAWL* and concludes the paper.

¹ Further details on the workflow patterns, including detailed definitions, product evaluations, animations, vendor feedback and an assessment of their overall impact can be found at www.workflowpatterns.com.

2 *new* YAWL: A patterns-based workflow language

The workflow patterns triggered the development of *YAWL* [AH05] — an acronym for *Yet Another Workflow Language*. Unlike other efforts in the BPM area, YAWL sought to provide a comprehensive modelling language for business processes based on formal foundations. The content of the YAWL language is an adaptation of Petri Nets informed by the workflow patterns. One of its major aims was to show that a relatively small set of constructs could be used to directly support most of the workflow patterns identified. It also sought to illustrate that they could coexist within a common framework. In order to validate that the language was capable of direct enactment, the *YAWL System*² was developed, which serves as a reference implementation of the language. Over time, the YAWL language and the YAWL System have increasingly become synonymous and have garnered widespread interest from both practitioners and the academic community alike³.

Initial versions of the YAWL System focussed on the control-flow perspective and provided a complete implementation of 19 of the original 20 patterns. Subsequent releases incorporated limited support for selected data and resource patterns, however this effort was hampered by the lack of a complete formal description of the patterns in these perspectives. Moreover, a recent review [RHAM06] of the control-flow perspective identified 23 additional patterns which illustrate a number of commonly used control-flow constructs, many of which YAWL is unable to provide direct support for, including the partial join, transient and persistent triggers, iteration and recursion.

In an effort to manage the conceptual shortcomings of YAWL with respect to the range of workflow patterns that have now been identified, a substantial revision of the language is proposed — termed *newYAWL* — which aims to support the broadest range of the workflow patterns in the control-flow, data and resource perspectives. *newYAWL* provides a comprehensive formal description of the workflow patterns, which to date have only partially been formalised. It has a complete abstract syntax which identifies the characteristics of each of the language elements. Associated with this is an executable, semantic model for *newYAWL* — presented in the form of a Coloured Petri Net — which defines the runtime semantics of each of the language constructs. The following sections provide an overview of the features of *newYAWL* in the control-flow, data and resource perspectives.

2.1 Control-flow perspective

Figure 1 identifies the complete set of language elements which comprise the control-flow perspective of *newYAWL*. All of the language elements in *YAWL* have been retained and perform the same functions. A more detailed discussion of YAWL can be found in [AH05]. Several new constructs have been added based on the full range of workflow patterns that have now been identified. These are:

² See <http://www.yawl-system.com> for further details of the YAWL System and to download the latest version of the software.

³ Hereafter in this paper, we refer to the collective group of YAWL offerings developed to date — both the YAWL language as defined in [AH05] and also more recent *YAWL System* implementations of the language based on the original definition (up to and including release Beta 8.1) — as *YAWL*.

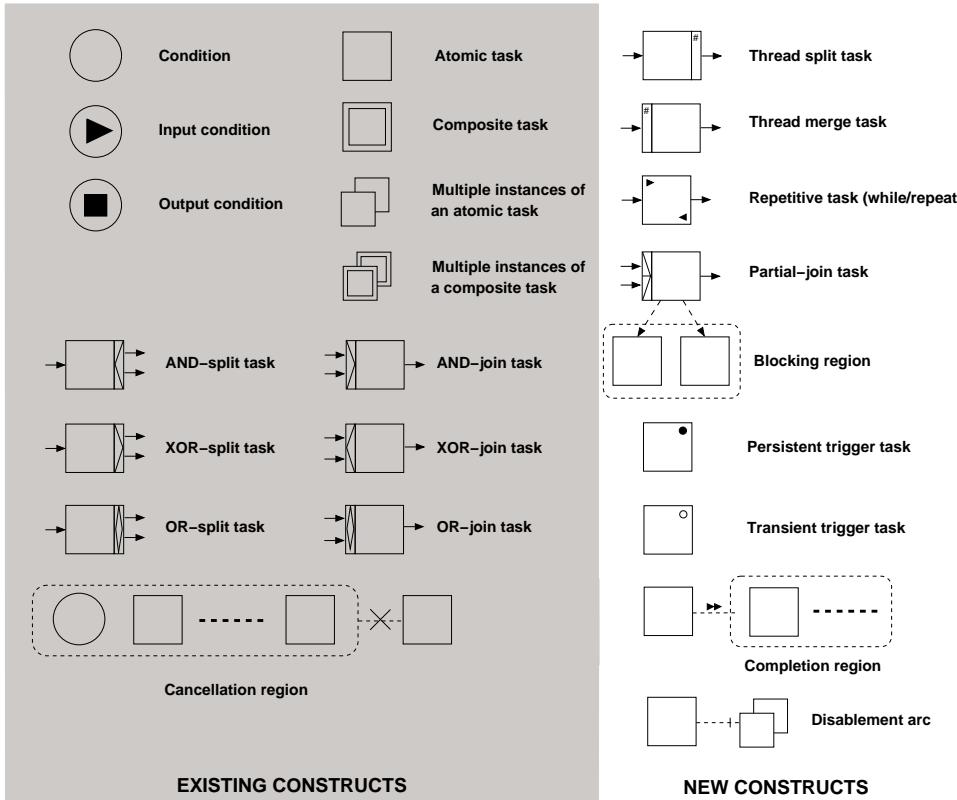


Fig. 1. newYAWL symbology

- the *Thread split* and *Thread merge* constructs, which allow the thread of control to be split into multiple concurrent threads or several distinct threads to be merged into a single thread of control respectively. The number of threads being created/merged is specified for the construct in the design-time model. Figure 2(a) illustrates these constructs. After the *make box* task, twelve threads of control are created ensuring that the *fill bottle* task runs 12 times before the *pack box* task can run (merging these threads before it commences);
- the *Partial join* (also known as the m-out-of-n join) allows a series of incoming branches to be merged such that the thread of control is passed to the subsequent branch when m of the incoming n branches are enabled. In Figure 2(b), the *cancel booking* task has a 1-out-of-3 join associated with it. If any of the incoming branches are enabled, then the *cancel booking* task is enabled (and any preceding tasks that are still executing in the associated cancellation region are withdrawn);
- the *Structured loop* (which supports while, repeat and combination loops) allows a task (or a sequence of tasks in the form of a subprocess) to execute repeatedly based on conditional tests at the beginning and/or end of each iteration. The loop is structured in form and it has a single entry and exit point. Figure 2(c) illustrates a repeat loop for the *check backup* task which executes repeatedly until all backups have been verified (i.e. it is a post-tested repeat loop);

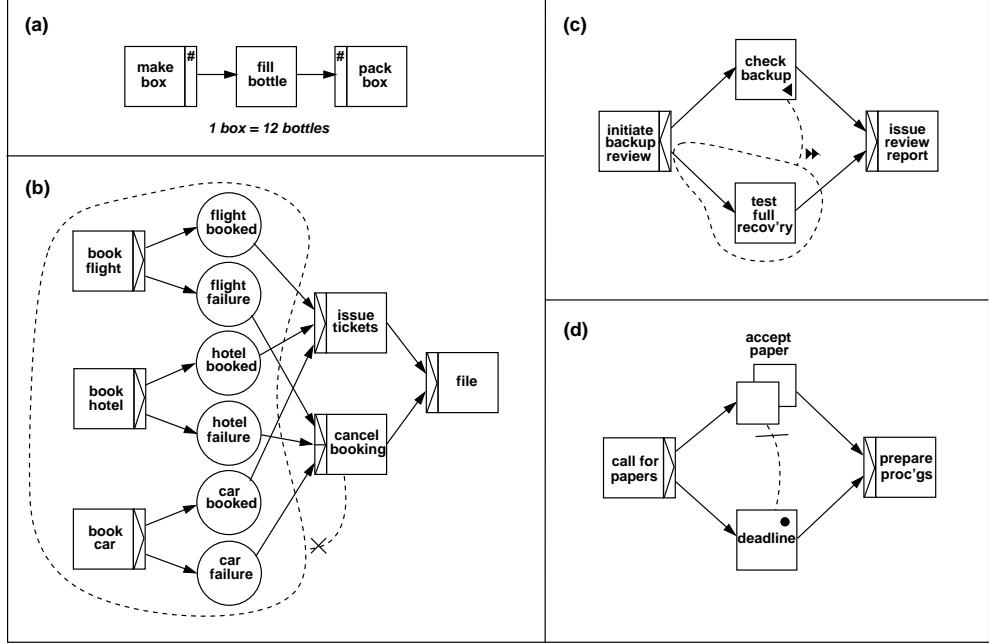


Fig. 2. Examples of *newYAWL* control-flow constructs

- the *Completion region* supports the forced completion of tasks which it encompasses. In Figure 2(c) the *test full recovery* task is forcibly completed once (all iterations of) the *check backup* task has finished. This allows the *issue review report* task to be immediately enabled;
- *Persistent triggers* and *Transient triggers* support the enablement of a task being contingent on a trigger being received from the operating environment. They are durable or transient in form respectively. Figure 2(d) illustrates a persistent trigger (assumedly associated with some form of alarm) which allows the *deadline* task to be enabled when it is received. As this trigger is durable in form, it is retained for future use if it is received before the thread of control arrives at the *deadline* task;
- the *Disablement arc* allows a dynamic multiple instance task to be prevented from creating further instances but allows for each of the currently executing instances to complete normally. Figure 2(d) has a disablement arc associated with the *deadline* task which prevents any further papers from being accepted once it has completed.

2.2 Data perspective

Whilst the control-flow perspective has received considerable focus in many workflow initiatives, the data perspective is often only minimally supported with issues such as persistence, concurrency management and complex data manipulation often being outsourced to third party products. In an effort to characterise the required range of data facilities in a workflow language, *newYAWL* incorporates a series of features derived from the data patterns. These include:

- Support for a variety of *distinct scopes* to which data elements can be bound. This allows the visibility and use of data elements to be restricted. The range of data scopes recognised include: *global* (available to all elements of all process instances), *folder* (available to the elements of process instances to which the folder is currently assigned), *case* (available to all elements in a given process instance), *block* (available to all elements of a specific process or subprocess definition for a given process instance), *scope* (available to a subset of the elements in a specific top-level process or subprocess definition for a given process instance), *task* (available to a given instance of a task) and *multiple-instance* (available to a specific instance of a multiple instance task);
- *Formal parameters* for specifying how data elements are transferred between process constructs (e.g. block to task, composite task to subprocess decomposition, block to multiple instance task). These parameters take a function-based approach to data transfer, thus providing the ability to support inline formatting of data elements and setting of default values. Parameters can be associated with tasks, blocks and processes;
- *Link conditions* for specifying conditions on outgoing arcs from OR-splits and XOR-splits that allow the determination of whether these branches should be activated;
- *Preconditions* and *postconditions* for tasks and processes. They are evaluated at the enablement or completion of the task or process with which they are associated. Unless they evaluate to true, the task or process instance with which they are associated cannot commence or complete execution; and
- *Locks* which allows tasks to specify data elements that they require exclusive access to (within a given process instance) in order to commence. Once these data elements are available, the associated task instance retains a lock on them until it has completed execution preventing any other task instances from using them concurrently. The lock is relinquished once the task instance completes.

2.3 Resource perspective

The resource perspective in *newYAWL* provides a variety of means of controlling and optimising the way in which work is distributed to users and the manner in which it is progressed through to ultimate completion. For each task, a specific *interaction strategy* can be specified which precisely describes the way in which the work item will be communicated to the user, how their commitment to executing it will be established and how the time of its commencement will be determined. Similarly, a detailed *routing strategy* can be defined which determines the range of potential users that can undertake the work item. The routing strategy can nominate the potential users in a variety of ways — they can be directly specified by name, in terms of roles that they perform or the decision as to possible users can be deferred to runtime. There is also provision for determining the range of potential users based on capabilities that individual users possess, the organisational structure in which the process operates or the results of preceding execution history. The routing strategy can be further refined through the use of constraints that restrict the potential user population. Indicative constraints may include: *retain familiar* (i.e. route to a user that undertook a previous work item), *four eyes principle* (i.e. route to a different user than one who undertook a previous work item), *random allocation* (route to a user at random from the range of potential users), *round robin allocation* (route to a

user from the potential population on an equitable basis such that all users receive the same number of work items over time) and *shortest queue allocation* (route the work item to the user with the shortest work queue).

newYAWL also supports two advanced operating modes that are designed to expedite the throughput of work by imposing a defined protocol on the way in which the user interacts with the system and work items are allocated to them. These modes are: *piled execution* where all work items corresponding to a given task are routed to the same user and *chained execution* where subsequent work items in a process instance are routed to the same user once they have completed a preceding work item. Finally, there is also provision for specifying a range of user privileges, both at process and individual task level, that restrict or augment the range of interactions that they can have with the process engine when they are undertaking work items.

3 Mapping *newYAWL* to Coloured Petri Nets

The language design for *newYAWL* is made up of two distinct components: (1) an *abstract syntax* that characterises the various constructs of which the language is comprised and the relationships between them, hence facilitating the capture of a *newYAWL* business process model from static perspective, and (2) a *semantic model* which describes the enactment of a *newYAWL* business process model.

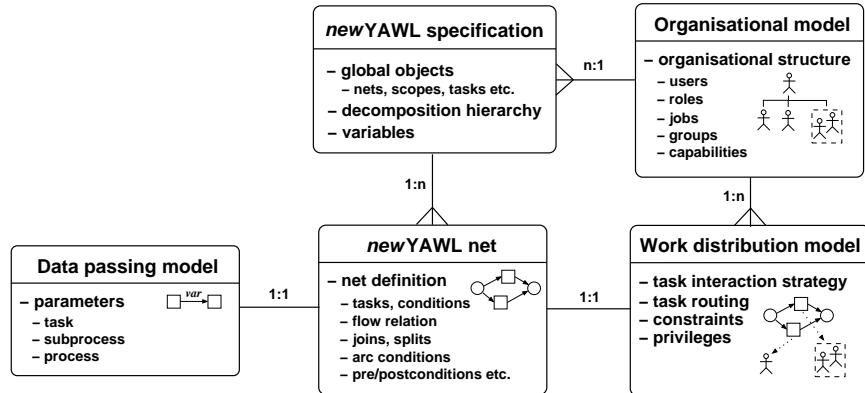


Fig. 3. Schema definition comprising the *newYAWL* abstract syntax

The *newYAWL* abstract syntax is composed of five distinct schemas that capture various aspects of a business process model. Each of these schemas is specified on a set-theoretic basis. Figure 3 summarises the content captured by each of the individual schemas and the relationships between them. Each process captured using the *newYAWL* abstract syntax has a single instance of the *newYAWL* specification associated with it. This defines elements that are common to all of the schemas and also captures the decomposition hierarchy. Each *newYAWL* specification is associated with an instance of the organisational model that describes which users are available to undertake tasks that comprise the process and the organisational context in which they operate.

A *new*YAWL process can be made up of a series of distinct subprocesses (where each subprocess specifies the manner in which a composite task is implemented) together with the top-level process. For each of these (sub)processes, there is an instance of the *new*YAWL net which describes the structure of the (sub)process in detail in terms of the tasks that it comprises and the sequence in which they occur. Associated with each *new*YAWL net is a data passing model which defines the way in which data is passed between elements in the process in terms of formal parameters operating between these elements. There is also a work distribution model that defines how each task will be routed to users for execution, any constraints associated with this activity and privileges that specific users may have assigned to them. The collective group of schemas for a specific process model is termed a *complete new*YAWL specification.

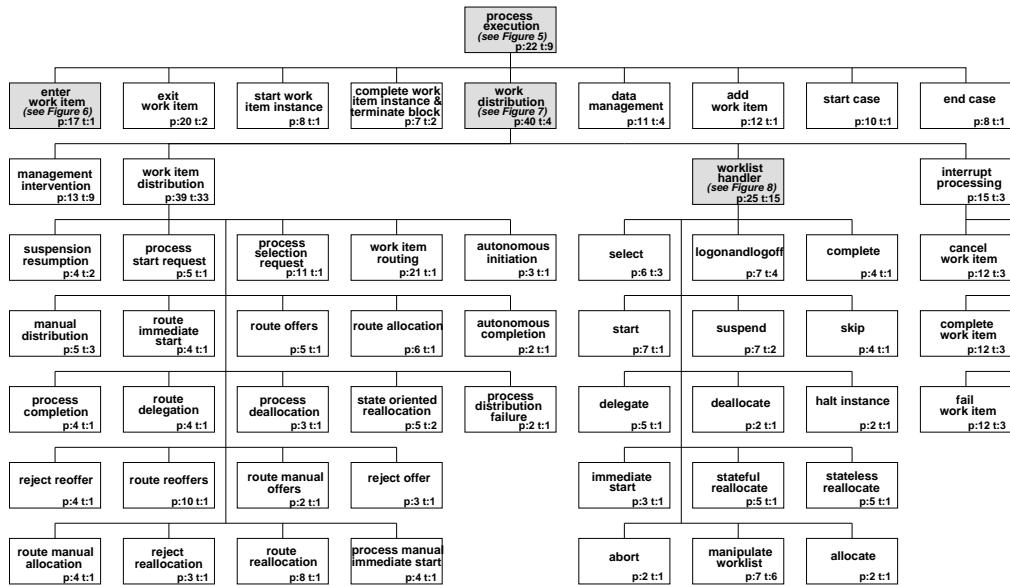


Fig. 4. *new*YAWL CPN model hierarchy (top-level in Figure 4)

The semantics of *new*YAWL are defined in terms of a series of interrelated Coloured Petri Nets developed using the CPN Tools environment. This approach to formalising the language offers the dual benefits of establishing a precise definition of the operation of each of the constructs which comprise *new*YAWL and also providing a means by which an instance of a *new*YAWL specification can be directly executed. There are 55 distinct CPNs which make up the semantic model. These are illustrated in Figure 4 along with the relationships between them. An indication of the complexity of individual nets is illustrated by the *p* and *t* values included for each of them which indicate the number of places and transitions that they contain. It is not possible to discuss the operation of all of these nets in the confines of this paper, however several of them (indicated by the shaded boxes and cross-references) are discussed in further detail in subsequent sections. A comprehensive description of the 55 CPNs which comprise the semantic model together with details of how it is initialised in order to facilitate the execution of a given *new*YAWL process model can be found in [RHEA07].

3.1 Overview of the semantic model

The semantic model logically divides into two main parts: (1) the control-flow and data sections and (2) the work distribution, organisational model and resource management sections. These roughly correspond to the *newYAWL* specification, *newYAWL* net and Data passing model, and the Organisational model and Work distribution model illustrated in Figure 3 respectively, which in turn seek to capture the majority of control-flow, data and resource patterns.

Figure 5, which is the topmost net in the semantic model, provides a useful summary of the major components and their interrelationship. The various aspects of control-flow, data management and work distribution information from the static *newYAWL* specification are encoded into the CPN model as tokens in individual places. The top level view of the lifecycle of a process instance is indicated by the transitions in this diagram connected by the thick black line. First a new process instance is started, then there are a succession of **enter** → **start** → **complete** → **exit** transitions which fire as individual task instances are enabled, the work items associated with them are started and completed and the task instances are finalised before triggering subsequent tasks in the process model. Each atomic work item needs to be distributed to a suitable resource for execution, an act which occurs via the **work distribution** transition. This cycle repeats until the last task instance in the process is completed. At this point, the process instance is terminated via the **end case** transition. There is provision for data interchange between the process instance and the environment via the **data management** transition. Finally where a process model supports task concurrency via multiple work item instances, there is provision for the dynamic addition of work items via the **add** transition.

The major data items shared between the activities which facilitate the process execution lifecycle are shown as shared places in this diagram. Not surprisingly, this includes both *static* elements which describe characteristics of individual processes such as the flow relation, task details, variable declarations, parameter mappings, preconditions, postconditions, scope mappings and the hierarchy of processes and subprocesses which make up an overall process model, all of which remain unchanged during the execution of particular instances of the process. It also includes *dynamic* elements which describe how an individual process instance is being enacted at any given time. These elements are commonly known as the *state* of a process instance and include items such as the current marking of the place in the flow relation, variable instances and their associated values, locks which restrict concurrent access to data elements, details of subprocesses currently being enacted, folder mappings (identifying shared data folders assigned to a process instance) and the current execution state of individual work items (e.g. *enabled*, *started* or *completed*).

There is relatively tight coupling between the places and transitions in Figure 5, illustrating the close integration that is necessary between the various aspects of the control-flow and data perspectives in order to enact a process model. The coupling between these places and the **work distribution** transition however is much looser. There are no static aspects of the process that are shared with other transitions in the model (i.e. the transitions underpinning **work distribution**) and other than the places which serve to communicate work items being distributed to resources for execution (and being started, completed or cancelled), the **variable instances** place is the only aspect of dynamic data that is shared with the work distribution

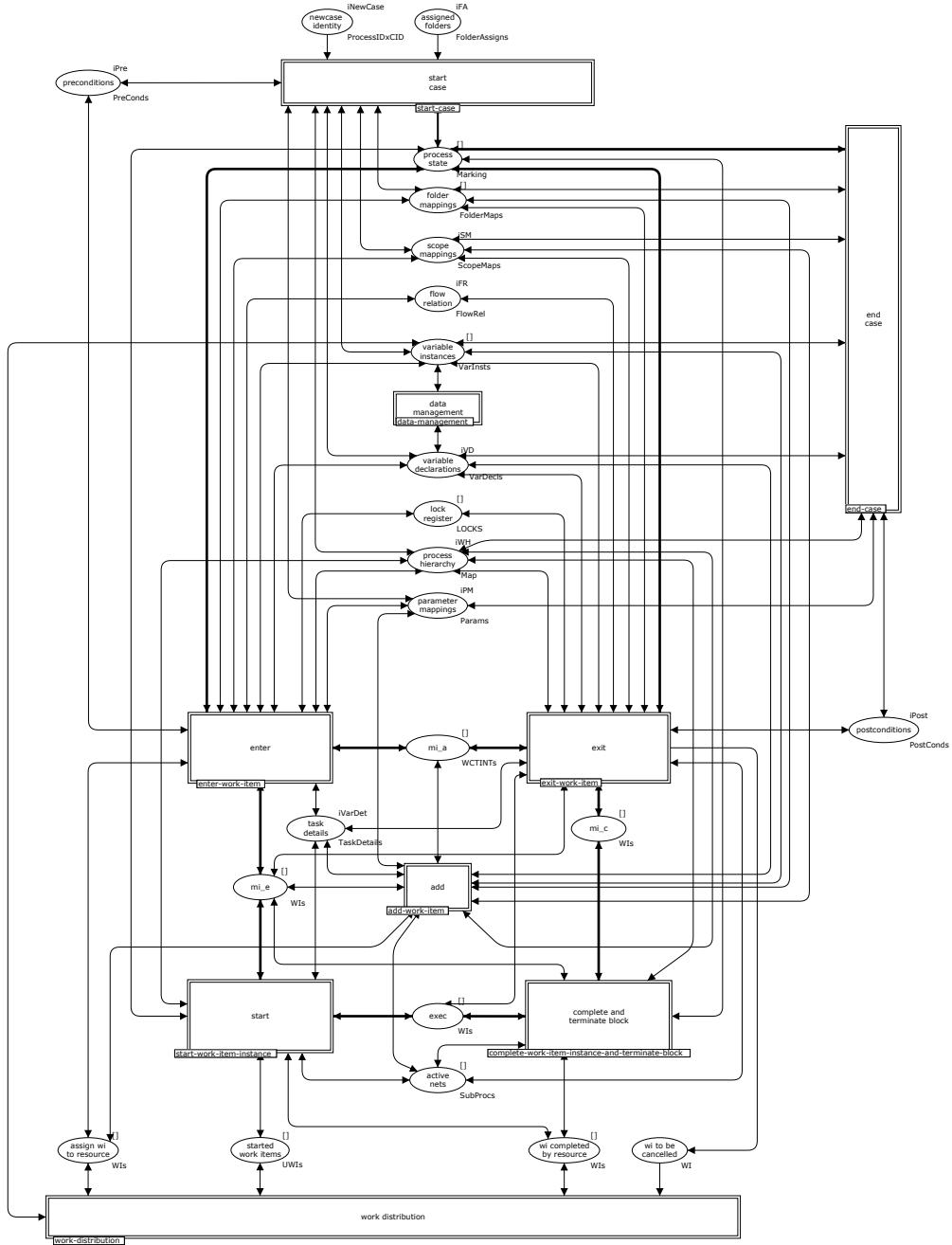


Fig. 5. Overview of the newYAWL semantic model

subprocess. The following sections focus on the two main parts of the *newYAWL* semantic model: (1) control-flow and data handling and (2) work distribution.

3.2 Control-flow and data handling

The actions comprising the control-flow and data handling processes are extremely complex both in terms of the range of concepts that they involve and the interrelationship between them. It is not possible to describe all aspects of these processes in the confines of this paper hence in this section we focus on one specific aspect of the overall work item lifecycle: *task instance enablement* and *work item creation*. Task instance enablement is the first step in work item execution. It is depicted by the `enter` transition in Figure 6. The first step in determining whether a task instance can be enabled is to examine the marking of the input places to the task. There are four possible scenarios:

- If the task has no joins associated with it, then the input condition to the task simply needs to contain a token;
- If the task has an AND-join associated with it, each input condition needs to contain a token with the same $ProcessID \times CID$ combination, where these two attributes uniquely identify a process and process instance (or case) respectively;
- If the task has an XOR-join associated with it, one of the input conditions needs to contain a token; and
- If the task has an OR-join associated with it, one (or more) of the input conditions needs to contain a token and a determination needs to be made as to whether in any future possible state of the process instance, the currently marked input conditions can retain at least one token and another input condition can also receive a token. If this can occur, the task is not enabled, otherwise it is enabled. This issue has been subject to rigorous analysis and an algorithm has been proposed [WEAH05] for determining exactly when an OR-join can fire. The *newYAWL* semantic model implements this algorithm.

Depending on the form of task that is being enabled (singular or multiple-instance), one or more work items may be created for it. If the task is atomic, the work item(s) is created in the same block as the task to which it corresponds. If the task is composite, then the situation is slightly more complicated and two things occur: (1) a “virtual” work item is created in the same block for each instance of the task that will be initiated (this enables later determination of whether the composite task is in progress or has completed) and (2) a new subprocess decomposition (or a new block) is started for each task instance. This involves the placement of a token in the input place to the subprocess decomposition which has a distinct subprocess CID. Table 1 indicates the potential range of work items that may be created for a given task instance. In order for a task to be enabled, all prerequisites associated with the task must be satisfied. There are five prerequisites for the `enter` transition to be able to fire:

- The precondition associated with the task must evaluate to true;
- All data elements which are inputs to mandatory input parameters must exist and have a defined value;
- All mandatory input parameters must evaluate to defined values;

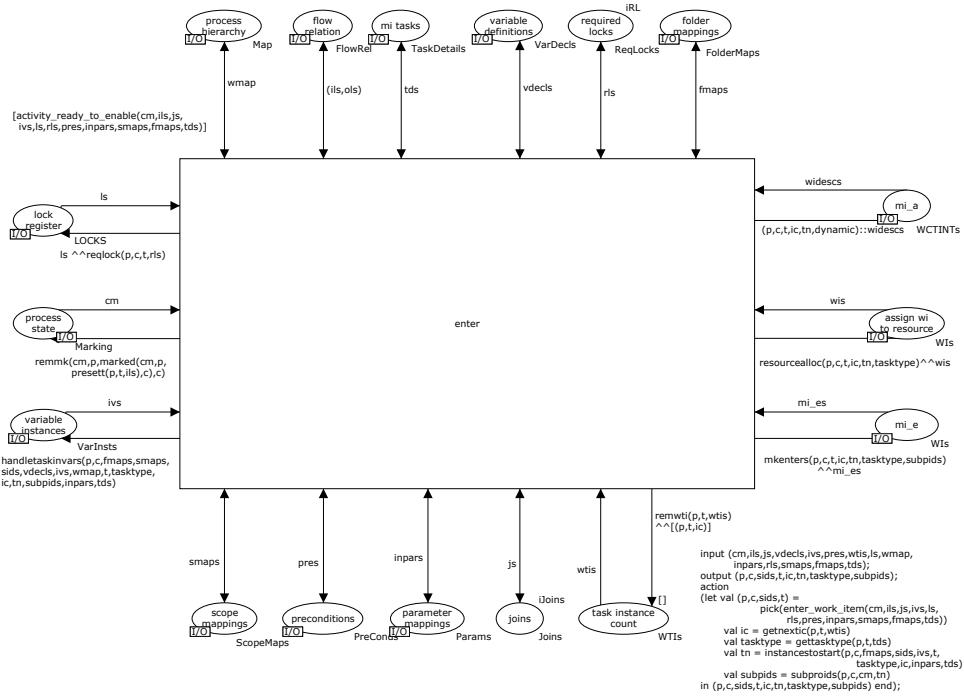


Fig. 6. Enter work item transition

- All locks which are required for data elements that will be used by the work items associated with the task must be available; and
- If the task is a multiple instance task, the multiple instance parameter when evaluated must yield a number of rows that is between the minimum and maximum number of instances required for the task to be initiated.

Once these prerequisites are satisfied, task enablement can occur. This involves:

1. Removing the tokens marking input conditions to the task corresponding to the instance enabled. The exact number of tokens removed depends on whether there is a join associated with the task or not and occurs as follows:
 - *No join*: one token corresponding to the *ProcessID* × *CID* combination that triggered the task is removed from the input condition to the task;
 - *AND-join*: one token corresponding to the triggering *ProcessID* × *CID* combination is removed from each of the input conditions to the task;
 - *XOR-join*: one token corresponding to the triggering *ProcessID* × *CID* combination is removed from *one* of the input conditions to the task; and
 - *OR-join*: one token corresponding to the triggering *ProcessID* × *CID* combination is removed from *any* of the input conditions to the task which currently contain tokens of this form.
2. Determining which instance of the task this is. The instance identifier (which corresponds to the *Inst* attribute) must be unique for each task instance and all work items and data elements associated with this task instance in order to

ensure that they can be uniquely identified. A record is kept of the next available instance for a task in the **task instance count** place.

3. Determining how many work item instances should be created. For a singular task (i.e. an atomic or composite task), this will always be a single work item, however for a multiple instance task (i.e. an atomic or composite multiple instance task), the actual number started will be determined from the evaluation of the multiple instance parameter which will return a composite result containing a number of rows of data. The number of rows returned indicates the number of instances to be started. In all of these situations, individual work items are created which share the same *ProcessID*, *CID*, *TaskID* and *Inst* values, however the *TaskNr* value is unique for each work item and is in the range *1...number of work items created*;
4. For all work items corresponding to composite tasks, distinct subprocess CIDs need to be determined to ensure that any variables created for subprocesses are correctly identified and can be accessed by the work items for the subprocesses that will subsequently be triggered;
5. Creating variable instances for data elements associated with the task. This varies depending on the task type and the number of work items created for the task:
 - For atomic tasks which only have a single instance, this will involve the creation of relevant task variables.
 - For atomic multiple instance tasks, this will involve the creation of both task variables and multiple instance variables for each task instance. The required multiple instance variables are indicated by the output data elements listed for the multiple instance parameter. and this set of variables is created for each new work item.
 - For composite tasks that only have a single instance, any required task variables are created in the subprocess decomposition that is instantiated for the task. Also, there may be block and scope variables associated with the subprocess decomposition that need to be created; and
 - For composite multiple instance tasks, any required block, scope, task variables and multiple instance variables are created for each subprocess decomposition that is initiated for the task.
6. Mapping the results of any input parameters for the task instance to the relevant output data elements. For multiple instance parameters, this can be quite a complex activity;
7. Recording any variable locks that are required for the execution of the task instance;
8. For all work items corresponding to atomic tasks (other than for automatic tasks which can be initiated without distribution to a resource), requests for work item distribution need to be created. These are routed to the **assign wi to resource** place and are subsequently dealt with by the **work distribution** transition; and
9. Finally, work items with an *enabled* status need to be created for this task instance and added to the **mi_e** place (which identifies work items corresponding to enabled but not yet started tasks) in accordance with the details outlined in Table 1.

Table 1. Task instance enablement in *newYAWL*

Task Type	Instances Initiated at Commencement	
<i>Atomic</i>	<i>Singular</i> Single work item created in the same block.	<i>Multiple Instances</i> Multiple work items created in the same block, each with a distinct <i>TaskNr</i> .
<i>Composite</i>	Single “virtual” work item created in the same block and a new subprocess is initiated for the block assigned as the task decomposition.	Multiple “virtual” work items created in the same block. Additionally a distinct subprocess is initiated for each work item created, each with a distinct subprocess <i>CID</i> and <i>TaskNr</i> .

The work distribution process in *newYAWL* provides an interesting contrast to the control-flow and data handling. Unlike the latter process which is comprised of a limited number of transitions which must coordinate state changes involving a large number of places with complex guard conditions, the work distribution process is much more diffuse. It involves multiple places which describe alternate states for a work item that is currently in progress and supports a variety of distinct transitions between these states. The work distribution process is discussed in the next section.

3.3 Work distribution

The main motivation for workflow systems is achieving more effective and controlled distribution of work. Hence the actual distribution and management of work items are of particular importance. The process of distributing work items is summarized by Figure 7. It comprises four main components⁴:

- the **work item distribution** transition, which handles the overall management of work items through the distribution and execution process;
- the **worklist handler**, which corresponds to the user-facing client software that advises users of work items requiring execution and manages their interactions with the main **work item distribution** transition in regard to committing to execute specific work items, starting and completing them;
- the **management intervention** transition, that provides the ability for a process administrator to intervene in the **work distribution** process and manually reassign work items to users where required; and
- the **interrupt handler** transition that supports the cancellation, forced completion and forced failure of work items as may be triggered by other components of the process engine (e.g. the control-flow process, exception handlers).

Work items that are to be distributed through this process are added to the **work items for distribution** place. This then prompts the **work item distribution** transition to determine how they should be routed for execution. This may involve the services of the process administrator in which case they are sent to the **management intervention** transition or alternatively they may be sent directly to one or

⁴ Note that the high-level structure of the work distribution process is influenced by the earlier work of Pesic and van der Aalst [PA07].

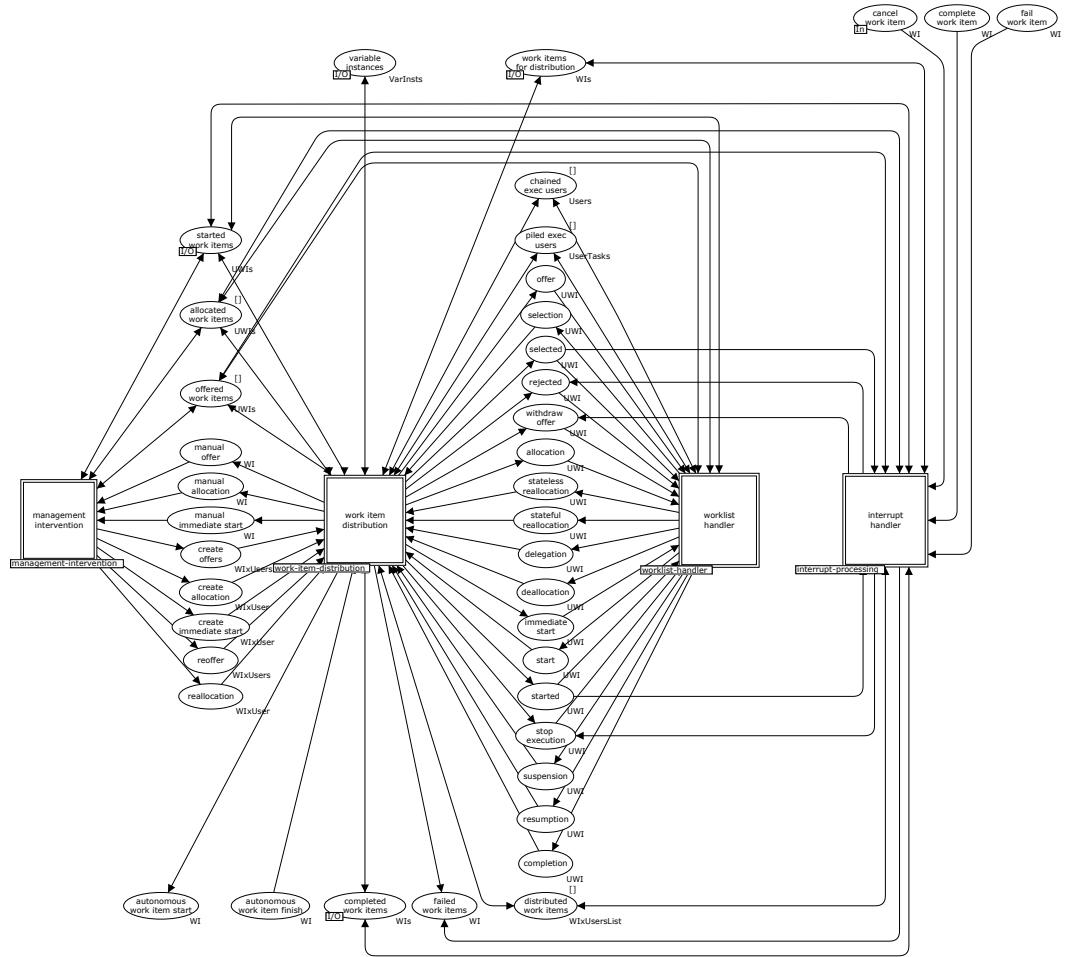


Fig. 7. Top level view of the main *work distribution* process

more users via the `worklist handler` transition. The various places between these three activities correspond to the range of requests that flow between them. In the situation where a work item corresponds to an *automatic* task, it is sent directly to the `autonomous work item start` place and no further distribution activities take place. An automatic task is considered complete when a token is inserted in the `autonomous work item finish` place.

A common view of work items in progress is maintained between the **work item distribution**, **worklist handler** and **management intervention** transitions via the offered work items, allocated work items and started work items places. There is also shared information about users in advanced operating modes that is recorded in the piled **exec users** and chained **exec users** places. Although there is significant provision for shared information about the state of work items, the determination of when a work item is actually complete rests with the **work item distribution** transition and when this occurs, it inserts a token in the completed

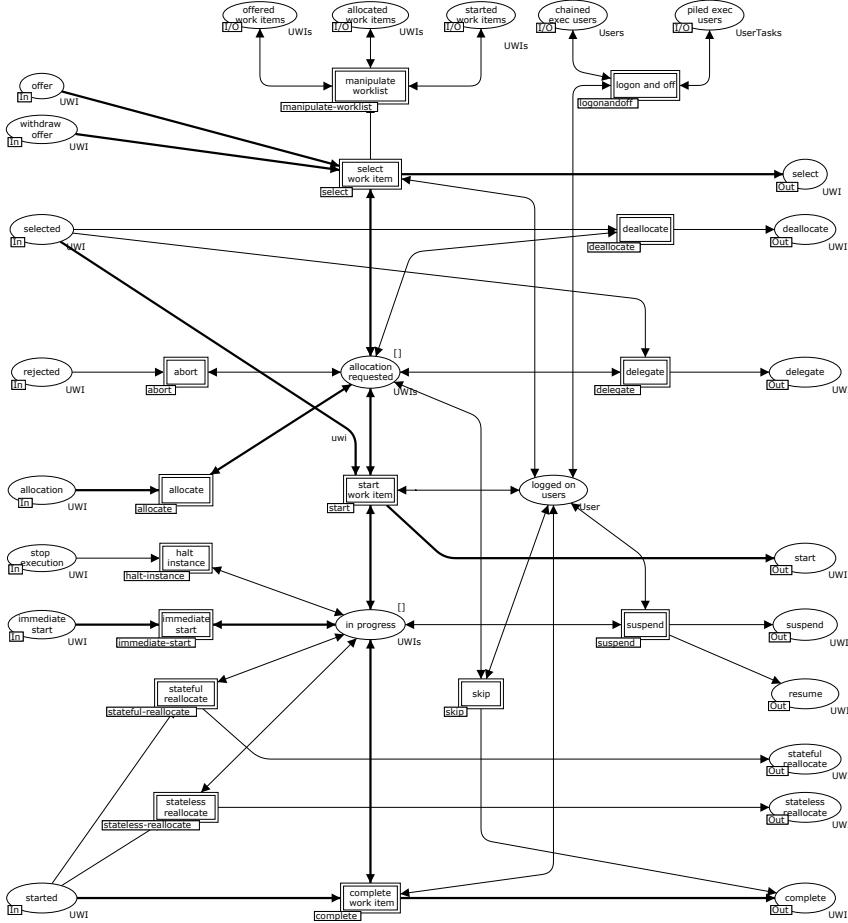


Fig. 8. Worklist handler process

work items place. Similarly, work item failures are notified via the failed work items place. The only exception to these arrangements are for work items that are subject to some form of interrupt (e.g. an exception being detected and handled). The interrupt handler transition is responsible for managing these occurrences on the basis of cancellation, forced completion and failure requests received in the cancel work item, complete work item and fail work item places respectively. All of the activities in the work distribution process are illustrated by substitution transitions indicating that each of them are defined in terms of significantly more complex subprocesses. It is not possible to present each of them in this paper, hence we focus on one of the more significant: the worklist handler process. The worklist handler is illustrated in Figure 8 and describes how the user-facing process interface (typically a worklist handler software client) operates and interacts with the work item distribution process. As previously, the main path through this process is indicated by the thick black arcs. There are various transitions that make up the process, these correspond to actions that individual users can request

in order to alter the current state of a work item to more closely reflect their current handling of it. These actions may simply be requests to start or complete it or they may be “detour” requests to reroute it to other users e.g. via **delegation** or **deallocation**. The manner in which these requests operate is illustrated by the shared places in Figure 7.

4 Related work

There have been numerous papers advocating approaches to workflow and business process modelling based on Petri Nets (cf. [Aal98],[EN93],[AAH98],[MR03]), however these tend to either focus on a single aspect of the domain (e.g. the control-flow perspective) or they are based on a relatively simplistic language. There have also been attempts to provide formal semantics using Petri Nets for many of the more widely used approaches to business process modelling including EPCs [Aal99], UML 2.0 Activity Diagrams [SH05] and BPMN [DDO07], although in each case arriving at a complete semantics has been hampered by inherent ambiguities in the informal descriptions for each of the formalisms. There has been minimal work on formalisation of the other workflow perspectives, one exception is [PA07] which investigates mechanisms for work distribution in workflows and presents CPN models for a number of the workflow resource patterns.

Historically, the modelling and enactment of processes have often been treated distinctly and it is not unusual for separate design and runtime models to be utilised by systems. Approaches to managing the potential disparities between these models have included the derivation of executable process descriptions from design-time models [DNLS⁺02] and the direct animation of design-time models for requirements validation [MLO⁺07]. The latter of these approaches which uses a strategy based on Coloured Petri Nets [Jen97] and CPN Tools [JKW07] as an enablement vehicle is one of a number of initiatives that have successfully used the CPN Tools offering as a means of executing various design-time modelling formalisms including Protos models [GAJV06], Sequence diagrams [RF06] and task descriptions [JLA06].

5 Experiences and conclusions

The *new* YAWL semantic model⁵ incorporates 55 distinct pages of CPN diagrams and encompasses 480 places, 138 transitions and in excess of 1500 lines of ML code. It took approximately six months to develop. The size of the model gives an indication of the relative complexity of formally specifying a comprehensive business process modelling language such as *new* YAWL. Indeed, it is only with the aid of an interactive modelling environment such as CPN Tools that developing a formalisation of this scale actually becomes viable. One of the major advantages of pursuing this approach to software development is that it provides a design that is executable. This allows fundamental design decisions to be evaluated and tested much earlier than would ordinarily be the case during the development process. Where suboptimal design decisions are revealed, the cost of rectifying them is significantly less than it would be later in the development lifecycle. There is also the opportunity to test alternate solutions to design issues with minimal overhead before a final decision is

⁵ This model is available at www.yawl-system.com/newYAWL.

settled on. A particular benefit afforded by this approach to formalisation is that the CPN hierarchy established during the design process provides an excellent basis on which to make subsequent architectural and development decisions.

The original motivations for this research initiative were twofold: (1) to establish a fully formalised business process modelling language based on the synthesis of the workflow patterns and (2) to demonstrate that the language was not only suitable for conceptual modelling of business processes but that it also contained sufficient detail for candidate models to be directly enacted. *newYAWL* achieves both of these objectives and directly supports 118 of the 126 workflow patterns that have been identified. It is interesting to note however that whilst the development of a model of this scale offers some extremely beneficial insights into the overall problem domain and provides a software design that can be readily utilised as the basis for subsequent programming activities, it also has its limitations. Perhaps the most significant of these is that the scale and complexity of the model obviates any serious attempts at verification. Even on a relatively capable machine (P4 1.6Ghz, 512Mb RAM), the model takes over 8 minutes to load. Moreover the potentially infinite range of business process models that the semantic model can encode, rules out the use of techniques such as state space analysis. This raises the question as to how models of this scale can be comprehensively tested and verified.

Notwithstanding these considerations however, the development of the semantic model delivered some salient insights into areas of newYAWL that needed further consideration during the formalisation activity. These included:

- recognition of the fact that at runtime the completion region construct can only bring affected work items to the point at which they *should* complete. It cannot force the completion to occur;
- recognition that when a self-cancelling task completes: (1) it should process the cancellation of itself last of all in order to prevent the situation where it cancels itself before all other cancellations have been completed and (2) it needs to establish whether it is cancelling itself before it can make the decision to put tokens in any relevant output places associated with the task;
- introduction of a consistent approach for handling the evaluation of any functions associated with a *newYAWL* specification e.g. for outgoing links in a XOR-split, pre/postconditions, pre/post tests for iterative tasks etc. This issue was ultimately addressed by mapping any necessary function calls to ML functions and establishing a standard approach to encoding the invocation of these functions and the passing of any necessary parameters and the return of associated results;
- adoption of a standard strategy for characterising parameters to functions in order to ensure that they could be passed in a uniform way to the associated ML functions that evaluated them;
- the establishment of a coherence protocol to ensure that reallocation of work items to alternate resources either by users or the process administrator are handled in a consistent manner in order to ensure that potential race conditions arising during reallocation do not result in the process engine, process administrator or the initiating user having differing views of the current state of work item allocations; and
- recognition that the current approach to privilege specification for users and tasks (where privileges need to be individually specified) is likely to be intractable for any large scale implementation of *newYAWL*.

There were also some learnings in regard to the CPN Tools offering during the course of this research. Whilst extremely powerful, there are several aspects of the CPN Tools environment that would benefit from the inclusion of additional capabilities. In particular, the ability to incrementally wind back the execution state of a given execution would be useful, as would the ability to save an execution state for later execution and analysis. The interaction facilities for the CPN model are particularly effective, however there are less features provided for tracing and altering the execution of ML code segments that form part of a CPN model. The inclusion of features such as these in future CPN Tools releases would be extremely beneficial.

The new YAWL semantic model will serve as the design blueprint for the next major version of the open-source YAWL System offering. This is currently being developed by the BPM Group at QUT.

References

- [AAH98] N.R. Adam, V. Atluri, and W.K. Huang. Modeling and analysis of workflows using Petri nets. *Journal of Intelligent Information Systems*, 10(2):131–158, 1998.
- [Aal98] W.M.P. van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
- [Aal99] W.M.P. van der Aalst. Formalization and verification of event-driven process chains. *Information and Software Technology*, 41(10):639–650, 1999.
- [AH05] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.
- [AHKB03] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, 2003.
- [DDO07] R.M. Dijkman, M. Dumas, and C. Ouyang. Formal semantics and automated analysis of BPMN process models. Technical Report 5969, Queensland University of Technology, Brisbane, Australia, 2007. <http://eprints.qut.edu.au/archive/00005969/>.
- [DNLS⁺02] E. Di Nitto, L. Lavazza, M. Schiavoni, E. Tracanella, and M. Trombetta. Deriving executable process descriptions from UML. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 155–165, New York, NY, USA, 2002. ACM Press.
- [EN93] C.A. Ellis and G.J. Nutt. Modelling and enactment of workflow systems. In M. Ajmone Marsan, editor, *Proceedings of the 14th International Conference on Application and Theory of Petri Nets*, volume 691 of *Lecture Notes in Computer Science*, pages 1–16, Chicago, IL, USA, 1993. Springer.
- [GAJVV06] F. Gottschalk, W.M.P. van der Aalst, M.H. Jansen-Vullers, and H.M.V. Verbeek. Protos2CPN: Using colored Petri nets for configuring and testing business processes. In K. Jensen, editor, *Proceedings of the 7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, volume PB-579 of *Daimi Reports*, pages 137–155, Aarhus, Denmark, 2006.
- [Jen97] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts*. Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1997.
- [JKW07] K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri nets and CPN Tools for modelling and validation of concurrent systems. *International Journal of Software Tools for Technology Transfer*, 9(3):213–254, 2007.
- [JLA06] J.B. Jørgensen, K.B. Lassen, and W.M.P. van der Aalst. From task descriptions via coloured Petri nets towards an implementation of a new electronic patient

- record. In K. Jensen, editor, *Proceedings of the 7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, volume PB-579 of *Daimi Reports*, pages 137–155, Aarhus, Denmark, 2006.
- [MLO⁺07] R.J. Machado, K.B. Lassen, S. Oliveira, M. Couto, and P. Pinto. Requirements validation: Execution of UML models with CPN Tools. *International Journal on Software Tools for Technology Transfer*, 9(3):353–369, 2007.
- [MR03] Daniel Moldt and Heiko Rölke. Pattern based workflow design using Reference nets. In W.M.P. van der Aalst, A.H.M. ter Hofstede, and M. Weske, editors, *Proceedings of the Business Process Management Conference 2003*, volume 2678 of *Lecture Notes in Computer Science*, pages 246–260, Eindhoven, The Netherlands, 2003. Springer.
- [PA07] M. Pesic and W.M.P. van der Aalst. Modelling work distribution mechanisms using colored Petri nets. *International Journal on Software Tools for Technology Transfer*, 9(3):327–352, 2007.
- [RAHE05] N. Russell, W.M.P. van der Aalst, A.H.M. ter Hofstede, and D. Edmond. Workflow resource patterns: Identification, representation and tool support. In O. Pastor and J. Falcão e Cunha, editors, *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*, volume 3520 of *Lecture Notes in Computer Science*, pages 216–232, Porto, Portugal, 2005. Springer.
- [RF06] O.R. Ribeiro and J.M. Fernandes. Some rules to transform sequence diagrams into coloured Petri nets. In K. Jensen, editor, *Proceedings of the 7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, volume PB-579 of *Daimi Reports*, pages 137–155, Aarhus, Denmark, 2006.
- [RHAM06] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow control-flow patterns: A revised view. Technical Report BPM-06-22, 2006. <http://www.BPMcenter.org>.
- [RHEA05] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow data patterns: Identification, representation and tool support. In L. Delcambre, C. Kop, H.C. Mayr, J. Mylopoulos, and O. Pastor, editors, *Proceedings of the 24th International Conference on Conceptual Modeling (ER 2005)*, volume 3716 of *Lecture Notes in Computer Science*, pages 353–368, Klagenfurt, Austria, 2005. Springer.
- [RHEA07] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. newYAWL: achieving comprehensive patterns support in workflow for the control-flow, data and resource perspectives. Technical Report BPM-07-05, 2007. <http://www.BPMcenter.org>.
- [RZ96] D. Riehle and H. Züllighoven. Understanding and using patterns in software development. *Theory and Practice of Object Systems*, 2(1):3–13, 1996.
- [SH05] H. Störrle and J.H. Hausmann. Towards a formal semantics of UML 2.0 activities. In P. Liggesmeyer, K. Pohl, and M. Goedicke, editors, *Proceedings of the Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik*, volume 64 of *Lecture Notes in Informatics*, pages 117–128, Essen, Germany, 2005. Gesellschaft für Informatik.
- [WEAH05] M.T. Wynn, D. Edmond, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Achieving a general, formal and decidable approach to the OR-join in workflow using Reset nets. In G. Ciardo and P. Darondeau, editors, *Proceedings of the 26th International Conference on Application and Theory of Petri nets and Other Models of Concurrency (Petri Nets 2005)*, volume 3536 of *Lecture Notes in Computer Science*, pages 423–443, Miami, USA, 2005. Springer-Verlag.
- [Wor95] Workflow Management Coalition. Reference model — the workflow reference model. Technical Report WFMC-TC-1003, 19-Jan-95, 1.1, 1995. <http://www.wfmc.org/standards/docs/tc003v11.pdf>.

Translating Colored Control Flow Nets into Readable Java via Annotated Java Workflow Nets

Kristian Bisgaard Lassen and Simon Tjell

Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
`{k.b.lassen,tjell}@daimi.au.dk`

Abstract. In this paper, we present a method for developing Java applications from Colored Control Flow Nets (CCFNs), which is a special kind of Colored Petri Nets (CPNs) that we introduce. CCFN makes an explicit distinction between the representation of: The system, the environment of the system, and the interface between the system and the environment. Our translation maps CCFNs into Annotated Java Workflow Nets (AJWNs) as an intermediate step, and these AJWNs are finally mapped to Java. CCFN is intended to enforce the modeler to describe the system in an imperative manner which makes the subsequent translation to Java easier to define. The translation to Java preserves data dependencies and control-flow aspects of the source CCFN. This paper contributes to the model-driven software development paradigm, by showing how to model a system, environment, and their interface, as a CCFN and presenting a fully automatic translation of CCFNs to readable Java code.

1 Introduction

In this paper, we document an approach to automatic generation of executable Java programs based on Colored Petri Net [13, 14] (CPN) models with a clear distinction between the environment and system domains [19]. We will define these restricted CPNs as Colored Control Flow Nets (CCFNs).

The approach that this paper presents is intended to contribute to the model-driven software paradigm by allowing the user to build, simulate, and analyze CCFN models in tools such as CPN Tools [9, 14], and later to automatically translate the CCFN models into Java code. The purpose of the code generation we present in this paper is not to produce the end product - i.e. the final implementation - but rather to take a step in the direction of tool-based support for generating rapid prototypes.

CPN models are useful for expressing functional requirements for reactive systems by representing required behavior at a high level of abstraction [10, 11]. Reactive systems [20] are a special class of computer systems that are characterized by a close coupling with the surrounding environment through an interface of sensors and actuators. Typical examples of reactive systems include vending machines, elevator controllers, and cruise controllers.

When a reactive system is modeled it is also necessary to model the parts of the environment interacting with the reactive system. The reactive system interacts in a predictive manner with its environment by exchanging observable events and state changes. The reason why it is generally necessary to model both the system and the environment is that a model of the system alone would be inactive without the simulation of incoming stimuli from the environment to which the system can react. This property is common to all reactive systems: No spontaneous behavior is exhibited. When we use the models for abstract representation of behavior, we want to be able to execute scenarios of interaction between the environment and system domains and this is why both domains must be represented in the model.

Our intention is that the translation generates readable Java. By readable, we mean that the generated Java should be readable in the sense that the translation builds a Java program, where behavioral constructs such as sequence, choice, or parallelism, are implemented in the same way as a programmer would have done. It should therefore be more easy to understand, edit, and maintain the generated Java code.

We do not map CCFNs directly to Java, but instead map CCFNs to Annotated Java Workflow Nets (AJWN) (we define AJWN later in this paper), and map AJWNs to Java as shown by the translations T_0



Fig. 1. Two-phase approach for mapping CCFN to Java.

and $T1$ in Figure 1. $T0$ maps the control-flow of the CCFN to a AJWN, which is essentially a Workflow net [1] (WF-net) where each transition is associated with a Java statement. The focus of this step is to map construct in the CCFN to Java statements, since the translation of the control-flow is straight-forward by the way CCFN is defined. $T1$ focus on mapping various behavioral constructs in the AJWN control-flow to a Java code. We found that this division of concerns made the translation more smooth and extensible, than if we had mapped CCFN directly to Java.

Our definition of CCFN will encourage and force the modeler to build models that behave like imperative programs. Transitions will correspond to statements and what the statements can do is to route the control-flow, and either update the state by setting the value of a variable, read an event, or write an event. In this paper, when we refer to the imperative paradigm, we talk about a situation where only those statement types are possible and control-flow is composed of statements.

Besides making a translation from CCFN to Java we have made a translation that is extensible with respect to the collection of Java statements and Java control-flow constructs you want to be able to map to. In this paper, we have chosen a sensible subset of Java to map to, but it is possible to extend either $T0$ or $T1$ to handle an even bigger subset of Java statements and control-flow constructs, as we discuss throughout paper.

The paper is structured as follows: in Section 2 we introduce and define the class of CPNs called CCFNs that we use for modeling reactive systems and their environments; Section 3 presents the intermediate language AJWN used in the translation; Sections 4 and 5 describe and define the translation algorithm respectively from CCFNs to AJWNs and AJWNs to Java code; in Sections 6 we discuss related work; and finally in Section 7 we conclude and present future work.

2 Modeling Reactive Systems with CPNs

In this section we describe how we model reactive systems, and how we represent these using CPNs. In Section 2.1 we motivate and explain how we wish to model reactive systems using CPNs and in Section 2.2 we introduce CCFNs that we will use for modeling reactive systems.

2.1 Colored Petri Nets and Reactive Systems

In Figure 2(a) we show a simple way to model reactive systems in CPNs. There are basically three parts: An environment modeled; a system; and finally some interface modeled as places. In the Figure 2(a) the interface is simply a single place where the tokens represent messages. In reactive systems there is often a clear distinction between events sent to the system and events sent to the environment. This distinction is reflected in the model by dividing the interface place in Figure 2(a) into two places that model each of these aspects as shown in Figure 2(b). In this paper, we will use CCFN to model both the environment and system, so that the environment and its interface is a CCFN model and the system and its interface is a CCFN model.

We generalize the idea of Figure 2(b) to include more than one kind of environment and system, by adding more substitution transitions; see Figure 2(c). This generalization allows a part of a model to represent both a system and an environment - i.e. the model of an environment may itself represent a system connected to yet another environment and so forth. In our approach we allow the modeler to model a reactive system as shown in Figure 2(c), and we are then able to translate each model of environment/system into Java code.

It should be noted that while we require that the parts of a model that are to be translated into Java code should be expressed by means of CCFN, it is still possible to express other parts using the full CPN language. For example, it would be possible to express the environment domain as a traditional CPN model and the system as a CCFN model. The only requirement in this context is that the interaction between these two domains should be performed through interface places as specified by CCFN.

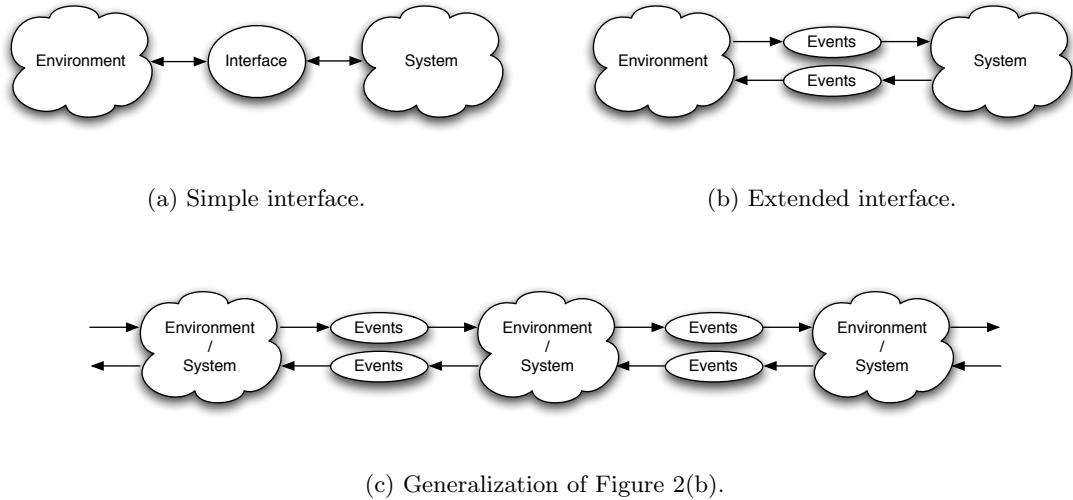


Fig. 2. Different ways to model reactive systems using CPNs.

2.2 Colored Control Flow Nets

CCFNs is designed to facilitate two design aspects: (1) Reactive systems support, as well as (2) support for building models in an imperative fashion. The reactive system design support that CCFNs offer is the same as what is presented in Figure 2(c), i.e. we allow the user to model multiple environments/systems and link them together using places to model events going in and out of the environment/system.

The reason why the second aspect of building imperative models is so important is that it makes the translation from CCFN to readable Java code (Java is imperative) more manageable. If we did not consider this aspect, we would have to map CPNs that contain non-imperative constructions to imperative constructions in Java, which is outside the scope of our research. It is through our definition of CCFNs, which we will come to shortly, that we allow the modeler to model reactive systems, while we enforce that he use an imperative style of modeling.

Before we give a formal definition of CCFN, let us look at the small example in Figure 3; notice it is not important to know exactly what the arc inscriptions mean. The CCFN models a small process that will read an event “input” and assign the value associated with “input” to the variable y ; do an operation square on y and assign the result to x ; and finally, generate an “output” event associated with the value of x until $x \geq 10$. When x reaches this value the process transition *Quit* becomes enabled and *Receive* disabled. The access to the variable x is performed through a function (*get*) that returns the value of a specific variable found in a **STATE** value. Each CCFN has a single place with the color set **STATE** and we will simply refer to this place as the *state place*. Two examples of operators *lt* ($<$) and *gte* (\geq) are used to evaluate the value of the variable in the guards. Later we will see that we distinguish between five types of transitions: *Enter process* and *Leave process* are a special kinds of transitions that are used to set up the CCFN so that it has

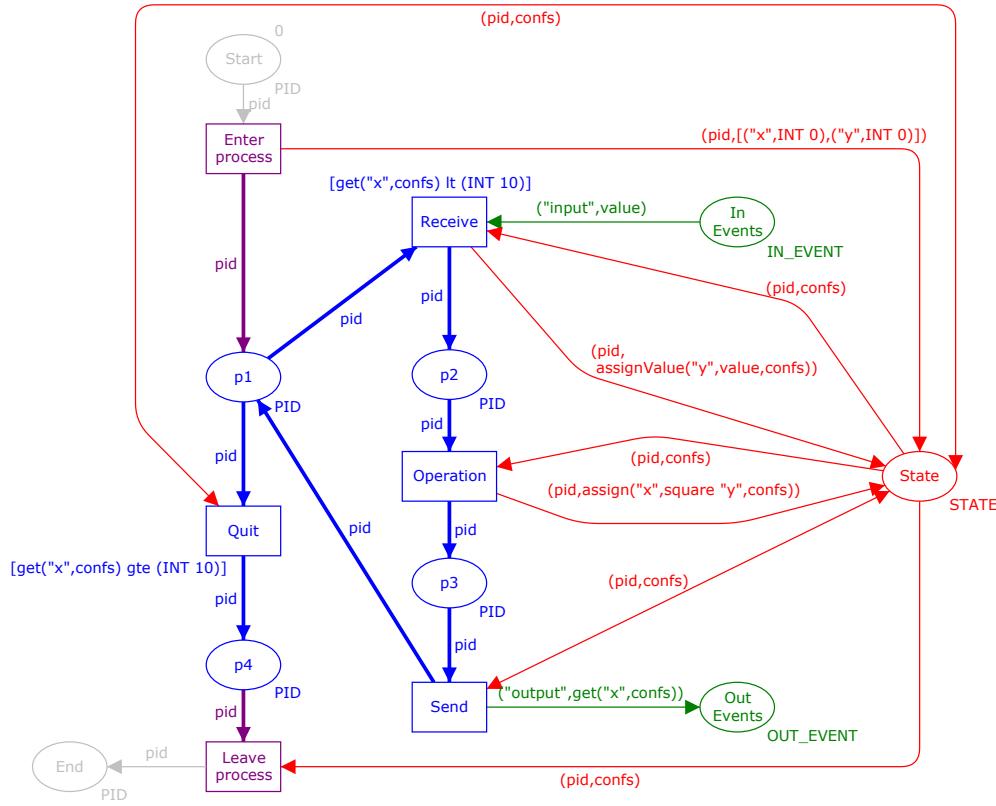


Fig. 3. Small example of a CCFN.

single entry and exit points; Operation and Quit we call *state update* transitions, since they access the state place; Receive is called a *read event* transition since it reads a value from an IN_EVENT place and write the result to the state place; Send is a *write event* transition since it reads a value from the state place and puts it on a OUT_EVENT place; and, finally there is a transition type that is not in this example, and it is called a *silent step* transition, and it simply a place that is only connected to places of type PID. silent step transition do not update the state place or read or write events, it is simply a silent step in the execution of the process; hence the name.

Before we define CCFN, we first introduce the allowed place types in a CCFN in Definition 1, and later the allowed arc inscriptions in Definition 2.

Definition 1 (CCFN place types). A place may either have the color set type PID (a process id), STATE, IN_EVENT, or OUT_EVENT, which are defined as follows:

```

colset PID = INTEGER
colset STATE = product PID * CONFIGURATION_LIST
colset IN_EVENT = CONFIGURATION
colset OUT_EVENT = CONFIGURATION

```

where

```

colset IDENTIFIER = STRING
colset VALUE = union STRING:STRING + INT:INT + BOOL:BOOL + VOID
colset CONFIGURATION = product IDENTIFIER * VALUE
colset CONFIGURATION_LIST = list CONFIGURATION

```

Notice that **CONFIGURATION_LIST** is similar to the concept of *environment* in a program execution, however, we choose not to use this term, in order to avoid confusion with the term *environment* used in the context of reactive systems. Another important point is that it is possible to define two different variables with the same identifier in a **CONFIGURATION_LIST**. It is up to modeler to ensure that this does not happen, either by hand or by formal analysis. The intentions of the place types are the following: **PID** will be used to model the control flow of a CCFN; **STATE** is for modeling the state of a process - i.e. the state of either the system or the environment - expressed in CCFN; **IN_EVENT** and **OUT_EVENT** are used to model incoming and outgoing events for the respective CCFN.

Definition 1 states that a place can only contain a process identifier, which is simply an integer; a state, which is a pair that binds a process identifier to a configuration; or, either an input or output event that is represented as a pair of identifier and value. Examples of process identifiers are $0, 1, 2, \dots$; states are $(29, [("x", INT(399)), ("y", BOOL(false)), ("z", STRING("foo"))])$, meaning that the process with identifier 29 is in a configuration list with three declared variables x, y , and z that are assigned values integer 399, boolean false, and string "foo"; and finally, an input or output event $("event1", VOID)$ for the event "event1" and no value, and $("event2", INT 4)$ for the event "event2" with the integer value 4.

In the following, we will write $[X]$ to denote any expression where $Type([X]) = X$, \mathbb{B} the boolean type, and \mathbb{S} for the string type. We are now ready to introduce the allowed arc inscriptions in a CCFN. The definition will be followed by a more detailed description of its elements.

Definition 2 (Allowed arc inscriptions in CCFNs). *The only allowed arc inscriptions in a CCFN are*

- (i) pid (used to describe a process id); color set **PID**.
- (ii) $(pid, confs)$ (used to describe a process and its configuration $confs$); color set **STATE**.
- (iii) $(pid, assign([IDENTIFIER], function, confs))$ (a pid and an $assign$ expression that evaluates to an updated configuration, where the variable specified by the identifier is assigned the result of calling the function); color set **STATE**. If variable is the empty string the result of the evaluation is not assigned to anything in the configuration.
- (iv) $(pid, assignValue([IDENTIFIER], [VALUE], confs))$ (a variation of (iii) that simply assign the variable specified by the identifier to the value in the configuration).
- (v) $([IDENTIFIER], value)$ (an identifier for a named event, and value has type **VALUE**); color set **EVENT**.
- (vi) $([IDENTIFIER], [VALUE])$ (an identifier for a named event, and an expression with type **VALUE**); color set **EVENT**.
- (vii) $([IDENTIFIER], get([IDENTIFIER], confs))$ (an identifier for a named event, get is a function that returns that value of the variable specified by an identifier); color set **EVENT**.

The reason why it is necessary to restrict the allowed arc inscriptions is to ease the translation of the CCFN as we will see later. Another consequence of the definition is that we only allow the modeler to update the token values by using imperative constructions, such as **assign**, **assignValue** (a special case of **assign**), and **get**.

The signature of the function **assign** is $IDENTIFIER * (\alpha \rightarrow CONFIGURATION_LIST \rightarrow VALUE) * CONFIGURATION_LIST \rightarrow CONFIGURATION_LIST$; i.e. **assign** takes a triple: the name of a variable; a function that when applied to a value with type α and a configuration list, then calculates a value; and, the current configuration list, so that **assign** returns the updated configuration. The signature of the function **assignValue** is $IDENTIFIER * VALUE * CONFIGURATION_LIST \rightarrow CONFIGURATION_LIST$. The signature of **get** is $IDENTIFIER * CONFIGURATION_LIST \rightarrow VALUE$. Examples of arc inscriptions are: pid , $(pid, confs)$, $assign("x", negate, confs)$, $("button pressed", [("isOn", BOOLEAN(false))])$, and $get("x", confs)$.

We are now ready to define CCFNs in Definition 3. For the sake of readability, we have omitted the arguments to **get**, **assign**, **assignValue** where used. See Definition 2 for a definition of these. In this paper, we refer to the preset and preset of a node $x \in P \cup T$ as $\bullet x = \{y \in P \cup T | (y, x) \in N(A)\}$ and $x\bullet = \{y \in P \cup T | (x, y) \in N(A)\}$.

Definition 3 (Colored Control Flow Nets (CCFNs)). *A non-hierarchical Colored Petri Net is a tuple $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ satisfying [13, Definition 2.5]. CPN is a Colored Control Flow Net (CCFN) iff*

- (i) (Color sets) $\Sigma = \{PID, STATE, IN_EVENT, OUT_EVENT\}$. These color sets are defined in Definition 1.
- (ii) (Places) $P = \{p_{start}, p_{end}\} \cup P_{pid} \cup \{p_{state}\} \cup P_{in_event} \cup P_{out_event}$, where
- $\forall A, B \in \{\{p_{start}, p_{end}\}, P_{pid}, \{p_{state}\}, P_{in_event}, P_{out_event}\} : A \neq B \Rightarrow A \cap B = \emptyset$.
 - $|P_{pid}| \geq 1$.
- (iii) (Transitions) $T = \{t_{start}, t_{end}\} \cup T_{regular}$, where
- $\{t_{start}, t_{end}\} \cap T_{regular} = \emptyset$.
- (iv) (Arcs and node function) $N(A) = \{(p_{start}, t_{start}), (t_{start}, p_{state}), (p_{state}, t_{end}), (t_{end}, p_{end})\} \cup F$, where
- $\{(p_{start}, t_{start}), (t_{start}, p_{state}), (p_{state}, t_{end}), (t_{end}, p_{end})\} \cap F = \emptyset$.
 - $\forall a_1, a_2 \in A : a_1 \neq a_2 \Rightarrow N(a_1) \neq N(a_2)$ (N is injective).
 - $\forall (x, y) \in F : x \notin \{p_{start}, t_{end}, p_{end}\} \wedge y \notin \{p_{start}, t_{start}, p_{end}\}$.
 - $\forall (p_{state}, t) \in F : (t, p_{state}) \in F$.
 - $\forall p_{in} \in P_{in_events}, \forall p_{out} \in P_{out_events}, \exists t \in T : (p_{in}, t), (t, p_{out}) \in F$.
- (v) (Color function) $C(p) = \begin{cases} PID & \text{if } p \in \{p_{start}, p_{end}\} \cup P_{pid} \\ STATE & \text{if } p = p_{state} \\ IN_EVENT & \text{if } p \in P_{in_event} \\ OUT_EVENT & \text{if } p \in P_{out_event} \end{cases}$
- (vi) (Guard function) $G(t) = \begin{cases} \text{get } R \ v, \ R \text{ is a boolean operator} \wedge \text{Type}(v) = \text{VALUE if } \bullet t = \{p\} \wedge |p \bullet| > 1 \\ \text{true} & \text{otherwise} \end{cases}$
- (vii) (Arc expression function) In the restriction of E we assume that pid and $confs$ are declared variables where $\text{Type}(pid) = PID$, $\text{Type}(confs) = CONFIGURATION_LIST$, and $\text{Type}(value) = VALUE$.
- $$E(a) = \begin{cases} pid & \text{if } N(a) \in P' \times T \cup T \times P', \text{Type}(P') = PID \\ (pid, confs) & \text{if } N(a) = (p_{state}, t_{end}) \vee N(a) = (t, p_{state}), t \in T_{regular} \\ (pid, [CONFIGURATION]) & \text{if } N(a) = (t_{start}, p_{state}) \\ (pid, assignValue) & \text{if } N(a) = (t, p_{state}) \wedge (p, t) \in P_{in_event} \times T_{regular} \\ (pid, assign) & \text{if } N(a) = (t, p_{state}) \\ (\$, value) & \text{if } N(a) \in P_{in_event} \times T_{regular} \\ (\$, [VALUE]) & \text{if } N(a) \in T_{regular} \times P_{out_event} \\ (\$, get) & \text{if } N(a) \in T_{regular} \times P_{out_event} \end{cases}$$
- (viii) (Initialization function) $I(p) = \begin{cases} [PID_{ms}], p = p_{start} \\ \emptyset, \text{otherwise} \end{cases}$

In the following, we will explain each part in the definition. (i) It is possible to use four different kinds of color sets in a CCFN and these are as in Definition 1.

(ii) We say that there are five sets of places. Notice that there are only one start, one end and one state place. The single start and end places are to describe that the control-flow starts in a single point, and ends in a single point. The single state place describe that each CCFN has a global state that is stored on the place p_{state} . A CCFN must have at least one PID place to route the control-flow and any number of in and out event places.

(iii) There are two kinds of transition: Those that we use to initiate and end the process, and a set of transition $T_{regular}$ which we will refer to as regular transitions.

(iv) The arcs consists of two parts where the first is used to setup the CCFN process so the start place is linked to the start transition and so on. Then we state that it is not possible to have two arcs between the same pair of nodes to simplify the function N ; this means the function N is now injective and especially that N^{-1} is always uniquely defined. Next we require that it is not possible to make arcs from regular transitions to one of the place p_{start} and transition t_{start} and it is not possible to have an arc going away from the transition t_{end} , except an arc to p_{end} . Next we say that if there is an arc going from p_{state} to a transition t there must be an arc going back from the same t . And, finally if there is an arc from IN_EVENT place to a transition t there may not be an arc to a OUT_EVENT and vice versa. Figure 4 gives four examples of how regular transition may be used in a CCFN, and we will use the terms *silent step*, *state update*, *read event*, and *write event* to refer to each of these. Definition 4 formalizes these four possibilities.

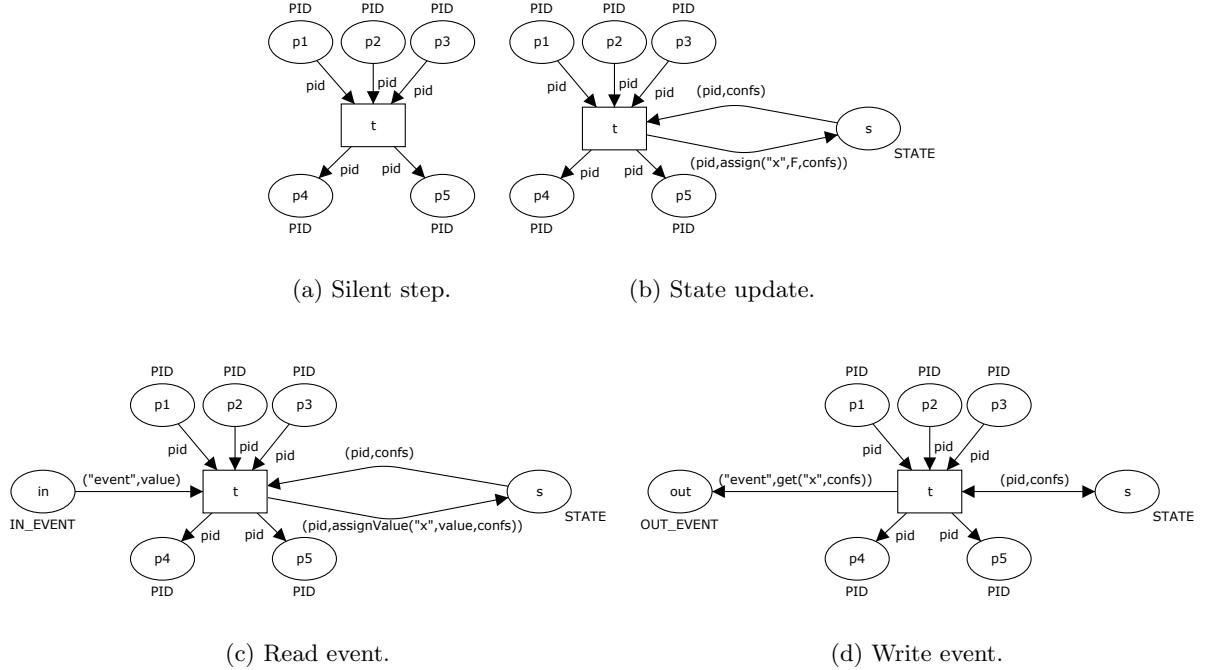


Fig. 4. Examples of how a regular transition can be used in a CCFN model.

(v) As we explained after Definition 1 the places are given place types corresponding to how we intend them to be used.

(vi) Transitions that are part of a free choice may have a boolean expression as guard; others have the guard true.

(vii) Figure 4 shows examples of the different cases for the arc expression function. Definition 2 explains the intention of these.

(viii) We initialize the CCFN by adding some multiset of PID on the place p_{start} .

Definition 4 (Transitions in CCFNs). Let be $CCFN = (\Sigma, P, T, A, N, C, G, E, I)$ be a Colored Control Flow Net as in Definition 3. Definition 3 states that there are four distinct sets of transitions that we name as follows:

Silent step: $\{t \in T \mid \forall (p_{in}, t), (t, p_{out}) \in N(A) : C(p_{in}) = C(p_{out}) = PID\}$.

State update: $\{t \in T \mid \exists (t, p_{state}), (p_{state}, t) \in N(A),$

$\forall (p_{in}, t), (t, p_{out}) \in N(A) \setminus \{(t, p_{state}), (p_{state}, t)\} : C(p_{in}) = C(p_{out}) = PID\}$.

Read event: $\{t \in T \mid \exists (t, p_{state}), (p_{state}, t), (p_{in_event}, t) \in N(A) : C(p_{in_event}) = IN_EVENT,$

$\forall (p_{in}, t), (t, p_{out}) \in N(A) \setminus \{(t, p_{state}), (p_{state}, t), (p_{in_event}, t)\} : C(p_{in}) = C(p_{out}) = PID\}$.

Write event: $\{t \in T \mid \exists (t, p_{state}), (p_{state}, t), (t, p_{out_event}) \in N(A) : C(p_{out_event}) = OUT_EVENT,$

$\forall (p_{in}, t), (t, p_{out}) \in N(A) \setminus \{(t, p_{state}), (p_{state}, t), (p_{out_event}, t)\} : C(p_{in}) = C(p_{out}) = PID\}$.

A CCFN where all places of type STATE, IN_EVENT, and OUT_EVENT are removed should correspond to a Sound Workflow Net (sound WF-net) as defined in [2]; we refer to this version of CCFN as $CCFN_{PID}$. Although WF-nets have been defined for classical Petri nets it is easy to generalize the definition to CPN as discussed in [3, 4]. The basic requirement is that there is one source place and one sink place and all other nodes (places and transitions) are on a path from source to sink. We can test soundness of CCFN by testing the net \overline{CCFN}_{PID} . \overline{CCFN}_{PID} is constructed from CCFN by removing all places with types IN_EVENT,

`OUT_EVENT`, and `STATE`, and by adding a new transition t and add two arcs from the end place to t , and from t to the start place, and finally settings the arc expression of the two arcs to pid . Moreover, we set all guard expressions to true. If all transitions in \overline{CCFN}_{PID} are live and if all places are bounded then we say that the CCFN is sound. Moreover, if all places have multi-set upper bounds $^{++} \sum_{pid \in PID} pid$ we say that the CCFN is safe. In the translation of CCFN, we will assume that the input CCFN is safe and sound in the sense described here.

3 Annotated Java Workflow Nets

In our translation we will need to transform CCFNs to an intermediate form. This form is expressed in a special kind of Petri nets that we call Annotated Java Workflow Nets (AJWNs); see Definition 5. Note that the framework for translating annotated WF-nets was introduced in [7], we refer to Section 6 on a discussion on how our approach related to what was done in [7].

Definition 5 (Annotated Java Workflow Nets). *An Annotated Java Workflow Net is a tuple $AJWN = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$, where*

- P is the set of places.
- T is the set of transitions.
- $F = (P \times T) \cup (T \times P)$ is a flow relation.
- $\tau : T \rightarrow JS$, where JS is the set of Java statements.
- $\tau_G : T \rightarrow BE$, where BE is a boolean expression.
- $\Gamma = \{(id_1, type_1, value_1), \dots, (id_n, type_n, value_n)\}$ is an initial configuration.
- $\Pi = (\{channel_1^{in}, \dots, channel_n^{in}\}, \{channel_1^{out}, \dots, channel_m^{out}\})$ a pair of in and out channels that carry events.
- A single input place p_s exists s.t. $\bullet p_s = \emptyset$ - i.e. a transition with no input transitions.
- A single output place p_e exists s.t. $p_e \bullet = \emptyset$ - i.e. a transition with no output transitions.
- $\forall x \in P \cup T : (x, p_{end}) \in F^*$ (F^* is the transitive closure of F).
- The Workflow net (P, T, F) is safe and sound [1].

In Figure 5 we see a small example of an AJWN. As we will see later the example is actually a translation that we define later of the CCFN of Figure 3.

4 Translation of Colored Control Flow Nets to Annotated Java Workflow Nets

In our approach we map CCFNs to AJFNs, and map the AJFNs to Java. In Definition 6 we give a translation of the first step from CCFNs to AJFNs. An explanation is given after the definition. For an application of the definition we refer to Figure 5, where the AJWN there is a translation of the CCFN in Figure 3.

Definition 6 (Translation of CCFNs to AJFNs). *Let $CCFN = (\Sigma, P^C, T^C, A, N, C, G, E, I)$ be a safe and sound Colored Control Flow Net. We define the corresponding Annotated Java Flow Net $ACFN = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$ as follows:*

- (i) P (places): $P := \{p \in P^C | C(p) = PID\}$.
- (ii) T (transitions): $T := T^C$.
- (iii) F (flow relation): $F := \{(p, t), (t', p') \in N(A) | C(p) = C(p') = PID \wedge t, t' \in T^C\}$.
- (iv) τ (Java annotations): See Definition 4 for a definition of the four possible ways a transition can be used in a CCFN.

$$\tau(t) = \begin{cases} ";" & \text{if silent step} \\ "x = F(x_1, \dots, x_n);;" & \text{if state update} \wedge E(N^{-1}(t, p_{state})) = (pid, assign("x", F(x_1, \dots, x_n), confs)) \\ "F(x_1, \dots, x_n);;" & \text{if state update} \wedge E(N^{-1}(t, p_{state})) = (pid, assign("", F(x_1, \dots, x_n), confs)) \\ "x = value;" & \text{if state update} \wedge E(N^{-1}(t, p_{state})) = (pid, assignValue("x", value, confs)) \\ "x = read("e", p_{in});;" & \text{if read event} \wedge E(N^{-1}(p_{in}, t)) = ("e", value) \\ "write("e", x, p_{out});;" & \text{if write event} \wedge E(N^{-1}(t, p_{out})) = ("e", get("x", confs)) \\ "write("e", value, p_{out});;" & \text{if write event} \wedge E(N^{-1}(t, p_{out})) = ("e", value) \end{cases}$$

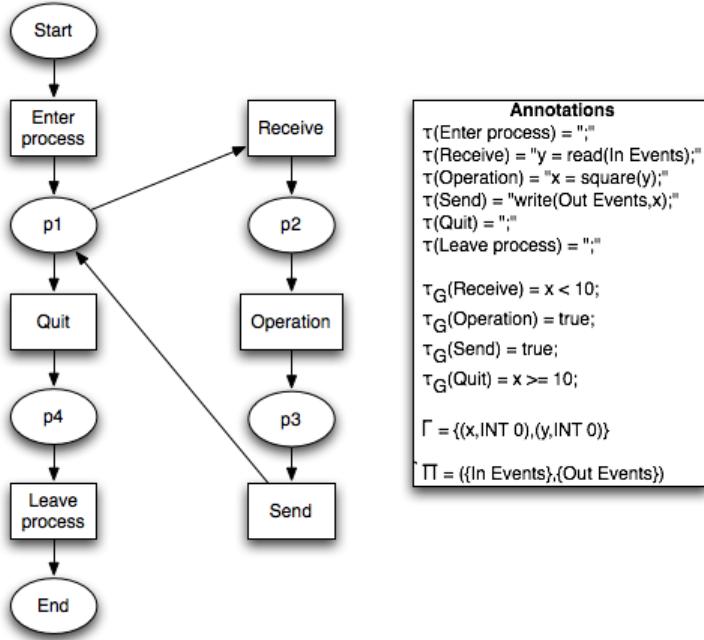


Fig. 5. AJWN of the CCFN in Figure 3.

(v) τ_G (Guards):

$$\tau_G(t) = \begin{cases} x \ R \ [VALUE], \ R \text{ is a boolean operator if } G(t) = \text{get}("x", \text{conf}) \ R \ [VALUE] \\ \text{true} \quad \text{otherwise} \end{cases}$$

- (vi) Γ (Initial environment): The initial configuration in a CCFN model is specified on the arc $(t_{\text{start}}, p_{\text{state}})$ and it is on the form $[(id_1, value_1), \dots, (id_n, value_n)]$ (see Definition 3). For each pair $(id, value)$ in the initial environment of CCFN, set $\Gamma(id) = value$.
- (vii) Π (Streams): $\Pi = (\{\text{java_in_stream}(p) | p \in P_{\text{in_events}}\}, \{\text{java_out_stream}(p) | p \in P_{\text{out_events}}\})$, where *java_in_stream* and *java_out_stream* are streams that implement the `InputStream` and `OutputStream` interfaces in Java.

The translation is rather straightforward if we look at places and transitions in steps (i) and (ii). In (iii) we map the arcs of the CCFN, where we map all arcs that are not connected to the place p_{state} or any of the in and out event places. (iv) We generate the Java annotations for each transition, by looking at what type of transition it is, and then by looking at the sort of arc expressions used on arcs around it: *silent step* transitions correspond to doing nothing and it is simply translated into the empty statement $";";$ *state update* transitions read the current state from the state place and update a value in the current state either by a constant or by calling a function, therefore the statement is either $"x = F(x_1, \dots, x_n);"$, for some variable x and function F with input x_1, \dots, x_n , or the statement $"x = C;"$ for some variable x and constant C ; *read event* and *write event* transitions are handled in a similar fashion as the two preceding types. (v) There are two allowed guard expressions in a CCFN: *true*, and expression on the form $\text{get}([\text{IDENTIFIER}], \text{conf}) \ R \ \text{value}$ (examples of such an expression is $\text{get}("x", \text{conf})$). (vi) and (vii) simply tell us how to find the initial configuration and which streams are defined. In (vii) we assume that some stream types are defined in the Java code already, so we do not assume they are of type `java.io.InputStream` or other specific types of streams. All we assume is that the input streams are blocking - i.e. a reading call would block if no data is available for reading through the stream and will continue to block until data becomes available.

The time complexity of the translation in Definition 6 is $O(|P \cup T|)$. It simply does a linear sweep of the nodes in the CCFN in mapping to AJWN.

5 Translation of Annotated Java Workflow Nets to Readable Java

Before we describe the translation let us first introduce some definitions. Definition 7 defines how we understand the union of two functions and when a union is possible. This definition is important for the definition of projection in Definition 8.

Definition 7 (Union of two functions). Let $F : A_1 \rightarrow B, G : A_2 \rightarrow B, A_1, A_2 \subseteq A$ and require that $\forall x \in A_1 \cap A_2 : F(x) = G(x)$. Then we define the union of F and G as

$$(F \cup G)(x) = \begin{cases} F(x), & x \in A_1 \setminus A_2; \\ G(x), & \text{otherwise.} \end{cases}, \forall x \in \text{dom}(F) \cup \text{dom}(G)$$

Definition 8 states what we mean by a projection, i.e. a restriction, of a AJWN. It simply defines a projection as the subnet you get by selecting a set of nodes, and then include all the arcs that have source and target in that set. The annotation functions of the AJWN are restricted in a similar fashion.

Definition 8 (Projection). Let $AJWN = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$ and $AJWN' = (P', T', F', \tau', \tau'_G, \Gamma', \Pi')$ be Annotated Java Workflow Nets and $X \subseteq P \cup T$ a set of nodes. $AJWN|_X = (P \cap X, T \cap X, F \cap (X \times X), \tau|_X, \tau_G|_X, \Gamma|_X, \Pi|_X)$ is the projection of $AJWN$ onto X . $AJWN \cup AJWN' = (P \cup P', T \cup T', F \cup F', \tau \cup \tau', \tau_G \cup \tau'_G, \Gamma \cup \Gamma', \Pi \cup \Pi')$ is the union of $AJWN$ and $AJWN'$.

A component as in Definition 9 is a special kind of projection. It is a set of nodes in a AJWN that contain a source and a sink and where all nodes in the component are on a path between source and sink in the AJWN.

Definition 9 (Component). Let $AJWN = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$ be an Annotated Java Workflow Net. C is a component of $AJWN$ if and only if

- (i) $C \subseteq P \cup T$,
- (ii) there exist different source and sink nodes $i_C, o_C \in C$ such that
 - $\bullet(C \setminus \{i_C\}) \subseteq C \setminus \{o_C\}$,
 - $(C \setminus \{o_C\})\bullet \subseteq C \setminus \{i_C\}$, and
 - $(o_C, i_C) \notin F$.

Definition 10 (Component notations). Let $AJWN = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$ be an Annotated Java Workflow Net and let C be a component of $AJWN$ with source i_C and sink o_C . We introduce the following notations and terminology:

- C is a PP-component if $i_C \in P$ and $o_C \in P$,
- C is a TT-component if $i_C \in T$ and $o_C \in T$,
- C is a PT-component if $i_C \in P$ and $o_C \in T$,
- C is a TP-component if $i_C \in T$ and $o_C \in P$,
- $\overline{C} = C \setminus \{i_C, o_C\}$,
- $PN||_C =$
 - $PN|_C$ if $i_C \in P$ and $o_C \in P$,
 - $PN|_C \cup (\{p_{(i,C)}\}, \{i_C\}, \{p_{(i,C)}, i_C\}) \cup (\{p_{(o,C)}\}, \{o_C\}, \{o_C, p_{(o,C)}\})$ if $i_C \in T$ and $o_C \in T$,¹
 - $PN|_C \cup (\{p_{(o,C)}\}, \{o_C\}, \{o_C, p_{(o,C)}\})$ if $i_C \in P$ and $o_C \in T$,
 - $PN|_C \cup (\{p_{(i,C)}\}, \{i_C\}, \{p_{(i,C)}, i_C\})$ if $i_C \in T$ and $o_C \in P$.

¹ Note that $p_{(i,C)}$ and $p_{(o,C)}$ are the (fresh) identifiers of the places added to make a transition bordered component place-bordered.

- $[PN]$ is the set of non-trivial components of PN , i.e., all components containing two or more transitions.

Definition 10 introduces some terminology that is useful in the context of components, such as the projection $PN||_C$. This projection is similar to $PN|_C$ in Definition 8, where a component is padded with places if it is a TT-, PT-, or a TP-component, so that the new projection always yields a AJWN that is bordered by places. It also introduces the concept of non-trivial component denoted $[AJWN]$ for some $AJWN$.

Definition 11 shows how a component may be removed from an AJWN and replaced by a single transition. In [7], we show that a similar kind of folding yields a sound and safe AJWN if the original AJWN was sound and safe.

Definition 11 (Fold). Let $AJWN = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$ be an Annotated Java Workflow Net and let C be a non trivial component of $AJWN$ (i.e., $C \in [AJWN]$). Function **fold** replaces C in $AJWN$ by a single transition t_C , i.e., $\text{fold}(AJWN, C) = (P', T', F', \tau', \tau'_G, \Gamma', \Pi')$ with:

- $P' = P \setminus \overline{C}$,
- $T' = (T \setminus C) \cup \{t_C\}$,
- $F' = (F \cap ((P' \times T') \cup (T' \times P'))) \cup \{(p, t_C) | p \in P \cap (\{i_C\} \cup \bullet i_C)\} \cup \{(t_C, p) | p \in P \cap (\{o_C\} \cup o_C \bullet)\}$.
- $\tau' = \tau[t_C := \text{translate}(C)]$ (**translate** generates Java code from C ; for more information on how **translate** works see Section 5.1).
- $\tau'_G = \tau_G$.
- $\Gamma' = \Gamma$
- $\Pi = \Pi$

We are now ready to define the algorithm by which we will translate AJWFs. We will explain the steps of the algorithm in detail after the definition. The overall idea behind the algorithm is to take a safe and sound AJWN and reduce it by matching a component, replacing the component by a transition using **fold** that also extends the annotation function by the translation of the component. The predefined components referred to in the algorithm are defined in the next section along with a description of how they are defined.

Definition 12 (Algorithm). Let $AJWN = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$ be a safe and sound Annotated Java Workflow Net.

- (i) $X := AJWN$
- (ii) while $[X] \neq \emptyset$ (i.e., X contains a non-trivial component)²
 - (iii) If there is a predefined component $C \in [X]$, select it and goto (vi).
 - (iv) If there is a component $C \in [X]$ that appears in the component library, select it and goto (vi).
 - (v) Select a component $C \in [X]$ to be manually mapped into Java and add it to the component library.
 - (vi) $X := \text{fold}(AJWN, C)$ and return to (ii).
 - (vii) Output the Java code attached to the transition in X .

In the next section, we will introduce the predefined component types and these are: SEQUENCE, CHOICE, WHILE, and PARALLEL. The sequence by which the component types are listed is also the precedence that the algorithm use when selecting a predefined algorithm. If the algorithm cannot find any SEQUENCE-component it will look for CHOICE-component, and so on.

If we do not match one of these general predefined components we look in a component library, which is basically a collection of pairs consisting of one AJWN and one matching Java translation. If we find a component in the AJWN that matches the AJWN part of an existing library component, we translate the component based on the translation information found in the library - i.e. we use the Java translation of the component as the annotation for the transition used to replace the component in the AJWN. If no library component can be found, after testing predefined component, the algorithm terminates, since no component in the net could be reduced.

² Note that this is the case as long as X is not reduced to a WF-net with just a single transition.

This algorithm can be extended by adding or removing types of predefined and library components, and changing the priority by which the algorithm selects components. For example, the algorithm could be extended to test for PARALLEL-components before SEQUENCE-components.

The time complexity of the algorithm in Definition 12 is $O(|T| \cdot 2^{|P \cup T|})$, because the matching of library components may need to compare any combination of nodes in the library component with the nodes input AJWN; $O(\cdot 2^{|P \cup T|})$. This is the most complex operation. We may need to do the operation in each iteration of the loop, which in worst case may need to run $O(|T|)$ times if we match a component in each iteration. This may seem discouraging at first, but in [16] we did a practical investigation on matching library components for real-world WF-nets, and we found that even with a library that had more than 100 components, processing and matching all components took under one second. This was also partly due to an optimized version of subgraph isomorphism for WF-nets that also would apply in this scenario.

5.1 Component Types and Their Translations

This subsection is devoted to describing component we have chosen to translate the function **translate** that was used in the Definition 11 of **fold**. We have chosen some component types that many people know, but as we discussed earlier the algorithm for translating AJWN to Java is extensible by which component translate, so it is possible to add even more components then the ones we present here.

In Figure 6 we show examples of the different component types that we translate. Later on in this section we give formal definitions of these component types and present a translation of them.

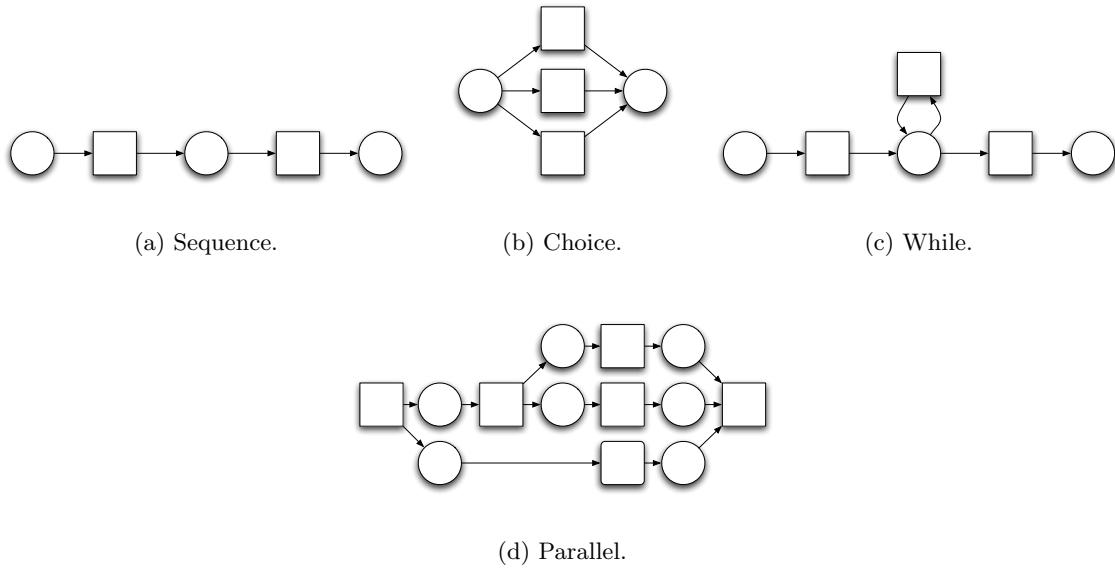


Fig. 6. Different component types.

Definition 13 (MAXIMAL SEQUENCE-component). Let AJWN be a safe and sound Annotated Java Workflow Net, C be a component of AJWN and let $AJWN|_C = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$.

- C is a SEQUENCE-component iff $\forall x \in P \cup T : |\bullet x| \leq 1 \wedge |x \bullet| \leq 1$.
- C is a MAXIMAL SEQUENCE-component is a component that is not contained in any other SEQUENCE-components.

We translate a SEQUENCE-component in a straightforward manner: we sort the transitions in T in a list $[t_1, \dots, t_n]$, where t_i is before t_j iff $(t_i, t_j) \in F^*$. Next, we simply translate the sequence into the Java code: $\tau(t_1); \dots; \tau(t_n);$.

Definition 14 (CHOICE-component). Let $AJWN$ be a safe and sound Annotated Java Workflow Net, C a component of $AJWN$, and $AJWN|_C = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$.

- C is a CHOICE-component iff $P = \{p_i, p_o\}$, $T = p_i \bullet = \bullet p_o$, and $\forall t \in p_i \bullet : t \bullet = \{p_o\}$.

We translate a CHOICE-component with transitions t_1, \dots, t_n by translating $AJWN|_C$ to the Java code: **if** ($\tau_G(t_1)$) $\{\tau(t_1)\}$ **else if** ($\tau_G(t_2)$) $\{\tau(t_2)\}$... **else** $\{\tau(t_n)\};$.

Definition 15 (WHILE-component). Let $AJWN$ be a safe and sound Annotated Java Workflow Net, C a component of $AJWN$, and $AJWN|_C = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$.

- C is a WHILE-component iff $P = \{p\}$, $T = \{t_i, t, t_o\}$ and $F = \{(t_i, p), (p, t), (t, p), (p, t_o)\}$.

We translate a WHILE-component with transitions and places given in Definition 15 as the Java code: $\tau(t_i); \text{while}(\tau_G(t) \wedge \neg \tau_G(t_o)) \{\tau(t)\}; \tau(t_o);$.

Definition 16 (MAXIMAL PARALLEL-component). Let $AJWN$ be a safe and sound Annotated Java Workflow Net, C a component of $AJWN$, and $AJWN|_C = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$.

- C is a PARALLEL-component iff it is an acyclic marked graph; i.e. $\forall p \in P : |\bullet p| \leq 1 \wedge |p \bullet| \leq 1$ and $\forall x \in P \cup T : (x, x) \notin F^*$.
- C is a MAXIMAL PARALLEL-component is a PARALLEL-component that is not contained in a larger PARALLEL-component.

To translate a PARALLEL-component we need to introduce the concept of a clean synchronization and branches in Definition 17. For an example of PARALLEL-components with clean and unclean synchronization please refer to Figure 7.

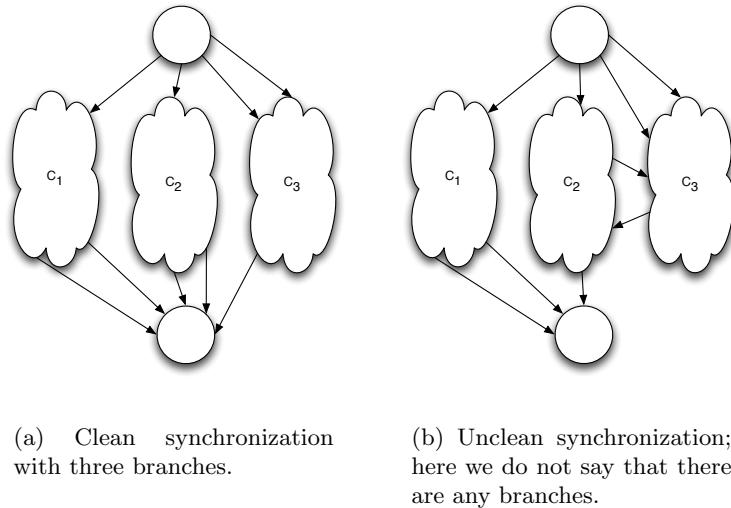


Fig. 7. Two examples of a PARALLEL-components.

Definition 17 (Clean Synchronization and Branches). Let AJWN be a safe and sound Annotated Java Workflow Net, C a component of AJWN, and $AJWN|_C = (P, T, F, \tau, \tau_G, \Gamma, \Pi)$. Assume with no loss of generality that C is a TT-component and that the source and sink are $t_i, t_o \in T$. If for each $p_i \in t_i \bullet$ we can find a $p_o \in \bullet t_o$, such that p_i and p_o are source and sink of a component and p_i, p_o are only used in a single component, then we say that the PARALLEL-component has clean synchronization.

We call the components we find between t_i and t_o branches of $AJWN|_C$, and we say that they are disjoint.

A MAXIMAL PARALLEL-component that only has clean synchronization is easy to translate to Java since it corresponds to a split in the WF-net that is later synchronized and no synchronization happens before this final synchronization. This simply corresponds to starting n Java threads in parallel and waiting for all of them to finish. However, clean synchronization is not always the case in MAXIMAL PARALLEL-component, which is why the following translation become more verbose.

When we translate a MAXIMAL PARALLEL-component, we assume without loss of generality that it has source and sink $t_i, t_o \in T$.

We split the translation of MAXIMAL PARALLEL-component $AJWN|_C$ in two cases: First we look at $AJWN|_C$ with clean synchronization (see Figure 7(a)), and later on we take the other case. In this case $AJWN|_C$ has n branches C_1, \dots, C_n . Here we generate the following Java code:

```

1    $\tau(t_i);$ 
2   Thread  $t_{C_1} = \text{new Thread()} \{ \text{public void run()} \{ \text{translate}(C_1); \} \}$ 
3   ...
4   Thread  $t_{C_n} = \text{new Thread()} \{ \text{public void run()} \{ \text{translate}(C_n); \} \}$ 
5    $t_{C_1}.\text{start}(); \dots; t_{C_n}.\text{start}();$ 
6    $\tau(t_o);$ 

```

In case the $AJWN|_C$ does not have clean synchronization (see Figure 7(b)), we have to do something a little more complicated. Before we begin to explain the steps, let us define two functions $\sigma : T \rightarrow T$ and $\kappa : T \rightarrow \mathcal{P}(T)$ that we will use to define relations between two transitions, σ , a function $\rho : T \rightarrow \text{legal Java identifier}$ to relate transitions to Java thread identifiers and a function $\varphi : T \rightarrow \text{Java code}$ to relate transitions to some Java code. The five steps to translate $AJWN|_C$ are the following:

1. For each join transition t ($|\bullet t| > 1$) we look at p_1 and p_2 , where $\{p_1, p_2\} \in \bullet t$. We introduce fresh place p_f , and fresh transitions t_{f_1}, t_{f_2} and update C by setting $P := P \cup \{p_f\}$, $T := T \cup \{t_{f_1}, t_{f_2}\}$, and $F := (F \cup \{(p_1, t_{f_1}), (p_2, t_{f_2}), (t_{f_2}, p_f), (p_f, t)\}) \setminus \{(p_1, t)\}$. Moreover we set $\sigma(t_{f_2}) = t_{f_1}$. We continue doing this until all join transitions are removed. This step removes all synchronization point and this means that C is converted to a tree where t_i is the root.
2. For each split transition t , we find all reachable transitions t_1, \dots, t_n . Set $\kappa(t) = \{t_1, \dots, t_n\}$. This step determines which threads will generated in the Java code.
3. Now we generate identifiers for each thread in the Java code. For each $t \in \text{dom}(\kappa)$ generate a fresh identifier id_t and set $\rho(t) = id_t$.
4. In this step we need to map each transition in C to some Java code. For the original members of C we have τ that carry the Java code of those, but for the new transitions that we added in step 1 we need to do something else. For this reason we introduce the extended annotation function τ_{ext} that we define as follows:

$$\tau_{ext}(t) = \begin{cases} \tau(t), & \text{if } t \in C; \\ join(\sigma(t)), & \text{if } t \in \text{dom}(\sigma); \\ ; & \text{otherwise.} \end{cases}$$

$join(t)$ is a short hand for synchronizing a perhaps not yet scheduled thread. We can write it as "while ($t.getState() == Thread.State.NEW$) {try {wait();} catch (InterruptedException ie) {} t.join();}".

Each member $t \in \text{dom}(\kappa)$ is a tree that starts with a sequence and either ends in that sequence or in a split by a split transition. In the first case, assume that the sequence of transitions is t_1, \dots, t_n and set $\varphi(t) = \tau_{ext}(t_1); \dots; \tau_{ext}(t_n);$. If the tree from t is not just a sequence but contains a split, then the tree

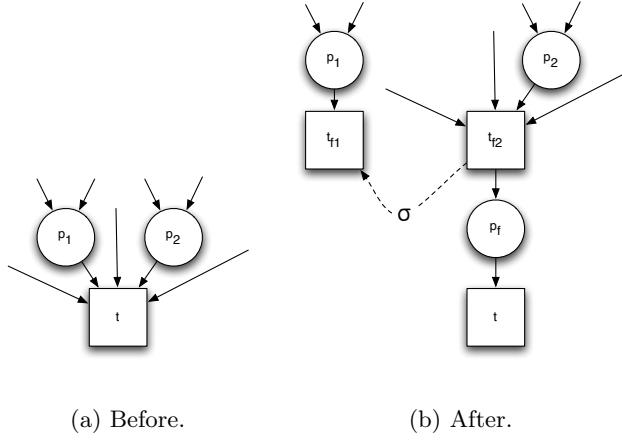


Fig. 8. Example of how step 1 in the MAXIMAL PARALLEL-component translation works.

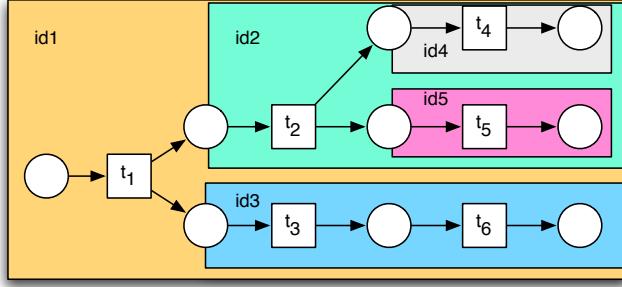


Fig. 9. Illustration of step 2 and 3 in the MAXIMAL PARALLEL-component translation.

will be a sequence with transition t_1, \dots, t_n followed by a transition split t_s where $\kappa(t_s) = \{t_{s_1}, \dots, t_{s_n}\}$. In this case we set $\varphi(t) = "\tau_{ext}(t_1); \dots; \tau_{ext}(t_1); \rho(t_{s_1}).start(); \dots \rho(t_{s_n}).start();"$.

5. At this point assume $dom(\kappa) = \{t_1, \dots, t_n\}$. We now generate the Java code. The code first declares all threads that we need and then starts the initial thread:

```

1 Thread ρ(t1) = new Thread() {public void run() {φ(t1);}};
2 ...
3 Thread ρ(tn) = new Thread() {public void run() {φ(tn);}};
4 ρ(ti).start();

```

In step 1 we remove all joins in $AJWN|_C$ and an example of this is found in Figure 8, which shows how a single join transition is transformed. Figure 9 illustrates the steps 2 and 3. In this figure $\kappa(t_1) = \{t_2, t_3, t_4, t_5, t_6\}$, $\kappa(t_2) = \{t_4, t_5\}$, $\kappa(t_3) = \{t_6\}$, $\kappa(t_4) = \kappa(t_5) = \kappa(t_6) = \emptyset$.

This final translation of acyclic marked graphs concludes the algorithm. Next we will show a small example of translating AJWNs to Java.

5.2 Example of the Translation of Annotated Java Workflow Net to Java

In this section we will look at a small generic example of an application of translation algorithm presented in the previous section. The example in Figure 10 focuses on the translation of the structural parts of the AJWN, so we do not consider the annotations of the AJWN since they do not affect the algorithm; we set the annotations of transition tN to $\tau(tN) = "tN();"$ and $\tau_G(tN) = gN$, where gN is some boolean expression. In the figure, all components are boxed and named in order to make it easier to see how the algorithm matches components. Our algorithm reduces the net by folding components in the following order:

1. Match MAXIMAL SEQUENCE-component $C1 \{p4, p5, p9, p13, t3, t8, t10\}$, replace with transition t_{C1} , and extend τ so that $\tau(t_{C1}) = \text{translate}(C1)$.
2. Match MAXIMAL SEQUENCE-component $C2 \{p8, t6, t14\}$, replace with transition t_{C2} , and extend τ so that $\tau(t_{C2}) = \text{translate}(C2)$.
3. Match CHOICE-component $C3 \{p2, p6, t4, t5\}$, replace with transition t_{C3} , and extend τ so that $\tau(t_{C3}) = \text{translate}(C3)$.
4. Match WHILE-component $C4 \{p8, t2, t7, t_{C2}\}$ and replace with transition t_{C4} , and extend τ so that $\tau(t_{C4}) = \text{translate}(C4)$.
5. Match MAXIMAL PARALLEL-COMPONENT $C5$ containing the rest of the net, replace with transition t_{C5} , and extend τ so that $\tau(t_{C5}) = \text{translate}(C5)$.

We map the AJWN into the Java code in Listing 1.1 and we have marked the start and end of the translation of each component.

6 Related Work

In [20], Wieringa describes his perception of a reactive system. Our approach matches well with this view on reactive systems as a part of the approach to describing the behavior of such systems. CCFN fit in the behavioral descriptions category in a tool box for reactive systems, where Wieringa uses state charts for that purpose. Other tools include E/R-diagrams to model the contents of the messages passed from the system to the environment and vice versa, and use communication diagrams, a variant of data flow charts, to describe the channels, processes, and storage in order to show how the system is structured internally and how it communicates with its environment.

It is fair to compare the class of CCFN to Colored Workflow Nets (CWNs) as defined and used in [6, 15] in the sense that it was developed to mirror an underlying computational paradigm: CCFN is intended to be used for modeling systems in an imperative manner, whereas CWNs are aimed at developing models representing resources, tasks, and case perspectives in the domain of workflow systems. This was enforced by the definition of CWN. CWNs were used as a starting point for mapping CWNs to annotated WF-nets, and these WF-nets were mapped to BPEL.

In [7], we mapped a special kind of annotated WF-net to BPEL. This is similar to what we did in Section 5 where we focused on a different sort of annotated WF-net and other component types, although some types are the same. The translations from the WF-net to BPEL and from CPN to Java are different, since BPEL is a language that directly supports many of the structural component types, whereas the imperative nature of Java complicates the translation.

Philippi [18] outlines three methods for translating high-level Petri nets (such as CPNs) to imperative code: structural-, simulation-, and reachability-based. He dismisses a structural approach as we propose in this paper, since he feels that high-level Petri net cover more behavioral constructs than what common imperative languages such as Java provide, so he thinks that such a translation is not possible. Instead he proposes a simulation-based approach which means that he translates the high-level Petri nets to an interpreter that interprets the state of the high-level Petri net. In other words, he constructs an interpreter that describes how transitions fire by calculating binding elements, what tokens are consumed and generated, and other Petri net related parts. This means that the code he generates is difficult to read since e.g. a sequence of transitions are not mapped to a sequence of statements, but instead a high-level CPN interpreter.

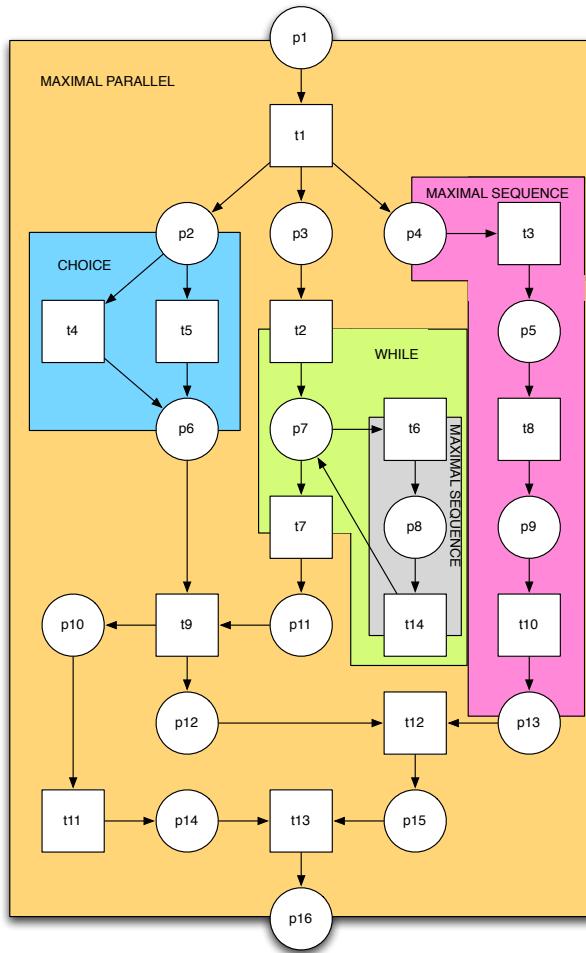


Fig. 10. AJWN to be folded using Definition 12. Notice that all the components of the AJWN are shown along with the type of the component.

Mortensen [17] translates CPNs to executable code. His method is a simulation-based one in which he takes the simulation image of a running CPN and extract the code it was build from (CPNs in his paper are compiled to Standard ML code), and maps this code to some implementation platform, including Java. He is also able to simply execute the simulation image in a Standard ML environment. In contrast to Philipi, Mortensen's approach is simply technical since he does not provide a generation process for the interpreter, but simply extracts it from already generated code from Design/CPN. In the paper, he mentions that he does not have any experience with the compilation to Java, but expects that it will be very slow. By his approach, Mortensen will not have any control over how the generated Java code will look like and does therefore not have any possibility to tweak the appearance of the generated code, as we are able to in our approach.

7 Conclusion and Future Work

The translation of this paper is intended as a move to bridge the gap between CPNs and Java and we feel that this is a step in the right direction. We think that translating any CPN to Java is not feasible, so we

Listing 1.1. Translation of the AJWN in Figure 10

```

1 // Start: MAXIMAL PARALLEL
2 final Thread t1 = new Thread(new Runnable()
3     {public void run() {
4         if (g1) {t4();} // Start: CHOICE
5         else {t5();} // End: CHOICE
6     }});
7 final Thread t2 = new Thread(new Runnable()
8     {public void run() {t11();}});
9 final Thread t3 = new Thread(new Runnable()
10    {public void run() {}});
11 final Thread t4 = new Thread(new Runnable()
12    {public void run() {
13        t2(); // Start: WHILE
14        while(g2) {
15            t6(); // Start: MAXIMAL SEQUENCE
16            t14(); // End: MAXIMAL SEQUENCE
17        }
18        t7(); // End: WHILE
19        while (t1.getState() == Thread.State.NEW)
20            try{wait();}
21            catch (InterruptedException ie) {}
22        t1.join();
23        t9();
24        t3.start();
25    }});
26 final Thread t5 = new Thread(new Runnable()
27     {public void run() {
28         t3(); // Start: MAXIMAL SEQUENCE
29         t8();
30         t10(); // End: MAXIMAL SEQUENCE
31         while (t4.getState() == Thread.State.NEW)
32             try{wait();}
33             catch (InterruptedException ie) {}
34         t4.join();
35         t12();
36     }});
37 t1();
38 t1.start(); t4.start(); t5.start();
39 while (t2.getState() == Thread.State.NEW)
40     try{wait();}
41     catch (InterruptedException ie) {}
42 t2.join(); t5.join();
43 t13(); // End: MAXIMAL PARALLEL

```

introduced CCFN as a restriction of CPN, to get a subset of CPN that could be mapped into readable Java. Although CCFN is simple in many ways, we think it is powerful enough to express many of the control-flows used in Java programs.

To help the translation to Java we introduced AJWN. Besides making the translation more smooth, we think it make the overall translation more extensible in two ways: (1) In Definition 4 we describe the four ways we allow a transition to occur in a CCFN and we were later able to present translations into Java statements of these ways. If anyone can find more ways to map transitions to Java statements they just have to change the definition of CCFN and the translation of these changes into Java statements, and leave the rest unchanged. (2) On the other hand, if someone has a better translations for component types or other component types, they can simply change the second part of the translation, and leave the first part unchanged. The algorithm in this paper is described in such a way that it is possible to implement this in a compiler.

The time complexity for the translation equals the sum two measures: (1) the complexity of the translation from CCFN to AJWN, which is $O(|PUT|)$ (P and T places and transition in CCFN), and (2) the complexity of the translation from AJWN to Java, which is $O(|T| \cdot 2^{|PUT|})$. In total, the complexity is $O(|T| \cdot 2^{|PUT|})$. However, as discussed in the end of Section 5, this is not an issue, since it is possible to find algorithms that, although they do not change the time complexity, they are so effective that translating CCFNs is not a problem in practice.

In this paper, we have introduced an algorithm for transforming CCFNs to Java code. There are, however, many things that would be sensible extension to the algorithm, but also areas where it would be interesting to test the applicability of the approach. In this section we will discuss both of these issues.

Extensions to Translation: Hierarchy The modeler should be able to modularize CCFN to keep models manageable. To do this, Definition 3 needs to be changed to allow for hierarchical constructs. Obviously, this will change the way we need to map the CCFN. Mapping could be handled in one of two ways: (1) mapping flattens the CCFN process definition into one net with no substitution transitions, and uses the translation algorithm that we have presented in this paper, or; (2) each page in the CCFN hierarchy is separately mapped to a Java class. The last proposal is the most complicated one since we have to give a Java translation of how and when control is transferred from a super page to a subpage and vice versa.

Extensions to Translation: Better Reactive System Support CCFNs were partly designed to be easy to map to Java, but also to support certain elements in the reactive systems terminology as described by Wieringa [20]. We have places for input and output events that in the sense of Wieringa model named events. It would be desirable to enable the modeling of interaction through shared states existing in the interface between the system and the environment. Such a shared state would be readable by both parties while only one party would be able to change (/control) the state. For example, a state representing a thermometer can be altered by the environment and read by the system. This extension can be done by extending Definition 3 in order to allow a CCFN to contain such places. The type of the place could simply be a **CONFIGURATION_LIST**.

Extensions to Translation: State machines Since a program written in an imperative language without parallelism is best described as a state machine, it would be desirable to be able to map components of AJWNs that are state machines - i.e. to be able to recognize components with the structural properties of state machines. [8, 12] describe algorithms to compile a goto graph into code consisting of loops and alternations. A state machine in a Petri net can be viewed as a goto graph, so it should be possible to adapt their theory to handle state machines in AJWNs as well.

Uses of the Translation: Model-driven Software Development It would be interesting to use this translation approach in a model-driven software development projects such as those presented in [6, 15]. The goal of the system should then be some reactive system.

Uses of the Translation: Library Components In the algorithm presented in Definition 12 we allow the user to provide library components if none of the standard components could be matched. This introduces an area we have not touched much in this paper: Not all AJWNs may be reducible. In fact, in [16] 100 models were reduced using the same component definition and folding technique as the one presented in this paper, and 76 unique non-reducible components were found. The models were designed by graduate students and

they were encouraged to implement advanced workflow patterns [5]. Although, the models were generally more complicated than what could be expected, it still shows that using this method for translating a graph, you must anticipate that some manual work is needed. It would therefore be interesting to study which non-reducible components that are typically found in the area of reactive system design.

References

1. W.M.P. van der Aalst. Verification of Workflow Nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426. Springer-Verlag, Berlin, 1997.
2. W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
3. W.M.P. van der Aalst. Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 1–65. Springer-Verlag, Berlin, 2004.
4. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2004.
5. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
6. W.M.P. van der Aalst, J.B. Jørgensen, and K.B. Lassen. Let's Go All the Way: From Requirements via Colored Workflow Nets to a BPEL Implementation of a New Bank System Paper. In R. Meersman and Z. Tari et al., editors, *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005*, volume 3760 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, Berlin, 2005.
7. W.M.P. van der Aalst and K.B. Lassen. Translating Unstructured Workflow Processes to Readable BPEL: Theory and Implementation. *Information and Software Technology*, 2006.
8. Zahira Ammarguellat. A Control-Flow Normalization Algorithm and its Complexity. *IEEE Trans. Softw. Eng.*, 18(3):237–251, 1992.
9. CPN Tools. www.daimi.au.dk/CPNTools.
10. João Miguel Fernandes, Jens Bæk Jørgensen, and Simon Tjell. Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net. In *Proc. of 6th International Workshop on Scenarios and State Machines (SCESM) at ICSE 2007*, 2007.
11. João Miguel Fernandes, Simon Tjell, and Jens Bæk Jørgensen. Requirements Engineering for Reactive Systems: Coloured Petri Nets for an Elevator Controller. In *Proc. of 14th Asia-Pacific Software Engineering Conference (APSEC)*, 2007.
12. Rainer Hauser and Jana Koehler. Compiling process graphs into executable code. In Gabor Karsai and Eelco Visser, editors, *GPCE*, volume 3286 of *Lecture Notes in Computer Science*, pages 317–336. Springer, 2004.
13. K. Jensen. *Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 1992.
14. Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 2007.
15. Jens Bæk Jørgensen, Kristian Bisgaard Lassen, and Wil M.P. van der Aalst. From Task Descriptions via Coloured Petri Nets Towards an Implementation of a New Electronic Patient Record. *Software Tools for Technology Transfer*, 2007.
16. Kristian Bisgaard Lassen and Wil M.P. van der Aalst. WorkflowNet2BPEL4WS: A Tool for Translating Unstructured Workflow Processes to Readable BPEL. In R. Meersman and Z.Tari, editors, *CoopIS/DOA/ODBASE*, 2006.
17. K. H. Mortensen. Automatic code generation from coloured petri nets for an access control system. In *Kurt Jensen (ed.): Second Workshop on Practical Use of Coloured Petri Nets and Design/CPN, Aarhus, Denmark*, pages 41–58, October 1999. InternalNote: Submitted by: khm@daimi.au.dk.
18. Stephan Philippi. Automatic code generation from high-level petri-nets for model driven systems engineering. *Journal of Systems and Software*, 79(10):1444–1455, 2006.
19. Simon Tjell. Distinguishing Environment and System in Coloured Petri Net Models of Reactive Systems. In *IEEE Second International Symposium on Industrial Embedded Systems*, 2007.
20. R. J. Wieringa. *Design Methods for Software Systems: YOURDON, Statemate and UML*. Science & Technology Books, 2002.

CPNunf: A tool for McMillan's Unfolding of Coloured Petri Nets

Visar Januzaj

Technische Universität München
Fakultät für Informatik
85748 Garching, Germany
visar.januzaj@in.tum.de
October 4, 2007

Abstract. In [11, 12], the branching processes and unfoldings of high-level Petri nets in general have been defined. We adopt this approach, in order to define branching processes of Coloured Petri Nets (CPNs). Further, we implement the approach, using McMillan's unfolding algorithm [15]. The experimental results demonstrate that our implementation successfully unfolds CPNs and thus alleviates the state space explosion, which can occur when generating the state space, e.g., by applying the state space generation methods [8] used in CPN Tools [17].

1 Introduction

In this paper, we will focus on McMillan's Petri net unfolding technique [15]. McMillan's approach, proposed to alleviate the well known state space explosion problem, is based on partial order semantics for Petri nets, so-called *branching processes*. A branching process, or an unfolding of a Petri net, is in fact a particular simple acyclic Petri net (*occurrence net*) which can be used to investigate the behaviour properties of a system (modelled as a Petri net), since it contains all information about the state space. However, the unfoldings can be infinitely long and thus not very useful for analysis. Nonetheless, McMillan introduced an approach to construct a finite initial part of the unfolding of a system, which still would represent all information about the state space. Such an initial part is called the *finite complete prefix*. A prefix is in fact much smaller than the state space and will never get larger (for a precise statement see [1]).

McMillan's approach not only attracted other research on the unfolding technique, that led to further improvements of his initial unfolding approach (such as [1, 4, 13]), but more importantly also led to the development of prefix-based model checkers, which are successfully used for the verification of distributed and concurrent systems (see, e.g., [2, 3, 11, 15, 16]).

Petri nets are in fact a powerful tool for modelling and analysis of distributed and concurrent systems, and depending on their modelling and expression power, they are classified in *low-level* and *high-level* Petri nets. The unfolding technique [15], together with its improvements mentioned above, can be applied only on the low-level family of Petri nets. This is due to the fact, that high-level Petri

nets have a more complex compound, e.g., the tokens are distinguishable, i.e., they can represent different values and/or data structures.

A naive way of unfolding high-level Petri nets is, to first transform high-level Petri nets into low-level ones and then apply the unfolding algorithm. The disadvantage of this approach is that the intermediate low-level Petri net (after the transformation) can become very large. Further, it can contain a huge number of places that never will get marked and transitions that will never fire.

However, in [11, 12] the branching processes of high-level Petri nets (in general) have been defined, i.e., high-level Petri nets are unfolded directly and no transformation is needed. Further, a relation between low-level and high-level branching processes has been established. This relation allows, as stated in [11, 12], "*to import results proven for branching processes of low-level nets rather than re-prove them*".

We will adopt the definitions of branching processes of high-level Petri nets presented in [11, 12] and introduce in this paper the branching processes of Coloured Petri Nets [5–7]. Furthermore, we implement a tool for investigating the unfolding of Coloured Petri Nets.

In [11, 12], the definitions of high-level branching processes are described by means of *M-nets* [11, 12], a family of high-level Petri nets. Therefor, we need to modify these definitions in order to define the branching processes of Coloured Petri Nets. In fact, only some modest modification need to take place.

We will informally introduce an algorithm for unfolding Coloured Petri Nets, as a combination of McMillan's unfolding algorithm [15] for low-level Petri nets and the Coloured Petri Nets transformation rules [5]. This will make it easier to understand the modified definitions of branching processes of Coloured Petri Nets.

We have to add here, that in [14] the unfoldings of n-safe Coloured Petri Nets, allowing only finite sets of colours, have been presented. Though, we have decided to rely on the approach presented in [11, 12], since it is superior (it allows infinite sets of colours), is more compact and easier to understand.

This paper is organised as follows: Section 2 handles the unfolding of low-level Petri nets and in Section 3 we discuss the unfolding of Coloured Petri Nets. Our implementation is briefly discussed in Section 4. We show our experimental results in Section 5 and we give some conclusions and directions for future work in Section 6.

2 Basic Notation

We assume that the reader is familiar with definitions and notations of Petri nets, as well as Coloured Petri Nets (CPNs, CP-nets) in particular, such as, marking, enabling, reachability, preset, postset, initial marking, binding elements, token elements, multi-sets etc. We will use the definitions and notations of Petri nets as described in [1]. For the definitions and notations of Coloured Petri Nets we will rely on [5–7]. In the following we will introduce McMillan's unfolding technique.

2.1 McMillan's Unfolding

McMillan's unfolding approach is based on partial order semantics for Petri nets, i.e., the states of a system are represented implicitly, using branching processes, which are in fact labelled acyclic Petri nets (occurrence nets).

The idea of McMillan's unfolding algorithm [15], roughly described below, is very simple: for each token in the initial marking of a given Petri net, make a copy b of the place p on which it resides and label it with p , and then carry out the following steps:

- Choose a transition t from the given Petri net.
- For each place p connected to t (those with arcs directed to t) find a copy b , labelled with p , and mark it with a token. If no such a copy can be found, then go to the first step. Note: for a given t , do not choose the same subset of places twice.
- If, for any two places a and b of the marked places, there is a path leading from one place to the other or there exists a place c containing different paths leading to a and b , while exiting c by different arcs, go to the first step.
- Make a copy e of t and label it with t . From each of the marked copies b draw an arc to e . Erase the tokens.
- For each place p , that t has an arc pointing to it, make a copy b , label it with p and draw an arc from e to b .

Following these steps one can easily unfold the Petri net presented in Fig.2(a). Its unfoldings are shown in Fig.2(b) and (c), i.e., unfoldings can have different sizes, depending on how long we proceed with it. And, depending on the behaviour of Petri nets, an unfolding can get infinitely long and as such not very useful for analysis.

Let us briefly introduce in the following the unfoldings of low-level Petri nets more formally and finally define the algorithm for generating the finite complete prefix, which can be used for verification.

The relations between any two distinct nodes of a Petri net, $x, y \in P \cup T$ (with P representing the set of places and T the set of transitions), are defined as follows:

- x and y are in *causal-relation*, if there exist a path in the net with at least one arc leading from x to y or from y to x , denoted $y < x$ and $x < y$, respectively.
- x and y are in *conflict-relation*, denoted $x \# y$, if
 $\exists t_1, t_2 \in T, t_1 \neq t_2 \wedge {}^\bullet t_1 \cap {}^\bullet t_2^1 \neq \emptyset : (t_1 < x \wedge t_2 < y)$, i.e., there exists a place containing different paths leading to x and y .
- x and y are in *concurrency-relation*, denoted $x \text{ co } y$, if
 $\neg(x < y) \wedge \neg(y < x) \wedge \neg(x \# y) = \text{true}$, i.e., x and y are neither in causal-relation nor in conflict-relation.

¹ ${}^\bullet x$ represents the preset of x and x^\bullet represents the postset of x .

Let us consider the branching process in Fig.2(b) regarding the relations described above, one can easily get some information about the behaviour of the original Petri net (Fig.2(a)), e.g., the nodes b_1 and e_5 are in causal-relation, i.e., the action t_1 occurs after the state th_1 . e_3 and e_4 are in conflict-relation, i.e., the actions t_2 and t_4 cannot take place in parallel. b_1 and b_4 are in concurrency-relation, i.e., the states th_1 and th_2 do not influence each other.

Having introduced the relations above, we can now define the occurrence nets and the branching processes for low-level Petri nets.

An *occurrence net* is an acyclic Petri net $O = (B, E, F)$, where B is the set of *conditions* (places), E is the set of *events* (transitions) and F is a flow relation, satisfying the following:

- $\forall b \in B, |\bullet b| \leq 1$, i.e., a condition has only one or none event in its preset,
- $\forall x \in (B \cup E) : |\{y \in (B \cup E) | y < x\}| < \infty$, i.e., each node has a finite set of nodes being in causal-relation with, and
- $\forall y \in B \cup E, \neg(y \# y)$, i.e., the nodes are not in self-conflict, thus, no two (or more) disjunctive paths starting at a condition are allowed to merge into a single node.

Definition 1. A **branching process** of a Petri net system $\Sigma = (P, T, F, M_0)$ is the pair $\beta = (O, p)$, where O is an occurrence net and p a labelling function satisfying the following properties:

- (i) $p(B) \subseteq P$ and $p(E) \subseteq T$, i.e., conditions are mapped to places and events to transitions.
- (ii) $\forall e \in E$, the restriction of p to $\bullet e$ is a bijection between $\bullet e$ (in Σ) and $\bullet p(e)$ (in β), similarly for e^\bullet and $p(e)^\bullet$, i.e., transition environments are preserved.
- (iii) the restriction of p to the set of conditions that have an empty preset, denoted $Min(O)$, is a bijection between $Min(O)$ and M_0 , i.e., β starts at M_0 .
- (iv) $\forall e_1, e_2 \in E : (\bullet e_1 = \bullet e_2 \wedge p(e_1) = p(e_2)) \Rightarrow (e_1 = e_2)$, i.e., there is no redundancy.

p represents the **homomorphism** from O to Σ .

As mentioned, depending on how long we unfold, we get different sizes of branching processes, i.e., a branching process can be a prefix of some other branching process. For example, the branching process in Fig.2(c) is a prefix of the branching process in Fig.2(b).

A branching process $\beta' = (O', p')$ is a *prefix* of another branching process $\beta = (O, p)$, denoted $\beta' \sqsubseteq \beta$, if O' is a subnet of O satisfying:

- $Min(O)$ belongs to O' , i.e., both β and β' start at the same set of conditions,
- if a condition b belongs to O' , then its input event $e \in \bullet b$ in O also belongs to O' (if it exists), and
- if an event e belongs to O' , then its input and output conditions, $\bullet e \wedge e^\bullet$, in O also belong to O' .

There exists a unique, up to isomorphism, maximal (w.r.t. \sqsubseteq) branching process β_{max} for any given Petri net system Σ , called the *unfolding* of Σ (see [1, 11–13]). The unfolding of the Petri net in Fig.2(a) is infinite.

In order to be able to define the finite complete prefix of an unfolding, we will introduce in the following some more definitions. These will also contribute on building a stronger connection between branching processes and their original Petri nets.

A set of events $C \subseteq E$ is called a *configuration* if the following is satisfied:

- $\forall e, e' \in C : \neg(e \# e')$, i.e., C is conflict-free, and
- $\forall e \in C : e' \leq e \Rightarrow e' \in C$, i.e., C is causally-closed.

The *local configuration* of an event e of a branching process, denoted $[e]$, represents the set of events E' , such that $e' \leq e, \forall e' \in E'$.

The set of events $\{e_1, e_3, e_5, e_6\}$ is a configuration, as well as the local configuration of e_6 . The sets $C' = \{e_1, e_2, e_4\}$ and $C'' = \{e_1, e_3, e_6\}$ are not. C' satisfies the second configuration-property but not the first, i.e., e_1 is in conflict with both other events. C'' satisfies the first property but not the second, because the event e_5 is missing.

Further, a set of conditions $B' \subseteq B$ is called a *co-set*, iff for all $b, b' \in B' : \neg(b < b' \vee b' < b \vee b \# b')$, i.e., b and b' are neither reachable from each other nor are they in conflict with each other. A maximal co-set $B' \subseteq B$ is called a *cut*.

The set of conditions $B' = \{b_1, b_2, b_3\}$ is a co-set, and adding b_4 to B' , $B' \cup \{b_4\}$, we get a cut, i.e., there is no other condition that can be added to B' without destroying the co-set property, thus, B' is maximal.

For a given finite configuration C the co-set $Cut(C) = (Min(O) \cup C^\bullet) \setminus {}^\bullet C$ is a cut. In addition, the set of places $Mark(C) = p(Cut(C))$ is a reachable marking in the original Petri net. Thus, a cut represents a reachable marking.

Let $C = \{e_1, e_3, e_5, e_6\}$ be a configuration of the branching process in Fig.2(b) and $Min(O) = \{b_1, b_2, b_3, b_4\}$ represent the set of conditions with an empty preset, we get the cut $Cut(C) = (Min(O) \cup C^\bullet) \setminus {}^\bullet C = \{b_4, b_{14}, b_{15}, b_{16}\}$, which represents the reachable marking $Mark(C) = p(Cut(C)) = \{th_1, th_2, f_1, f_2\}$.

One can conclude that, a marking M of a Petri net system Σ is *represented* in a branching process β of Σ iff β contains a finite configuration C such that $Mark(C) = M$. Further, every reachable marking of a Petri net system is represented in its unfolding, and every marking represented in a branching process is reachable in the original Petri net system, see [1, 11–13].

There are certain rules, during unfolding, that one has to follow, in order to find extensions of already built branching processes. These extensions that come into consideration are called possible extensions and are defined as follows:

Definition 2. A *possible extension* of a branching process β of a Petri net system Σ is a pair (t, X) ² such that:

- (i) $p(X) = {}^\bullet t$, and
- (ii) (t, X) is not a part of β .

² X is a co-set of conditions of β , and t a transition of Σ

FINITE PREFIX ALGORITHM:

input: $\Sigma = (N, M_0)$ ³ — an n -safe net system
output: a finite complete prefix Fin of Σ

```

 $Fin := \{(s_1, \emptyset), \dots, (s_n, \emptyset)\}$ 
 $pe := PE(Fin)$ 
 $cut-off := \emptyset$ 

while  $pe \neq \emptyset$  do
    choose an event  $e = (t, X)$  in  $pe$  such that  $[e]$  is minimal w.r.t.  $\prec$ 
    if  $[e] \cap cut-off = \emptyset$  then
        add to  $Fin$   $e$  and a condition  $(s, e)$  for every output place  $s$  to  $t$ 
         $pe := PE(Fin)$ 
        if  $e$  is a cut-off event of  $Fin$  then
             $cut-off := cut-off \cup \{e\}$ 
         $pe := pe - \{e\}$ 

```

Fig. 1. The finite complete prefix algorithm.

The set of all possible extensions of β is denoted $PE(\beta)$.

A possible extension of the branching process in Fig.2(c) would be the pair $(t_1, \{b_7, b_8, b_9\})$, which is already represented in Fig.2(b).

When dealing with Petri nets that have an infinite unfolding, e.g., like the one in Fig.2(a), the number of possible extensions is infinite as well. However, our aim is the construction of a finite complete prefix of an unfolding. In order to achieve this, we will introduce in the following two other core components of the prefix generation process, namely the adequate order and cut-off event.

A partial order \prec on the finite configurations of the unfolding of a net system is an *adequate order* if:

- \prec is well-founded,
- $C_1 \subset C_2 \Rightarrow C_1 \prec C_2$, and
- \prec is preserved by finite extensions, i.e., if $C_1 \prec C_2$ and $Mark(C_1) = Mark(C_2)$, then the isomorphism I^4 satisfies $C_1 \oplus E \prec C_2 \oplus I(E)$ for all finite extensions $C_1 \oplus E$ of C_1 .

Let \prec be an adequate order on the configurations of the unfolding of a net system, and let β be a prefix of the unfolding containing an event e . The event e is a *cut-off event* of β (w.r.t. \prec) if β contains a local configuration $[e']$ such that:

- $Mark([e]) = Mark([e'])$, and
- $[e'] \prec [e]$.

⁴ I is a mapping from the finite extensions of C_1 onto the finite extensions of C_2 . A configuration C and a set of events E is called an *extension* of C , denoted $C \oplus E$, if $C \cup E$ is a configuration such that $C \cap E = \emptyset$.

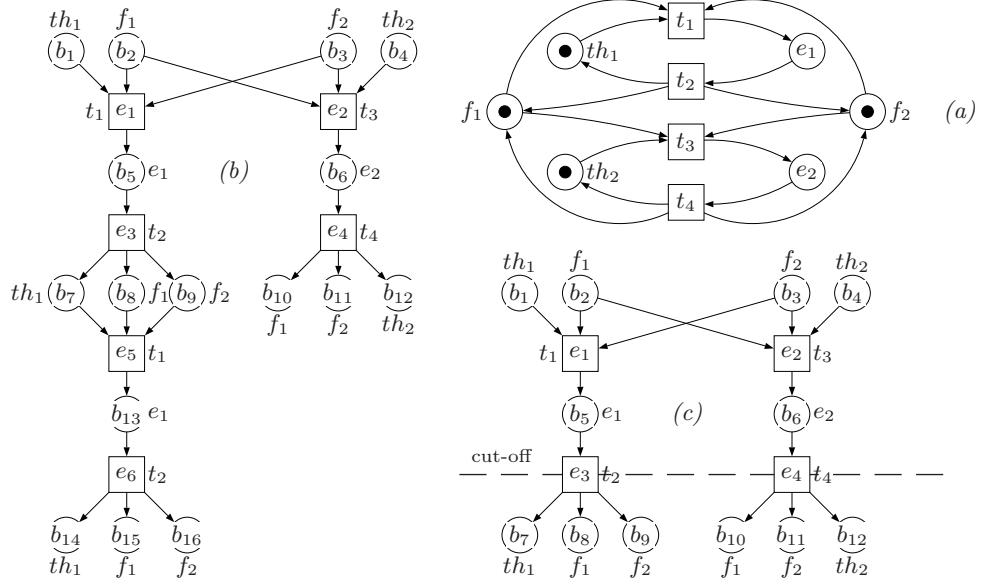


Fig. 2. A Petri net (a), one of its branching process (b) and its prefix (c).

The conditions e_3 and e_4 build the cut-off events set of the net in Fig.2(a). The dashed line through the events e_3 and e_4 in Fig.2 (b) represents the *cut-off-point* of these events, i.e., further construction is stopped, and the already constructed part (up to the dashed line) represents the finite complete prefix. It is easy to see that the part after these two events repeats itself, and thus the prefix contains all the necessary information about the reachable states. A branching process $\beta = (O, p)$ of a Petri net system $\Sigma = (P, T, F, M_0)$ is *complete* iff for every reachable marking M in Σ there exists a configuration C in β , not containing any cut-off event, such that:

- $Mark(C) = M$, indicating that M is represented in β , and
- $\forall t \in T$ enabled in M there exists a configuration $C \cup \{e\} : e \notin C \wedge p(e) = t$.

Further, the number of events in the complete prefix can never exceed the number of reachable states of a Petri net, see [1, 11–13]. Thus, the size of the prefix is never larger than the corresponding state space graph.

Now we can put all these definitions together and define the finite complete prefix algorithm, which is shown in Fig.1. The correctness of the algorithm, *Fin* is finite and *Fin* is complete, is proved in [1], and thus omitted.

3 Unfoldings of Coloured Petri Nets

There exist two ways of unfolding CPNs: *unfolding per transformation*, i.e., CPNs are first transformed into low-level Petri nets and then the unfolding technique

is applied, and *direct unfolding*, i.e., CPNs are unfolded directly from their high-level state.

In the following we will discuss both approaches and we will define the *branching processes of Coloured Petri Nets*.

3.1 Unfolding per transformation

In [5] the rules for transforming CPNs into low-level Petri nets are defined. These rules are shown below.

Definition 3. Let a (non-hierarchical) CP-net $\Omega = (P, T, A, \Sigma, V, C, G, E, I)$ be given. Then we define the **equivalent low-level Petri net** as $LLPn = (P', T', A', E', I')$ where:

1. $P' = TE(p)$, $\forall p \in P$, i.e., for each possible token element of a CPN place a low-level Petri net place is generated.
2. $T' = BE(t)$, $\forall t \in T$, i.e., for each possible binding element⁵ of a CPN transition a low-level Petri net transition is generated.
3. $A' = \{((p, c), (t, b)) \in P' \times T' | (E(p, t) < b >)(c) \neq 0\} \cup \{((t, b), (p, c)) \in T' \times P' | (E(t, p) < b >)(c) \neq 0\}$,
i.e., we have an arc iff an occurrence of t with a binding b removes/adds at least one c -token from/to p .
4. $\forall ((p, c), (t, b)) \in A' \cap (P' \times T') : E'((p, c), (t, b)) = (E(p, t) < b >)(c)$ and
 $\forall ((t, b), (p, c)) \in A' \cap (T' \times P') : E'((t, b), (p, c)) = (E(t, p) < b >)(c)$,
i.e., we weight the arcs with the number of c -tokens which an occurrence of t with the binding b removes/adds to p .
5. $\forall (p, c) \in P' : I'(p, c) = (I(p))(c)$, i.e., the number of c -tokens in $I(p)$.

The proof about the equivalency of (non-hierarchical) CPNs and their transformations (low-level Petri nets) is carried out in [5] and, being beyond the scope of this work, it is omitted.

Following the rules defined above one can transform a (non-hierarchical) CP-net into a low-level Petri net. Then by applying the unfolding algorithm, introduced in the previous section, the finite complete prefix is generated. The prefix can be used to analyse the properties of the original CP-net.

Nonetheless, considering the first transformation rule, it is not possible to transform CPNs allowing infinite sets of colours. This is due to the fact, that CPN places of types of infinite sets of colours produce during the transformation (1. rule) an infinite number of low-level Petri net places. This means, we cannot generate a low-level Petri net with a finite set of places and transitions. Further, a finite complete prefix cannot be constructed.

Another disadvantage of this approach is that, even if we use only finite sets of colours, the transformation may generate a huge number of low-level Petri net places and transitions, that will never get marked and will never get enabled, respectively.

⁵ All elements of $B(t)$ automatically satisfy the guard $G(t)$.

Let us consider the CP-net Ω in Fig.3, allowing only finite sets of colours⁶, and its transformation in Fig.4(a). The transformation contains 13 places and 8 transitions, which number can grow for larger colour sets and still have only 4 places that can get marked and 2 transitions that are enabled (considering the current initial marking). The place names of the form X_y represent a place X holding the colour y , and the transition names of the form X_{ij} represent a transition X and the binding of variables n and k to values i and j , respectively. Note: In Fig.4(a) for the transition T'_{10} , where $j = 0$, means that variable k is not bound, since it does not appear on the arcs of transition T' .

The unfolding of Ω is shown in Fig.4(b), which has a relatively small size considering the transformation of Ω .

Similar results are observed in [11, 12], where this issue is discussed more detailed, and the experiments show that especially data-based high-level Petri nets can generate large intermediate low-level Petri nets.

However, we want to allow infinite sets of colours, since this allows us to analyse a broader family of CPNs and, as they write in [11, 12], "*it is often convenient to assign to a place the type \mathbb{N} rather than $\{0, \dots, n\}$, since n might be not known in advance*". Further, we want to prevent the generation of unnecessary large intermediate low-level Petri nets.

In the following we will discuss a different approach, which allows the use of infinite sets of colours.

3.2 Direct unfolding

In this section, we informally introduce an approach for unfolding CPNs, allowing finite as well as infinite sets of colours, directly from their high-level state, i.e., no transformation needs to take place. This approach is a combination of the transformation rules introduced in the previous section and McMillan's unfolding description represented in Section 2.1:

- 0: Similar to the first transformation rule (Definition 3.1.), transform each token element (p, c) of the current marking into a condition b and label it with (p, c) .
- 1: Choose a transition t .
- 2: For each place $p \in \bullet t$, find all possible copies of p in the occurrence net, i.e., all conditions b labelled by (p, c) , and mark them with c . If no such copy can be found, go to step 1. Note: for a chosen t , do not choose the same subset of places in the occurrence net twice.
- 3: For each possible binding b_p of $B(t)$ ⁷, regarding the current marking, create a binding element (t, b_p) and apply the following:
 - (a) similar to the second transformation rule (Definition 3.2.), transform (t, b_p) into an event e and label it with t ,
 - (b) for each $M(p), p \in t^\bullet$, generated after the occurrence of the actual binding element (t, b_p) , apply step 0,

⁶ The allowed colour sets are, as defined on the top right corner in Fig.3, sets of integers: $\{1..2\}$, $\{1..3\}$ and $\{2..5\}$.

⁷ $B(t)$ represents the set of all bindings of t .

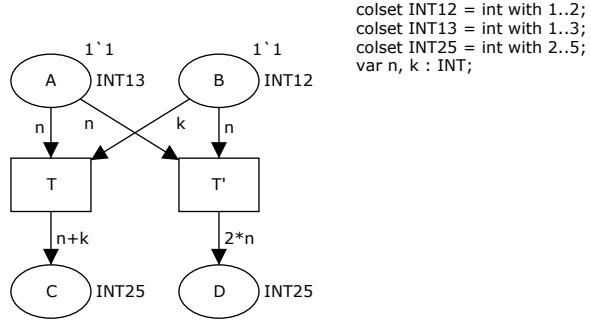


Fig. 3. A simple Coloured Petri Net model.

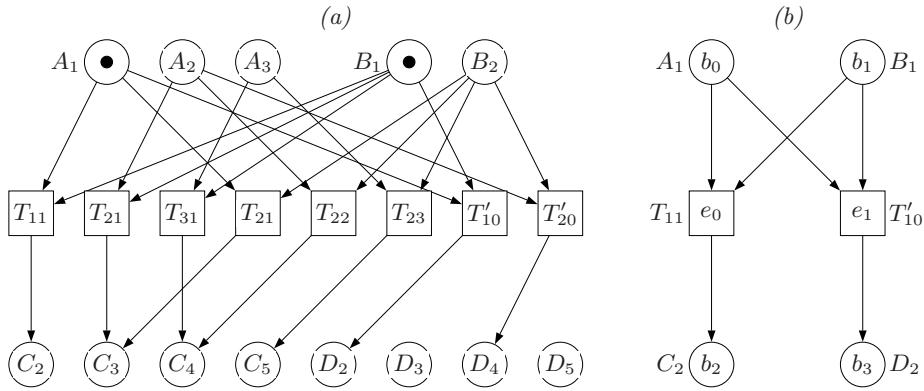


Fig. 4. The low-level Petri net representation (a) of the transformed CP-net in Fig.3 and its unfolding (b).

- (c) connect e to its pre- and postset accordingly, similar to the third transformation rule (Definition 3.3.).⁸
- 4: Go to step 1.

If we follow the steps of our above roughly sketched unfolding approach, we can easily unfold the CP-net shown in Fig.3. Apart from step 0 and 4, in an optimal case, one would run the steps 1 - 3 only twice. The resulting unfolding is the same as the one shown in Fig.4(b), where the transformation took place first, i.e., it is possible to unfold CPNs directly from their high-level state. One can easily notice, that the conditions represent a subset of all token elements, denoted TE , and the events a subset of all binding elements, denoted BE , of the CP-net Ω .

⁸ It is very important to note that each binding element (t, b) must satisfy the guard $G(t)$, and the preset of each event e must be a co-set, otherwise e should not be generated.

This leads to the definition of branching processes for high-level Petri nets as introduced in [11, 12], which we will discuss in the following.

3.3 Branching processes

Since the branching processes in [11, 12] were presented based on a certain family of high-level Petri nets, the so called M-nets [11, 12], we will introduce these here by means of Coloured Petri Nets. In fact, the only changes that need to take place are the notions "legal place instances" and "legal firings" (see [11, 12]), which correspond to the for CPNs users well known notions "token elements" and "binding elements", respectively. Being only a matter of different notions meaning the same thing, we do not think that some explicit proof is necessary to show the equivalency of those notions.

Definition 4. A homomorphism from an occurrence net $O = (B, E, F)$ to a CP-net Ω is a mapping $h : B \cup E \rightarrow TE \cup BE$ such that:

- $h(B) \subseteq TE$ and $h(E) \subseteq BE$ (conditions are mapped to token elements and events to binding elements).
- $\forall e \in E, h(\bullet e)_{MS} = \bullet h(e)$ and $h(e^\bullet)_{MS} = h(e)^\bullet$ (the environments of binding elements are preserved).
- $h(\text{Min}(O))_{MS} = M_0$ (conditions with empty preset are mapped to the initial marking).
- $\forall e_1, e_2 \in E: (\bullet e_1 = \bullet e_2 \wedge h(e_1) = h(e_2)) \Rightarrow (e_1 = e_2)$ (there is no redundancy).

A branching process of a Coloured Petri Net Ω is a pair $\beta = (O, h)$ such that O is an occurrence net and h is a homomorphism from O to Ω .

Relying on the results presented in [11, 12], most of the definitions and results proven for branching processes of low-level Petri nets, can be easily imported for branching processes of Coloured Petri Nets, and for each CP-net Ω there exists a unique, up to isomorphism, maximal branching process β_{max} of Ω , called the *unfolding* of Ω , for a precise statement see [11, 12].

Since unfolding might be infinite, we are interested on finite complete prefixes of Coloured Petri Nets, which can be gained by using one of the existing finite complete prefix generating algorithms, e.g., [15, 13, 1]. We will use in this work McMillan's algorithm, as introduced in [15]. It is important to add, that different algorithms, e.g., [1, 13], perform better than the one we use here.

Similarly we can define the possible extensions of branching processes of Coloured Petri Nets, which is the only thing that needs to be modified, in order to use the unfolding algorithm presented in Fig.1 to unfold CP-nets. These modifications affect in fact only the function $PE()$, which we will not discuss here. More on this, see, e.g., [12, 11, 1, 3].

Definition 5. A possible extension of a branching process $\beta = (O, h)$ of a CP-net Ω is a pair $((t, b), X)$, where X is a co-set in β and $(t, b) \in BE(t)$ is an enabled binding element, such that:

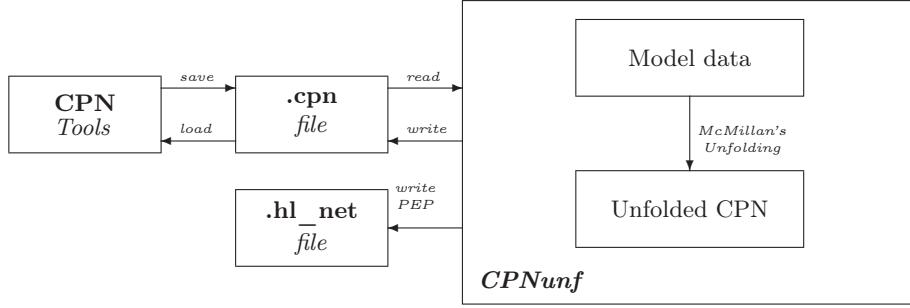


Fig. 5. *CPNunf*'s structure.

- (i) $h(X_{MS}) = \bullet t$, and
- (ii) $((t, b), X)$ is not a part of β .

4 Implementation

In this section, we will not go into details of our implementation, denoted *CPNunf*, but rather highlight the restrictions of CPNunf and the representation of the generated prefix in CPN Tools.

CPNunf is realised using the programming language JAVA. We use the *Java Architecture for XML Binding* [18] to load the contents of the *.cpn* - XML files into Java-Object-Trees, as well as for writing it back in such a *.cpn* - file format. This process is represented by the arcs *read* and *write* in Fig.5, where the structure of CPNunf is shown. The box *Model data* represents the data of a CPN model, which is used as an input for the unfolding process. An unfolded CPN model (box *Unfolded CPN*) is then written in a *.cpn* file, so that it can be loaded into CPN Tools. Furthermore, CPNunf can convert CPN models into the *.hl_net* - file format, accepted by the PEP⁹ tool [19] and other existing unfolders, such as PUNF [10], which is able to unfold high-level Petri nets (M-nets [11, 12]). We used this feature to check the correctness of our tool, by comparing our results with those of PUNF.

4.1 Restrictions

We restrict ourselves to non-hierarchical and non-timed CPNs. Further, we assume that the CPNs we use are finite and n-safe, for $n \geq 1$.

However, we allow the use of infinite and finite colour sets of integers. We think that for the first step it is most important to show that the theory of unfolding Coloured Petri Nets, presented in the previous section, can be successfully applied, than implementing an unfolding *engine* that allows all possible colour sets. Further, once the essence of the unfolding technique for Coloured Petri Nets

⁹ PEP is a tool for modelling, compilation, simulation and verification of high-level/low-level Petri nets.

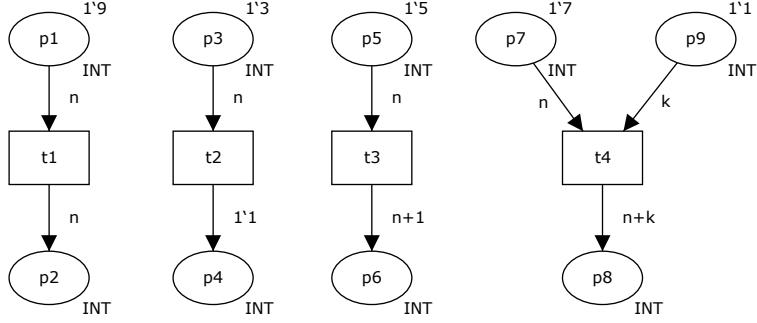


Fig. 6. Restrictions.

has been implemented, one can easily extend its features. In fact, there is only a small number of things one has to modify, e.g., the evaluation of more complex arc expressions and the calculation of corresponding binding elements.

Let us look at some other restrictions we make. In the following, we informally present the arc expressions that the CP-nets in this small family can have:

- $P \rightarrow T$: the allowed expression on arcs from places to transitions
 - $\langle \text{var} \rangle$
- $T \rightarrow P$: the allowed expressions on arcs from transitions to places
 - $\langle \text{var} \rangle$
 - $\langle \text{var} \text{ op } \text{var} \rangle$
 - $\langle \text{var} \text{ op } \text{const} \rangle$ (or $\langle \text{const} \text{ op } \text{var} \rangle$)
 - $\langle 1' \text{const} \rangle$

`var` denotes the variables which are of integer type, `const` is a constant, i.e., an integer value, and `op` denotes the operation that are allowed, namely, addition, subtraction and multiplication.

In Fig.6 one can see how the restrictions introduced above can be used in a CPN model. As it is easy to see, the arcs are not allowed to carry more than one token colour. In other words the arcs have weight 1.

4.2 Prefix representation in CPN Tools

Since the finite complete prefixes are represented by occurrence nets, they offer the possibility of running simulations on them, e.g., when one wants to visually see how and where a deadlock can occur. To make this possible for the constructed prefixes of CPNs, we represent the prefixes in a CPN format, i.e., in a `.cpn` XML file format, so that it can be loaded in *CPN Tools*.

We have chosen the colour set $E = \{e\}$ to be the type of the conditions of the prefix. Since e is the only colour in the colour set E and it represents a token that does not carry any data, thus *empty*, it makes the perfect *neutral* representative of token elements in the original CP-net. For the arc expressions we write:

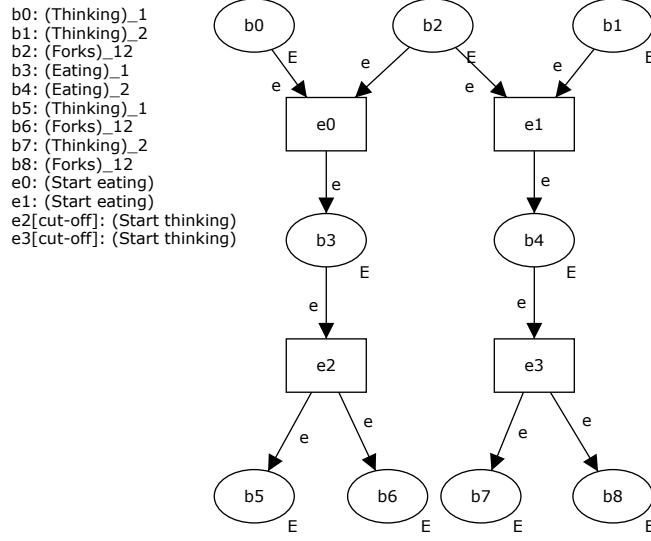


Fig. 7. The CPN-like representation of a finite complete prefix.

$$E(a) = e, \forall a \in A, \text{ i.e., all arcs are labelled by } e.$$

To easier know which condition and event belong to which place and transition, respectively, we add some text (as *Auxiliary*) holding this information.

A representation of a prefix as a CP-net is shown in Fig.7. In fact this prefix belongs to a slightly modified version of the CP-net in Fig.8(b) representing the Dining Philosophers problem. In the text on the left side in Fig.7 the names of the conditions and events are listed. Besides the names of the conditions we see the names of places, representing the token elements, i.e., $(\text{Thinking})_1$ represents the pair $(\text{Thinking}, 1) \in TE$. Near the names of events we see the names of transitions. We do not represent the binding elements, since this information can be extracted from the token elements shown near the names of conditions the event is connected to. We cannot know exactly which variable was bound to which value, but we know which token colours are consumed and which are produced when a transition in a certain marking occurs. Further, we mark the events that represent the cut-off events, e.g., $e2[\text{cut-off}]$ indicates that e_2 is a cut-off event. Through this we can see which parts of the net repeat itself.

To run simulations on such prefix representations one can initialise the places (conditions) with the empty colour e , i.e., those places(conditions) that have an empty preset, which represent the unfolding of the initial marking in the original CP-net, namely $\text{Min}(O)$, e.g., in Fig.7 such conditions are b_0 , b_1 and b_2 .

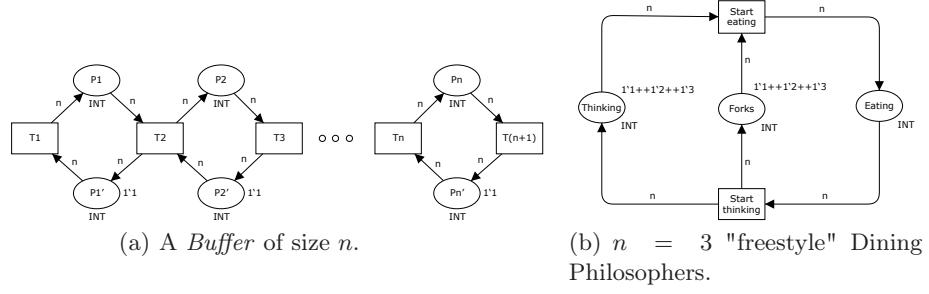


Fig. 8. Concurrent CPNs.

5 Experimental results

All the experiments were conducted on a PC with a PENTIUM® 4/2.66 GHz processor and 512MB main memory.

The experiments represent the performance of CPNunf when unfolding CPNs, and that of CPN Tools' state space methods [8] used for constructing the state space. We used for our experiments CPN Tools version 1.4.

We present only some of the more representative experiments we conducted during this work. The CP-nets we have chosen (see Appendix) can be divided in two classes: CP-nets involving a lot of concurrency and those without concurrency at all. In each of these classes we have CP-nets that grow on the number of token colours and those that grow on the size, i.e. the number of places and transitions. We have to add here, that it is not possible to compare CPNunf's performance with other unfolding tools, since none of the existing unfolders can unfold n -safe CP-nets allowing infinite (as well as finite) sets of colours. The first known high-level Petri net unfold, PUNF [10], which is as a result of the work in [11, 12], supports only (strictly) 1-safe M-nets (see [11, 12]) and thus not possible to compare its performance with that of CPNunf in all of the conducted experiments. However, we used PUNF to check the correctness of CPNunf for small 1-safe examples. This was done by the conversion of CP-nets into the from PUNF accepted file format `.hl_net`, using one of the extra features of CPNunf.

5.1 Concurrent CPNs

For our experiments in this class of CP-nets we use the CPN models presented in figures 8(a) and 8(b) in the Appendix.

In Fig.8(a) (*Buff(n)*) we present a CP-net modelling a buffer of size n . In Fig.8(b) (*FreeStyle(n)*) we present a CP-net modelling the *freestyle* Dining Philosophers, where we allow the philosophers to eat and think as they want, i.e., they can eat by only using one of the forks. In this case each fork belongs to a philosopher which can be used only by this philosopher.

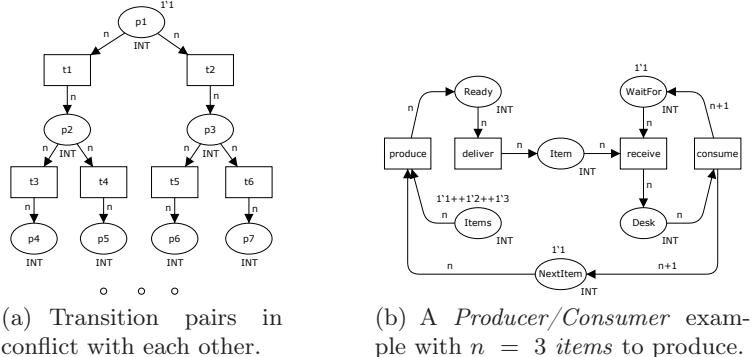


Fig. 9. Non-Concurrent CPNs.

The experimental results for the unfolding and the state space of these two CP-nets are represented in Table 1 (see Appendix). Reading from the left to the right, in the first column is the number n which represents, e.g., the number of freestyle philosophers (Fig.8(b)) and the number of buffer cells (Fig.8(a)). The second column represents the number of places $|P|$, and the third column represents the number of transitions $|T|$ of a CP-net. The size of the state space (graph) generated by CPN Tools is shown in the fourth and fifth column, representing the number of states/nodes $|N|$ and the number of arcs $|A|$, respectively. The sixth, seventh and eighth column represents the size of the prefix generated by CPNunf: $|B|$ = the number of conditions, $|E|$ = the number of events, and $|E_{cut}|$ = the number of cut-off events. We have to add here that E represents all events together with the cut-off events in E_{cut} . B represents all conditions as well as all conditions in the postset of each cut-off event in E_{cut} . The last two columns represent the time (in seconds) that CPN Tools and CPNunf need for the generation of the state space and of the finite complete prefix, respectively. In both examples, *Buff(n)* and *FreeStyle(n)*, the size of the state space grows exponentially, i.e., 2^n . The time for constructing the state space (graph) increases as well, as can be seen in Table 1. While during the unfolding the number of non-cut-off events grows linear to n , i.e., for *Buff(n)* we have $\sum_{i=1}^n i$ non cut-off events, and for *FreeStyle(n)* we have $2 \cdot n$. For the case $n = 20$ the test for constructing the state space has been interrupted after 16 hours. The size of the state space $|N|$ and the number of arcs $|A|$, for $n = 20$, as shown in Table 1, have been calculated by hand according to the previous results for $n < 20$.

5.2 Non Concurrent CPNs

In this class of experiments we have chosen the CP-nets in Fig.9(a) and Fig.9(b) (see Appendix). In the following we will discuss the results for each of the CP-nets.

Conflict pairs The CP-net $\text{Conflict}(n)$ in Fig.9(a) represents a CPN model with places on which transition pairs are in conflict with each other, i.e., n is the number of such places where transition pairs are in conflict. The results are similar for both approaches, state space and unfolding, see Table 1. We did though not construct any larger CP-net, i.e., for $n > 31$, because it was challenging enough to keep the overview over the CP-net for $n = 31$, involving 63 places, 62 transitions and 124 arcs. Especially, when constructing it by hand, as we did. Nonetheless, we think that for such nets, even for greater n , both approaches would perform similarly, since these kind of CP-nets are neither complex nor challenging enough. The sizes of the prefix and the state space grow linear to the size of the CP-net, in fact, the state space and the prefix have exactly the same size as the original CP-net.

Sequential We will discuss here the CP-net in Fig.9(b), which models a Producer/Consumer with n -items. It is a net where only the number of token colours grows, i.e., the number of items to produce and consume. We can observe from the results in Table 1 that for $n = 1..20$ the times for constructing the state space and the prefix are relatively similar. Beginning with $n > 20$ the performance of CPNunf starts to let go, and for $n = 130$ CPNunf needs 2.65 seconds for the construction of the prefix, whereas CPN Tools needs only 0.54 seconds for the state space construction.

Remark: Every time a place p is unfolded, the transitions in the postset of this place are marked (or promoted) as possible extension candidates, and when such a transition is chosen, then the possible correct co-sets are calculated out of *all* the copies of p , i.e., (p,c) token elements in the occurrence net, and the unfolding is extended accordingly.

In our example, the place *NextItem* is unfolded sequentially, holding different colours, and each time it gets unfolded it promotes transition *produce*, i.e., each time *produce* is chosen, we calculate *all* possible co-sets out of 130 instances (conditions in the unfolding) of the place *Items* and $n \leq 130$ instances of *NextItem*, depending on how far we have proceeded with the unfolding. We do this, in order not to forget any possible co-set. The problem here is, that each time *produce* is chosen only one co-set can be built and thus extend the unfolding. The calculation of co-sets is time consuming, especially when a large number of conditions in the unfolding has to be considered, and this affects the performance of CPNunf. Thus, CPNunf has a drawback, when a transition has a large number of promoters out of which only one co-set at a time can be built. Probably by saving the information about the already calculated co-sets, one would get better performance. One may ask though, why the performance of CPNunf is still good in the example in Fig.8(b), even though the transition *Start eating* has a large number of promoters? *Start eating* has indeed a large number of promoters, though it gets promoted only once, namely at the beginning when the initial marking is unfolded.

Unfortunately, it was not possible to compare the CPNunf results with PUNF for the *Producer/Consumer* example, since, as mentioned, PUNF does not ac-

cept n -safe nets, for $n > 1$. Nonetheless, the number of non-cut-off events does not exceed the number of reachable states, and as in the previous example the state space and the prefix grow linear to each other.

6 Conclusions

We have introduced in this paper the branching processes of Coloured Petri Nets, adopted from the approach presented in [11, 12]. In addition we briefly presented our tool, CPNunf, for unfolding finite and n -safe Coloured Petri Nets allowing finite and infinite sets of colours. The experiments show that CPNunf can successfully be used for the prefix generation of Coloured Petri Nets, and that it performs better than CPN Tools' state space methods [8], when dealing with CPNs with a lot of concurrency, and thus alleviate the state space explosion problem. Nonetheless, CPNunf has its weak points, as was the case with the *Producer/Consumer(n)* example. This drawback is caused by transitions which have a large number of promoters, since we calculate the co-sets all over again, each time a transition is promoted. Keeping track of the already built co-sets, i.e., not to calculate them each time from the beginning, would probably solve this problem.

The generated prefixes can be used to investigate the behaviour properties of CPNs by applying prefix-based model checkers. We conducted some experiments, which we did not show here, and the results were promising, e.g., we could check the liveness properties for the CP-net *Buff(20)*, whereas with CPN Tools we could not even generate the state space graph.

We think that by extending CPNunf, i.e., to allow other colour sets than INT, and by developing advanced prefix-based model checkers, one can investigate even more efficiently behaviour properties of CPNs.

Acknowledgments

I would like to thank Bernd Finkbeiner at the University of Saarland and Lars M. Kristensen at the University of Aarhus for their valuable advice and support. My thanks go to Maciej Koutny and Victor Khomenko at the University of Newcastle upon Tyne for their valuable comments and helpful advice on the unfolding technique.

References

1. J. Esparza, S. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. *Formal Methods in System Design* 20(3) (2002) 285-310.
2. J. Esparza and C. Schröter: Unfolding Based Algorithm for the Reachability Problem. *Fundamenta Informaticae* 46 (2001) 1-17.
3. K. Heljanko: *Deadlock and Reachability Checking with Finite Complete Prefixes*. Research Reports 56. Helsinki University of Technology (1999).

4. K. Heljanko: Minimizing finite complete prefixes. In *Proceedings of the Workshop Concurrency, Specification and Programming* (1999) 83-95.
5. K. Jensen (1992) *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. EATCS Monographs on Theoretical Computer Science.
6. K. Jensen and L. M. Kristensen: *Coloured Petri Nets - Modelling and Verification of distributed systems*. Draft Manuscript (January 2005).
7. K. Jensen and L. M. Kristensen: *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Lecture Notes (January 2006).
8. K. Jensen, S. Christensen and L. M. Kristensen: *CPN Tools State Space Manual*. Manual (2006).
9. V. Khomenko: *CLP Documentation and User Guide. Version 6.01*. Manual (2002).
10. V. Khomenko: *PUNF Documentation and User Guide. Version 3.01β*. Manual (2002).
11. V. Khomenko: *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD thesis. University of Newcastle upon Tyne (2003).
12. V. Khomenko and M. Koutny: Branching Processes of High-Level Petri Nets. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)* (2003).
13. V. Khomenko, M. Koutny and W. Vogler: Canonical Prefixes of Petri Net Unfoldings. In *Proceedings of the Conference on Computer-Aided Verification (CAV'02)* (2002).
14. V. E. Kozura: Unfoldings of Colored Petri Nets. In *Perspectives of System Informatics. 4th International Andrei Ershov Memorial Conference (PSI 2001)* (2001) 268-278.
15. K. L. McMillan: A Technique of State Space Search Based on Unfolding. *Formal Methods in System Design* 6(1) (1995) 45-65.
16. S. Melzer and S. Römer: Deadlock checking using net unfoldings. In *Proceedings of the Conference on Computer-Aided Verification (CAV'97)* (1997).
17. <http://wiki.daimi.au.dk/cpntools>
18. <http://java.sun.com/xml/jaxb/about.html>
19. <http://theoretica.informatik.uni-oldenburg.de/~pep/>

APPENDIX

Buffer(n)										
n	CP - net		StateSpaceGraph			Unfolding			Time in seconds	
	P	T	N	A	B	E	E _{cut}	CPN Tools	CPNunf	
1	2	2	2	2	3	2	1	0.00	0.00	
2	4	3	4	5	5	4	1	0.00	0.00	
3	6	4	8	12	13	7	1	0.00	0.00	
4	8	5	16	28	21	11	1	0.00	0.00	
5	10	6	32	64	31	16	1	0.01	0.01	
6	12	7	64	144	43	22	1	0.03	0.01	
7	14	8	128	320	57	29	1	0.06	0.01	
8	16	9	256	704	73	37	1	0.18	0.01	
9	18	10	512	1536	91	46	1	0.35	0.03	
10	20	11	1024	3328	111	56	1	1.57	0.03	
11	22	12	2048	7168	133	67	1	2.64	0.04	
12	24	13	4096	15360	157	79	1	8.09	0.06	
13	26	14	8192	32768	183	92	1	22.85	0.07	
14	28	15	16384	69632	211	106	1	84.76	0.09	
15	30	16	32768	147456	241	121	1	335.60	0.12	
16	32	17	65536	311296	273	137	1	1250	0.15	
17	34	18	131072	655360	307	154	1	5011	0.20	
20	40	21	2 ²⁰	2 ²⁰ . 5.75	421	211	1	-	0.37	

FreeStyle(n)										
n	CP - net		StateSpaceGraph			Unfolding			Time in seconds	
	P	T	N	A	B	E	E _{cut}	CPN Tools	CPNunf	
1	3	2	2	2	5	2	1	0.01	0.00	
2	3	2	4	8	10	4	2	0.01	0.00	
3	3	2	8	24	15	6	3	0.03	0.00	
4	3	2	16	64	20	8	4	0.03	0.00	
5	3	2	32	160	25	10	5	0.03	0.00	
6	3	2	64	384	30	12	6	0.06	0.00	
7	3	2	128	896	35	14	7	0.12	0.01	
8	3	2	256	2048	40	16	8	0.40	0.01	
9	3	2	512	4608	45	18	9	1.29	0.01	
10	3	2	1024	10240	50	20	10	3.82	0.01	
11	3	2	2048	22528	55	22	11	17.26	0.01	
12	3	2	4096	49152	60	24	12	71.53	0.01	
13	3	2	8192	106496	65	26	13	291.54	0.01	
130	3	2	2 ¹³⁰	2 ¹³⁰ . 130	650	260	130	-	0.48	

Conflict(n)										
n	CP - net		StateSpaceGraph			Unfolding			Time in seconds	
	P	T	N	A	B	E	E _{cut}	CPN Tools	CPNunf	
1	3	2	3	2	3	2	0	0.00	0.00	
3	7	6	7	6	7	6	0	0.00	0.00	
7	15	14	15	14	15	14	0	0.01	0.01	
15	31	30	31	30	31	30	0	0.01	0.01	
31	63	62	63	62	63	62	0	0.10	0.01	

Producer/Consumer(n)										
n	CP - net		StateSpaceGraph			Unfolding			Time in seconds	
	P	T	N	A	B	E	E _{cut}	CPN Tools	CPNunf	
1	6	4	5	4	8	4	0	0.00	0.00	
2	6	4	9	8	14	8	0	0.00	0.00	
3	6	4	13	12	20	12	0	0.01	0.00	
4	6	4	17	16	26	16	0	0.01	0.00	
5	6	4	21	20	32	20	0	0.01	0.00	
6	6	4	25	24	38	24	0	0.01	0.01	
7	6	4	29	28	44	28	0	0.01	0.01	
8	6	4	33	32	50	32	0	0.01	0.01	
9	6	4	37	36	56	36	0	0.01	0.01	
10	6	4	41	40	62	40	0	0.03	0.01	
20	6	4	81	80	122	80	0	0.03	0.04	
30	6	4	121	120	182	120	0	0.04	0.11	
40	6	4	161	160	242	160	0	0.04	0.17	
50	6	4	201	200	302	200	0	0.07	0.21	
60	6	4	241	240	362	240	0	0.10	0.27	
70	6	4	281	280	422	280	0	0.12	0.42	
80	6	4	321	320	482	320	0	0.17	0.56	
90	6	4	361	360	542	360	0	0.23	0.83	
100	6	4	401	400	602	400	0	0.26	1.27	
110	6	4	441	440	662	440	0	0.29	1.52	
120	6	4	481	480	722	480	0	0.37	2.03	
130	6	4	521	520	782	520	0	0.54	2.65	

Table 1. State space graph vs. Unfolding.

Designing coloured Petri net models: a method

Christine Choppy¹, Laure Petrucci¹, and Gianna Reggio²

¹ LIPN, Institut Galilée - Université Paris XIII, France

² DISI, Università di Genova, Italy

Abstract. When designing a complex system with critical requirements (e.g. for safety issues), formal models are often used for analysis prior to costly hardware/software implementation. However, writing the formal specification starting from the textual description is not easy. An approach to this problem has been developed in the context of algebraic specifications [5]. Here, we present a similar method, giving precise and detailed guidelines for writing coloured Petri nets.

Keywords: specification method, modelling method, coloured Petri net

1 Introduction

While formal specifications are well advocated when a good basis for further development is required, they remain difficult to write in general. Among the problems are the complexity of the system to be developed, and the use of a formal language. Hence, some help is required to start designing the specification, and then some guidelines are needed to remind some essential features to be described. [5] proposes a method, providing detailed and precise guidelines, for the development of specifications written using CASL [1], the Common Algebraic Specification Language, and CASL-LTL [13], an extension for dynamic systems specification, as target languages. However, this method could be used with quite a variety of target languages.

Petri nets have been successfully used for concurrent systems specification. Among their attractive features, is the combination of a graphical language and an effective formal model that may be used for formal verification. Expressiveness of Petri nets is dramatically increased by the use of high-level/coloured Petri nets, and also by the addition of modularity features allowing for quite large case studies.

While the use of Petri nets becomes much easier with the availability of high quality environments and tools, to our knowledge, little work has been devoted to a specification method for Petri nets. The aim of this work is to provide guidelines for coloured Petri net specification on the grounds of the aforementioned specification method. An initial approach was presented in [2]. In this paper, further work is achieved in different directions. We show how the relevant items can be identified in the description. We adapted in detail the method so as to encompass the coloured Petri net target, and achieved a full treatment of properties.

It is important to mention some facts about this work. First, the example developed here (which is classical for Petri nets) was designed by the authors of the paper who are not specialists in Petri nets. Hence, they had no prior knowledge of the usual coloured net modelling the problem, and were only given a textual description of the problem. Actually, the model obtained is slightly different from the usual one. Second, they have also tried out other examples, starting with place/transition nets. The other author did validate their approach afterwards. Third, this approach was successfully used by our master students. Therefore, its application to a large category of problems seems rather promising. It can however still be refined, so as to take into account more detailed features such as hierarchy/modularity, as mentioned in our conclusions.

The paper is structured as follows. The different steps of our method (finding events and state observers, looking for properties, modelling with a coloured net, checking the properties) are explained and illustrated on a case study, in sections 3, 4, 5 and 6 respectively. Finally, section 7 concludes and indicates issues for future work.

2 The method

The goal of the proposed method is to obtain a coloured Petri net modelling a given system hereafter denoted by the **System**. The general approach is described in Fig. 1.

The proposed method is based on two key ingredients (or constituent features, using the terminology of [5]) that are *events* and *state observers*. *Events* are, as usual, something happening in the life of the **System** (e.g. an action of some component, or a change in some part of the **System** or in the value of a condition) and are considered as atomic, with zero-duration, and no two events may happen simultaneously (thus, in the case two actions are happening together, there will be a unique event). A *state observer* instead defines something that may be observed on the states of the **System**, defined by the values of some type.

A first step consists in deriving the state observers and the events characterising the **System** from its textual description (Sect. 3). Associated properties are then determined and expressed, leading to possible modifications of state observers and events (Sect. 4). When reaching a stable set of events, state observers and properties, the coloured net can be built (Sect. 5) and the properties checked (Sect. 6). This analysis may lead to modifications of the model, in which case the process should be repeated.

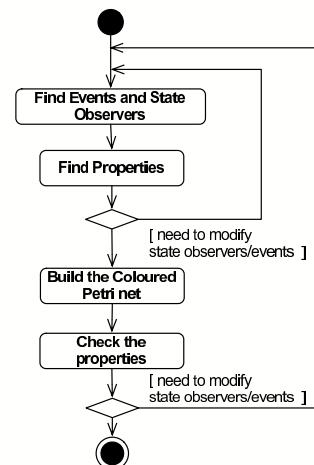


Fig. 1: Design method

3 Identifying events and state observers

3.1 Guidelines to find events and state observers

The first task of the proposed method is then to find the events and the state observers that are relevant for the System. We propose a standard technique to perform this activity: a grammar-based analysis of an informal description of the System, as advocated by classical object-oriented methods (see e.g. [6]).

More precisely, the starting point should be a *processing narrative*, as used in [12], for the System, that is a text in natural language describing its behaviour. If the informal description does not completely present the behaviour of the System, but, for example, motivates the need for building the System or justifies some of its features, then it should be modified both eliminating and adding new parts.

The text should then be examined, and the verbs, the nouns (or better the verbal and the noun phrases), and the adjectives should be outlined. Unless the same words are used for different meanings, phrases are outlined only once.

This may be achieved on two copies of the text, one where the verb phrases are outlined, and the other for the noun phrases. To save space in this paper we used only one copy of our case study, using two different compatible styles, and this lead us to note that verb phrases and noun phrases can be nested.

In general, the outlined verbs (or verbal phrases) should lead to find out the events, while the outlined nouns and adjectives should lead to find out the state observers and the datatypes. The style used has an influence, e.g. the use of active/passive forms. Since events are also changes in parts of the System, and some actions may not be explicit (e.g. the water reaches the maximum level, or the engine is broken), a careful attention should be paid to verbs like “to be”, “to become”, “to reach”, etc. Thus all outlined verbs are listed, grouping together the synonyms or different phrases refering to the same concept, and each one is examined in order to decide whether it should yield an event. Each event should then be given a name (an identifier), and should be accompanied by a short sentence describing it. If two events are always simultaneous, they should be joined into a unique event. Similarly, the outlined nouns and adjectives are listed, grouping synonyms, and examined in order to decide whether they yield datatypes or state observers.

Some potential cases are given below:

- if the noun denotes an active subpart of the System, it should not become a state observer, however it may be the case that the state of this subpart should be observed (e.g. if there is a user sending messages, check if the state of the user is relevant)
- similarly, for names of structural parts (or passive subparts) of the System (e.g. if two processes communicate by means of a channel, check if the status of the channel is relevant, for example, if it matters that it may be broken)
- the noun denotes data, it may be that it refers to some aspect of the System, and thus there should be an associated state observer.

- if the adjective refers to the System or to part of it, it should become a state observer of the form “is the adjective applicable to the System/its part?”

Each outlined state observer should then be given a name (an identifier) and a type, and should be accompanied by a short sentence describing what it observes in the System.

All the datatypes needed to type the state observers should be listed apart, together with a (chosen) name and if possible a definition or some operations.

Note 1. It may be quite helpful to group the events and the state observers, and, if these lists are not short, to add a title to these groups (e.g. events concerning the sender, or the receiver). These groupings are of course adopted in the above lists, and should be kept in further tables and formulas, so as to facilitate reading, eye-checking, and future modifications.

Note 2. To have to decide if a verbal phrase should be an event, and a noun phrase a state observer may lead to ask questions about the behaviour of the system (e.g. to decide if two actions are simultaneous).

Three lists are resulting from this step: (i) events, (ii) state observers, (iii) datatypes.

3.2 Case study: identifying events and state observers

The distributed database is a small example taken from [10] (vol. 1, pp. 21–25) which describes the communication among a set of database managers in a distributed system. The managers are supposed to keep their databases as identical as possible. Hence, each update must be followed by broadcasting the update to all the other managers, asking them to perform a similar update.

Even though this well-known example is small, it is complex enough to show how our method could help to specify it and obtain a coloured net model.

The informal description of this case study is given below with emphasis on **verbal phrases**, **noun phrases**, or **both** (when nested).

Informal description This example describes a very simple *distributed database with n different sites* (*n is a positive integer, which is assumed to be greater than or equal to 2*). Each **site maintains its own copy** of the whole database. On each site, a *local database manager handles all operations*.

Each manager is **allowed to update its own copy of the database**. Then, in order to **keep subsequent consistency** among all copies, it must **send a message to all the other managers** (*so that they can perform the same update* on their own copy of the database). In this example we are not interested in the actual update data.

Hence the **messages sent on the network to ensure the cooperation** between the different database managers require to **keep track only of the header information, i.e. the sender and the receiver** that are **two different database managers**. When a database manager makes an update, it

must then **inform** (by sending a message) all other ($n-1$) managers. Before a similar operation can take place again, all the updates should be finished. Therefore, the manager who has asked for an update has to wait until all other managers have sent back an *acknowledgement*. When a database manager is informed of a new update, it must achieve the corresponding update on its local copy and send back an acknowledgement.

Informal description analysis The first task to achieve is to analyse the textual description (as described in Sect. 3.1) so as to find out relevant elements about the *events*, the state of the system (expressed in terms of *state observers*), and the *data* involved (either directly mentioned in the text, or returned by the state observers). We first list the verb phrases and the noun phrases and discuss for each whether it leads to relevant information. Redundant texts (that describe the same thing) are grouped together. Then, the events, state observers and datatypes lists are extracted.

Verbs (verbal phrases)

- **maintains its own copy** \Rightarrow no event (a qualification of site)
- **handles all operations** \Rightarrow no event (a qualification of database manager)
- **allowed to update its own copy of the database** \Rightarrow no event (a qualification of database manager)
- **keep subsequent consistency** \Rightarrow no event (a motivation of some actions)
- for the following verbal phrases
 - **send a message to all the other managers**
 - **inform (by sending a message) all other ($n-1$) managers**
 - **has asked for an update**
- \Rightarrow the inform EVENT is to inform that an initial update was done and that a manager **has asked for an update** (corresponding to the one it just did)
- **perform the same update** \Rightarrow the corrUpd EVENT is that a similar update (corresponding to the new one) is achieved
- **to ensure the cooperation between the different database managers** \Rightarrow no event (a motivation of some actions)
- **keep track only of the header information, i.e. the sender and the receiver** \Rightarrow this qualifies what is considered in the messages
- **Before a similar operation can take place again, all the updates should be finished** \Rightarrow two EVENTs here, one (allUpd) is that **all updates are finished**, and the other (update) is to **perform an initial update** (referred to by **similar operation**)
- **wait until all other managers have sent back an acknowledgement** \Rightarrow **wait** cannot be associated with an event
- **all other managers have sent back an acknowledgement** \Rightarrow recAllAck EVENT
- **a database manager is informed of an initial update** \Rightarrow informed EVENT

- **it must achieve the corresponding update** \Rightarrow the corrUpd EVENT is that a database manager achieves the update corresponding to the initial one made (already mentioned)
- **and send back an acknowledgement** \Rightarrow the updAck EVENT is that a local database manager sends back an acknowledgement

List of events

- inform: a database manager informs (by sending a message) all other ($n - 1$) managers that an initial update was made
- allUpd: all updates are finished
- update: a database manager performs an initial update
- recAllAck: all other managers have sent back an acknowledgement, thus all acknowledgements are received
- informed: a database manager is informed of an initial update
- corrUpd: a database manager achieves the update corresponding to the initial one made
- updAck: a database manager sends back an acknowledgement

Note that since communication is asynchronous, to inform and to be informed are two different events. The issue whether there should be a distinction between events allUpd and recAllAck should be solved.

Nouns (noun phrases)

- *distributed database with n different sites* \Rightarrow this refers to the whole system, so it does not apply to a state observer or data
- *n is a positive integer, greater than or equal to 2* \Rightarrow a constant value of type integer (or natural)
- sites of the distributed database and the local managers
 - *site* \Rightarrow this is a “structural part”, each site is referred to by its identifier (that is a datatype), and a question is whether its state should be characterised
 - *local database manager* \Rightarrow associated with each site, and a question is whether its state should be characterised
 - *its local copy* \Rightarrow it is managed by the local database manager
- several parts of the description mention messages, and will lead to the definition of the MESSAGE datatype
 - *message ... so that they can perform the same update* \Rightarrow a datatype for messages requiring an update;
 - *messages sent on the network* \Rightarrow one question is whether the communication is synchronous or not, and it is decided in this case that it is asynchronous. Therefore, a state observer provides the messages sent in the network.
 - *the header information, i.e. the sender and the receiver* \Rightarrow the message datatype should include the sender and the receiver
 - *acknowledgement* \Rightarrow a datatype for another kind of messages

List of state observers

- **inTransit**: Set(MESSAGE) returns the messages in transit in the network

Let us note that, given the study of the noun phrases reported above, we have only one state observer at this point which is not much. More state observers will emerge from the next step when working on properties.

List of datatypes

- DBM: identities of the sites
- INT: integers, with a constant value n greater than 2
- MESSAGE: messages that are either update requests or acknowledgements, and provide only the sender and the receiver of the message. At this stage, we can provide a provisional definition for this type:

MESSAGE ::= Req (DBM,DBM) | Ack (DBM,DBM)

4 Finding the properties

4.1 Guidelines to find the properties

Let us assume that we have the three lists (events, state observers and datatypes) produced in the previous step. Now we consider the task of finding the most relevant/characteristic properties of the System and of its behaviour, and to express them in terms of the identified events and state observers (using also the identified datatypes). Our method helps to find out these properties by providing precise guidelines (inspired by [5]) for the net designer to examine all relevant relationships among events and state observers, and all aspects of events and state observers.

The behaviour of the System can be seen as the set of all its possible “lives”, where a *life* is a sequence of states and events

$s_0 e_1 s_1 e_2 \dots s_{n-1} e_n s_n e_{n+1} s_{n+1} \dots$

where each state s_i defines the values of the state observers, and s_0 is an initial state.

For each state observer SO returning a value of type DT (declared as SO: DT), we look for:

- properties on the values returned by SO (e.g. assuming DT = INT, SO should always return positive values);
- properties relating the values observed by SO with those returned by other state observers (e.g. the value returned by SO is greater than the value returned by state observer SO₁).

For each event EV we look for pre and postconditions and there may be other properties (e.g. liveness and incompatibility between events).

precondition is what must hold before EV happens, i.e. a condition on the state observers such that if s is a state of the System in which EV happens, then this condition holds on s

postcondition is what must hold after EV happened, i.e. a condition on the state observers such that if s is a state of the System after EV happened, then this condition holds on s

- more properties** Consider a life of the System where EV happens
 $s_0 e_1 s_1 e_2 \dots s_{n-1} e_n s_n$ EV $s'_1 e'_1 s'_2 e'_2 \dots$
 $s_0 e_1 s_1 e_2 \dots s_{n-1} e_n s_n$ is a possible past of EV, whereas $s'_1 e'_1 s'_2 e'_2 \dots$ is a possible future of EV.
- on the past** properties on the possible pasts of EV (e.g. the System was in a state such that the values returned by the state observers satisfy some condition, or a given event happened)
- on the future** properties on the possible futures of EV (e.g. the System will reach a state such that the values returned by the state observers satisfy some condition, or a given event will happen)
- vitality** when it should be possible for EV to happen (e.g. if state s satisfies some condition, then EV may happen in s , if state s satisfies some condition, then eventually EV will happen, ...)
- incompatibility** the events EV₁ such that there cannot exist a state of the System in which both EV and EV₁ may happen

Obviously, there may be some conditions fulfilled by the possible initial states of the System. Conversely, we propose to characterise the final states when relevant.

- initial condition** a property about state observers that must hold in any initial state of the System.
- final condition** a property about state observers that must hold in any final state of the System (irrelevant if the System never terminates).

While writing the properties it may happen that:

- we discover the need for operations over the datatypes, or that their definition should be made more precise and detailed \Rightarrow modify the definition of the data types accordingly
- we need new state observers and perhaps new datatypes to express what they observe (e.g. to express some property about an event) \Rightarrow add them
- we need new events, or an event has to be split into several other ones, or different events turn out to be the same \Rightarrow add/split/identify the events as required.

4.2 How to find the properties: case study

The text analysis did not bring much in terms of state observers, therefore event properties are first expressed in natural language, and once the properties are identified, the corresponding state observers will emerge. The only event leading to properties other than the pre/postconditions is **update**. When expressing properties, we use primed notations for the value of state observers after an event has taken place. Recall that - and + denote deletion and addition of an element to a set or to a multiset.

Event properties

update: (a database manager d performs an initial update)

precondition no update is taking place (thus we introduce the state observer updating: BOOL) and d is inactive, i.e. in the inactive state (thus we introduce the state observer inactive: Set(DBM)):

$$\text{updating} = \text{false} \wedge d \in \text{inactive}$$

postcondition d performed an update (thus we introduce the state observer updated: DBM+, where the values of datatype DBM+ are those of DBM plus None), d is not inactive anymore, and an update is taking place: $\text{inactive}' = \text{inactive} - d \wedge \text{updated}' = d \wedge \text{updating}' = \text{true}$

more it should always be eventually possible to make an initial update

inform: (a database manager d informs, by sending a message, all other managers that an initial update was performed)

precondition d performed an initial update:

$$\text{updated} = d$$

postcondition d is waiting for the other sites to perform the subsequent updates (thus we introduce the state observer waiting: DBM+), and the messages sent to require the subsequent updates are in transit on the network. Moreover, we add to the datatype DBM an operation AllUpdReq producing all update request messages:

$$\text{updated}' = \text{None} \wedge \text{waiting}' = d \wedge \text{inTransit}' = \text{inTransit} + \text{AllUpdReq}(d)$$

informed: (a database manager d is informed of an initial update)

precondition there is a message in transit requiring d to make a subsequent update from the site d1, and site d is inactive:

$$\text{Req}(d1,d) \in \text{inTransit} \wedge d \in \text{inactive}$$

postcondition the request message is received by site d, i.e. it is included in the received messages, thus we introduce the state observer recMsg: Set(MESSAGE), and d is performing the required update, i.e. it is in the performing state, thus we introduce the state observer performing: Set(DBM):

$$\text{inTransit}' = \text{inTransit} - \text{Req}(d1,d) \wedge \text{inactive}' = \text{inactive} - d \wedge$$

$$\text{performing}' = \text{performing} + d \wedge \text{recMsg}' = \text{recMsg} + \text{Req}(d1,d)$$

corrUpd: (a database manager makes the update corresponding to the initial one)

The occurrence of this event corresponds to a database manager reaching the state performing, thus it is useless and we drop it from the events list.

updAck: (a database manager d sends back an acknowledgement)

precondition the database manager d performed the update, so it was in the performing state, and has received a request message from some d1: $d \in \text{performing} \wedge \text{Req}(d1,d) \in \text{recMsg}$

postcondition the database manager d is now in the inactive state, and an acknowledgement message is in transit on the network:

$$\text{performing}' = \text{performing} - d \wedge \text{inactive}' = \text{inactive} + d \wedge$$

$$\text{recMsg}' = \text{recMsg} - \text{Req}(d1,d) \wedge \text{inTransit}' = \text{inTransit} + \text{Ack}(d, d1)$$

allUpd: (all subsequent updates are finished)

This is the same as recAllAck, since an acknowledgement is sent when an update is done. Thus this event will be removed from the event list.

recAllAck: (all acknowledgements are received by database manager d)

precondition d is waiting for the acknowledgments, and all acknowledgement messages are in transit on the network (we assume that they are all received together):

$$\text{waiting} = d \wedge \text{AllAcks}(d) \subseteq \text{inTransit}$$

postcondition d is not waiting, the update acknowledgement messages are not in transit on the network anymore, and no update is taking place.

$$\text{waiting}' = \text{undefined} \wedge \text{updating} = \text{false} \wedge$$

$$\text{inTransit}' = \text{inTransit} - \text{AllAcks}(d) \wedge \text{inactive}' = \text{inactive} + d$$

Thus, while expressing the properties of the events, we have identified the following new state observers:

(New) List of state observers

inTransit: Set(MESSAGE) returns the messages in transit on the network

inactive: Set(DBM) returns the set of the inactive database managers.

updated: DBM+ returns the database manager that did the initial update, or None

waiting: DBM+ returns the database manager that is waiting, after having informed the others that a subsequent update is required, or None.

performing: Set(DBM) returns the database managers performing the subsequent updates

recMsg: Set(MESSAGE) returns the update request messages received by the database managers

updating: BOOL returns true if an update is taking place, and false otherwise.

(New) datatypes and operations over the DBM datatype

- DBM+ ::= ... : DBM | None
- AllUpdReq: DBM → Set(MESSAGE)
- AllUpdReq(d) = { Req(d, d1) | d1:DBM, d ≠ d1 }
- AllAcks: DBM → Set(MESSAGE)
- AllAcks(d) = { Ack(d1, d) | d1:DBM, d ≠ d1 }

State observers properties

inTransit: Set(MESSAGE) (returns the messages in transit on the network)

- Requests from two different database managers are not in transit simultaneously:

$$\text{Req}(d1, d1') \in \text{inTransit} \wedge \text{Req}(d2, d2') \in \text{inTransit} \implies d1 = d2$$

- Acknowledgements to two different database managers are not in transit simultaneously:

$$\text{Ack}(d1, d1') \in \text{inTransit} \wedge \text{Ack}(d2, d2') \in \text{inTransit} \implies d1' = d2'$$

- There are messages in transit on the network iff an update is taking place:

$$\text{inTransit} \neq \emptyset \equiv \text{updating} = \text{true}$$

- inactive:** Set(DBM) (returns the set of the inactive database managers)
- An inactive database manager is neither waiting nor performing nor did an update:
 $d \in \text{inactive} \implies (d \neq \text{waiting} \wedge d \notin \text{performing} \wedge d \neq \text{updated})$
 - A database manager is either inactive, performing, waiting or just did an initial update:
 $d \in \text{inactive} \vee d \in \text{performing} \vee d = \text{waiting} \vee d = \text{updated}$
- updated:** DBM+ (returns the database manager that did the initial update, or None)
- A database manager that did the initial update is neither waiting nor performing nor inactive:
 $\text{updated} \neq \text{None} \implies (\text{updated} \neq \text{waiting} \wedge \text{updated} \notin \text{performing} \wedge \text{updated} \neq \text{inactive})$
 - If there is a database manager that did the initial update then no other one is waiting, and vice versa
 $\neg(\text{updated} \neq \text{None} \wedge \text{waiting} \neq \text{None})$
- waiting:** DBM+ (returns the database manager that is waiting, after having informed the others that a subsequent update is required, or None)
- A database manager that is waiting neither just did an initial update nor is performing nor is inactive:
 $\text{waiting} \neq \text{None} \implies (\text{waiting} \neq \text{updated} \wedge \text{waiting} \notin \text{performing} \wedge \text{waiting} \neq \text{inactive})$
- performing:** Set(DBM) (returns the database managers performing the subsequent updates)
- A performing database manager is neither waiting, nor inactive nor just did an initial update:
 $d \in \text{performing} \implies (d \neq \text{waiting} \wedge d \notin \text{inactive} \wedge d \neq \text{updated})$
- recMsg:** Set(MESSAGE) (returns the update request messages received by the database managers)
- A message cannot be received and in transit on the network simultaneously:
 $\text{recMsg} \cap \text{inTransit} = \emptyset$
- updating:** BOOL (returns true if an update is taking place, and false otherwise)
- If an update is taking place, not all database managers are inactive, and if one of them is waiting then there are messages travelling on the network or received:
 $\text{updating} = \text{true} \implies (\exists d. d \notin \text{inactive}) \wedge (\text{waiting} \neq \text{None} \implies \text{inTransit} \cup \text{recMsg} \neq \emptyset)$

initial state Initially all database managers are inactive, no update is taking place, and there is no message in transit on the network nor received
 $[(\forall d. d \in \text{inactive}) \wedge \text{waiting} = \text{None} \wedge \text{updated} = \text{None} \wedge \text{performing} = \emptyset] \wedge \text{updating} = \text{false} \wedge \text{inTransit} = \emptyset \wedge \text{recMsg} = \emptyset$

final state There should not be any final state, since the distributed database system will never terminate.

5 Modelling using coloured Petri nets

5.1 Building the Coloured Petri Net

At this point, we can assume that we have the list of state observers and events (plus the list of used datatypes with their operations) resulting from the previous steps, and that for each event the pre/postconditions have been expressed. Recall that we have collected also other properties about the state observers and the events, that will be checked in the last step of the method, once the net is built.

We now show how starting from the above elements, derived from the analysis of the System, we can build a coloured Petri net modelling the System itself. Obviously, the net cannot be built in all cases, so we present a canonical form for events, state observers and pre/postconditions that allow the procedure to result in a coloured net. Whenever the form is not canonical, it is possible to do some refactoring replacing the used state observers, events, and datatypes with other ones able to model the System in an equivalent way; analogously it is possible to replace the pre/postconditions with equivalent formulae. This scheme is sketched in Fig 2. We present later various patterns showing how to do the refactoring in some quite common cases.

The *canonical form* requires that:

1. each state observer has type $MSet(T)$ for some type T ;
2. the pre/postconditions have the following form¹

$$\begin{aligned} \text{pre } & (\wedge_{i=1,\dots,n} exp_i \leq SO_i) \wedge (\wedge_{j=n+1,\dots,m} exp_j \leq SO_j) \wedge cond, \\ \text{post } & (\wedge_{i=1,\dots,n} SO'_i = SO_i - exp_i + exp'_i) \wedge (\wedge_{j=n+1,\dots,m} SO'_j = SO_j - exp_j) \wedge \\ & (\wedge_{h=m+1,\dots,r} SO'_h = SO_h + exp'_h) \wedge cond', \end{aligned}$$

where

- SO_l ($l = 1, \dots, r$) are all distinct,
- the free variables occurring in exp_l and exp'_l ($l = 1, \dots, r$) may occur in $cond$ and in $cond'$,
- no state observer occurs in $cond$, $cond'$, exp_l and exp'_l ($l = 1, \dots, r$),
- and $cond$ and $cond'$ are first order formulae.

The pre/postconditions on event EV in canonical form require that:

- before EV occurs some values are contained in SO_1, \dots, SO_n and that such values are deleted and that other values are added when EV occurs;
- before EV occurs some values are contained in SO_{n+1}, \dots, SO_m and that such values are deleted when EV occurs, but nothing is added;
- some values are added to SO_{m+1}, \dots, SO_r when EV occurs.

¹ \leq , $+$ and $-$ denote respectively the inclusion, union and the difference between multisets.

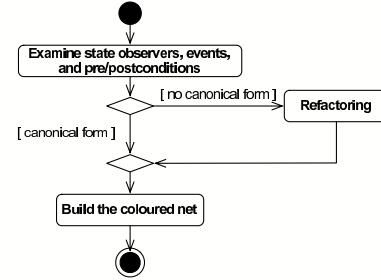


Fig. 2: Deriving the coloured net

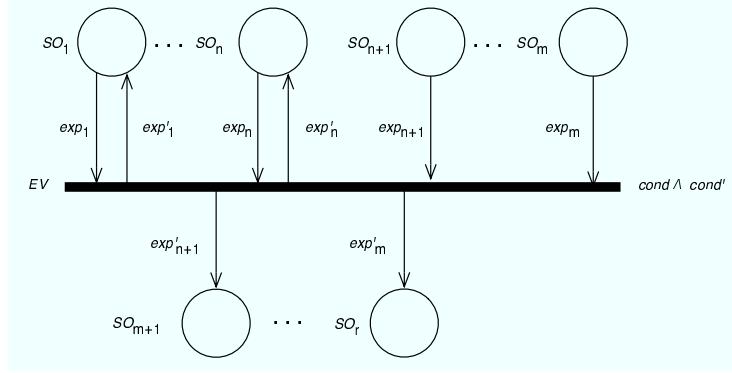


Fig. 3. Deriving arcs from pre/post-conditions

– whereas $cond$ expresses some condition over exp_1, \dots, exp_m , and $cond'$ some condition over exp'_1, \dots, exp'_r .

Thus it will be possible to realise the behaviour of EV described by these pre/postconditions by the flowing of valued tokens throughout the places of a coloured net.

Assume that all elements are in the canonical form. The coloured Petri net is defined as follows. The state observers and the events determine the places and the transitions, while the pre/postconditions determine the arcs. Each state observer $SO : MSet(T)$ becomes a place named SO coloured by T , and each event EV becomes a transition, named EV . If the pre/postconditions of an event EV have the same form as in (2), then the set of arcs is as pictured in Fig. 3.

Petri net design patterns The method proposed in this paper offers various patterns that may help to refactorise the state observers, the events and the pre/postconditions to reach a canonical form, and thus to be able to generate a coloured Petri net. Similar to design patterns in [8] in a specific case several patterns may be applicable. In this subsection we present the two patterns that will be applied in the case study.

Black-Box Value Pattern This pattern may be applied whenever we want to reflect in the Petri net that a state observer $SO : T$ observes values considered as black-blox, that is we are not interested in exploiting the structure (if any) of the observed values, i.e. of T .

Assume we have a state observer $SO : T$ where SO appears in the pre/postconditions only in atoms having either the form $SO = exp$ or $SO' = exp'$, and SO does not appear in exp nor exp' .²

² This is not restrictive at all, since a complex formula $cond$ in which SO appears can always be transformed into an equivalent one $SO = exp \wedge cond[exp/SO]$.

The logical specification of the System may be refactorised by replacing SO with $\underline{\text{SO}} : \text{MSet}(\mathbf{T})$, while the pre/postconditions should be transformed as follows:

- pre: $\text{SO} = \text{exp} \wedge \text{cond}$ post: $\text{SO}' = \text{exp}' \wedge \text{cond}'$ should become:
pre: $\text{exp} \leq \underline{\text{SO}} \wedge \text{cond}$ post: $\underline{\text{SO}}' = \underline{\text{SO}} - \text{exp} + \text{exp}' \wedge \text{cond}'$
- pre: $\text{SO} = \text{exp} \wedge \text{cond}$ post: cond' where SO does not appear, should become:
pre: $\text{exp} \leq \underline{\text{SO}} \wedge \text{cond}$ post: $\underline{\text{SO}}' = \underline{\text{SO}} - \text{exp} \wedge \text{cond}'$
- pre: cond where SO does not appear, post: $\text{SO}' = \text{exp}' \wedge \text{cond}'$ should become:
pre: $X \leq \underline{\text{SO}} \wedge \text{cond}$ (X having type \mathbf{T}) post: $\underline{\text{SO}}' = \underline{\text{SO}} - X + \text{exp}' \wedge \text{cond}'$

We can prove that if initially $\text{SO} = \text{exp}$, and thus $\underline{\text{SO}} = \{\text{exp}\}$, then always $\text{size}(\underline{\text{SO}}) = 1$ (thus we are sure that SO is correctly realised to return in each state a unique value).

Set Value Pattern This pattern may be applied to a state observer $\text{SO} : \text{Set}(\mathbf{T})$ whenever we want the elements of the observed sets to be realised as tokens typed by \mathbf{T} flowing in the Petri net.

Assume we have the state observer $\text{SO} : \text{Set}(\mathbf{T})$, and that the pre/postcondition have the form:

pre: $(\text{exp}_1 \cup \text{exp}_2) \subseteq \text{SO} \wedge \text{cond}$ post: $\text{SO}' = (\text{SO} - \text{exp}_2) \cup \text{exp}_3 \wedge \text{cond}'$
where exp_1 , exp_2 and exp_3 are sets (also empty), and SO does not appear in exp_1 , exp_2 , exp_3 , cond and cond' .

The precondition requires that SO includes the elements in $\text{exp}_1 \cup \text{exp}_2$, whereas the occurrence of EV requires that only the elements in exp_2 are removed from SO , and that the elements of exp_3 are newley added to SO .

The logical specification of the System may be refactorised by replacing SO with $\underline{\text{SO}} : \text{MSet}(\mathbf{T})$, and transforming the pre/postconditions as follows:

pre: $(\text{exp}_1 + \text{exp}_2) \leq \underline{\text{SO}} \wedge \text{cond}$
post: $\underline{\text{SO}}' = \underline{\text{SO}} - (\text{exp}_1 + \text{exp}_2) + (\text{exp}_1 + \text{exp}_3) \wedge \text{cond}'$ Note that to get the canonical form exp_1 should first be deleted and then added again to SO .

Unfortunately this refactoring does not guarantee that $\underline{\text{SO}}$ always returns a set, thus we should add the following property to the System, to be checked at a later stage:

$$\neg \exists x . \{x, x\} \leq \underline{\text{SO}} \text{ (i.e. } \underline{\text{SO}} \text{ is a set).}$$

5.2 Modelling with coloured Petri nets: Case study

In Tab. 1 and 2, we summarise the events, their pre/postconditions, the state observers and the datatypes used to model the distributed database system.

The logical specification of the database system produced up to now is not in the canonical form that allows to generate a coloured Petri net, first of all because it uses state observers not typed by multisets. Hence, we now apply the patterns proposed in Section 5.1. For the sake of simplicity, we use the same

Events	State Observers	Datatypes
update	inTransit: Set(MESSAGE)	DBM
inform	inactive: Set(DBM)	MESSAGE ::= Req(DBM, DBM) Ack(DBM, DBM)
informed	updated: DBM +	DBM + ::= ...: DBM None
updAck	waiting: DBM +	BOOL ::= true false
recAllAck	performing: Set(DBM)	
	recMsg: Set(MESSAGE)	
	updating: BOOL	

Table 1. Elements of the distributed database specification

Event	pre	post
update	updating = false \wedge $d \in \text{inactive}$	updating' = true \wedge $\text{inactive}' = \text{inactive} - \{d\} \wedge$ $\text{updated}' = d$
inform	updated = d	updated' = None \wedge waiting' = d \wedge $\text{inTransit}' = \text{inTransit} \cup \text{AllUpdReq}(d)$
informed	$\text{Req}(d_1, d) \in \text{inTransit} \wedge$ $d \in \text{inactive}$	$\text{inTransit}' = \text{inTransit} - \{\text{Req}(d_1, d)\} \wedge$ $\text{inactive}' = \text{inactive} - \{d\} \wedge$ $\text{performing}' = \text{performing} \cup \{d\} \wedge$ $\text{recMsg}' = \text{recMsg} \cup \{\text{Req}(d_1, d)\}$
updAck	$d \in \text{performing} \wedge$ $\text{Req}(d_1, d) \in \text{recMsg}$	$\text{performing}' = \text{performing} - \{d\} \wedge$ $\text{recMsg}' = \text{recMsg} - \{\text{Req}(d_1, d)\} \wedge$ $\text{inactive}' = \text{inactive} \cup \{d\} \wedge$ $\text{inTransit}' = \text{inTransit} \cup \text{Ack}(d, d_1)$
recAllAck	waiting = d \wedge $\text{AllAcks}(d) \subseteq \text{inTransit}$	waiting' = None \wedge updating = false \wedge $\text{inTransit}' = \text{inTransit} - \text{AllAcks}(d) \wedge$ $\text{inactive}' = \text{inactive} \cup \{d\}$

Table 2. Pre/postconditions of the distributed database specification

names for the new state observers introduced by the refactoring. In Tab. 3 and 4 present the resulting events, state observers, datatypes and pre/postconditions. Note that the datatypes are unchanged.

The coloured net derived from this canonical form is presented in Fig. 4. We used the coloured Petri nets tool CPNtools [7], and its associated CPNML language. The declarations are shown in Fig. 5. The DBM+ colour set is declared DBMP as a union of database managers ($\text{man}:DBM$) and value None.

6 Checking the properties

6.1 Checking the properties of the System

The previous steps of our design method did exhibit several properties which must be satisfied by the System. These properties should be expressed accord-

Events	State Observers	Datatypes
update	inTransit: MSet(MESSAGE)	DBM
inform	inactive: MSet(DBM)	MESSAGE ::= Req(DBM, DBM) Ack(DBM, DBM)
informed	updated: MSet(DBM+)	DBM+ ::= ...: DBM None
updAck	waiting: MSet(DBM+)	BOOL ::= true false
recAllAck	performing: MSet(DBM) recMsg: MSet(MESSAGE) updating: MSet(BOOL)	

Table 3. Elements of the distributed database specification, in canonical form

Event	pre	post
update	false \leq updating \wedge d \leq inactive \wedge X \leq updated	updating' = updating - false + true \wedge inactive' = inactive - d \wedge updated' = updated - X + d
inform	d \leq updated \wedge X \leq waiting	updated' = updated - d + None \wedge waiting' = waiting - X + d \wedge inTransit' = inTransit + AllUpdReq(d)
informed	Req(d1, d) \leq inTransit \wedge d \leq inactive	inTransit' = inTransit - Req(d1, d) \wedge inactive' = inactive - d \wedge performing' = performing + d \wedge recMsg' = recMsg + Req(d1, d)
updAck	d \leq performing \wedge Req(d1, d) \leq recMsg	performing' = performing - d \wedge recMsg' = recMsg - Req(d1, d) \wedge inactive' = inactive + d \wedge inTransit' = inTransit + Ack(d, d1)
recAllAck	d \leq waiting \wedge AllAcks(d) \leq inTransit \wedge X \leq updating	waiting' = waiting - d + None \wedge inTransit' = inTransit - AllAcks(d) \wedge updating' = updating - X + false \wedge inactive' = inactive + d

Table 4. Pre/postconditions of the canonical distributed database specification

ing to the language accepted by the coloured Petri nets tool to be used. Then the properties should be checked using the tool. One possibility is to build the occurrence graph and check that all states generated satisfy the properties.

In case some properties do not hold, the designer should look up for the causes of the problem by e.g. closely examining the states not satisfying the property and the paths leading to these states. This will give insight to locate the source of the problem. The model will then have to be modified accordingly, and the properties check repeated until all properties hold. It might also be the case that some properties derived from the informal specification are not correctly expressed. Then the properties should be changed and the new ones checked.

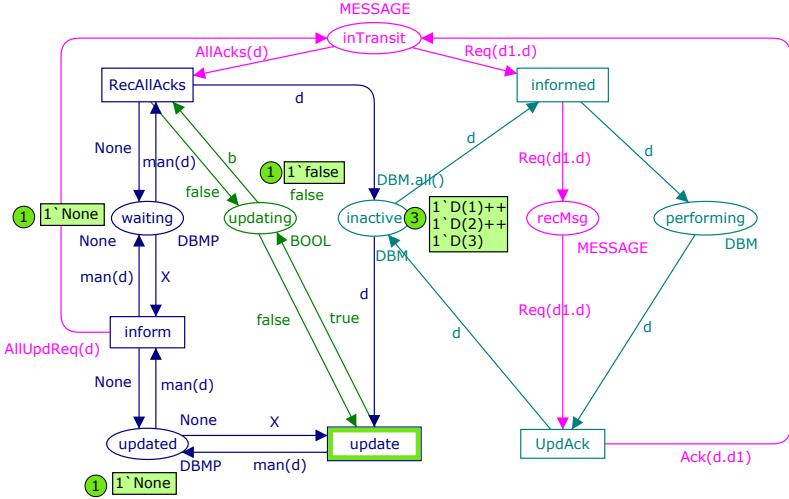


Fig. 4. Distributed database coloured Petri net

```

val nbdbm=3;
colset DBM=index D with 1..nbdbm;
colset DBMP=union man:DBM + None;
colset PAIRDBM = product DBM * DBM;
colset MESSAGE=union Req:PAIRDBM + Ack:PAIRDBM;
var d,d1:DBM;
var X: DBMP;
var b:BOOL;
fun AllUpdReq d = filter (fn Req(x,y) => (x<>y) andalso (x=d) | Ack(_,_) => false)
    (MESSAGE.all());
fun AllAcks d = filter (fn Req(_,_) => false | Ack(x,y) => (x<>y) andalso (y=d))
    (MESSAGE.all());

```

Fig. 5. Declared types and functions for the distributed database Petri net

6.2 Checking the properties of the case study

The various properties were checked on the state space graph. For most of them, it boils down to checking that the set of graph nodes satisfying the negated property is empty. We now explain a representative excerpt of the properties, the others are given in [4], together with a picture of the analysis page.

The property of Fig. 6 is part of the state observers properties. It states that a *database manager is either inactive, performing, waiting or just did an initial update*:

$$d \in \text{inactive} \vee d \in \text{performing} \vee d = \text{waiting} \vee d = \text{updated}$$

`DBM.all()` is the set of database managers, thus the union (denoted `++`) of the places (`inactive`, `performing`, `updated` and `waiting`) database manager multisets should be equal (negated by `<><>`) to `DBM.all()`. Note that places `waiting` and `updated` may contain value `None` and this value should not be considered, and this is taken care of by function `RemoveNone()`.

```

fun RemoveNone dbms =
    ext_col (fn man db => db) (filter (fn None => false | _ => true) dbms);

fun alldbmsonce() = PredAllNodes(fn n =>
    Mark.Database'inactive 1 n ++
    Mark.Database'performing 1 n ++
    RemoveNone(Mark.Database'updated 1 n) ++
    RemoveNone(Mark.Database'waiting 1 n) <><> DBM.all());

```

Fig. 6. Property: state of all database managers

The following property refers to one of the properties brought up by the description analysis. *There is at most one site waiting or that did an initial update (updated):*

$$\neg(\text{updated} \neq \text{None} \wedge \text{waiting} \neq \text{None})$$

To check this property, we find the maximum number of tokens in places waiting and updated together, without considering the `None` value. This is done by checking this number for each state in the graph and taking the maximum of the previous result and the value for the current state, using the function in Fig. 7. After examining all states, the result is 1, therefore the property is satisfied.

```

fun maxupdate () = SearchAllNodes(fn _ => true,
    fn n => size(RemoveNone(Mark.Database'waiting 1 n)) +
        size(RemoveNone(Mark.Database'updated 1 n)),
    0, Int.max);

```

Fig. 7. Property: only one database manager did an initial update

Finally, *if an update is taking place, not all database managers are inactive, and if one of them is waiting then there are messages travelling on the network or received:*

$$\text{updating} = \text{true} \implies (\exists d. d \notin \text{inactive}) \wedge (\text{waiting} \neq \text{None} \implies \text{inTransit} \cup \text{recMsg} \neq \emptyset)$$

This property, expressed in Fig. 8 is split into two functions: `upd1()` to check the first part concerning the database managers, and `upd2()` to check the second part concerning the messages on the network.

7 Conclusion

Designing a formal specification has proved to be important to check properties of a system prior to hardware and software costly implementation. However, even if such an approach reduces both the costs and the experimenting time, designing a formal model is difficult in general for an engineer.

As mentioned in the introduction, to our knowledge little work was devoted to a specification method for Petri nets. However, we would like to mention some work done by M. Heiner and M. Heisel in [9] to combine place/transition

```

fun upd1() = PredAllNodes(fn n =>
  if (Mark.Database'updating 1 n == 1'true andalso
      Mark.Database'waiting 1 n <><> 1'None)
  then (Mark.Database'inTransit 1 n ++
        Mark.Database'recMsg 1 n == empty)
  else false);

fun upd2() = PredAllNodes(fn n =>
  if (Mark.Database'updating 1 n == 1'true)
  then (Mark.Database'inactive 1 n == DBM.all())
  else false);

```

Fig. 8. Property: an update is taking place

nets with Z specifications so as to reduce the net complexity. In [11], another approach is to rely on problem frames concepts to structure the problem before developing the Petri net.

This paper gives guidelines to help with the design process. It has proven successful with people who are not used to model with Petri nets, hence a positive point w.r.t. the applicability of the design methodology.

The main idea is to derive key features from the textual description of the problem to model, in a rather guided manner so as to deduce the important entities handled, and then to transform all this into Petri net elements. At the same time, some properties inherent to the system appear, that are also formalised and should be proven valid on the model at an early stage. When a coloured net is obtained, with these properties satisfied, further analysis can be carried out, leading to possible changes in the specification.

Our method is inspired by the one developed in [5] for simple dynamic systems specification with the CASL-LTL algebraic specification language, which also requires to look for state observers, events (or rather elementary interactions), and datatypes, but in addition provides an extensive list of potential properties one should look for. This way of handling properties has the advantage of giving ideas on the potential properties, with the drawback of systematic long lists.

While in [2], the initial approach presented kept these properties list, here we adapted the method so as to guide the search for properties in a "light" way.

In [3], we developed this method for place/transition nets with several examples. Place/transition nets could easily be used when the involved datatypes are boolean or natural numbers (and of course if the size and complexity of the problem is reasonable). Since this is a simpler case (tokens do not have a value, and the matching mechanism with the arc labels is very simple), we could develop a more systematic guidance of the specification development. In [4], we address also the issue of the choice of the appropriate family of Petri nets that may be hinted by the datatypes needed in a case study.

In this paper, we have used the classical distributed database problem as a running example so as to explain the design methodology step by step.

For this case study we present a choice for state observers that take the whole state of the system as an argument. We could have taken another option, to have

a function yielding, for any database manager site, its state, and clearly the way to the coloured net would have been less straight. Future work will detail even more the different ways to transform a state observer into a place.

In this work, we have stuck to commonly used datatypes, but a designer could write his own complex types and functions to be used by his coloured net. Reflecting them in the net is then more complex and must be done in a rigorous way so as to ensure the applicability and the success of the approach.

Moreover, a large specification is often designed in a modular way. This is not tackled here, but including such features, e.g. hierarchies in coloured nets, is an important issue that we plan to address in the future. For instance, if repeated patterns are found in the Petri net, then they can be put in subnets, and a hierarchy may be introduced. If the net exhibits some symmetries, some folding may occur, and the appropriate colors are introduced. An evaluation of the method will be carried out in the near future.

Acknowledgements We thank the anonymous referees for their careful reading and fruitful comments.

References

1. M. Bidoit and P. Mosses. *CASL User Manual, Introduction to Using the Common Algebraic Specification Language*. LNCS 2900. Springer-Verlag, 2004.
2. C. Choppy and L. Petrucci. Towards a methodology for modelling with Petri nets. In *Proc. Workshop on Practical Use of Coloured Petri Nets, Aarhus, Denmark*, pages 39–56, Oct. 2004. Report DAIMI-PB 570, Aarhus, DK.
3. C. Choppy, L. Petrucci, and G. Reggio. A method for modelling with place/transitions nets. Technical report, Université Paris 13, 2006.
4. C. Choppy, L. Petrucci, and G. Reggio. A method for modelling with coloured nets. Technical report, Université Paris 13, 2007.
5. C. Choppy and G. Reggio. A formally grounded software specification method. *Journal of Logic and Algebraic Programming*, 67(1-2):52–86, 2006.
6. P. Coad and E. Yourdon. *Object-Oriented Analysis*. Prentice-Hall, Englewood Cliffs, N.J., 1991.
7. The CPN Tools Homepage. <http://www.daimi.au.dk/CPNtools>.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
9. M. Heiner and M. Heisel. Modelling Safety-Critical Systems with Z and Petri Nets. In *Proc. SafeComp '99*, LNCS 1698, pages 361 – 374. Springer-Verlag, 1999.
10. K. Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use, vol. 1, vol. 2 et vol. 3*. Monographs in Theoretical Computer Science, Springer-Verlag, London, UK, 1995.
11. J. Jorgensen. Addressing Problem Frame Concerns Using Coloured Petri Nets and Graphical Animation. In *International Workshop on Advances and Applications of Problem Frames*, 2006.
12. R. S. Pressman. *Software Engineering: A Practitioner's Approach, 6th edition*. McGraw-Hill, 2005.
13. G. Reggio, E. Astesiano, and C. Choppy. CASL-LTL : A CASL Extension for Dynamic Reactive Systems Version 1.0– Summary. Technical Report DISI-TR-03-36, DISI – Università di Genova, Italy, 2003. Available at <ftp://ftp.disi.unige.it/person/ReggioG/ReggioEtAll03b.pdf>.

From Requirements via Colored Workflow Nets to an Implementation in Several Workflow Systems

R.S. Mans¹, W.M.P. van der Aalst¹, P.J.M. Bakker², A.J. Moleman², K.B. Lassen³ and J.B. Jørgensen³

¹ Department of Information Systems, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands. {r.s.mans,w.m.p.v.d.aalst}@tue.nl

² Academic Medical Center, University of Amsterdam, Department of Innovation and Process Management, Amsterdam, The Netherlands. {p.j.bakker,a.j.moleman}@amc.uva.nl

³ Department of Computer Science, University of Aarhus, IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark. {krell,jbj}@daimi.au.dk

Abstract. Care organizations, such as hospitals, need to support complex and dynamic workflows. Moreover, many disciplines are involved. This makes it important to avoid the typical disconnect between requirements and the actual implementation of the system. This paper proposes an approach where an Executable Use Case (EUC) and Colored Workflow Net (CWN) are used to close the gap between the given requirements specification and the realization of these requirements with the help of a workflow system. This paper describes a large case study where the diagnostic trajectory of the gynaecological oncology care process of the Academic Medical Center (AMC) hospital is used as reference process. The process consists of hundreds of activities. These have been modeled and analyzed using an EUC and a CWN. Moreover, based on the CWN, the process has been implemented using four different workflow systems. This shows the applicability of our approach and allows for an evaluation of different approaches towards flexibility in workflow systems.

1 Introduction

For some time now, especially in academic hospitals, there has been a need for support in controlling and monitoring health care processes for patients [29]. In general, there is the need to support the diagnostic and therapeutic trajectory of health care processes.

One of the objectives of hospitals is to increase the quality of care for patients [16]. However, what we also see is that on the governmental side and on the side of the health insurance companies, more and more pressure is put on hospitals to work in the most efficient way as possible. Moreover, in the future, an increase in the demand for care is expected.

Workflow technology can be seen as an interesting vehicle for the support and monitoring of health care processes. Workflow Management Systems (WfMS) support processes by managing the flow of work such that the work is done at the right time by the proper person [3]. Advantages of successfully applying workflow technology are that processes supported by workflow systems can be executed faster and more efficiently. In addition, processes can be monitored, which has also as consequence that processes can be executed faster.

The difficulties that hospitals have to cope with when they want to support their health care processes, and the need for supporting their health care processes emerges from the fact that healthcare processes are *diverse*, *flexible* and that *several specialties* can be involved in the treatment process. So, what we find is that, for example, for a group of patients with the same diagnosis, the number of different examinations and treatments can be high and the order in which they are done can vary greatly. Also, because of intermediary results of diagnostic examinations, the way a patient reacts to the offered treatment, and the condition of the patient itself, it may be necessary to continuously adapt the care process for a particular patient [14].

Actually, there is a large gap between a running hospital process and its implementation in different workflow systems. The main focus of this paper is to present the steps we took to bridge this gap. Moreover, we will also explicitly focus on *how* and *when* we used the formal modeling language *Colored Petri Nets (CPNs)*[20, 27] in the steps that we took.

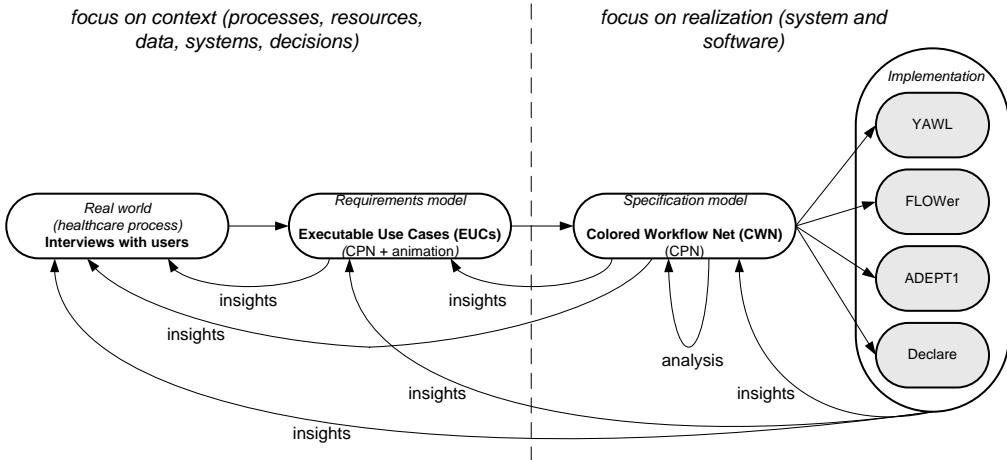


Fig. 1. Overall approach

The steps that we took for bridging the gap between a running hospital process and its implementation in different workflow systems are given in Figure 1. First of all, the healthcare process under consideration is the diagnostic trajectory of patients visiting the gynaecological oncology outpatient clinic in the AMC hospital, a large academic hospital in the Netherlands. This process covers what can happen from the moment that a patient is referred to the AMC hospital for treatment till the patient is eventually diagnosed or dismissed. As can be seen in Figure 1, we first had interviews with people involved in the healthcare process and made a CPN model out of it. This involved creating an *Executable Use Case (EUC)* [23], which is a CPN model augmented with a graphical animation. EUCs are formal and executable representations of work processes to be supported by a new IT system and can be used in a prototyping fashion to specify, validate, and elicit requirements. We will present the EUC that has been created and elaborate on how it has been used. Afterwards, we converted the EUC into a *Colored Workflow Net (CWN)*, which is closer to an implementation of the healthcare process in a workflow system. Finally, the CWN serves as a basis for implementation of the healthcare process in four different workflow systems. Also, as the CWN is a formal model of the workflow to be executed and serves as input for implementation of the process in different workflow systems we will, in Section 5, focus on the analysis of the CWN to ensure that a correct CWN had been made.

Moreover, an additional goal of implementing a hospital process in different workflow systems is that we wanted to identify the requirements that have to be fulfilled by workflow systems, in order to be successfully applied in an hospital environment. To this end, we choose to implement the healthcare process in workflow systems which already provide a certain kind of flexibility. As a consequence, as workflow management systems, we choose YAWL [4], FLOWer [11], ADEPT1 [37] and Declare [34]. However, in this paper we will not elaborate on the requirements identified.

The approach described above is closely related to the approach described in [7, 24]. In [7, 24], EUCs and a CWN have been used to go from an informal description of a real world process to an implementation of the same process in a certain workflow system. However, we studied an existing healthcare process of a hospital in detail, whereas in [7, 24] rather small cases are used. To give an idea about the size of the healthcare process, it needs to be indicated that the EUC consists of *689 transitions, 601 places and 1648 arcs* and that the CWN consists of *324 transitions, 522 places and 1221 arcs*. Moreover, we made an implementation in four different workflow systems instead of only one workflow system and systematically collected feedback from the care organization (AMC). Furthermore, in [7] there was only user involvement in the EUC phase, whereas in [24] there has not been any user involvement.

This paper is structured as follows: Section 2 introduces the approach followed. Section 3 introduces the EUC and the gynaecological oncology healthcare process of the AMC hospital that we studied. In Section 4, Colored Workflow Nets are introduced and used to model the selected process. This is followed by the implementation of the healthcare process in four different workflow systems, in Section 6. In Section 5, we will focus on the analysis of the CWN. Related work is given in Section 7. The paper finishes with the conclusions in Section 8.

2 Approach

In this section, we will first elaborate in general on the approach that has been followed, to go from a real-life process to the implementation of this in several workflow systems. Afterwards, the separate steps will be considered in more detail. The steps followed in the approach were already shown in Figure 1. So, we started with a real-life case, and for which we created a EUC, a CPN model augmented with a graphical animation. Afterwards, the EUC is converted into a CWN, which is also a CPN model. The CWN is then used as basis for the implementation of the healthcare process in four different workflow systems; namely YAWL [4], FLOWer [11], ADEPT1 [37], Declare [34].

We will now elaborate in more detail on the steps that have been followed, especially on the second and third step that has been followed. As can be seen in Figure 1, the model used in the second and third step are CPN models. CPNs have been chosen because they provide a well-established and well-proven language suitable for describing the behavior of systems with characteristics like concurrency, resource sharing, and synchronization. In this way, they are well-suited for modeling workflows or work processes [3]. The CPN language itself, is supported by *CPN Tools* [13]. CPN Tools has been used to create, simulate, and analyze the CPN models that are presented in this paper.

As can be seen in Figure 1, our approach started with interviewing users. The users were involved in the diagnostic trajectory of the gynaecological oncology healthcare process of the AMC hospital in Amsterdam. In these interviews we focussed on identifying the work processes which could then be modeled as a EUC. The EUC consists of a CPN model, which describes the real-life process, and an animation layer on top of it, which can be shown to the users of which we modeled the process.

In the CPN model, any concepts and entities deemed relevant can be used. So, we can use the CPN language in an unrestricted manner which means that tokens, places and transitions may refer to any concept or entity, i.e. also concepts which are not directly part of the process but still relevant from the users point of view.

Remember that, according to the definition given in [23], that EUCs are formal and executable representations of work processes to be supported by a new IT system and can be used in a prototyping fashion to specify, validate, and elicit requirements. Actually, we have used the model part of the EUC for modeling the healthcare process and have used the animation layer as a vehicle for validating the model. In other words, we used EUCs for checking together with the users involved, whether we modeled their work processes correctly, instead of specifying, validating and eliciting requirements. Only the animations have been shown to the user whereby the CPN model, which is lying beneath the animation layer, remains hidden for the user. This has been the main reason for using EUCs. As we were modeling the work processes of doctors and nurses, we assumed that they were not able to understand the CPN model themselves. Therefore, we used animations which are understandable for end users and in this way provided a suitable opportunity for validating our model.

Furthermore, EUCs have to ability to “talk back to the user”, which is not possible with a static CPN model. As EUCs provide executable descriptions of a work process, they can be used in a trial-and-error fashion. When users remark that something is wrong or missing, the EUC can easily be adapted and again shown to the user.

After we validated our model, we took the underlying CPN model of the EUC and translated it into a *workflow model*. By making this workflow model we restricted ourselves to the workflow domain. More specifically, by making the workflow model, we restricted ourselves to concepts and entities which are common in workflow languages. *Compared to the EUC model, we now only used a fixed library of concepts and entities, whereas in the EUC any concept or entity deemed relevant may be used.* In addition, the workflow model

contains actions that will be supported by the new system in interaction with human users, and it also contains actions that are to be fully automated by the new system. Actions that are not going to be supported by the new system, are left out.

More specifically, as workflow model we used a, so called, Colored Workflow Net (CWN) as modeling language. A CWN is a CPN model restricted to the workflow domain and can be seen as a high-level version of the traditional *Workflow Nets* (WF-nets) [1].

When the CWN model had been finished, we used it as a basis for the process models used to configure each of the four workflow systems. This way we implemented four systems to support the gynaecological oncology healthcare process. As workflow systems, YAWL, FLOWer, ADEPT1 and Declare have been chosen. All these four workflow systems provide a certain kind of flexibility, which in this context is deemed relevant. The reason for this is that we want to support a healthcare process and for supporting healthcare processes it is obvious that a certain kind of flexibility is needed, and which needs to be provided by a workflow system. In Section 6, we will elaborate more on these systems and discuss which kind of flexibility is exactly provided by each system.

As is indicated in Figure 1, during the construction of the EUC and the CWN and the implementation in the different workflow systems, additional insights can be obtained about previous phases. So, there can be often iterations back and forth between the components in the figure. However, we only had feedback from the people involved in the process during the interview, EUC and CWN phase. In other words, during and after finishing the implementation in the four workflow systems, we did not have any feedback from the people involved in the process. Although this could have given us valuable feedback about the process, we think we already clearly identified the process during the construction of the EUC and CWN.

In Figure 1, we see that there exists a dashed line between the first two blocks and the last two blocks. This dashed line represents a shift of focus when going from the left side of the line to the right side of the line. At the left side of the dashed line the focus is on the *context*, whereas at the right side the focus is on the *realization*. So, with *context* we mean that the focus is on processes, resources, data, systems and decisions and with *realization* we mean that the focus is on the system and software itself. Within this shift of focus, the CPN modeling language, which has been used for the EUC and the workflow model, provides a smooth transition between the two foci. In addition, we believe this addresses the classical “disconnect” which exists between business processes and IT. Furthermore, as on the left side of the dashed line one of the foci on processes, this means that the approach via an EUC and CWN only works in case we are dealing with process oriented systems, like document handling systems. This can also be derived from the fact that both the EUC and CWN are process models.

It is important to mention that all the models and translations between models have been done manually. In the end, this leads to a *full* implementation of a *complete* healthcare process in four different workflow systems. Because of the specific nature of the EUC model and the CWN, it is obvious that the CWN cannot be generated automatically from the EUC. This is because a decision needs to be made between what needs to be supported by the workflow system and what not, and which needs to be reflected within the CWN. However, in principle, a (semi-) automatic translation from the CWN to each of the workflow systems is possible. In [7, 24], it has been demonstrated that it is possible to (semi-) automatically generate BPEL code or a YAWL model from a CWN model.

We already indicated that we studied a large healthcare process; the EUC consists of 689 transitions and the CWN consists of 324 transitions. Moreover, for creating the EUC and the CWN more than 100 man hours were needed for each model. As we also needed to get acquainted with the workflow systems used, configuring each workflow system also took more than 80 man hours per system. Additionally, around 60 man hours were needed for interviewing and getting feedback from the people involved in the process.

3 Executable Use Case for the Gynaecological Oncology Healthcare Process

In this section, we first introduce the gynaecological oncology healthcare process which we studied. After that, we will consider one part of the healthcare process in more detail and for this part we will elaborate on how the

animations have been set-up within the EUC. In other words, for the selected part of the model, we elaborate on what is shown in the EUC and explain how still the routing within the model can be influenced. Finally, we will elaborate on the obtained experiences. Note that given the size of the CPN model of the EUC (689 transitions, 601 places, 1648 arcs and 2 colorsets) it is only possible to show a small fragment of the overall model.

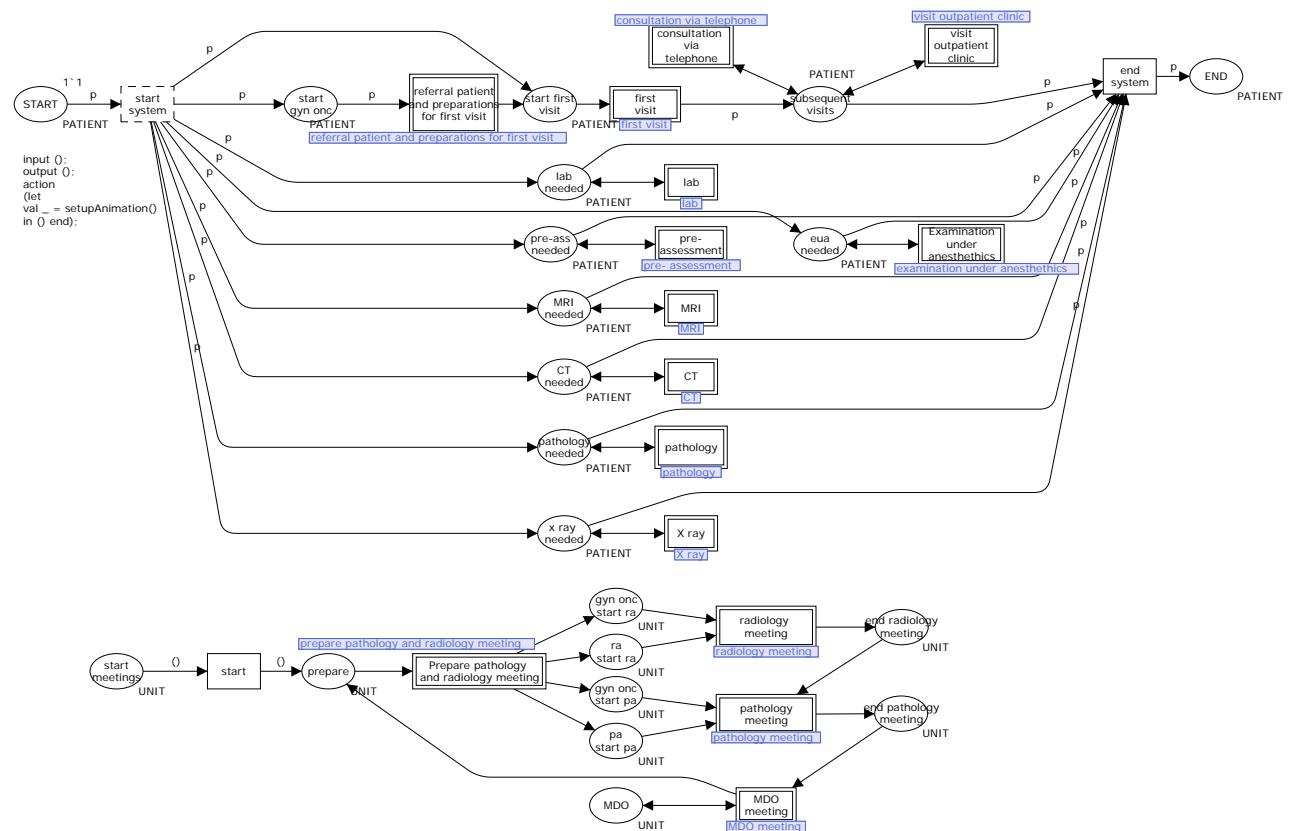


Fig. 2. General overview of the gynaecological oncology healthcare process.

In Figure 2, the topmost page of the CPN model of the EUC is shown, which gives a general overview of the diagnostic trajectory of the gynaecologic oncology healthcare process in the AMC hospital. In the remainder of this paper, we will simply refer to the gynaecological oncology healthcare process itself, instead of the diagnostic trajectory of the gynaecological oncology healthcare process.

Actually, as can be seen in Figure 2, two different processes have been modeled which are relevant for the gynaecological oncology healthcare process. The *first* process, which is modeled in the lower part of the picture, deals with the diagnostic trajectory that is followed by a patient when referred to the AMC hospital for treatment, till the patient is diagnosed. In the first part of the process we have that already some diagnostic examinations can be ordered, before actually the patient visits the hospital for the first time, like a MRI or a CT-scan. Moreover, also some administrative activities are already done before the patient visits the hospital for the first time. Later on, in this section, we will elaborate on more detail on this specific part of the process.

During the first visit of the patient to the hospital, the doctor examines the patient and decides whether he/she is confident with the already ordered examinations or that some new examinations need to be ordered.

In addition, the doctor decides about the next appointment(s) he want to have with a patient. Afterwards, the nurse is responsible for the arrangement of the dates of the additional examinations and the next appointment(s) with the doctor which can be either again a visit to the outpatient clinic or an appointment by telephone. These appointments are made together with the patient. Furthermore, also some administrative activities are done, like giving additional information about the treatment and handing over some folders.

As already indicated, the next appointment of the doctor with the patient can be either via telephone, or again a visit to the outpatient clinic. In general, at these appointments, the doctor decides about which examinations need to be ordered, canceled or replaced. The same holds for appointments of the doctor with a patient. In addition, some administrative activities need to be done, mostly by the nurse.

Actually, the doctor can order a lot of different examinations, and also at different specialties. For example, at the radiology department he can order an X-ray or a CT-scan or at the anaesthesiology department he can order a pre-assessment. The interactions with these specialties and also the process within these specialties are modeled at the bottom of Figure 2.

The *second* part of the process, which is modeled in the upper part of the picture in Figure 2, deals with the weekly organized meetings, on Monday afternoon, for discussing the status of patients and what needs to be done in the future for these patients. These three different meetings are called “radiology meeting”, “pathology meeting” and “MDO” and respectively people from radiology, pathology and people involved in the therapeutic trajectory are involved, as well as doctors from the gynaecological oncology itself.

Remark that some connections exist between the two processes. However, as we only focussed on the activities and ordering of activities within one process, we did not put any effort in making these connections explicit.

Now, after introducing the gynaecological oncology process in general, we want to focus on a specific part of the process. More specifically, we focus on the very beginning of the process (transition “referral patient and preparation for first visit”), in which a doctor of a referring hospital calls a nurse or doctor of the AMC and after which an appointment is made for the first visit of the patient and some appointments for diagnostic examinations are already made. The appointment making part of the process is shown in the upper part of Figure 3. For example, we see that the first visit of the patient needs to be planned and that it is possible to make appointments for an “MRI”, “CT” or “pre-assessment”.

In Figure 3, we see how the CPN model and the animation layer are related within the EUC. At the top, we see the CPN model that is executed in CPN Tools. At the bottom we see the animation that is provided within the BRITNeY tool [12], the animation facility of CPN Tools. The CPN model and the animation layer are connected by adding animation drawing primitives to transitions in the CPN model, which update the animation. The animation layer shows for the last executed activity in the CPN model, which *resources*, *data* and *systems* are involved in executing the activities and it also shows which *decisions* are made at the activity. This allows for focusing on what happens in the *context* of the process. In addition, for the last executed activity in the model, a separate panel is shown which indicates which activities are enabled and may be executed. One of the enabled activities in the panel can be selected and executed, which changes the state of the process and in this way, we can directly influence the routing within a process. Remark, that in BRITNeY already functionality is available for showing a panel with enabled bindings, but we slightly adapted it to our needs, so that only the enabled activities are shown instead.

In this way, the animation layer provides a view on the current state of the process and shows which next activities may be executed. When an activity is executed in the CPN model it is reflected by updates to the animation layer. Consequently, the CPN model and the animation layer remain synchronized.

In the snapshot shown in Figure 3, the animation visualizes activity “make document and stickers”. In addition, the panel at the top right side of the snapshot in Figure 3, shows which activities can be executed after that activity “make document and stickers” has been executed. Moreover, in the animation we see that a nurse of the outpatient clinic is responsible for executing the “make document and stickers” activity and that no decisions need to be made. We also see that a computer is needed for executing the activity. Moreover, the panel at the top right side shows that, amongst others, the activities “plan MRI” and “send fax to pathology” may be executed now.

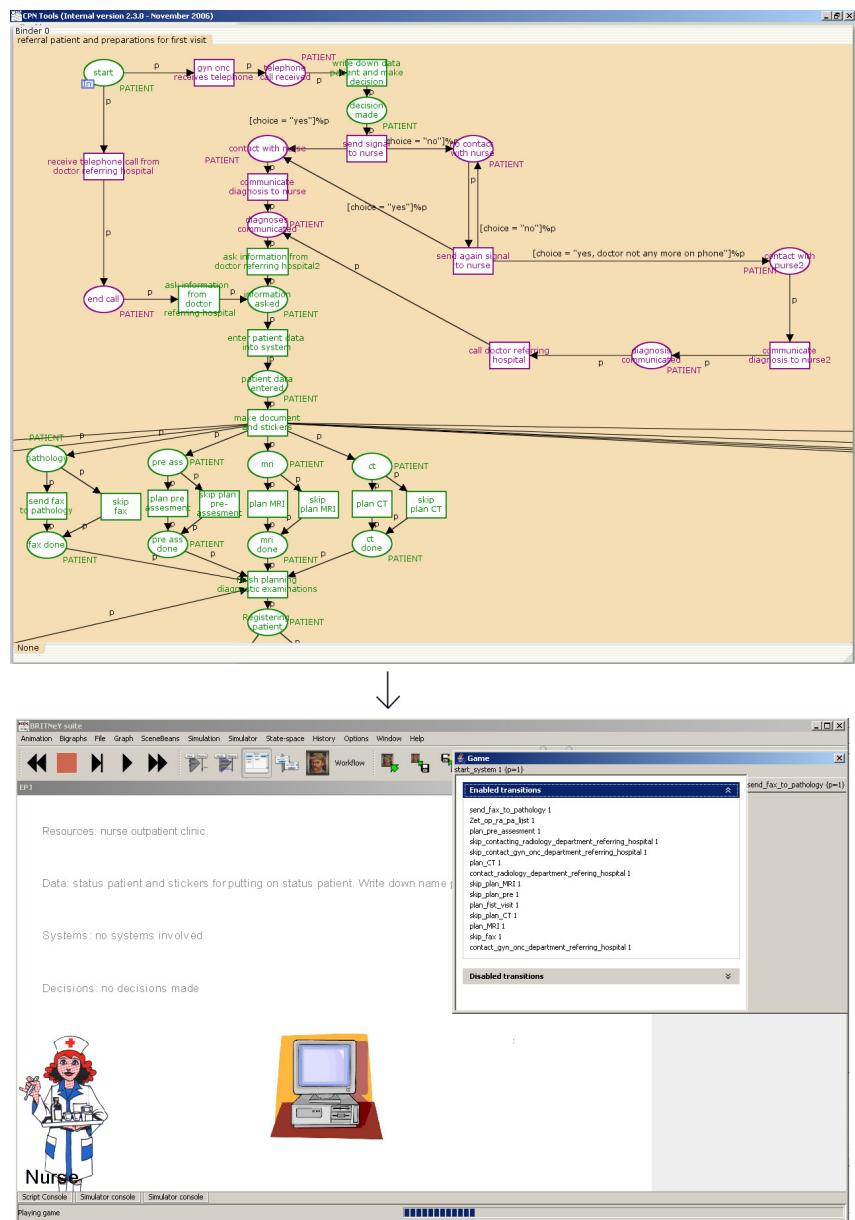


Fig. 3. Animation belonging to the “make document and stickers” activity. In addition, the panel at the top right side shows which activities are enabled now.

We have shown the animations to the people that were involved in the gynaecological oncology process. Before starting with the animations, we explicitly asked the people whether they wanted to indicate whether something was wrong, missing, or superfluous, with regard to the animation shown, and the enabled activities shown in the panel. In general, the people were very positive and indicated that in this way, they were able to check whether the process modeled in the EUC corresponded with their workprocess. Also, they gave useful feedback about activities that had not been modeled or were placed in the wrong order, and whether the information which was shown for each activity, was correct or not. However, it needs to be indicated that later

on, when making the CWN model, we found out that some activities were missing and which we did not discover with the EUC approach. But in general, we can say that the EUC approach was really helpful in validating the model and we believe that better results have been obtained than when we would have shown the plain CPN models or process schemas of a workflow management system to the people involved.

4 Colored Workflow Net for the Gynaecological Oncology Healthcare Process

In this section, we first introduce the Colored Workflow Net (CWN) that has been made for the gynaecological oncology healthcare process and shortly discuss the differences between EUCs and CWNs. After that, we will consider the same part of the CWN in detail as we considered in detail for the EUC. For the selected part of the CWN, we elaborate on what is shown regarding the different perspectives of the CWN. Finally, the differences with the corresponding part of the EUC are discussed. Note that also in this case, given the size of the CWN model (324 transitions, 522 places and 1221 arcs and 53 colorsets) it is only possible to show a small fragment of the overall model.

As can be seen in Figure 1, both the EUC and CWN are CPN models. Remember that a CWN is a *workflow* model in which we restricted ourselves to concepts and entities which are common in workflow languages, whereas in the EUC any concept or entity deemed relevant may be used. Furthermore, as can be seen in Figure 1, the focus of the CWN is on the *realization*, which means that the CWN only contains activities which will be supported by the workflow system. Moreover, as workflow systems also cover the resource and data perspective, it is clear that in addition to the EUC, which only covers the control-flow perspective, the CWN should also cover the resource, data and operation⁴ perspective. Moreover, the resource, data and operation perspective are covered by the CWN by using ML-functions, where the animation in the EUC only textually showed resources, data, and systems.

The syntactical and semantical requirements for a CWN have been defined in [7]. Moreover, a CWN abstracts from implementation details and language/application specific issues. According to [7], a CWN should be a CPN with only places of type **Case**, **Resource** or **CxR**. Tokens in a place of type **Case** refer only to a case and the corresponding attributes (e.g. name patient, patient id), and tokens in a place of type **Resource** refer only to resources. Finally, tokens in a place of type **CxR** refer to both a case and a resource. There are some subtle differences between the conventions in [7] and the conventions we have used to construct the CWN, and of which we mention only the two most important ones. First of all, the data attributes of the original CWNs in [7] may only be name-value pairs of the string type. In this way, we consider its use as quite limited as in practice also often lists are used, and therefore we decided to also allow *list* types for the value part of the name-value pair. For example, patients which need to be discussed during the weekly pathology meeting are put on a, so called “pathology list”. To this end, in the CWN model, we need to have a data attribute with name “pathology list” and where the value part is of a list type. Furthermore, we decided to separate the case data from the case, so that case data can be accessed everywhere in the model. For this we use the concept of a fusion place in CPN Tools.

In Figure 4, we see the CWN for the EUC CPN which has been shown in Figure 3. In Figure 4, there exist some connections between transitions and places of the type **Resource** and **CaseData**. In addition, by using guards, which belong to a transition, explicit references are made to the resource and data perspective. Places of the type **Resource** contain information about the availability of different kinds of resources and places of the type **CaseData** contain information about the case data belonging to a case and is a product of a case identifier and the data belonging to that case. Note that tokens in places of type **CasID** contain unique identifiers for each case and that tokens in places of type **CaseData** are a product of a case identifier and case data. In this way, activities can inspect or change case data for a certain case. Furthermore, tokens in a place of type **CidxR** refer to both a case id and a resource.

For example, in Figure 4, we see that the activity “enter patient data into system” need to be performed by a nurse, and which is indicated by the guard at the top right side of the activity. Furthermore, activity “send

⁴ The operation perspective describes the elementary operations performed by resources and applications.

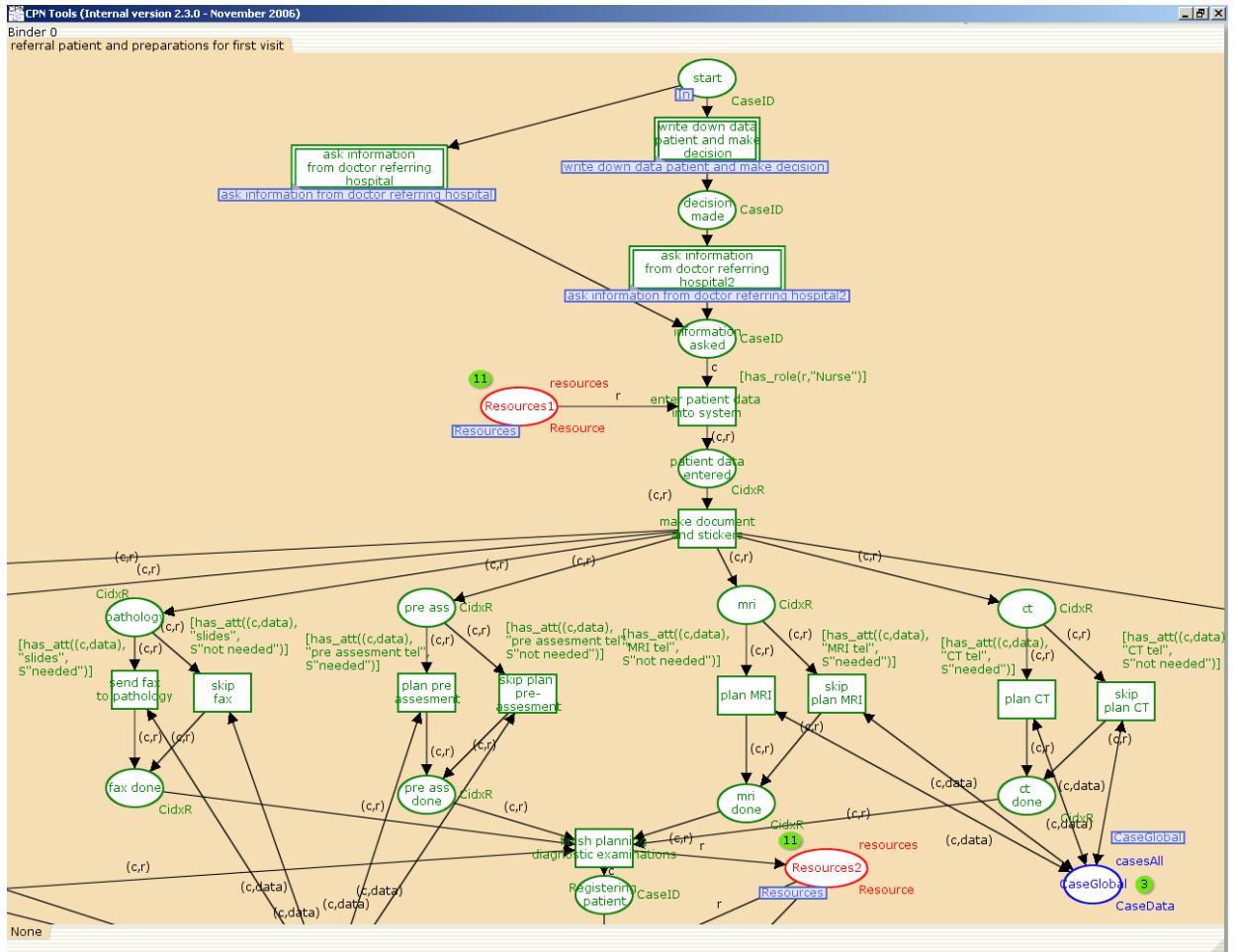


Fig. 4. CWN for the EUC shown in Figure 3.

fax to pathology” may only be performed if the pathology slides of the referring hospital need to be sent to the AMC, and which is specified by the guard with the texts “slides” and “needed” in it. From this guard, it also becomes clear that we need to have a data attribute with name `slides`, but in the model we may also have other data attributes like `name`, `patient id`, or `pathology list`.

If we compare the CWN of Figure 4 with the EUC CPN of Figure 3, we see that there exist some differences. First of all, some activities which are shown in the EUC CPN do not appear in the CWN, as they are not supported by the workflow system. The activities of the CWN in Figure 4 which can be directly mapped to activities of the EUC in Figure 3 are colored green. In other words, they are preserved. For example, we see that activities “enter patient data into system” and “plan MRI” are both in the EUC and CWN, so they are preserved. The activities of the EUC which are not supported by the workflow system, and as a consequence do not pop up in the CWN, are colored purple. For example, the activities “send signal to nurse” and “communicate diagnosis to nurse” of Figure 3 are not supported by the workflow system.

Moreover, the places “Resources1” and “Resources2”, which are colored red, are only present in the CWN as it contains information about the availability of resources and which is needed for the organizational perspective of the CWN. Also, the place “Case global”, which is colored blue, is only present in the CWN as it contains

the corresponding data attributes for each case instance and which is needed for the data perspective of the CWN. Furthermore, the guards belonging to transitions explicitly reference the resource and data perspective.

When we compare the EUC and CWN with each other it is clear that both cover the control flow perspective. However, the CWN also covers the resource, data and operation perspective. As these perspectives are also covered by workflow systems it is clear that the CWN, when compared to the EUC, is the next and also useful step towards the implementation of a process in a certain workflow system.

5 Analysis

In this section, we will focus on the analysis of the CWN model. Within CPN Tools there are two possibilities for the analysis of the CWN model, namely; simulation / animation and state space analysis [21]. Simulation can be used to investigate different scenarios and explore the behaviors of the model. Moreover, simulation also allows for performance analysis. However, performing several simulations does not guarantee that there are no errors within the model and in this way does not hold as a proof for correctness of the model. Therefore, state space analysis needs to be used for verification as it ensures that all possible executions are covered. In other words, with a state space analysis, the full state space of a CPN model is computed which makes it possible to *verify*, in the mathematical sense of the word, that the model possesses a certain formally specified property. The state space analysis of CPN Tools can handle state spaces up to 200.000 nodes and 2.000.000 arcs [22] and provides, amongst others, visual inspection and query functions for investigating (behavioral) properties of the model.

Since we are dealing with workflows we are interested in the so-called *soundness property*. Soundness for workflow nets is defined in [1] as: *for any case, the procedure will terminate eventually and the moment the procedure terminates there is a token in the sink place (i.e. a place with no outgoing arcs) and all the other places are empty. Moreover, there should be no dead transitions.* To check for soundness of the CWN, we need to abstract from resources. Moreover, as the CWN is exceptionally large (324 transitions, 522 places and 1221 arcs and 53 colorsets) we also need to simplify the colorsets and verify things in a hierarchical manner. To be more precise, for each transition on the top page of the CWN which was linked to a subnet, we checked the soundness of the subnet. Note that such a subnet can also contain subnets. Checking the soundness of such a subnet has been done according to the following procedure. First, we only focussed on the subnet itself by removing all nodes and subnets from the cpn file which did not belong to the subnet being considered. In other words, only the subnet being considered was kept in the CPN file. Second, we removed all colorsets and all data attributes which were not relevant for the subnet being considered. A data attribute was not considered relevant when there was not any function in the subnet which actually accessed the data attribute. After finishing the two preceding pre-processing steps, we were actually ready to check whether the subnet was sound. For the actual check for soundness we added an extra transition t^* in the subnet which connects the only output place with the only input place and is also called *short-circuited net* [1]. According to [1], if the short-circuited net is live and bounded, then the original net is sound. If some error had been found, the subnet was adapted and again checked for its soundness. This last step has been repeated till the subnet was sound. Afterwards, if some errors had been found, the CWN model had been adapted in the same way. A limitation of the approach is that a subnet which is checked for its soundness should not be too large and / or have many colorsets.

For example, for the CWN, shown in Figure 4, we originally found out that for the activities “skip plan MRI” and “skip plan CT” no double-headed arc had been used between these transition and place “CaseGlobal”. This could be concluded from the liveness properties of the state space report of the short-circuited net which are shown in Figure 5. In Figure 5, it can be seen that there are no live transition instances whereas for the short-circuited net all transitions had to be live and all places had to be bounded in order for the original net to be sound. From one of the dead markings the errors could be located and easily be fixed. So, For the CWN shown in Figure 4 all transitions are live and all places are bounded for the short-circuited net which means that it is sound. Moreover, for this sound CWN, it took 1901 seconds to calculate the full state space which consists of 39808 nodes and 156800 arcs.

Liveness Properties	
Dead Markings	96 [6392,5864,5863,5861,5860,...]
Dead Transition Instances	None
Live Transition Instances	None

Fig. 5. Liveness Properties section of the state space report generated for the erroneous CWN.

In general, for each subnet we found around one or two errors, but we also had subnets which were error-free. Furthermore, although there are more interesting structural properties which can be checked for, we only checked for soundness.

6 Realization of the system in different workflow systems

In this section, we will focus on the realization of the system in four different workflow systems. As workflow systems, the systems YAWL, FLOWer, ADEPT1 and Declare have been chosen. All these workflow systems provide a certain kind of flexibility, which in the context of implementing a healthcare process in different workflow systems, is deemed relevant. It is clear that for supporting healthcare processes some flexibility needs to be provided by the workflow system. Moreover, these workflow systems have been chosen as we wanted to identify the requirements that have to be fulfilled by workflow systems, in order to be successfully applied in an hospital environment. However, in this paper we will not elaborate on the requirements identified. Furthermore, we could easily obtain each of these systems.

For each of the four workflow systems that provide flexibility, we will discuss the kind of flexibility which is provided.

As input for the implementation, the CWN will be used. For each system, we will exemplify how the CWN model is mapped to the modeling language used in the workflow system itself. This is done by manually mapping the CWN of Figure 4 to the modeling language used in the workflow system itself. Differences are compared and, in this way, also an impression of the workflow system itself is obtained.

At the end of this section we will elaborate on the applicability of a CWN for implementation of a process in a workflow system.

6.1 YAWL / Worklets

YAWL (Yet Another Workflow Language) [4] is an open source workflow management system⁵, which is based on the well-known workflow patterns⁶ [5] and is more expressive than any of the other languages available today. Moreover, instead of only supporting the control-flow perspective and data perspective, YAWL also supports the resource perspective.

YAWL supports the modeling, analysis and enactment of flexible processes by, so called, *worklets* [9] and which can be seen as a kind of process fragments. Specific activities in a process are linked to a repertoire of possible actions. Based on the properties of the case and other context information, the right action is chosen. The selection process is based on a set of rules. Also, during enactment it is possible to add new actions to the repertoire.

⁵ YAWL can freely be downloaded from www.yawl-system.com

⁶ More information about the workflow patterns can be found on www.workflowpatterns.com

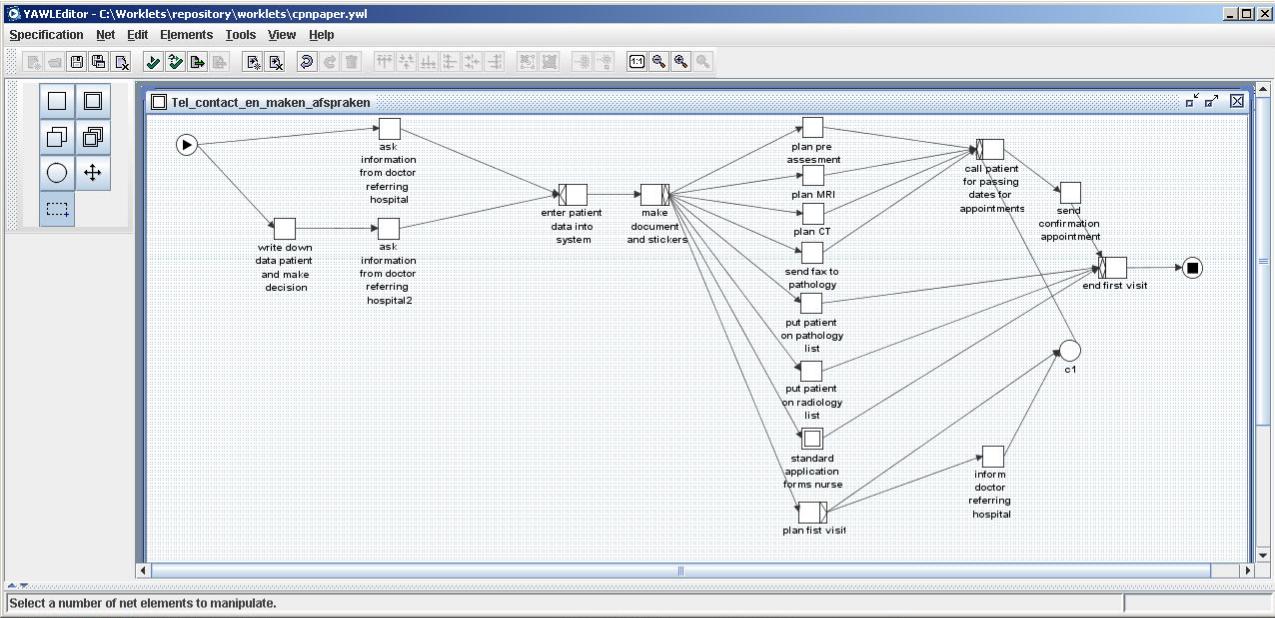


Fig. 6. Screenshot of the YAWL editor.

In Figure 6, we see how the CWN of Figure 4 is mapped onto the YAWL language. Given the fact that YAWL can be seen as a superset of CWNs, it was easy to translate the CWN of Figure 4 in YAWL. So, it was possible to directly translate the transitions into YAWL tasks. The places of the CWN model can also be directly translated into YAWL conditions, but due to syntactical sugarering there is no need to add all places as conditions to the YAWL model. For example, the “make document and stickers” activity in YAWL is an OR-split, which has as meaning that one or more of the outgoing paths may be followed, which means that it is optionally to follow the paths to, for example, the “plan MRI” and “plan CT” activities. In this way, the skip activities which appear in the CWN are not needed in the YAWL model.

Finally, the YAWL model consists of 231 nodes and 282 arcs and for which it took around 120 hours to construct a model that could be executed by the YAWL workflow engine.

6.2 FLOWer

FLOWer is a commercial workflow management system provided by Pallas Athena, the Netherlands⁷. *FLOWer* is a case-handling product [8]. Case-handling adds flexibility by focussing on the data aspect rather than on the control-flow aspect. Case-handling offers four core features [8]. The first one is that all information available within a case is available (i.e., present the case as a whole rather than showing just bits and pieces), which avoids “context tunneling”. Second, the decision of which activities are enabled is based on the information which is available within the case, instead of the activities which are already executed. Third, work distribution is separated from authorization. This allows for having additional types of roles, like skipping or redoing activities in the process. In this way, many more (implicit) scenarios are possible within the process. Moreover, a fourth distinguishing feature of *FLOWer* is that workers are allowed to view and add/modify data before or after the corresponding activities have been executed.

In Figure 7, we see how the CWN of Figure 4 is mapped onto the *FLOWer* language. In this case, it was also quite easy to translate the CWN in *FLOWer*. Namely, it was possible to directly translate the transitions

⁷ <http://www.pallas-athena.com/>

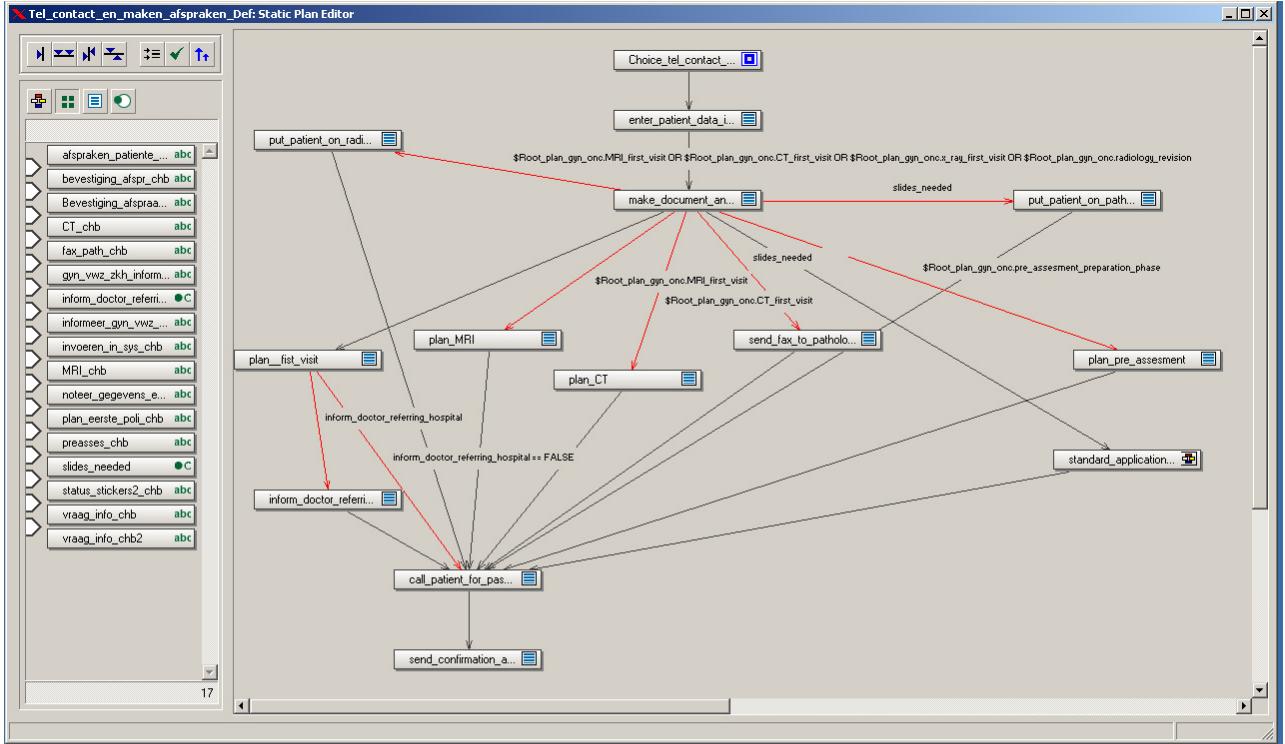


Fig. 7. Screenshot of the FLOWer editor.

into FLOWer activities and also the causal relationships could be taken into account. In Figure 7, all nodes are activities except the node with name “choice tel contact”. This node represents the deferred choice which needs to be made in the beginning of the process, as can be seen in Figure 4. So, in the beginning of the process, either the “ask information from doctor referring hospital” activity can be chosen or the “write down data patient and make decision” activity can be chosen, but not both.

Finally, the FLOWer model consists of 236 nodes and 190 arcs and for which it took around 100 hours to construct a model that could be executed by the FLOWer workflow engine.

6.3 ADEPT1

ADEPT1 is an academic prototype workflow system [37], developed at University of Ulm, Germany. ADEPT1 supports *dynamic change* which means that the process of one individual case can be adapted. So, it is allowed to deviate from the pre-modeled process template (skipping of steps, going back to previous steps, inserting new steps, etc.) in a secure and safe way. That is, the system guarantees that all consistency constraints (e.g., no cycles, no missing input data when a task program will be invoked) which have been ensured prior to the dynamic (ad hoc) modification of the process instance are also ensured after the modification.

In Figure 8, we see how the first part of the CWN of Figure 4 is mapped onto the ADEPT language. In the ADEPT language, the activities are represented by rectangles. However, as in the ADEPT language we only have an XOR and an AND split and join, we need to introduce activities which are empty, i.e they are not executed by users. For example, the “make stickers and document” activity in ADEPT is an AND-split which is followed by several empty XOR-splits in order that several activities, like “plan MRI” or “plan CT” can be performed or skipped.

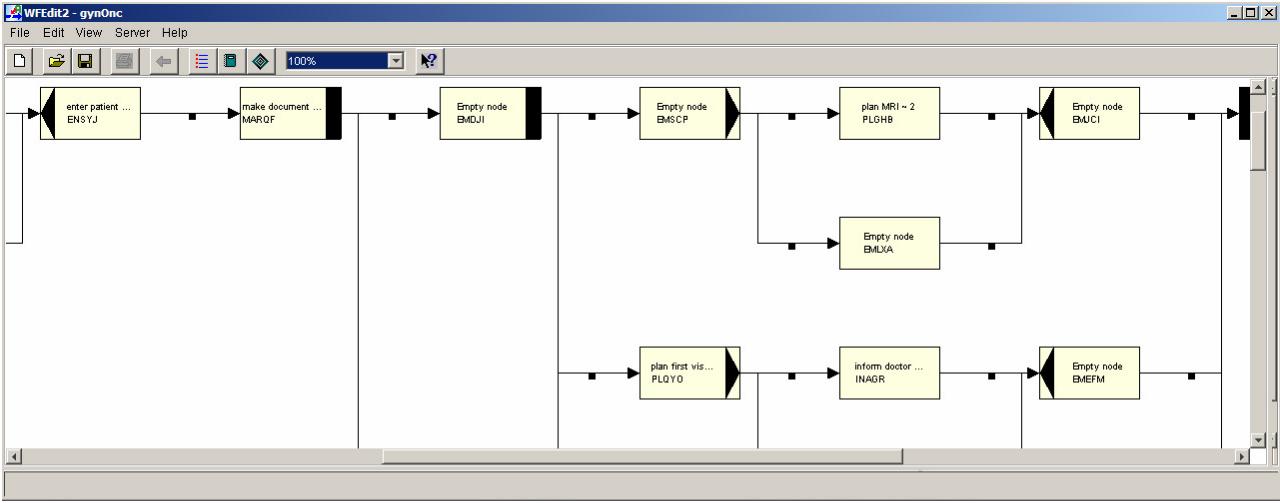


Fig. 8. Screenshot of the ADEPT1 editor. AND-splits/joins are represented by a black rectangle in a node and XOR-splits/joins are represented by a black triangle in a node.

Finally, the ADEPT model consists of 40 nodes and 53 arcs and for which it took around 8 hours to construct a model that could be executed by the ADEPT1 workflow engine.

6.4 Declare

Declare is an academic prototype workflow system [34], developed at Eindhoven University of Technology, the Netherlands⁸. In Declare the language used for specifying processes, which is called *ConDec*, is a *declarative* process modeling language, which means that it is specified *what* should be done. *Imperative* process modeling languages, like YAWL and FLOWer, specify *how* it should be done, which leads to over-specified processes. By using a declarative rather than an imperative / procedural approach, ConDec aims at an under-specification of the process where workers have room to “maneuver”.

The ConDec language allows for modeling and enacting dynamic processes and is based on *Linear Temporal Logic (LTL)* [25]. In this way, it can be specified which behavior is forbidden. Moreover, within Declare it is very important to mention that it is assumed that users already know what should be done. The users can execute activities in any order and as often as they want, but they are bound to certain rules, which are specified in the ConDec language.

Furthermore, Declare also supports *dynamic change*, so that the process of individual cases can be adapted. In terms of Declare, this means that it is allowed to deviate from the pre-modeled process template by adding or removing activities or constraints. Also, model correctness is guaranteed, which means that it is checked by Declare whether the changes are allowed or not for the cases for which the changes are intended to be applied.

In Figure 9, we see how the CWN of Figure 4 is mapped onto the ConDec language. In the ConDec language, the activities are represented by rectangles. Moreover, each different LTL formula, which can be used in the model, is represented by a different *template*, and can be applied to one or more activities, dependent on the arity of the LTL formula which is used. Note that the language is extendible, i.e., it is easy to add a new construct by selecting its graphical representation and specifying its semantics in terms of LTL.

The activities of the CWN model could easily be translated to activities in the ConDec language. However, as the ConDec language is a declarative process modeling language, and not an imperative language, the causal relationships of the CWN model could not easily be translated to the ConDec model. Moreover, we also had

⁸ <http://is.tm.tue.nl/staff/mpesic/declare.htm>

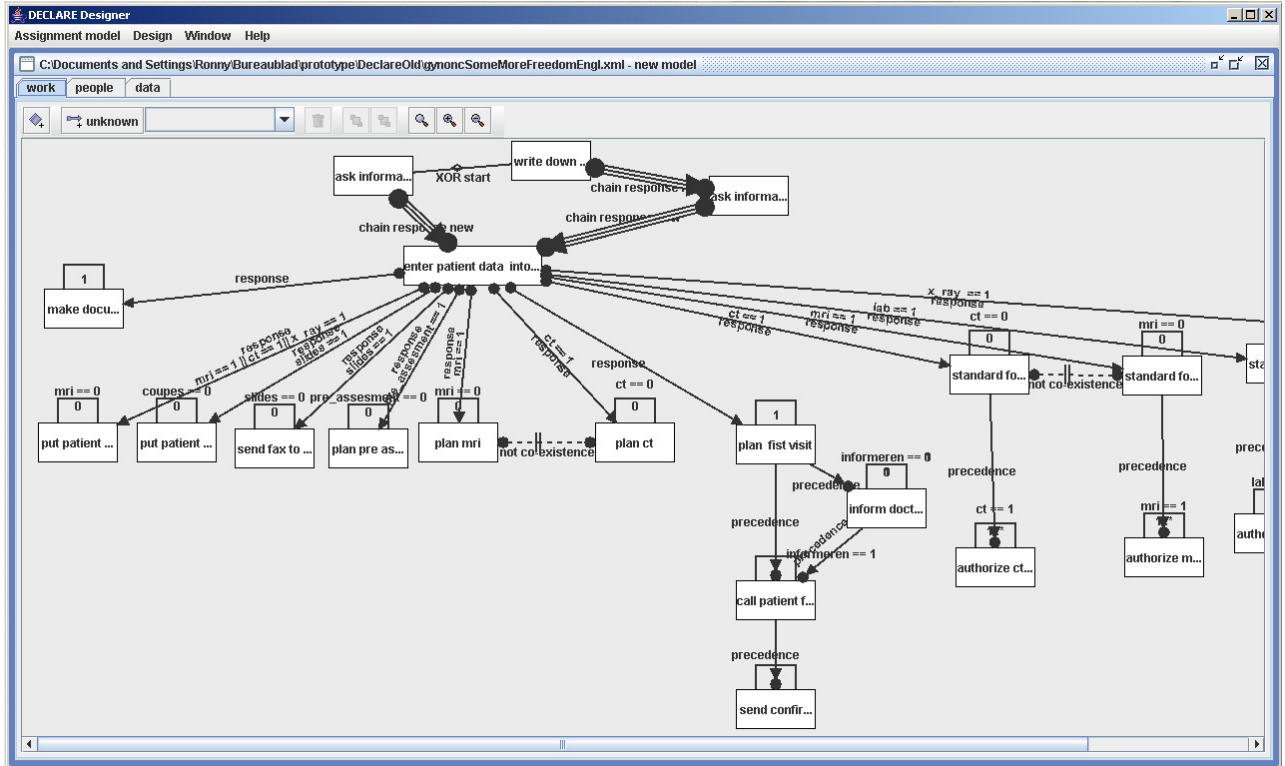


Fig. 9. Screenshot of the Declare editor.

to take into account that within Declare it is assumed that users already know what should be done. To this end, we made the ConDec model, in such a way, that we ruled out forbidden behavior and that the users know as early as possible what should be done, and from that moment on, the user itself is responsible for deciding which activities should be done and in which order.

In Figure 9, we see that after activity “enter patient data into system” a lot of subsequent activities need to be done, and which is indicated by a *response* arc going from the “enter patient data into system” activity to these activities, like “plan ct” and “plan mri”. However, it is only modeled that these activities need to be done afterwards, but not in which order. Also, during runtime, it is indicated to the user which activities need to be done and then the user himself can decide in which order he wants to execute the activities and how often. Moreover, with ConDec it is also easy to model that the execution of one activity rules out the execution of another activity, and the other way around. For example, the “not-coexistence” arc between activities “plan mri” and “plan ct” specifies that only one of these activities may be executed, but not both.

For Declare we did not implement the full healthcare process as in Declare there is only support for scalar data types, like integer and boolean, but not for more advanced data types like lists, etc. We only implemented the process which is depicted in Figure 4 and which consists of only 10 percent of the whole CWN model. Nevertheless, the model that we made in Declare still consists of 23 nodes and 44 LTL formulae and for which it took around 12 hours to construct a model that could be executed by the Declare workflow engine.

6.5 Applicability of CWNs for implementation of the system in different workflow systems

To conclude this section, we will elaborate on the applicability of CWNs for implementing the healthcare process into the four different workflow systems. In other words, we try to answer the question of how easy is

it to implement the CWN in a workflow system. We demonstrate how CWNs can be used as a starting point for implementation in a workflow system and evaluate the different systems. To this end, we use five different criteria, which are listed in the top row of table 1. The different workflow systems are given in the first column.

As first criterion, we have the number of nodes and arcs. For all the first three workflow systems considered, the nodes in the CWN which referred to activities could directly be translated to activities into the workflow language of the workflow systems and also only these nodes needed to be copied. However, for ADEPT1 we had to use artificial nodes as only XOR and AND split and joins are possible.

As indicated in [7], a CWN covers the *control flow*, *resource* and *data* perspective. Furthermore, as for the transitions in our CWN it is also required to manipulate the data attributes of a case, the CWN also covers the *operation perspective*[1]. For these last four criteria, we indicated how much effort in man hours it took to specify each perspective in each workflow system. Please remember that for all systems, we created a model which could be executed by the systems workflow engine. However, within Declare and ADEPT1 we only defined 10 percent of the model that we had within YAWL and FLOWer.

With regard to the second criterion, effort required for the control flow perspective, it was easy to translate the control flow perspective of the CWN to the YAWL, FLOWer and ADEPT1 workflow language, as they are all imperative languages as is also the case for the CWN. As the ConDec language of Declare is an declarative process modeling language and that in Declare it is also assumed that the users already know what should be done, it is clear that the control flow perspective of the CWN could not directly be translated to the ConDec language and which also took quite some effort. This can also be derived when we have a look at the normalized man hour values for the second criterion. From these normalized values we see that it takes more time to implement the control flow perspective in Declare compared to the other systems and also compared to the total implementation time within Declare ⁹.

With regard to the third criterion, effort required for the organizational perspective, the resource related parts of the CWN could easily be translated. All workflow systems considered supported the possibility to define roles which could be linked to activities.

The data perspective, the fourth criterion, could easily be translated to the workflow language of YAWL, FLOWer as these systems support directly or indirectly the same data types as defined in the CWN. As both Declare and ADEPT1 only support scalar data types, the data attributes of the CWN which uses lists could not be covered by both languages. As it is more time consuming to define advanced data types compared to simple scalar types, this explains why in Declare and ADEPT1, when looking to the normalized values, relatively less time was needed for implementation of the data perspective compared to the other workflow systems.

With regard to the fifth criterion, effort required for the operation perspective, we see in the table that the operation perspective of the CWN was hardly useful for defining the operation perspective in the different workflow systems. The reason for this is, that each workflow language has its own specific way/language for defining, for example, guards and how data attributes need to be manipulated. Furthermore, specifying the operational perspective in the CWN itself was also quite difficult. Also, as in Declare and in ADEPT1 we only had to deal with simple scalar types within the operation perspective, this explains why in Declare, when looking to the normalized values, less time was needed for implementation of the operation perspective compared to the other workflow systems.

To conclude, we can say that the CWN is of help for the implementation of a certain process in a workflow system. Furthermore, building a EUC and a CWN allows for a *separation of concerns*. More specifically, EUCs are good for capturing the requirements of a process, without thinking of how it is realized. CWNs, on its turn, are good because the control-flow, resource, data and operation perspectives are defined. As an EUC has captured the requirements of a process it is a valuable input for a CWN. Using this development process, we are sure that these concerns are dealt with at the right time as we have to deal with them anyways. The only disadvantage is that the operational perspective of the CWN cannot be translated to the different workflow systems. Furthermore, for Declare and ADEPT1 we used normalized values for the last four criteria. We think

⁹ As in ADEPT1 and Declare only 10 percent of the model has been defined, normalization for these systems is done by multiplying the hour values by 10.

these normalized values are still representative, as for the implementation of the other 90 percent indeed a better insight into the system will be obtained, but on the other hand, parts of the process need to be implemented in another way than other parts. In this way, we assume that these two facts compensate each other.

Table 1. Translation of the CWN into the workflow languages of YAWL, FLOWer and Declare. Please remember that within Declare and ADEPT1 we only defined 10 percent of the model that we had within YAWL and FLOWer.

	number of nodes / arcs	effort control flow perspective (hours)	effort organizational perspective (hours)	effort data perspective (hours)	effort operation perspective (hours)
YAWL	231 / 282	20	5	30	65
FLOWer	236 / 190	20	5	20	55
ADEPT1	40 / 53	2	1	1	4
Declare	23 / 44	6	1	1	4

Furthermore, in principle, a (semi-) automatic translation from the CWN to each of the workflow systems is possible. More specifically, the control flow and resource perspective are subject to an automatic translation. With regard to the data perspective, also data attributes which are name-value pairs of the string type are subject to automatic translation. However, for the operation perspective and for name-value pairs for which a list type has been used, more carefulness is needed. For both of them, a semi-automatic translation will be needed.

7 Related Work

From literature, it can be derived that workflow systems are not applicable for the healthcare domain [10, 28]. The current generation of workflow systems adequately supports administrative and production workflows but they are less adequate for healthcare processes which have more complex requirements [10]. In addition, in [35, 36], it has been indicated that so called “careflow systems”, systems for supporting care processes in hospitals, have special demands with regard to workflow technology. One of these requirements is that flexibility needs to be provided by the workflow system [31, 40]. Unfortunately, current WfMS are falling short with regard to providing flexibility, which is also seen as an important problem in literature [6, 8, 9, 15, 26, 38]. Also, once a workflow-based application has been configured on the basis of explicit process models, the execution of related process instances tends to be rather inflexible [2, 37, 39, 43]. Consequently, the lack of flexibility has significantly limited the application of workflow technology. The workflow systems that we chose in this paper (YAWL, FLOWer, ADEPT1 and Declare) allow for more flexibility than classical workflow systems.

Moreover, with regard to the requirements for applying workflow technology in the healthcare domain, in [18] it is indicated that real time patient monitoring, detection of adverse events, and adaptive responses to breakdown in normal processes is needed. As adaptive workflow systems are rarely implemented, this makes current workflow implementations inappropriate for healthcare [41]. Also, [14, 16, 32, 31, 40] stress that workflow systems have to support dynamic adaptation of running workflows to handle the flexibility of healthcare (therapy) processes. In case of breakdowns, managing exceptions is unavoidable [17]. Furthermore, in a real clinical setting, it is a critical challenge for any workflow management system how capable it is to respond effectively when exceptions occur [33].

Furthermore, what is lacking is that no support is provided for the multidisciplinary nature of healthcare processes. In [42, 19, 30], the processes for only one department in a hospital are supported by a workflow management system. So, a requirement is that support needs to be provided for the support of cross-departmental healthcare processes which is stressed in [29, 14, 40]. Finally, a completely different requirement

is that autonomous, independently developed applications needs to be integrated which is risky, costly and time-consuming [28].

This paper uses the approach initially proposed in [7, 24] where also an EUC and CWN have been used to go from an informal inscription of a real world process to an implementation of the same process in a certain workflow system. For [24], an electronic patient record system has been implemented in the YAWL system and is, in this way, related to the healthcare domain. However, for both papers, it holds that rather small cases have been used and that only an implementation has been done in one workflow system. Furthermore, in [7] there was only user involvement in the EUC phase, whereas in [24] there has not been any user involvement. In our case, we modeled a much larger and real healthcare process of a hospital using four different workflow systems. Moreover, the approach was evaluated by the people involved in the process.

The use of EUCs has also been inspired by the work done in [23], in which EUCs were used to prototype an electronic patient record system for hospitals in Aarhus, Denmark.

8 Conclusions

In this paper, we have focussed on the implementation of a large hospital process consisting of hundreds of activities, into different workflow systems. To ease the implementation process, we have first made an EUC, followed by a CWN. This CWN was used as input for configuring the different workflow systems. The approach followed has been described in Figure 1. As the approach to go from a EUC to a CWN and finally an implementation in different workflow systems, nicely bridges the gap between the modeling of a real-world process and the implementation of the process in a workflow system, three other important lessons have been learned.

The first lesson is that EUCs are a great help for validating the modeled real-world process used in this paper. Instead of showing the model itself to the people involved in the process, which is very difficult to understand for these people, we have used the EUC for showing to the people involved how we modeled their workprocess. Within the EUC, we focussed on animating relevant information about the last executed activity to the user and we also focussed on showing the activities, which can be executed afterwards. In this way, the people involved could easily check whether the process modeled in the EUC corresponded to their actual workprocess, which was also confirmed by the people themselves.

The second lesson learned is that the CWN helps on elaborating how the process considered, needs to be made ready for implementation in a workflow system. As the CWN is a workflow model, we had to restrict ourselves to concepts and entities that are common in workflow languages. So, activities that will not be supported by the workflow system are left out in the CWN and also the resource, data and operational perspective needs to be handled in a structured / uniform way.

Furthermore, it is clear that the CWN is also helpful for implementations of the process in a workflow system. The nodes of the CWN referring to activities can be directly translated and also the control flow, resource and data perspective of the CWN can easily be translated to the workflow language of a workflow system. This does not hold for the operation perspective of the CWN and, together with the previous observations, forms the third lesson learned.

Additionally, EUCs and CWNs are useful as they respectively capture the requirements of a process and define the control-flow, resource, data and operation perspective. With regard to man hours needed for our approach, interviewing and getting feedback from the people involved in the process took around 60 hours, and creating the EUC and CWN took more than 100 man hours for each model. Finally, the configuration of the workflow systems took around 240 hours.

A direction for future research is to develop animations specific for CWN. In this way, before a process will be supported by a workflow system, the people can already become acquainted with how their process will be supported by a workflow system and already start experimenting. Moreover, the case study also provided valuable insights into the requirements for workflow technology in care organizations. Besides the need for flexibility, it revealed the need for a better integration of the patient flow with the planning of appointments and peripheral systems supporting small and loosely coupled workflows (e.g. lab tests).

Acknowledgements

We would like to thank Lisa Wells and Kurt Jensen for their support in using CPN Tools. Also, we would like to thank Michael Westergaard for his support in using the BRITnEY tool. Furthermore, we also want to thank the people of the gynaecological oncology, radiology, pathology and anesthetics department, and especially prof. Burger of the gynaecological oncology department for the time spent, on explaining the gynaecological oncology healthcare process.

References

1. W.M.P. van der Aalst. Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 1–65. Springer-Verlag, Berlin, 2004.
2. W.M.P. van der Aalst, P. Barthelmess, C.A. Ellis, and J. Wainer. Workflow Modeling using Proplets. In O. Etzion and P. Scheuermann, editors, *7th International Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 198–209. Springer-Verlag, Berlin, 2000.
3. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2002.
4. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.
5. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
6. W.M.P. van der Aalst and S. Jablonski. Dealing with Workflow Change: Identification of Issues and Solutions. *International Journal of Computer Systems, Science, and Engineering*, 15(5):267–276, 2000.
7. W.M.P. van der Aalst, J.B. Jørgensen, and K.B. Lassen. Let's Go All the Way: From Requirements via Colored Workflow Nets to a BPEL Implementation of a New Bank System Paper. In R. Meersman and Z. Tari et al., editors, *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005*, volume 3760 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, Berlin, 2005.
8. W.M.P. van der Aalst, M. Weske, and D. Grünbauer. Case Handling: A New Paradigm for Business Process Support. *Data and Knowledge Engineering*, 53(2):129–162, 2005.
9. M. Adams, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Facilitating Flexibility and Dynamic Exception Handling in Workflows. In O. Belo, J. Eder, O. Pastor, and J. Falcao e Cunha, editors, *Proceedings of the CAiSE'05 Forum*, pages 45–50. FEUP, Porto, Portugal, 2005.
10. K. Anyanwu, A. Sheth, J. Cardoso, J. Miller, and K. Kochut. Healthcare Enterprise Process Development and Integration. *Journal of Research and Practice in Information Technology*, 35(2):83–98, May 2003.
11. Pallas Athena. *Case Handling with FLOWer: Beyond workflow*. Pallas Athena BV, Apeldoorn, The Netherlands, 2002.
12. CPN Group, University of Aarhus, Denmark. BRITnEY Suite Home Page. <http://wiki.daimi.au.dk/britney/britney.wiki>.
13. CPN Group, University of Aarhus, Denmark. CPN Tools Home Page. <http://wiki.daimi.au.dk/cpntools/>.
14. P. Dadam, M. Reichert, and K. Kuhn. Clinical Workflows - The Killer Application for Process-oriented Information Systems? In W. Abramowicz and M.E. Orlowska, editors, *BIS2000 - Proc. of the 4th International Conference on Business Information Systems*, pages 36–59, Poznan, Poland, April 2000. Springer-Verlag.
15. C.A. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In N. Comstock, C. Ellis, R. Kling, J. Mylopoulos, and S. Kaplan, editors, *Proceedings of the Conference on Organizational Computing Systems*, pages 10 – 21, Milpitas, California, August 1995. ACM SIGOIS, ACM Press, New York.
16. U. Greiner, J. Ramsch, B. Heller, M. Löffler, R. Müller, and E. Rahm. Adaptive Guideline-based Treatment Workflows with AdaptFlow. In K. Kaiser, S. Miksch, and S.W. Tu, editors, *Proceedings of the Symposium on Computerized Guidelines and Protocols (CGP 2004)*, Computer-based Support for Clinical Guidelines and Protocols, pages 113–117, Prague, 2004. IOS Press.
17. M. Han, T. Thiery, and X. Song. Managing Exceptions in the Medical Workflow Systems. In *Proceeding of the 28th international conference on Software engineering*, pages 741 – 750, Shanghai, China, 2006. ACM Press.

18. Y. Han, A. Sheth, and C. Bussler. A taxonomy of adaptive workflow management. 1998.
19. T. Wendler J. von Berg, J. Schmidt. Business Process Integration for Distributed Applications in Radiology. In *Third International Symposium on Distributed Objects and Applications (DOA'01)*, pages 10–19, Rome, Italy, 2001.
20. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1*. EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1997.
21. 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.
22. K. Jenssen, S. Christensen, and L.M. Kristensen. *CPN Tools State Space Manual*. Department of Computer Science, University of Aarhus, January 2006.
23. J.B. Jørgensen and C. Bossen. Executable Use Cases: Requirements for a Pervasive Health Care System. *IEEE Software*, 21(2):34–41, 2004.
24. J.B. Jørgensen, K.B. Lassen, and W.M.P. van der Aalst. From Task Descriptions via Coloured Petri Nets Towards an Implementation of a New Electronic Patient Record. In *Proceedings of the Seventh Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2006)*, volume 579, pages 17–36. University of Aarhus, 2006.
25. E.M. Clarke Jr., O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts and London, UK, 1999.
26. M. Klein, C. Dellarocas, and A. Bernstein, editors. *Adaptive Workflow Systems*, Special Issue of Computer Supported Cooperative Work, 2000.
27. L.M. Kristensen, S. Christensen, and K. Jensen. The Practitioner’s Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.
28. R. Lenz, T. Elstner, H. Siegele, and K. Kuhn. A Practical Approach to Process Support in Health Information Systems. *Journal of the American Medical Informatics Association*, 9(6):571–585, December 2002.
29. R. Lenz and M. Reichert. IT Support for Healthcare Processes - Premises, Challenges, Perspectives. *Data and Knowledge Engineering*, 61:49–58, 2007.
30. T. Greiser R. Röhrig M. Meister M. Sedlmayr, T. Rose and A. Michel-Backofen. Automating Standard Operating Procedures in Intensive Care. Accepted for CaiSE2007.
31. L. Maruster, W.M.P. van der Aalst, A.J.M.M. Weijters, A. van den Bosch, and W. Daelemans. Automated Discovery of Workflow Models from Hospital Data. In C. Dousson, F. Höppner, and R. Quiniou, editors, *Proceedings of the ECAI Workshop on Knowledge Discovery and Spatial Data*, pages 32–36, 2002.
32. S. Miksch, R. Kosara, and A. Seyfang. Is Workflow Management Appropriate for Therapy Planning? In *Proceedings of EWGLP 2000*, pages 53–69, Amsterdam, 2000. IOS Press.
33. S. Panzarasa and M. Stefanelli. Workflow management systems for guideline implementation. *Neurological Sciences*, 27:245–249, June 2006.
34. M. Pesic and W.M.P. van der Aalst. A declarative approach for flexible business processes management. In J. Eder and S. Dustdar, editors, *Business Process Management Workshops (Proceedings BPM 2006 International Workshops)*, volume 4103 of *Lecture Notes in Computer Science*, pages 169–180, Vienna, Austria, September 4–7 2006. Springer.
35. S. Quaglini, M. Stefanelli, A. Cavallini, G. Micieli, C. Fassino, and C. Mossa. Guideline-based Careflow Systems. *Artificial Intelligence in Medicine*, 20(1):5–22, 2000.
36. S. Quaglini, M. Stefanelli, G. Lanzola, V. Caporusso, and S. Panzarasa. Flexible Guideline-based Patient Careflow Systems. *Artificial Intelligence in Medicine*, 22(1):65–80, 2001.
37. M. Reichert and P. Dadam. ADEPTflex - Supporting Dynamic Changes of Workflows Without Loosing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.
38. S. Rinderle, M. Reichert, and P. Dadam. Correctness Criteria For Dynamic Changes in Workflow Systems: A Survey. *Data and Knowledge Engineering*, 50(1):9–34, 2004.
39. S. Sadiq, O. Marjanovic, and M.E. Orlowska. Managing Change and Time in Dynamic Workflow Processes. *International Journal of Cooperative Information Systems*, 9(1-2):93–116, 2000.
40. M. Stefanelli. Knowledge and Process Management in Health Care Organizations. *Methods Inf Med*, 43:525–535, 2004.
41. J. Sutherland and W.-J. van den Heuvel. Towards an intelligent hospital environment: Adaptive workflow in the or of the future. 2006.
42. T. Wendler, K. Meetz, and J. Schmidt. Workflow Automation in Radiology. In H.U. Lemke, editor, *Proceedings of Computer Assisted Radiology and Surgery (CAR98)*, pages 364–369. Elsevier, 1998.
43. M. Weske. Formal Foundation and Conceptual Design of Dynamic Adaptations in a Workflow Management System. In R. Sprague, editor, *Proceedings of the Thirty-Fourth Annual Hawaii International Conference on System Science (HICSS-34)*. IEEE Computer Society Press, Los Alamitos, California, 2001.

Requirements Engineering for Reactive Systems with Coloured Petri Nets: the Gas Pump Controller Example*

João M. Fernandes¹, Simon Tjell², and Jens Bæk Jørgensen²

¹ Dept. of Informatics, Universidade do Minho, Braga, Portugal

² Dept. of Computer Science, University of Aarhus, Aarhus, Denmark

jmf@di.uminho.pt, tjell@daimi.au.dk, bjb@daimi.au.dk

Abstract. The contribution of this paper is to present a model-based approach to requirements engineering for reactive systems, and more specifically to controllers. The approach suggests the creation of a CPN model based on several diagrams, for validating the functional requirements of the system under development. An automatic gas pump controller is used as case study. We propose a generic structure for the CPN model to address the modelling of the controller, the physical entities which the controller interacts with, and the human users that operate the system. The CPN modules for modelling the behaviour of the human users and the controller are instances of a generic module that is able to interpret scenario descriptions specified in CPN ML.

1 Introduction

A reactive system is “a system that is able to create desired effects in its environment by enabling, enforcing, or preventing events in the environment” [9]. This characterisation implies that in requirements engineering for reactive systems it is useful, and often necessary, to describe not only the system itself, but also the environment in which the system must operate [1].

In this paper, we are particularly interested in controllers, i.e., a type of reactive systems that control, guide or direct their environment. This work assumes that a controller (to be developed) and its surrounding environment are linked by a set of physical entities, as depicted in fig. 1. This structure clearly identifies two interfaces A and B that are relevant to two different groups of stakeholders, users and developers, in the task of requirements analysis.

From the user’s or client’s point of view, the system is composed of the controller and the physical entities. Typically, the users are not aware of this separation; they see a device and they need only to follow the rules imposed by interface B to use it. In fact, they may not even know that there is a computer-based system controlling the system they interact with.

* This research work was conducted while J.M. Fernandes was on a sabbatical leave at DAIMI, University of Aarhus and was partly supported by project SOFTAS (POSC/EIA/60189/2004).

From the developer's point of view, the environment is also divided in two parts with different behavioural properties; the physical entities have predictable behaviour while the human actors may exhibit disobedience with respect to their expected behaviour. Thus, the description of the behaviour of the environment must consider the physical entities (usually called sensors and actuators) which the system interacts with through interface A. In some cases, these physical entities are given, and software engineers cannot change or affect them during the development process, but need to know how they operate. Additionally, some relevant behaviour of the human users that interact with the system through interface B must be taken into consideration and actually reflected in the CPN model.

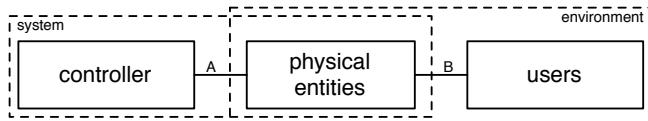


Fig. 1. A controller and its environment.

This paper presents a model-based approach to requirements engineering for reactive systems, and more specifically to controllers. The approach aims at obtaining a CPN model that describes the requirements through scenarios combined with a description of the behaviour of the physical entities which the controller interacts with. We propose a generic structure for the CPN model to hold two important properties: (1) *controller-and-environment-partitioned*, which means that it constitutes a description of both the controller and its environment, and that it distinguishes between these two domains and between desired and assumed behaviour; (2) *scenario-based*, meaning that it was constructed on the basis of the behaviours described in scenario descriptions. Our proposal continues the results presented in [8, 3, 2] and is illustrated in the development of a gas pump controller, which is a well-known example in the literature [4].

The paper is structured as follows. Sect. 2 introduces the Automatic Gas Pump case study that is used in this paper. In sect. 3, we present the main requirement models, in the form of use case diagrams and sequence diagrams, that were created for the case study. The CPN model for the case study, obtained with our approach, is discussed in sect. 4. We make some conclusions in sect. 5.

2 Case Study

As case study, we consider an Automatic Gas Pump, which is a computer-based system that permits customers to buy fuel in a self-served way. There exists one storage tank for each type of fuel (diesel, gasoline 92 octane, and gasoline 95 octane). The pump must be deactivated for a given type of fuel, when the

quantity of fuel in the associated tank is less than a given threshold (to be defined). There are also three different nozzles, one for each type of fuel.

To fill a car's tank with fuel, first the customer must insert a credit card and introduce the PIN code. If the card is valid and the introduced PIN code is correct, the customer may start to fill the car's tank with fuel, by picking a nozzle. When a given nozzle is picked by the customer, the price per litre of the respective type of fuel is shown in the display. While the fuel is being pumped, the pump must show in real-time the quantity pumped and the respective price.

After the nozzle has been returned to the holster, the credit card company is contacted and requested to withdraw from the customer's account an amount equal to the price of the fuel that has been tanked and to credit it to the station's account (the credit card company retains a fixed percentage of the transaction that is deduced to the station). The customer may also get a printed receipt, if the same credit card is reinserted in the pump, no later than five minutes after returning the nozzle.

Based on the general structure of a reactive system (fig. 1), our approach suggests the development to be started by creating a so-called entity diagram, that depicts the controller system to be developed and all the entities in its environment.

This entity model, which can be seen like a context diagram as proposed by several software methods, has an important role in the approach, since it defines without ambiguities the scope of the controller and identifies the entities that exist in its environment. This clear separation between controller and environment must be preserved in the subsequent models, since our aim is to obtain a CPN model that is controller-and-environment-partitioned.

Fig. 2 shows the entity diagram for the case study. It clearly identifies the name and direction of each message that flow in interfaces A or B. This diagram serves as a reference for the development process and in the next sections for each diagram proposed we identify which parts of the entity model is being addressed.

3 Use Cases and Scenarios for the Case study

In this section, we show the artefacts (models and diagrams) that we suggest to use before constructing the CPN model. These artefacts allow the developers to formalise the user requirements and serve as a basis for obtaining a CPN model. The artefacts are shown here in a specific (ideal) order, but in an engineering development context it is expected that an iterative process must be followed.

The use case diagram for the automatic gas pump controller is depicted in fig. 3. With respect to fig. 1, the use case diagram covers interface B (between the users and the system), and identifies the functionalities provided by the controller.

The use cases are briefly described below:

- **UC1 buy fuel** permits the customer to fill the car's tank with the chosen type of fuel.

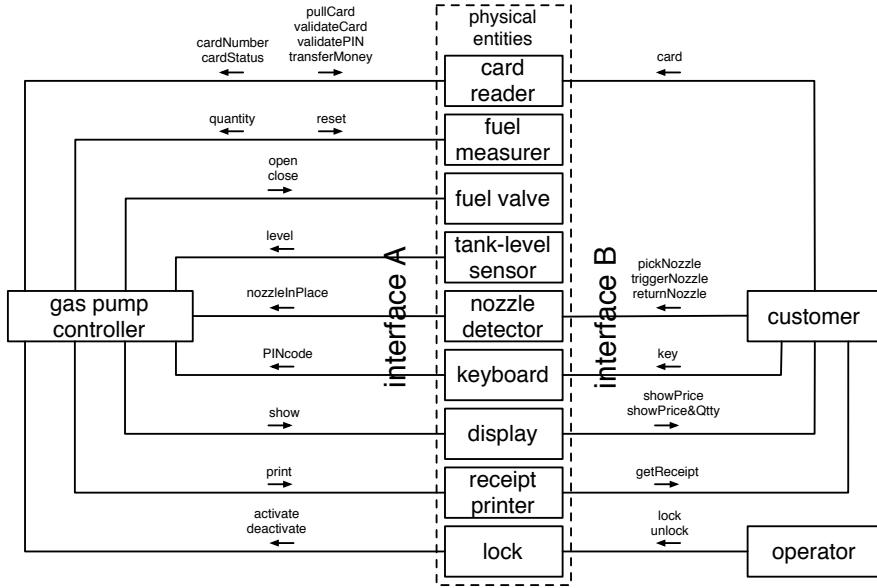


Fig. 2. An entity diagram for the Automatic Gas Pump, with a clear identification of the messages that flow in interfaces A (between the Gas Pump Controller and the Physical Entities) and B (between the Physical Entities and the Users).

- **UC2 initiate payment** validates if the customer has a valid credit card and if its PIN code is correctly entered in the keyboard. If this is the case, the pump is unblocked to allow fuel to be pumped.
- **UC3 get receipt** prints a receipt, if the customer reinserts the credit card no later than five minutes after returning the nozzle to its resting position.
- **UC4 de/activate pump** activates or deactivates the pump. The state of the pump must be easily visible to the customer.

As usual, the use case diagram identifies and names the use cases that the gas pump controller must support, and shows the external actors participating in the use cases. The actors in the use case diagram are the humans, customers and operators, that use the gas pump.

To describe the individual use cases in detail, their textual descriptions can be supplemented with sequence diagrams that specify some behavioural scenarios accommodated by the use cases. The scenarios describe desired behaviour of the gas pump controller in its interaction with the human actors and cover interface B with respect to fig. 1. These scenarios are thus adequate to be discussed with the client and also the final users of the system, since they permit a graphical and easy-to-understand representation of the user requirements, and omit design and implementation issues.

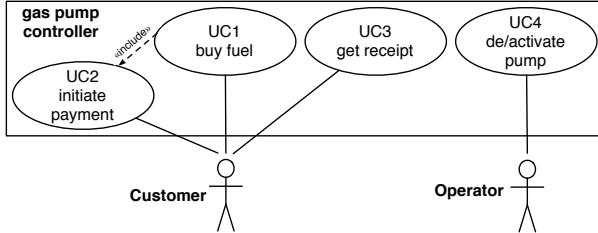


Fig. 3. Use case diagram for the Automatic Gas Pump controller.

As an example, the description of the main scenario for UC1 is presented next, including references to the sequence diagram that is depicted in fig. 4(a):

1. The customer starts the payment by introducing a valid credit card and typing the corresponding PIN code;
2. If the credit card is valid and the PIN code correct, the customer picks the nozzle of the wanted type of fuel;
3. The system shows the information related to the selected type of fuel (price per litre) and “0” as the number of litres pumped;
4. While the nozzle is being used, the customer can pump fuel to the car’s tank and the system updates the display showing the volume of pumped fuel and its respective price;
5. When the customer finishes pumping fuel, he returns the nozzle to its rest position (in the holster);
6. The system withdraws the amount corresponding to the price of the pumped fuel from the customer’s account, retains its commission, and credits the rest to the station’s account.

Alternative scenarios for a use case can be created, namely when it is sufficiently rich and complex. Fig. 4(b) shows an alternative scenario for UC1 that describes a situation where the user initially introduces a valid credit card, types its correct PIN code, picks a nozzle, but cancels the transaction by returning the nozzle to the holster (i.e., without putting fuel in the car’s tank). Therefore, at the end, the system does not transfer money from the customer’s account to the account of the station.

Similar textual descriptions and sequence diagrams exist for the other use cases. There is a dependency relationship between UC1 and UC2, meaning that to complete its execution, UC1 needs the functionalities provided by UC2. This dependency is specified in the use case diagram by an *include* relation and in the sequence diagrams by *ref* operators.

The next step in the modelling process is to refine the scenario descriptions, to introduce more detailed information in order to permit further development tasks to be conducted. In our approach, this entails two different things. Firstly, it is important to refine the user-level sequence diagrams by indicating the particular physical entity with which the users do exchange messages at each point in time.

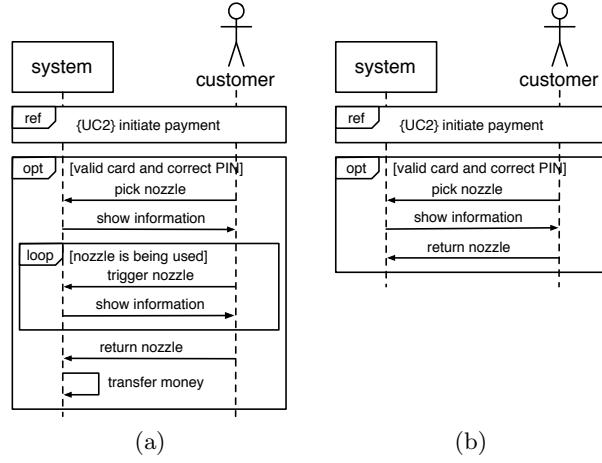


Fig. 4. Sequence diagram at the user level for UC1: (a) the main scenario, and (b) an alternative scenario.

The sequence diagram depicted in fig. 5 is an example of a scenario that details the exchange of messages in interface B. This can be seen in contrast to the diagram in fig. 4, where the user exchanges messages with the whole system, seen as a monolithic structure.

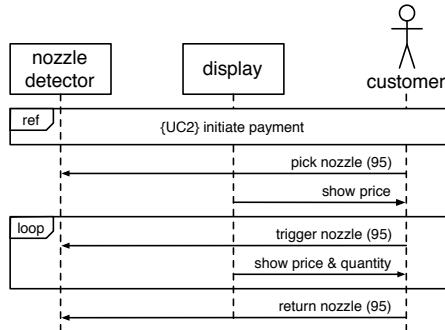


Fig. 5. The behaviour of the customer and the physical entities, during the main scenario of UC1.

Secondly, the messages that flow in interface A need also to be considered in our approach. This permits developers to introduce details about how the controller actually reacts to stimuli from the physical entities, that were supposedly initiated by the user. These refined scenario descriptions can be considered as part of the system requirements. The sequence diagram depicted in fig. 6 is an example

of a scenario that details the exchange of messages in interface B. This diagram is used to specify the behaviour of the gas pump controller, and more particularly the interaction of the controller with the physical entities. Therefore, the controller must be considered as the central element of that sequence diagram.

In summary, sequence diagrams as the one shown in fig. 5 describe requirements expressed as scenarios for the use cases, while sequence diagrams like the one in fig. 6 should be considered as specifications for a given scenario of a use case. This distinction assumes that “a requirement is a desired relationship among phenomena of the environment of a system, to be brought about by the hardware/software machine that will be constructed and installed in the environment, while a specification describes machine behaviour sufficient to achieve the requirement” [6].

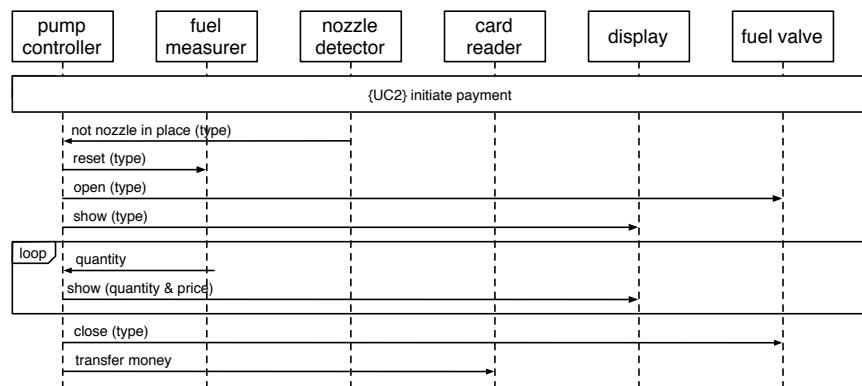


Fig. 6. The behaviour of the gas pump controller when interacting with the physical entities, during the main scenario of UC1.

4 The CPN model for the case study

The next development step is to construct a CPN model that represents all the behaviours described by the collection of considered sequence diagrams. The CPN modelling language was chosen, since CPN models are executable and formal, can provide a good balance between graphical and textual constructs, can address both the behaviour and the data of the system, and handle modelling aspects such as concurrency and locality in a graceful manner [7].

The construction of the CPN model is based on scenarios, which is important to guarantee that the model reflects all the partial behaviours identified and discussed with the clients and users of the system under development. Additionally, the CPN model must be structured in such a way that the separation between the controller and the environment, as expressed in fig. 1, is preserved and easy

to identify. Therefore, the approach ensures that the CPN is constructed to be controller-and-environment-partitioned and scenario-based.

4.1 Top-level Module

Fig. 7 shows the topmost module of the hierarchical CPN model for the case study, constructed from the sequence diagrams and following the structuring principles proposed in this paper. The module contains three substitution transitions: **Human Actors**, **Physical Entities**, and **Controller**. These three substitution transitions represent different domains and are used for modelling the functional requirements, the behavioural domain knowledge, and the behaviour of the controller, respectively.

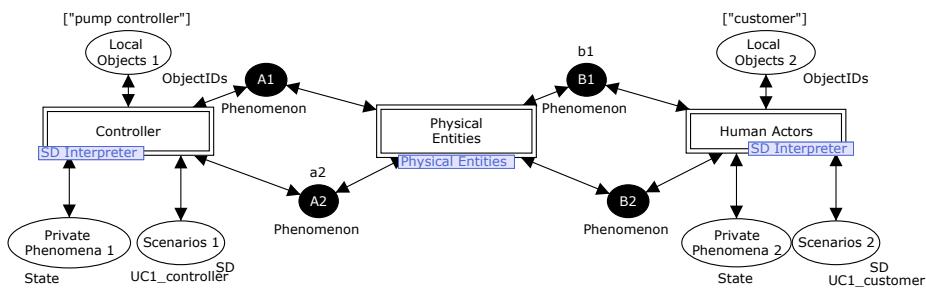


Fig. 7. The topmost module of the CPN model

The structure in fig. 7 is generic to reactive systems with a close interaction with the physical environment and operated by human actors. The structure embodies the guidelines that we are proposing for the modelling of such systems, their requirements, and their environment. The basic idea of the structure is to assist the modeller in maintaining a proper separation between the three modelling domains. The structure allows the description of scenarios for the behaviour of human actors and for the behaviour of the controller at an abstract level, both by means of high-level sequence diagrams, which are translated into a textual form for interpretation and execution by the **Human Actors** and **Controller** modules, respectively. Additionally, we use a regular CPN module (**Physical Entities**) for describing the behavioural properties of the physical entities through which the customer and the controller interact. By “regular”, we mean a CPN that directly uses the graphical constructs (places, transitions, arcs, etc.) to describe the behaviour of the considered domain.

The three domains interact through a collection of shared phenomena [5]. A *shared phenomenon* is a state or an event that is observable by both domains while being controlled by only one domain. In contrast, a *private phenomenon* is only observable within the controlling domain (not to be confused with the controller domain). The controlling domain is the domain that is able to affect

a shared phenomenon, i.e., to cause changes to a shared state or to generate a shared event. An observing domain is able to react on, but not affect, an observed phenomenon. No external domains are able to observe and thereby react on phenomena that are private to other domains. The shared phenomena perspective helps in the task of identifying the interfaces through which the domains are interacting. This allows us to enforce a strict partitioning of the representations of each of the domains in the CPN model, in order to make it useful for requirements engineering. In the top module of the CPN model (fig. 7), the interfaces of shared phenomena are emphasized as black places, each one denoted by a letter and a number:

- A1:** Shared phenomena between the controller and the physical entities, and controlled by the controller.
- A2:** Shared phenomena between the controller and the physical entities, and controlled by the physical entities.
- B1:** Shared phenomena between the physical entities and the human actors, and controlled by the physical entities.
- B2:** Shared phenomena between the physical entities and the human actors, and controlled by the human actors.

4.2 Physical Entities Module

The customer does not interact directly with the pump controller. In fact, he might not even be aware of the existence of a pump controller, i.e., a computer-based system controlling the system he interacts with. Instead, the customer does interact with the physical entities. The **Physical Entities** module is used for describing the behaviour of the actuators and the sensors that connect the controller with its physical environment. This behaviour is also referred to as the *indicative* (or given) properties of the environment; the physical entities have given behaviour patterns, which serve as a framework for the operation of the controller. These patterns of behaviour must be taken into account when the controller itself is designed, because they form part of the resulting behaviour of the environment when the controller is deployed. Furthermore, we consider that the physical entities are not integrated parts of the controller itself and this is the reason why they are explicitly modelled as a separate domain.

Fig. 8 shows the internals of the **Physical Entities** module (with just a subset of the entities). Each physical entity is represented by a substitution transition. Two internal states (the white places **Nozzle Triggered** and **Fuel Valves**) are used for modelling phenomena that are private to the physical entities; i.e., they are hidden from both the customer and the pump controller. The black and grey places are connected to the black places in the top module. A simple colour coding scheme is applied: black places hold locally controlled shared phenomena, while grey places hold remotely controlled shared phenomena.

Each substitution transition in the **Physical Entities** module encapsulates the behaviour of one particular physical entity; as an example, fig. 9 depicts the

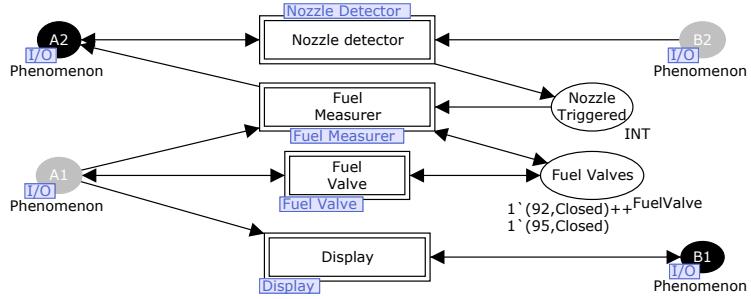


Fig. 8. The Physical Entities module.

module for the Fuel Measurer. The description is restricted by the fact that communication is performed exclusively through the interface of shared phenomena. The result is a collection of descriptions of the indicative behavioural properties of the physical environment. This part of the environment differs significantly from the part of the environment containing human actors (modelled in the Human Actors module) by the lack of free will and by the resulting deterministic nature. The physical entities exhibit strict reactive behaviour and do not generate events or change states in a spontaneous manner. Once the behavioural properties of the physical entities have been described, the descriptions can be maintained for executing various scenarios and for experiments with various possible design specifications for the pump controller.

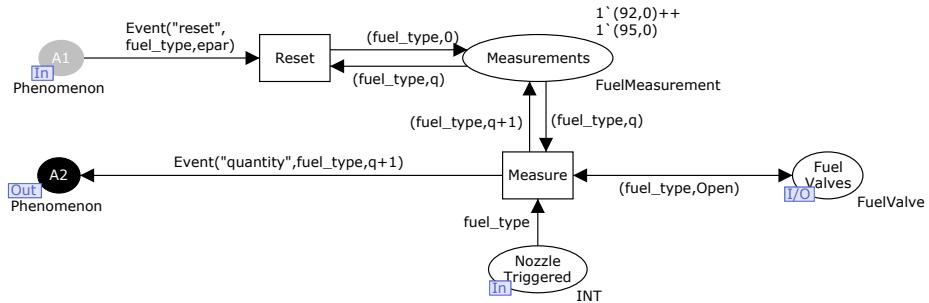


Fig. 9. The Fuel Measurer module.

In the approach to requirements engineering of reactive systems that we suggest in this paper, the modeller is only expected to specify CPN model structure, when the behaviour of the physical entities is being described. Everything else (i.e., controller and human users) is modelled by parameterizing a generic CPN module.

4.3 Controller and Human Actors Modules

Both the controller and the human actors are represented in fig. 7 by substitution transitions that refer to the SD Interpreter module. This generic module acts as an interpreter for textual CPN ML representations of the basic elements of UML 2.0 sequence diagrams. This module is utilized both for executing scenarios in which the user interacts with the system and for representing the behaviour of the controller.

As shown in fig. 7, the instances of the SD Interpreter module are parameterized through three places: one place specifies which objects (as found in the sequence diagram) are local to the instance (Local Objects 1 and 2), another place specifies possible private phenomena to the domain (Private Phenomena 1 and 2), and a third place specifies the behaviour as a scenario in the form of a sequence diagram (Scenarios 1 and 2). Each instance communicates with the physical entities through its set of shared phenomena. The communication consists of messages about the occurrence of events or changes to shared states. Furthermore, shared states can be part of the predicates used in the sequence diagrams.

Fig. 10 shows the internals of the SD Interpreter module. This module is basically the specification of a machine, which is able to execute sequence diagrams specified in CPN ML. The execution may be affected by incoming events and state changes and may itself cause state changes and generate events through the interface of shared phenomena. When a sequence diagram is executed, the modeller needs to specify which objects are local. All communication between local and non-local objects is performed through shared phenomena.

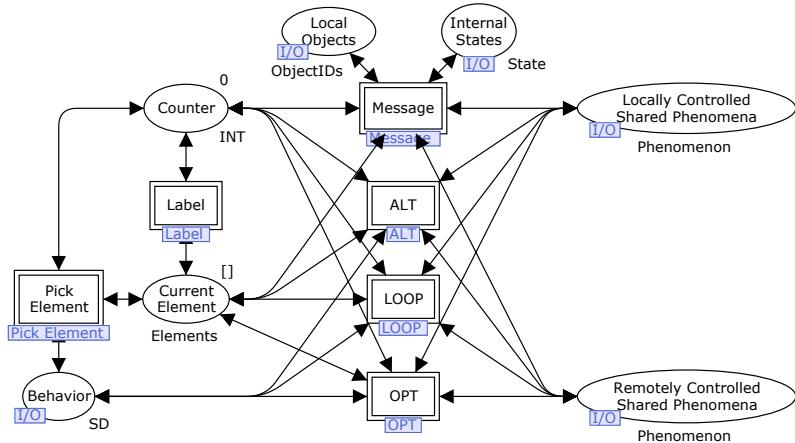


Fig. 10. The SD Interpreter module.

Fig. 11 illustrates the CPN ML representation of the sequence diagram in fig. 5 (a similar representation exists for the sequence diagram in fig. 6). This sequence

diagram specifies a scenario of UC1 as seen by the customers through their interaction with the physical entities. Here, it has been translated into a list value, which is placed (as a token) in the place **Scenarios** 2 and interpreted by the **Human Actors** substitution transition.

A simple language has been developed to represent the basic features of UML 2.0 sequence diagrams, such as optionals (OPT), alternatives (ALT), loops (LOOP), and messages (arcs). Each of these features is handled by a separate substitution transition in the **SD Interpreter** module. The interpreter utilises the parameterised knowledge about local objects (and the derived implicit knowledge about remote objects) to determine the direction of messages (events or state changes) during the execution of the list representation of a sequence diagram.

If a message is outgoing (i.e., generated by a local object), this is reflected by the interpreter altering a local phenomenon, either by generating a new event token or by modifying the value of a state token in the place called **Locally Controlled Shared Phenomena**. Alternatively, if a message is incoming (i.e., generated by a remote object), the interpreter halts until this message is detected in the place called **Remotely Controlled Shared Phenomena**. This is the basic mechanism for the synchronisation of the instances of sequence diagram interpreters with the physical entities modelled in regular CPN modules. In the example of fig. 5, the **customer** object is local to the **Human Actors** instance, while the physical entities are remote. The ALT, LOOP, and OPT operators do not involve any exchange of messages, but rely on the interface of shared places in order to evaluate predicates that may involve shared states. When the interpreter encounters a predicate in one of these operators, the current value of a relevant shared state is investigated to evaluate the predicate.

The basic operation of executing the CPN ML representation of a sequence diagram as performed by the **SD Interpreter** module can be described as follows: The interpreter traverses the CPN ML list one element at a time. This is controlled by a counter (maintained in the place **Counter**) that somehow resembles a program counter. The substitution transition **Pick Element** picks out the next element of the list based on the current state of the counter found in the single token value found in the **Counter** place. A single element is produced in the **Current Element** place and from here it is consumed and handled by one of these substitution transitions based on its type: **Message**, **ALT**, **LOOP**, **OPT**, or **LABEL**. As an example of how specific elements are handled, Fig 12 shows the contents of the **LOOP** substitution transitions that handles the elements used for representing loop structures of arbitrary levels of depth found in sequence diagrams. It can be seen how shared phenomena are evaluated through access to the interface places described earlier (**Remotely Controlled Shared Phenomena** and **Locally Controlled Shared Phenomena**). This makes it possible for the interpreter to evaluate the predicates that may exist in the definition of a specific loop in order to determine when to enter and leave the loop based on shared phenomena.

Fig. 13 documents the collection of colour sets that are used throughout the model.

```

val UC1_customer =
UC2_customer ^^
[
Message(("customer", "nozzle-detector"),
EventOccurrence("pick-nozzle", 95, EventParameter(0))),
Message(("display", "customer"),
StateChange("display", 0, AnyStateParameter)),
LOOP_HEAD(1, INT_(5), NoPredicate, "a", "b"),
Label("a"),
Message(("customer", "nozzle-detector"),
EventOccurrence("trigger-nozzle", 95, EventParameter(0))),
Message(("display", "customer"),
StateChange("display", 95, AnyStateParameter)),
LOOP_TAIL(),
Label("b"),
Message(("customer", "nozzle-detector"),
EventOccurrence("return-nozzle", 95, EventParameter(0)))
];

```

Fig. 11. The CPN ML representation of the sequence diagram found in fig. 5

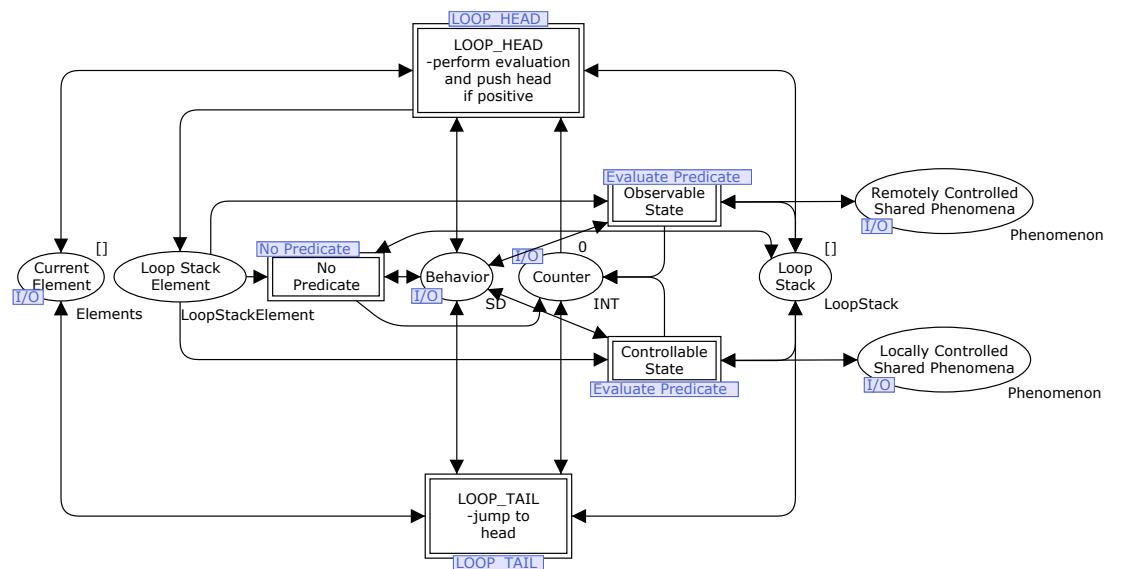


Fig. 12. The LOOP module.

```

colset ObjectID = STRING;
colset ObjectIDs = list ObjectID;
colset EventID = STRING;
colset Direction = product ObjectID * ObjectID;
colset EventParameter = INT;
colset OptionalEventParameter =
    union EventParameter:EventParameter + AnyEventParameter;
colset EventIndex = INT;
colset StateID = STRING;
colset Event = product EventID * EventIndex * EventParameter;
colset StateIndex = INT;
colset StateParameter = INT;
colset OptionalStateParameter =
    union StateParameter:StateParameter + AnyStateParameter;
colset StateChange = product StateID*StateIndex * OptionalStateParameter;
colset EventOccurrence =
    product EventID * EventIndex * OptionalEventParameter;
colset State = product StateID * StateIndex * StateParameter;
colset PredicateType =with NEQ | EQ | GT | LT | GTE | LTE;
colset StateChangeOrEvent =
    union EventOccurrence:EventOccurrence + StateChange:StateChange;
colset Message = product Direction * StateChangeOrEvent;
colset Predicate =
    product StateID * StateIndex * PredicateType * StateParameter;
colset OptionalPredicate =
    union Predicate:Predicate + NoPredicate + NonDeterministic;
colset Label = STRING;
colset Phenomenon = union State:State + Event:Event;
colset OPT_HEAD = product OptionalPredicate * Label * Label;
colset INTorINF = union INT_:INT + INF;
colset LOOP_HEAD =
    product INT * INTorINF * OptionalPredicate * Label * Label;
colset LOOP_TAIL = UNIT;
colset ALT_ELEMENT = product Predicate * Label * Label;
colset ALT_ELEMENT_ELSE = product Label * Label;
colset ALT_HEAD = Label;
colset ALT_TAIL = UNIT;
colset AltStackElement = product ALT_HEAD * BOOL;
colset AltStack = list AltStackElement;
colset Element = union Message:Message + Label:Label +
    OPT_HEAD:OPT_HEAD + LOOP_HEAD:LOOP_HEAD +
    LOOP_TAIL:LOOP_TAIL + ALT_HEAD:ALT_HEAD +
    ALT_ELEMENT:ALT_ELEMENT + ALT_ELEMENT_ELSE:ALT_ELEMENT_ELSE +
    ALT_TAIL:ALT_TAIL;
colset LoopStackElement = product LOOP_HEAD * INT * INT;
colset LoopStack = list LoopStackElement;
colset Elements = list Element;
colset SD = Elements;
colset Lst = list INT;
colset SCState = State;
colset OptionalEvent = union Event_:Event + NoEvent;
colset SCTransition =
    product SCState * OptionalEvent *
    OptionalPredicate * SCState * OptionalEvent;
colset SCStates = list SCState;
colset SCTransitions = list SCTransition;
colset SC = product ObjectID * SCState * SCStates * SCTransitions;
colset FuelValveState = union Open + Closed;
colset FuelValve = product INT * FuelValveState;
colset FuelMeasurement = product INT * INT;

```

Fig. 13. The colour sets used in the model

4.4 Discussion

The reflections behind the work presented in this paper are inspired by the work of Jackson and particularly by his work on Problem Frames [5]. The reactive system we deal with in this paper fits well in the Commanded Behaviour Problem Frame specified in [5] but our approach from the Problem Frames approach in a central point: we explicitly model the human actors observing the states and events of the domain of physical entities. This is necessary in order to synchronise the execution of scenarios between the human actors and the physical entities (and the system as a whole).

The main purpose of the CPN model we present in this paper is to provide the modeller of a reactive system with a generic structure that can be used as a starting point for capturing functional requirements and knowledge about the physical environment in a sensible way.

The requirements are specified as a collection of scenarios describing use cases in which the final system must be able to interact according to the expected behaviour. To validate the scenarios, the CPN model suggests the behaviour of the controller to be specified at a relatively-high abstract level. This permits to base a prototypical design of the controller on sequence diagrams that describe scenarios of use cases. At the same time, different sequence diagrams are used to describe scenarios of the behaviour of the human actors; and thereby required behaviour of the entire system consisting of the physical entities in combination with the controller.

The specification of the behaviour of the controller is relatively abstract, since it does not necessarily include descriptions of any internal components of the controller. At a later point in the development process, such components may be introduced by refining the sequence diagrams used to describe the controller behaviour (as the one found in fig. 6). The abstract description of the controller behaviour is necessary to permit the modeller to execute the scenarios specified for the human actors in a simulated environment with responses from the system. This is as an important property of the modelling approach, since it may be helpful in the complex task of specifying and validating the functional requirements.

5 Conclusions

The contribution of this paper is a model-based approach to requirements engineering for reactive systems. Its application is illustrated in an automatic gas pump controller. The approach suggests the creation of a CPN model based on the requirements expressed as use cases and sequence diagrams, for validating the functional requirements of the system under development.

A generic structure is proposed for the CPN model, so that it is possible to address the modelling of the controller, the physical entities which the controller interacts with, and the human users that operate the system. We suggest the CPN modules for modelling the behaviour of the human users and the controller

to be instances of a generic module that is able to interpret scenario descriptions specified in CPN ML. This proves to be a good solution, since the size of the CPN module remains the same independently of the number of considered scenarios.

In contrast, for modelling the behaviour of the physical entities (actuators and sensors) we use regular CPN modules, i.e., modules that directly use the graphical constructs of the CPN language (places, transitions, arcs, etc.), to model behaviour.

The CPN language is a good choice for modelling these two types of modules, since it allows the complexity of the model to be split between graphical and textual constructs, and also between the data and the control perspectives.

As future work, we plan to extend the SD Interpreter module to handle all UML 2.0 sequence diagrams constructs, and also to apply our approach to other types of reactive systems, like for example interactive systems, workflow systems, and robotic systems.

References

1. J. Desel, V. Milijic, and C. Neumair. Model Validation in Controller Design. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 467–95. Springer, 2004.
2. J.M. Fernandes, S. Tjell, and J.B. Jørgensen. Requirements Engineering for Reactive Systems: Coloured Petri Nets for an Elevator Controller. Technical report, DAIMI, University of Aarhus, Denmark, July 2007.
3. J.M. Fernandes, S. Tjell, J.B. Jørgensen, and O. Ribeiro. Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net. In *6th Int. Workshop on Scenarios and State Machines (SCESM 2007), at ICSE 2007*. IEEE CS Press, 2007.
4. D. Heimbolt and D. Luckham. Debugging Ada Tasking Programs. *IEEE Software*, 2(2):47–57, 1985.
5. M. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2001.
6. Michael Jackson and Pamela Zave. Deriving Specifications from Requirements: an Example. In *17th International Conference on Software Engineering (ICSE '95)*, pages 15–24, New York, NY, USA, 1995. ACM Press.
7. K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *Software Tools for Technology Transfer*, 2007. In Press. DOI: 10.1007/s10009-007-0038-x.
8. O. R. Ribeiro and J. M. Fernandes. Some Rules to Transform Sequence Diagrams into Coloured Petri Nets. In *7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2006)*, pages 237–56, 2006.
9. R.J. Wieringa. *Design Methods for Reactive Systems: Yourdon, Statemate, and the UML*. Morgan Kaufmann, 2003.

On the Use of Coloured Petri Nets for Visual Animation *

Óscar R. Ribeiro João M. Fernandes

Dep. Informática / CCTC, Universidade do Minho, Braga, Portugal

Abstract. This paper reports on an exercise on constructing a visual animation layer for a behaviourally-intensive reactive system. We assume that the requirements of the system under consideration are described by use cases, and the behaviour of each use case is detailed by a collection of scenario descriptions. These use cases and scenarios are translated into a Coloured Petri Net (CPN) model, which is subsequently complemented with animation-specific elements. We describe how the CPN model must be structured to facilitate the animation process, and we present the supporting tools for creating the animation. We consider an elevator controller system as a case study, to demonstrate that a CPN model complemented with a visual animation layer constitutes a solid basis for addressing behavioural issues in an early phase of the development process, namely during the validation task.

1 Introduction

Validation consists on checking if a system or a model satisfies the user expectations. One of the key issues to have a successful validation is to adopt a process where users can actively discuss the requirements of the system under development. To accomplish the validation task, one can use a visual animation [1] that employs domain-specific vocabulary, since it allows users and developers to be confident that the right system is being built.

When developing a reactive system [2], which usually has an intensive behaviour and a rich set of interactions with its environment, requirements validation is an important task to accomplish before deciding any design and implementation issues.

The Unified Modeling Language (UML) is currently the standard notation adopted in industry to model software systems. This work uses two UML diagrams: Use Case Diagrams and Sequence Diagrams. Use cases specify the set of functionalities presented by a system as seen by its users, and permit, due to their simplicity, the dialogue between clients and developers. A sequence diagram is used to capture a behavioural scenario of a given system, which can be seen as a sequence of steps describing interactions between the actors and that system. We suggest each use case to be described by a set of sequence diagrams.

* This work has been supported by the grant with reference SFRH/BD/19718/2004 from “Fundação para a Ciência e Tecnologia”.

Coloured Petri Net (CPNs) [3] constitute a graphical modelling language appropriate to describe the behaviour of systems with characteristics like concurrency, resource sharing, and synchronization. The CPN Tools [4, 5] is a well established tool supporting the CPN modelling language and allowing the execution of animations in accordance with the CPN model.

This paper describes the construction of a visual animation layer for the problem domain of the elevator controller case study. The visual animation can be used during the validation to facilitate the dialogue between the system developers and the clients. This animation layer described in this work is intended to be controlled by an executable CPN model. We give some guidelines to create the CPN model from the sequence diagrams present in the requirements of the system under development. We also introduce a tool to support the development of the animation layer.

This paper is structured as follows. Section 2 describes the elevator controller case study to be considered in this paper. In Section 3, we present the development of an animation layer for the elevator controller. We introduce also some tools to support this development. Section 4 introduces some guidelines to construct a CPN model for animation purposes. The conclusions are presented in Section 5.

2 Case Study

In this section we introduce the case study used in this paper, an elevator controller. We consider that this controller manages an elevator system with two cars in a building with six floors. This case study is an adaptation from the description presented in the technical report [6].

Fig. 1 depicts the context diagram for the elevator controller, where the main sensors and actuators present in the considered top-most entities (**Floor** and **Car**) are shown. Each **Floor** contains two **Location Sensors**, one for each car, to detect when the respective car is at or is arriving to the floor. In each **Floor** there are **Hall Buttons** to allow the passengers to call an elevator car indicating the direction (up or down) he wishes to travel. Obviously, the first and the last floors have only one button to select the unique direction that is possible to travel (up for the first floor and down for the top-most floor). There is a **Floor Door** in each floor to protect the car's shaft, and the doors only open when the corresponding car is stopped at the floor.

Each of the two **Cars** has one **Car Motor** to move up or down along the floors. Each car has a **Door** which has: (1) a **Door Timer** to automatically close the door after a given amount of time; (2) a **Door Sensor** to detect if there is something obstructing the door during closure; (3) a **Door Motor** to open/close the door of the elevator; and (4) a **Car Door**, which includes two sensors, to indicate either if the door is closed or totally opened. The **Car Door** is mechanically linked with the **Door Motor**, and also with the corresponding door in each floor.

Inside the car a passenger can find a **Floor Indicator** that shows the current floor of the car, a **Direction Indicator** that shows the direction being followed by

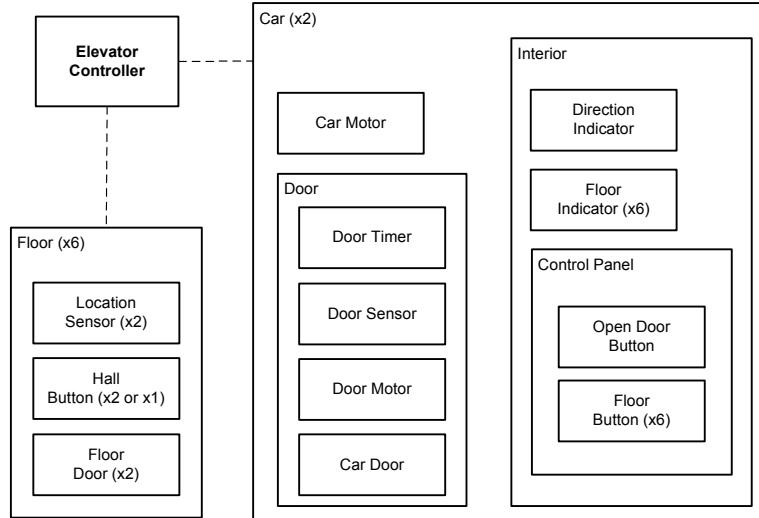


Fig. 1. Context Diagram for the Elevator Controller

the car, and a Control Panel that contains an Open Door Button to open the doors and six Floor Buttons to select the destination floor.

The behaviour of the elevator controller is triggered by the passengers' actions. The elevator controller has the responsibility of managing the movements of the cars in accordance to the requests of the passengers through pushing on the buttons.

Our approach proposes the usage of a use case diagram to depict the main functionalities provided by the system to its users. In this paper, with the purpose of illustrating the suggested approach, we just consider the use case "Service Floor". This use case is responsible for moving an elevator car from an origin floor to a destination floor, by request of a passenger either in one of the building's hall or inside an elevator car.

Fig. 2 depicts the sequence diagram with the main scenario of the "Service Floor" use case. This sequence diagram uses some high-level operators present in the UML 2.0, namely the `opt`, and the `loop` operators. These operators permit the description of several scenarios in a unique sequence diagram. In order to abstract from the scenarios presented in the sequence diagram, there are along the left-hand side of the diagram some textual annotations where some variables being used in the messages (and guards) are informally declared. For example, the diagram in Fig. 2 abstracts from the car, and the origin and destination floors. With the textual annotation "*Elevator car c is at Floor Fo, and the next requested floor is the Fd*", we are declaring the variables *c*, *F_o* (origin floor) and

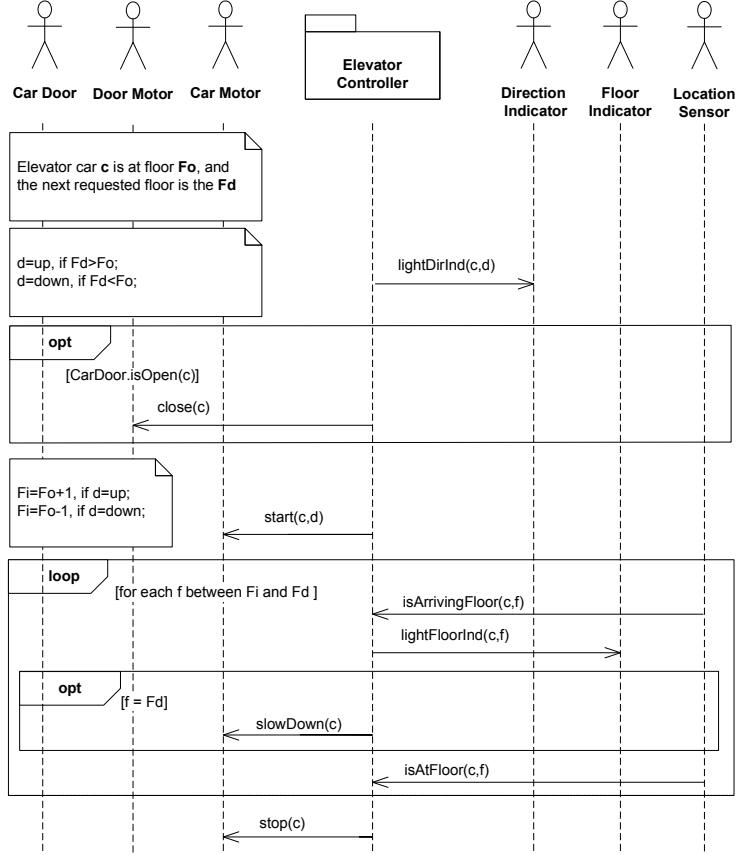


Fig. 2. Sequence Diagram describing the “Service Floor” use case.

F_d (destination floor), to be used as parameters for the scenario present in the diagram. With these variables, we are able to define variable d that represents the direction followed by the car, in the textual annotation “ $d=\text{up}$, if $F_d > F_o$; $d=\text{down}$, if $F_d < F_o$ ”. The textual annotation “ $F_i = F_o + 1$, if $d=\text{up}$; $F_i = F_o - 1$, if $d=\text{down}$ ”; defines the variable F_i , representing the next floor from the origin floor.

The scenario presented in Fig. 2 describes the following behaviour:

1. The passenger in the current floor is notified about the direction the car will take (message `lightDirInd`);
2. If the car door is open (high-level operator `opt`)
 - (a) The car door is closed (message `close`)
3. The car is moved in direction to the destination floor (message `start`);
4. While the destination floor is not reached (high-level operator `loop`):

- (a) The location sensor informs the elevator controller that the car is arriving to the next floor (message `isArrivingFloor`);
 - (b) The Floor Indicator corresponding to the next floor is activated (message `lightFloorInd`);
 - (c) The Car Door must slow down its speed, when the next floor is the destination floor (message `slowDown`);
 - (d) The location sensor informs the elevator controller that the car is at the next floor (message `isAtFloor`);
5. The car stops (message `stop`).

In this main scenario of the “Service Floor” use case, we are assuming that during its execution there are no other interaction from the passengers with the buttons. These situations could origin some other variations to this main scenario, but in this work we will consider only the main scenario of the use case.

3 Building an Animation to the Case Study

In this section, we present the development of an animation layer for the elevator controller introduced in the previous section. We describe also how to use the animation, and which tools are involved in its deployment.

3.1 Initial Considerations

The visual animation to be built is intended to reproduce the behaviour of the elevator controller, specified by the considered collection of sequence diagrams. It can be used during the validation to facilitate the dialogue between the system developers and the clients, which is a critical facility to ensure that both parts agree on what is to be developed.

One can start the construction of a visual animation, after some initial work on the analysis task has been accomplished. For this purpose, it is useful to have the context diagram, since it enumerates the main entities of the environment with which the controller interacts. Typically, these entities are strong candidates to be represented in the animation, since the system’s behaviour depends on them.

It is also important to have the use case descriptions, together with their corresponding sequence diagrams, since they describe which messages are received by the elements in the environment, and how these elements react on those messages. These artefacts specify the behaviour that must be covered by the animation layer.

The construction of the animation must be synchronised with the activity of creating the CPN model, because the CPN model must include some elements which are specific to control the animation as explained in next section.

The animation has been created using the SceneBeans tool [7, 8], which is a framework for creating and controlling animations, using the Java programming

language. There is a XML-based file format to define animations and a parser to translate those XML files into animation objects for SceneBeans.

In the SceneBeans architecture there are as basic elements scene graphs, behaviours and animations. A *scene graph* is implemented in JavaBeans by a direct acyclic graph, which draws a two-dimensional image. In the leaf nodes of a scene graph there are primitive shapes (such as circles, ellipses, rectangles). An intermediate node either combines or transforms its subgraphs. The combination can be done in two ways: putting one subgraph on the top of another; or choosing one from the set of subgraphs. In a transform node it is possible to apply a linear transformation followed by a translation to its subgraphs (for example rotation, scaling or translation) or to change the way that its subgraphs are rendered (for example, changing the colour in which a node is drawn).

Associated to each graphic element in the animation, there are some behaviours (not to be confused with the behaviour of the controller) to animate some of the properties of the element. A *behaviour* in SceneBeans is implemented by a Java bean that controls a time value, and when the value changes it announces an event. This permits the animation of the visual appearance of the scene graph. Notice that there is a so-called *animation* thread that manages the frame rate of the overall animation signals the passage of time. There is also the possibility to define commands to call the execution of a set of behaviours and to announce an event when they finish.

3.2 Static part of the animation

The behaviour of the elevator controller, as the behaviour of reactive systems in general, lies in the interaction with its environment, by sending messages to the environment, which in response can also send messages in the opposite direction (i.e., to the controller). In this work we consider the elements in the environment as the actors of the elevator controller system. The actors for the “Service Floor” use case are the ones that participate in the sequence diagram in Fig. 2.

An elevator can be visually represented by a picture with the floors and the cars, where the cars can go up/down across the floors. While the cars are moving, the elevator controller must update the information shown in the panels inside each elevator car and attend the requests from the passengers.

For the elevator controller, only a subset of the entities of the environment are relevant for animation, since the passenger is not aware of (or does not interact with) all of them. This means that the animation layer only includes the relevant entities. In contrast, the CPN model does specify all the environment’s entities. In the elevator controller, the passenger interacts only with the following six entities: Hall Button, Car Door, Direction Indicator, Floor Indicator, Open Door Button, and the Floor Button. The validation focuses essentially on the reactions of the controller to requests made by the passengers. If some flaw on the behaviour of the elevator is detected during validation, the developer may also need to analyse all the entities of the environment (even those that do not appear in the visual animation), to identify and understand the cause of the error.

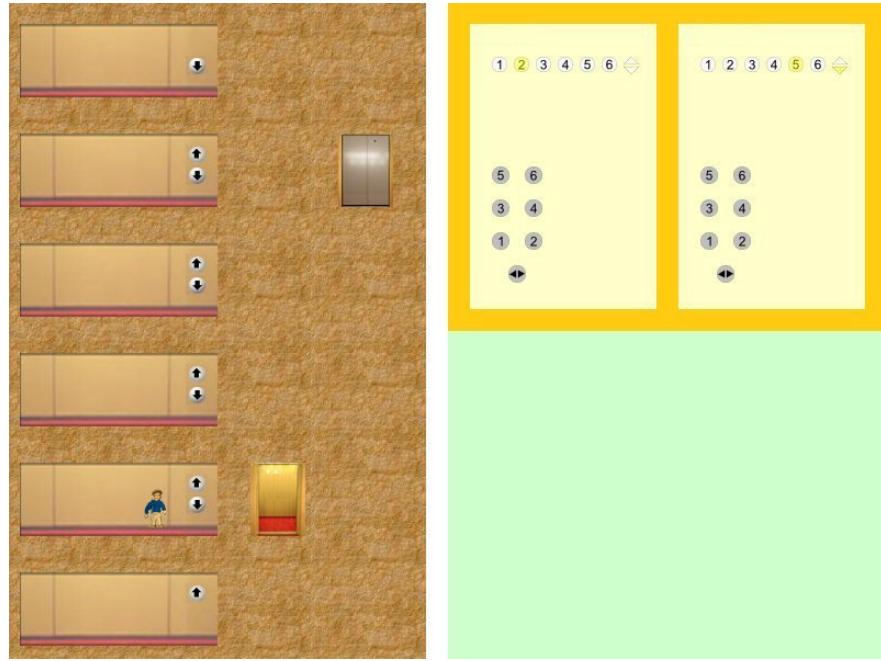


Fig. 3. An animation of the elevator system.

Fig. 3 shows a screenshot obtained from the animation of the elevator controller. On the left part of the figure, there is a representation of each floor with the buttons to call an elevator. We can also see the two elevator cars in Fig. 3: the left-hand side elevator is at the second floor with its door open, and the right-hand side elevator is at the fifth floor with its door closed.

On the right part of the figure, the interior of both cars are depicted. Each car has a floor indicator, which has one light for each floor, and only the light of the current floor is on. There are also two lights, one to indicate the up direction and another one to indicate the down direction, showing the direction being followed by the car. Fig. 3 shows that the left-hand side car has the light with the number two in yellow indicating that this light is on, and thus that the car is currently at the second floor. The left-hand side car currently has no direction, and the light direction of the right-hand side car indicates that the car is going down.

The structure of the animation is essentially based on importing some icons to represent an element of the system or on drawing a geometric artefact using a XML tag. This structure is present in the leaf nodes of the scene graph, and for example to include the image door.png we use the XML tag primitive, as follows:

```

1 <primitive type="sprite">
2   <param name="src" value="./images/door.png"/>
3 </primitive>
```

3.3 Dynamic part of the animation

In this subsection we show how to define the dynamic part of an animation in the SceneBeans XML-based format.

As we said before, in the leaf nodes of a scene graph there are primitive shapes and in the intermediate nodes either combination or transformation of its subgraphs using a set of parameters. Each parameter can be associated to a behaviour that needs to be previously defined. To allow the user to control the existing behaviours, there are commands, and each one includes a sequence of behaviour invocations. Thus, calling a command results on the animation of some of the parameters present in the intermediate nodes.

With respect to the animation of our case study, let us consider how to specify in the XML-based format the animation of the **Direction Indicator** entity and, in particular, how the message `lightDirInd` is handled. This message allows the **Direction Indicator** to change its state, among its possible values (up, down, and idle). The **Direction Indicator** is composed of two triangular lights. If the car is going up (down), the top light “ Δ ” is on (off) and the bottom light “ ∇ ” is off (on). If the car is stopped, the idle direction is represented by delighting both lights.

For example, to indicate that the car is going up, we use the following XML code, whose XML tags `param` in lines 2-4 are used to set the parameters `from`, `to` and `duration` of the behaviour:

```

1 <behaviour algorithm="move" event="ldi" id="showlightDirInd(rightCar,up)">
2   <param name="from" value="-1000"/>
3   <param name="to" value="750"/>
4   <param name="duration" value="0.0001"/>
5 </behaviour>
```

This block of code defines a behaviour that is based on a specific movement of an animation icon from a given point ('from') to another point ('to') during a given time ('duration'). This behaviour will be associated to the parameters in the nodes of the scene graph.

To indicate, during the animation, that the car is going up, it is needed to put an icon (showing the top light on, and the bottom light off) at the position of the **Direction Indicator** entity in the animation picture. This is achieved by the following XML code:

```

1 <transform type="translate">
2   <param name="translation" value="(-1000,50)"/>
3   <animate param="x" behaviour="showlightDirInd(rightCar,up)"/>
4   <animate param="x" behaviour="hidelightDirInd(rightCar,up)"/>
5     <primitive type="sprite">
6       <param name="src" value="./images/direction_indicator_up.png"/>
7     </primitive>
8 </transform>
```

In lines 3 and 4, behaviours `showlightDirInd` (presented in the previous block of code) and `hidelightDirInd` are associated to the parameter “x” of the transformation node, in order to change the x-axis position of the icon, i.e., they move the icon horizontally in the animation picture. These behaviours are invoked through their inclusion in a command definition as we present in the next block of code.

There is one icon to represent each state of the **Direction Indicator** entity, and `showlightDirInd` behaviour moves the respective icon to a visible part of the animation, and the `hidelightDirInd` behaviour moves the respective icon to a non-visible part of the animation. Thus, the change to a new state is animated showing the icon of the new state and hiding the other two icons.

To allow the external invocation of these behaviours in order to animate the changing of the **direction indicator** in the car on the right-hand side to indicate the up direction, the following command announces an event with the same name.

```

1 <command name="lightDirInd(rightCar,up)">
2   <start behaviour="hidelightDirInd(rightCar,down)"/>
3   <start behaviour="hidelightDirInd(rightCar,idle)"/>
4   <start behaviour="showlightDirInd(rightCar,up)"/>
5   <announce event="lightDirInd(rightCar,up)"/>
6 </command>
```

There are similar code blocks for the other two possible directions and for the other car.

3.4 Scripting Language

We have created a script to facilitate the manipulation of the XML tags in the XML-file that specify the animation. Ruby [9, 10] is a scripting language that follows the principles of object-oriented programming. The Ruby script uses components from the library REXML [11].

With Ruby we can easily manipulate the XML to be generated, namely when repetitive parts of the XML code follow a given pattern. To generate the XML code for the animation, a Ruby script was created. Next we show part of that script that generates the XML code for the animation of the Direction Indicator.

```

1 require 'builder'
2 require 'sbXMLgen.rb'

3 xmlBuilder = Builder::XmlMarkup.new(:target => $stdout, :indent => 3)

4 xmlBuilder.instruct! :xml, :version => "1.0"
5 xmlBuilder.animation("width" =>"800", "height" => "600" ){
6   carIds = ["left","right"]
7   carInteriorCoord = {"left" => Tuple.new(420,20), "right" => Tuple.new(610,20)}
8   lstBhLightDI = Hash.new()

9   lstCommands = Hash.new()
10  carIds.each{ |carId|
11    bhparams = Hash.new()
12    durQuick = 0.0001
13    xHide = -1000 }
```

```

14     xVisible = carInteriorCoord[carId].getX+7*20
15     bparams["show"] = BehaviourParams.new(xHide,xVisible,durQuick)
16     bparams["hide"] = BehaviourParams.new(xVisible,xHide,durQuick)
17     directions = ["up", "down", "idle"]
18     movs = ["show", "hide"]
19     bh = Hash.new()
20     movs.each{ |m| bh[m] = Hash.new() }
21     lstBhLightDI[carId] = {"up"=>[], "down"=>[], "idle"=>[]}
22     mkCmdStrDI= lambda{|mov, cld, dir| "#{mov}lightDirInd(#{cld}Car,#{dir})"}
23
24     directions.each{ |d|
25       movs.each{ |mov|
26         cmdStrDI = mkCmdStrDI.call(mov,carId,d)
27         bh[mov][d] = Behaviour.new(cmdStrDI, "move", bparams[mov], "xpto")
28         bh[mov][d].toXML(xmlBuilder)
29         lstBhLightDI[carId][d].push(Tuple.new("x", cmdStrDI))
30         if (mov=="show") then
31           d_tmp = directions.dup
32           d_tmp.delete(d)
33           lstBhToStart = Array.new()
34           lstBhToStart.push(cmdStrDI)
35           d_tmp.each{ |od|
36             cmdStrDIod = mkCmdStrDI.call("hide",carId,od)
37             lstBhToStart.push(cmdStrDIod)
38           }
39           cmdStrDIname = mkCmdStrDI.call("",carId,d)
40           lstCommands[cmdStrDIname] = lstBhToStart.dup
41
42     }
43   }
44   lstCommands.each{ |k,v| make_command(xmlBuilder, k , v) }
45   xmlBuilder.draw {
46     carIds.each{ |carId|
47       draw_DirectionIndicator(xmlBuilder, carInteriorCoord[carId],
48                               lstBhLightDI[carId])}
48 }

```

In line 1, the package from the REXML library to build XML tags is included. When we define an object as a `Builder` (see line 3 where we define the variable `xmlBuilder`) we are able to generate a XML tag by using the name of the tag to be generated as as a method for the object, for example in line 5 is created the following XML structure:

```

1 <animation height="600" width="800">
2 ...
3 </animation>

```

Inside the tag `animation` it is included the XML code generated by the lines 6-47 of the Ruby script.

In line 6, the variable `carIds` is defined as an array with the car identifiers. It is possible to create an iteration over this variable, as is shown in line 10. This permits an easy integration of new similar cars in the animation, because we have a dynamic structure based on the elements into the array `carIds`.

Line 2 includes some code from file “`sbXMLgen.rb`” created by us, in order to save some functions to be used on the generation of XML for animations, independently from the case being considered. For example, the code in line 26 creates an object of the class `Behaviour` which has the following definition:

```

1 class Behaviour
2   def initialize(id, algorithm, params, event, announce = "no")
3     @id      = id
4     @algorithm = algorithm
5     @params = params # class BehaviourParams
6     @event   = event
7     @toBeAnnounced = announce
8   end
9   def toXML(xb)
10    xb.behaviour("id" => @id, "algorithm"=> @algorithm , "event" => @event) {
11      @params.toXML(xb)
12    }
13  end
14 end

```

This a simple definition of a Ruby class, where we can find the same parameters as the behaviour XML tag, and the definition of the method `toXML` to generate the corresponding XML code. The advantage is that we can use this class to have a less verbose (i.e., easier to read by humans) code to specify a behaviour. Consequently, if we repeat this process for all other XML tags present in the SceneBeans XML-based format, we obtain a less verbose way to define an animation layer.

We believe that this Ruby Script constitutes an abstract way to deal with the XML-based stuff, in particular it is an easy way to work with parameterising in the visualizations. When comparing it with the `forall` construct we think that the Ruby script is a more flexible and user-friendly way to manipulate the parameters. The `forall` construct has the advantage that it is defined in the same XML-file as the rest of the animation definitions.

4 Coloured Petri Net Models

In this section, we describe some guidelines that we suggest to be taken into account when creating a CPN model for animation purposes. These guidelines are discussed and are exemplified with respect to the case study.

The CPN model that is constructed from a set of scenario descriptions is an executable model that can drive a graphical animation layer showing elements and concepts from the problem domain. Additionally, the CPN model needs to include a mechanism to manage how animation events are handled. The animation layer, in the SceneBeans XML-based format for the case study considered in this work, is presented in the previous section.

The BRITNeY suite [12, 13] animation tool is used to connect the execution of the CPN model in the CPN tools with the SceneBeans objects corresponding to the animation specified in XML-based file. We use the SceneBeans plug-in present in the BRITNeY suite to display and interact with a SceneBeans animation.

As stated before, we consider that the requirements process includes the creation of a set of use cases. The behaviour of each use case is detailed by a collection of scenario descriptions, which can be represented by sequence diagrams. In version 2.0 of UML, sequence diagrams have many high-level flow operators.

The translation from these sequence diagrams to a CPN model is based on the general principles described in [14], which associate a transition on the CPN model for each message in the sequence diagram and define some mechanisms in the CPN model to represent the high-level operators present in the sequence diagrams.

We describe the construction of a CPN model to execute the scenarios described by the sequence diagram in Fig. 2, and also the changes that need to be accomplished to allow the obtained CPN model to regulate the animation introduced in the previous section.

4.1 Mapping sequence diagrams into a CPN model

Firstly, we suggest that each sequence diagram is translated to a CPN model where there is a substitution transition for each message or high-level operator in the sequence diagram. The places between substitution transitions guarantee the order between the messages in the sequence diagram, and their colour set needs to include the necessary information to allow the parallel execution of many scenarios.

Fig. 4 shows a CPN model that was obtained from the sequence diagram in Fig. 2, where messages, and high-level operators in the sequence diagram are represented by substitution transitions in the CPN model. For example, the message `lightDirInd` is represented by the substitution transition with the same name, and the first `opt` operator in the sequence diagram is represented by the substitution transition “`opt (Is car door open?)`”. The messages inside the `opt` operator are present inside the corresponding subpage. The subpages contain the necessary details to animate the messages in the sequence diagram.

Places in the CPN of Fig. 4 have the colour set `ScenarioUC2`, which provides the necessary information to execute a scenario of “Service Floor” use case, namely the origin and destination floors and the car being used. In other words, the definition of the colour set `ScenarioUC2` comes from the textual annotation “*Elevator car c is at floor Fo, and the next requested floor is the Fd*”, from the sequence diagram in Fig. 2, where the variables `c`, `Fo` and `Fd` are informally declared. These implicit and informal declarations of variables by a textual annotation allow for the usage of the same sequence diagram in different situations, by using the variables as a parameter in messages, or even in other textual annotations. The definition of the colour set `ScenarioUC2` in the CPN ML programming language has the following code:

```

1 colset ScenarioUC2tmp = record
2           c: CarId *
3           fo: FloorNumber *
4           fd: FloorNumber ;
5 fun hasDiffFloors(s:ScenarioUC2tmp)= (#fo s) <> (#fd s) ;
6 colset ScenarioUC2 = subset ScenarioUC2tmp by hasDiffFloors;

```

`CarId` and `FloorNumber` are defined as integers to identify a car and a floor, respectively. The colour set `ScenarioUC2tmp` is created essentially to be used in

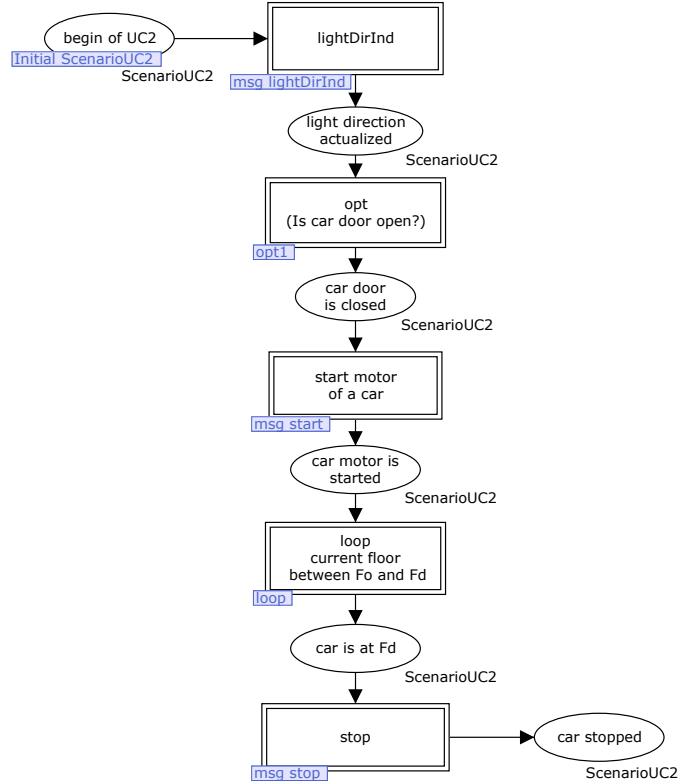


Fig. 4. CPN model representing the sequence diagram of UC2.

the definition of the colour set `ScenarioUC2`. When executing an instance of the “Service Floor” use case, it is implicit that the origin and destination floors are different. This is specified using the predicate `hasDiffFloors` to restrict the colour set `ScenarioUC2tmp` to obtain the colour set `ScenarioUC2`, whose elements are guaranteed to have different origin and destination floors.

The colour set `ScenarioUC2` is used to distinguish between parallel executions of the use case, and thus the colour set must identify the car, the origin floor, and the destination floor. To start the execution of the CPN model it is necessary to define the input parameters using the place `begin of UC2`, where several tokens can be put to allow the parallel execution of different UC2 instances.

The places in the CPN model of Fig. 4 ensures that the order between messages in the sequence diagram is maintained when firing the transitions in the CPN model. Each token in the place `begin of UC2` means that a “Service Floor” has been requested to be executed, i.e., a given car must travel from a origin to a destination floor. It is assumed that the environment is in conditions to allow the execution of this scenario, provided by other use cases (not considered in this

paper). The conditions to permit the execution of an instance of the “Service Floor” use case are that the selected car must be at the origin floor, with its motor stopped.

A token in the place light direction actualized means that the direction indicator light is now indicating the direction that the car is taking to go from the origin to the destination floor, and allows the next transition to be enabled.

4.2 Data representation for the environment

Secondly, it is important to define a data representation of the main elements in the environment of the system under development. This data description is used to represent the behaviour of each message in the sequence diagram.

To obtain a description of the system’s environment, its elements are specified as data in the CPN model using the CPN ML programming language, defining a colour set for each element in the environment. In our case study, the cars and the floors are the top-most entities of the environment. For example, the colour set **Car** is a record with an identification of the car, the door of the car, the motor to move the car, and the sensors and actuators inside the car.

```

1 colset Car = record
2     id      : CarId      *
3     motor   : CarMotor   *
4     door    : CarDoor    *
5     interior : CarInterior ;

```

Where the **CarId** colour set is an integer, the **CarMotor** colour set is a tuple containing its moving speed and the direction being followed. The **CarDoor** colour set is defined as a record stating if the door is open or closed, and representing also the motor, the sensor and the timer of the door.

```

1 colset CarDoor = record
2     door      : Door      *
3     doorMotor : DoorMotor *
4     doorSensor: DoorSensor*
5     doorTimer : DoorTimer ;

```

Similarly, the entities inside a car are defined by the colour set **CarInterior** and they include the lights to indicate the direction being followed by the car, the lights to indicate the current floor (there is a light for each floor), and a control panel where we can find a button to open the door, and buttons to allow the passenger to select one of the existing floors.

```

1 colset CarInterior = record
2     directionIndicator : Direction   *
3     floorIndicator      : FloorNumber *
4     controlPanel        : ControlPanel;

```

We consider that the components in the car are part of the colour set, such as the car door, the door motor, the door sensor, the door timer and the car motor.

The textual annotation “ $d = up$, if $F_d > F_o$; $d = down$, if $F_d < F_o$,” in the sequence diagram of Fig. 2 is represented by the function `calcDirection` which takes a `ScenarioUC2` colour set and gives a direction based on the origin and destination floors, and is defined as follows:

```

1 colset Direction = with up | down | idle ;
2 fun calcDirection(a:ScenarioUC2) =
3     case (Int.compare(#fo p,#fd p)) of
4         LESS    => up
5         |GREATER => down
6         |EQUAL   => idle ;

```

Some of the messages in the sequence diagram have the parameter direction, which can be calculated using the values of the `ScenarioUC2`. For example, the message `lightDirInd` uses the parameter direction.

4.3 Animation of messages in the sequence diagrams

Thirdly, one must detail how the execution of each substitute transition in the CPN model representing a message in the sequence diagram is animated in the SceneBeans animation.

The communication between the CPN model and the SceneBeans animation is done using an animation object which can be used in the code segments of CPN model to invoke some commands to be executed in the SceneBeans animation. The CPN model contains the following declaration of the ”anim” as a SceneBeans object:

```

1 structure anim = SceneBeans(val name = "Elevator Controller Animation");

```

Fig. 5 shows the subpage associated to the message `lightDirInd`. The animation of a message in the sequence diagram is divided in two transitions of the CPN model: the first one (`lightDirInd`) to invoke the command in the animation, and the second one (`ack lightDirInd`) to wait for the feedback from the animation, informing that the command has been executed. Thus, a token in the place `waiting for lightDirInd` event means that the animation is updating the appearance of the direction indicator in the animation. This mechanism, that waits for the event from the animation, ensures the synchronization between the animation and the execution of the CPN model.

To ask for an execution of a command in the SceneBeans animation we use the “`invokeCommand`” method which can be used for an object with an SceneBeans animation. The parameter for this method is a string which must correspond to a command in the animation. The command `lightDirInd` is used to animate the Direction Indicator, as defined in the previous section, and has two parameters: (1) the name of the car, and (2) the direction that the car is taking. To ease the creation of the command identifier, the following function can be created:

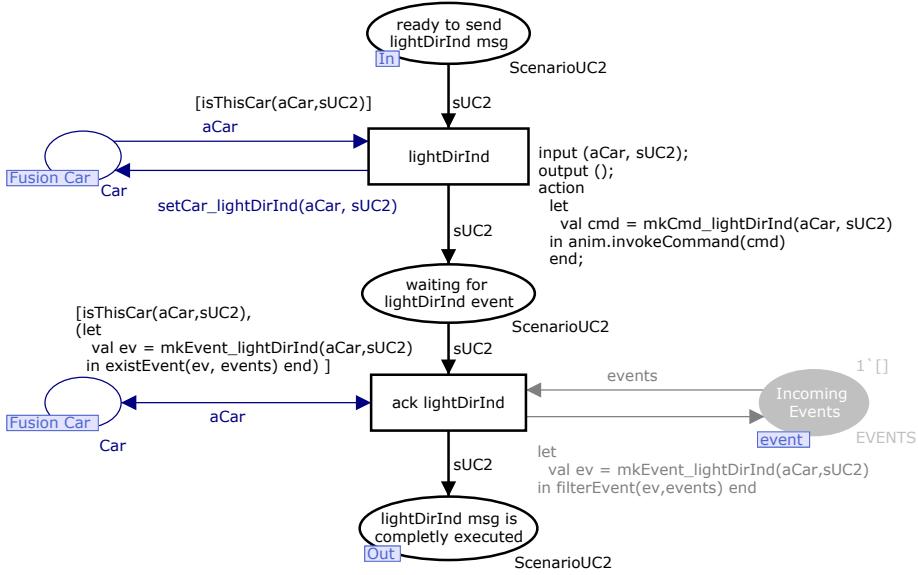


Fig. 5. CPN model for the execution of the message `lightDirInd`.

```

1 fun mkCmd_lightDirInd(aCar:Car, sUC2: ScenarioUC2) =
2 let
3   val sc      = carName(aCar)
4   val strd = showDir(calcDirection(#floors sUC2))
5 in "lightDirInd(\"^sc ^"Car,"^ strd ^")" end

```

To verify the incoming of the corresponding event of the command `lightDirInd` it is necessary to test if the event is in the list of incoming events. Our CPN model is constantly aware of the events being generated by the animation through the module in Fig. 6. The running fusion place is used to test if the animation is already started.

4.4 Initial conditions for scenario execution

Fourthly, we suggest to add a module to the CPN model where the initial conditions for the scenario execution and the selection of the SceneBeans XML file to used are introduced.

The CPN model in Fig. 7 analyses the initial conditions of the environment that the user wants to simulate and subsequently initialises the animation. The initial conditions are introduced in the pre-places (in green) of transition `Initialise`. The firing of the `Initialise` transition invokes the necessary commands to start running the animation according to the specified conditions. The Fig. 7 constitutes a top page of the CPN model where for each scenario a separated

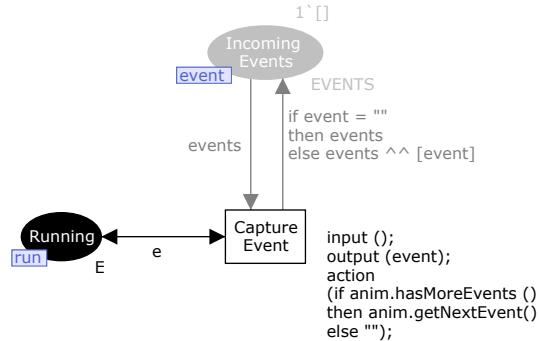


Fig. 6. Subpage of CPN model to capture events from the SceneBeans animation.

subpage exists and for which the start place can be found in Fig. 7 thought a fusion place. The running place is used to allow the execution of CPN model in subpage in Fig.6. There are also fusion places to connect the places with the initial values with places that represent the environment values in the subpages.

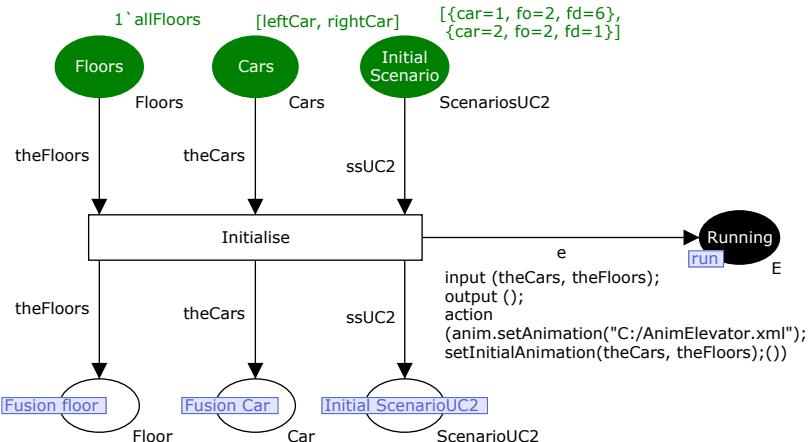


Fig. 7. CPN-model to initialise the environment values and the SceneBeans animation.

Let us see now how to change the CPN model to allow the consideration of a different numbers of cars and floors in the animation. To do this we need to adapt both the CPN model and the ruby file presented in page 9.

The changes in the CPN model are done in the topmost page presented in Fig. 7, changing the initial marking of the pre-places for **Initialise** transition. It is also necessary to change the value of a constant that represents the number of floors and another that represents the number of cars. These changes must be complemented with the corresponding changes in the Ruby file namely in what

concerns with the variables that represent the available cars and the available floors. And also the position of different elements in the graphic animation.

Supposing that we want to introduce a new car in the middle of the other two existing cars we define a constant, e.g. centerCar, with its data representation, to be included in the list of initial marking in place “Cars” of Fig. 7, which results in the list [leftCar, centerCar, rightCar]. If we want to execute a scenario related to this new car we need to add the scenario description to the list of initial scenarios to be executed. Now we will refer to the Ruby script code listed in page 9. Firstly in line 5, it is necessary to increase the height of the animation to have some space to introduce the icon for the additional car. In line 6 we add the identification label that we want to use for the identification of the new car, and in line 7 we add to the hash table the coordinates for the new car. As we can see in line 10, the code contains a loop over the identifiers of the cars.

5 Conclusions

In this paper we have described the creation of an animation layer for an elevator controller system case study. We consider that the animation layer is controlled by a CPN model, which has some additional elements that are specific to control the animation layer. The created animation is used during the validation task.

The idea of use an animation of a CPN model for the requirements validation is not new, for example Machado et al. present an approach to support the validation of workflow requirements for the interaction between people for a case study from a real project where animation were used [15]. Although these authors also consider that the CPN models are obtained from sequence diagrams, with this work we want to improve the mapping from sequence diagrams into the CPN model in order to support the parallel execution of many scenarios. We consider that the case study of Elevator controller is an example where the parallel execution of many scenarios is useful.

The CPN model is obtained from sequences diagrams that represents a set of scenario descriptions present in the use case behaviour. This CPN model explicitly model the entities on the environment that are important for the animation of the elevator controller system. It is possible to execute a given scenario in the CPN model for an initial state of the environment selected by the user. The mechanisms that were added to the CPN model to manipulate the animation are easily identified, thus it can be eliminated from the CPN model allowing the reuse of the CPN model in other tasks of the development process.

The animation layer consists on a representation of the problem domain in a user-friendly language, where the relevant entities of the system under development have a graphic representation with associated animations to represent the different behaviour of the entity. We use the SceneBeans tool to create the animation layer, which is defined using a XML-based file format. To assist on the management of XML-based code we present a script that permits an abstraction from the details associated to the usage of XML tags.

References

1. Dulac, N., Viguier, T., Leveson, N., Storey, M.A.: On the use of visualization in formal requirements specification. In: Proc. of IEEE Joint Int. Conf. on Requirements Engineering. (2002) 71–80
2. Wieringa, R.J.: Design Methods for Reactive Systems: Yourdon, Statemate, and the UML. Morgan Kaufmann (2003)
3. Jensen, K.: Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use. Volume 1-3. Monographs in Theoretical Computer Science. EATCS Series. Springer (1992-97)
4. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. Int. Journal on Software Tools for Technology Transfer (STTT) **9**(3-4) (June 2007) 213–54
5. CPN Tools. Online: www.daimi.au.dk/CPNtools.
6. Blanco, R.M.: Requirements Specification for an Elevator Controller. Technical report, School of Computer Science, University of Waterloo, Canada (2005)
7. SceneBeans. Online: www-dse.doc.ic.ac.uk/Software/SceneBeans/.
8. Magee, J., Pryce, N., Giannakopoulou, D., Kramer, J.: Graphical animation of behavior models. In: Proc. of the 22nd Int. Conference on Software Engineering (ICSE'00), New York, NY, USA, ACM Press (2000) 499–508
9. Thomas, D., Fowler, C., Hunt, A.: Programming Ruby: The Pragmatic Programmers' Guide, Second Edition. Pragmatic Bookshelf (October 2004)
10. Ruby Programming Language. Online: www.ruby-lang.org/en/.
11. Ruby: REXML. Online: www.germane-software.com/software/rexml/.
12. Westergaard, M., Lassen, K.B.: The BRITNeY Suite Animation Tool. In Springer-Verlag, ed.: Proc. of 27th Int. Conference on Applications and Theory of Petri Nets (ICATPN'06). Volume 4024 of LNCS. (2006) 331–40
13. Westergaard, M.: The BRITNeY Suite: A Platform for Experiments. In: 7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2006). (2006)
14. Ribeiro, O.R., Fernandes, J.M.: Some Rules to Transform Sequence Diagrams into Coloured Petri Nets. In: 7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2006). (2006)
15. Machado, R.J., Lassen, K.B., Oliveira, S., Couto, M., Pinto, P.: Requirements Validation: Execution of UML Models with CPN Tools. Int. Journal on Software Tools for Technology Transfer (STTT) **9**(3-4) (June 2007) 353–369

Towards Modelling and Validation of the DYMO Routing Protocol for Mobile Ad-hoc Networks*

Kristian L. Espensen, Mads K. Kjeldsen, and Lars M. Kristensen

Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N, DENMARK,
`{espensen, keblow, kris}@daimi.au.dk`

Abstract When mobile devices come within physical range of one another it is possible for them to form a mobile ad-hoc network (MANET). Messages in MANETs are routed between the devices and can thereby reach further than the range of a single device. To facilitate multi-hop communication between nodes in a MANET, a routing protocol is needed. In this paper we consider the Dynamic MANET On-demand (DYMO) routing protocol and focus on the route establishment procedures of the protocol. The DYMO protocol is currently under development by the Internet Engineering Task Force (IETF). The aim of our project is to use Coloured Petri Nets to construct a complete model of the DYMO protocol, to formally verify key properties using state spaces, and use the constructed CPN model via gradual refinement for the actual implementation of the DYMO protocol. The CPN model presented in this paper contains the basic parts of the protocol and we are currently working on a full model of the DYMO protocol. We also present results from an initial state space analysis of the constructed model. By using different scenarios we validate the protocols ability to establish routes and judge the usefulness of the routing information contained in the routing messages.

1 Introduction

There are basically two types of mobile networks — infrastructured and infrastructureless. In an infrastructured wireless networks (such as GSM networks and WLANs), mobile nodes connect to a base station which functions as a gateway to a fixed infrastructure. In an infrastructureless wireless network, known as a *mobile ad hoc network* (MANET) [15], there are no fixed routers or base stations. All nodes can move around in the network and they can connect in an arbitrary way. MANETs are often characterised by nodes entering and leaving the network at a high rate and sometimes the link between two nodes is unidirectional (asymmetric). The link could become bidirectional over time, but the nature of asymmetric networks is something to keep in mind when designing routing protocols. Since there are no dedicated routers in the network the nodes must function as routers themselves. A typical application of MANETs is emergency search-and-rescue operations in remote areas where no preexisting communication infrastructure is available.

There are two main approaches to routing protocols in MANETs: *proactive* and *reactive* [17]. The proactive protocols are table driven and each node maintains a route to every other node in the MANET. In MANETs, the nodes often have limited memory, processing power, and battery capacity. A reactive routing protocol is therefore often more suitable since it creates routes *on-demand* which generates less traffic than proactive protocols. The Ad hoc On-demand Distance Vector routing protocol (AODV) [14] and the Dynamic Source Routing protocol (DSR) [10] are well-known examples of reactive routing protocols. AODV only supports bidirectional links, whereas DSR also supports unidirectional links. The Dynamic

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

MANET On-demand (DYMO) [2] routing protocol is currently being developed by the IETF MANET working group [11] and builds upon the AODV and DSR protocols. The DYMO protocol specification [2] is currently an Internet-draft in its 10th revision and is expected to become a Request for Comments (RFC) document in the near future.

The recent discussions on the mailing list [12] of the MANET working group have revealed several complicated issues in the DYMO protocol specification, in particular related to the processing of routing messages and the handling of sequence numbers. This combined with the experiences of our research group with implementing the DYMO protocol [18,19] and conducting initial modelling the DYMO protocol [6] has motivated us to initiate a project aiming at constructing a Coloured Petri Net (CPN) model [8,9] of the complete DYMO protocol specification with the goal of validating the correctness of the protocol using state space analysis, and via a set of refinement steps to use the CPN model (preferably via automatic code generation) as a basis for implementing the DYMO protocol. This paper presents our initial work on this project. We present a CPN model of the basic route establishment procedures of the DYMO protocol. In addition to modelling DYMO, the modelling of the wireless mobile network itself is aimed at being generally applicable for modelling MANET protocols. Finally, we present our initial state space analysis results concerning the establishment of routes and processing of routing messages.

The modelling and validation of routing protocols for MANETs has also been considered by other researchers. The AODV protocol has been considered in [20,21]. An abstract CPN model of routing in MANETs focusing on the DSDV protocol was presented in [22] and the WARP Routing Protocol was modelled and verified using the SPIN model checker [7] in [3]. The LUNAR routing protocol was modelled and verified using the UPPAAL tool [16] in [13] and the RIP and AODV protocols were verified in [1] using a combination of the SPIN model checker and the HOL theorem prover [5]. Closest to our work is the modelling of the DYMO protocol presented in [23]. A main difference between our modelling approach and the work of [23] is that [23] presents a highly compact CPN model of the DYMO protocol consisting of just a single module aimed at conducting state space analysis. Our aim is to eventually use the CPN model as a basis for implementing the DYMO protocol, and we have therefore decided on a more verbose modelling approach which includes organising the CPN model into several modules to break up the complexity of modelling the complete DYMO protocol and which makes it easier to later refine parts of the CPN model as required. Furthermore, [23] uses simulation to investigate properties of the protocol whereas we present some initial state space analysis results. On the other hand, [23] models a larger subset of the DYMO protocol, including the route maintenance procedures and intermediate nodes appending information to the routing messages and report on several problematic issues in the specification of the DYMO protocol. Compared to [23] we have chosen an explicit approach for modelling node mobility and the topology of the MANET. At the end of this paper we give a more detailed discussion comparing our CPN model to the CPN model of [23].

The rest of this paper is structured as follows. Section 2 gives an overview of the DYMO protocol and its basic operation. Section 3 presents the CPN model and provides additional details about the DYMO protocol. In Sect. 4 we present our initial investigations of the behaviour of the DYMO protocol using state space analysis. Finally, in Sect. 5 we summarise our initial findings and discuss future work. The reader is assumed to be familiar with the CPN modelling language [9] and the basic ideas of state space analysis.

2 Overview of the DYMO protocol

The operation of the DYMO protocol can be divided into two parts: *route discovery* and *route maintenance*. The route discovery part is used to establish routes between nodes in the network when required for communication between two nodes. A route discovery begins with an *originator node* multicasting a Route Request (RREQ) to all nodes in its immediate range. The RREQ has a *sequence number* to enable other nodes in the network to judge the freshness of the route request. The network is then flooded with RREQs until the request reaches its *target node* (provided that there exists a path from the originating node to the target node). The target node then replies with a Route Reply (RREP) unicasted hop-by-hop back to the originating node. The route discovery procedure is requested by the IP network layer on a node when it receives an IP packet for transmission and does not have a route to the destination. The IP packet will then be queued in the network layer waiting for DYMO to establish the route and inform the IP layer that a route has been discovered.

A small example scenario illustrating route discovery is shown in figure 1. The scenario consists of six nodes numbered 1–6. A edge between two nodes indicates that the two nodes are within transmission range of each other. As an example, node 1 is within transmission range of nodes 2 and 3.

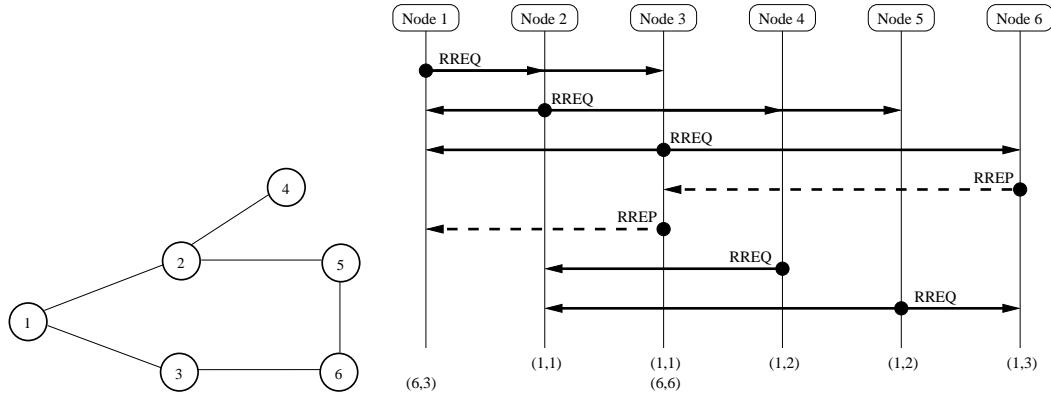


Figure 1. A simple MANET topology with six nodes (left) and MSC for route discovery process (right).

The message sequence chart (MSC) shown in Fig. 1 (right) depicts one possible exchange of messages in the DYMO protocol when the originating node 1 establishes a route to target node 6. Solid arcs represent multicast and dashed arcs represent unicast. In the MSC, node 1 multicasts a RREQ which is received by nodes 2 and 3. When receiving the RREQ from the node 1, nodes 2 and 3 will create an entry in their routing table specifying a route back to the originator node 1. Since nodes 2 and 3 are not the target of the RREQ they both multicast the received RREQ to their neighbours (nodes 1, 4 and 5, and nodes 1 and 6, respectively). Node 1 will discard these messages as it was the originator of the RREQ. When nodes 4 and 5 receive the RREQ they will add an entry to their routing table specifying that the originator node 1 can be reached via node 2. When node 6 receives the RREQ from node 3 it will discover that it is the target node of the RREQ, add an entry to its routing table specifying that node 1 can be reached via node 3, and unicast a RREP back to node 3. When node 3 receives the RREP it will add an entry to its routing table stating that node 6 is within direct range, and

use its entry in the routing table that was created when the RREQ was received to unicast the RREP to node 1. Upon receiving the RREP from node 3, node 1 will add an entry to its routing table specifying that node 6 can be reached using node 3 as a next hop. The RREQ will also be multicasted by node 4, but when node 2 receives it it will discard it as it will be considered *inferior*. Node 5 also multicasts the RREQ, but nodes 2 and 6 will also consider the RREQ to be *inferior* and therefore discard the RREQ message. The two last lines in the MSC specifies the entries in the routing table of the individual nodes as a pair (*target, nexthop*). The first line specifies the entries that was created as a result of receiving the RREQ and the second line specifies entries created as a result of receiving the corresponding RREP. It can be seen that a bidirectional route has been discovered and established between node 1 and node 6 using node 3 as an intermediate hop. The routing table entries in the other nodes pointing back to the originator node 1 will eventually timeout as no RREP are being received from the target node.

As illustrated above, RREQ and RREP messages are used to perform route discovery. These messages contain among other things information about the originator of the message, which node is the target, the hop count between the nodes, and a sequence number. The intermediate nodes on the forwarding route extract information about the other nodes from both RREQ and RREP. This information is used to maintain a routing table held by each node. It is important that the routing table is updated correctly to avoid old routing information and routes with possible loops to be propagated in the network. The protocol specifies how to judge the quality of new incoming routing information. The new routing information is compared with the information held in the routing table and is classified as *stale*, *loop-possible*, *inferior* or *superior*. Only superior routing information is used to update the routing table and only messages with superior routing information are processed further.

Nodes in a MANET are continuously entering and leaving. Because of this, routes need to be maintained and a node therefore monitors the nodes it is directly connected to. The DYMO protocol has a mechanism to notify nodes about a broken route. This is done by sending a Route Error (RERR), thereby informing nodes using the route that a new route discovery is needed.

The CPN model presented in this paper does not model all operations specified in the DYMO specification. We have not modelled timeouts, i.e., route table entry timeouts (cf. [2], Sect. 5.2.3), setting timeouts when creating and updating routing entries with new routing information (cf. [2], Sect. 5.2.2) and updating route lifetime during packet forwarding (cf. [2], Sect. 5.5.2.). Actions to be taken if a node loses its sequence number, e.g., as a result of a reboot or crash (cf. [2], Sect. 5.1.4) is not in our model since this operation depends on the timeout concept. Active link monitoring (cf. [2], Sect. 5.5.1) is used to detect broken links and is a subject for future work. Furthermore, the following optional DYMO operations are not in our current version of the model: intermediate DYMO Router RREP creation (cf. [2], Sect. 5.3.3), adding additional routing information to a RM (cf. [2], Sect. 5.3.5) and simple internet attachment and gatewaying (cf. [2], Sect. 5.8).

3 The DYMO CPN model

The CPN model is a hierarchical model organised in 12 modules. Figure 2 shows the module hierarchy of the CPN model. Each node in Fig. 2 corresponds to a module and System represents the top-level module of the CPN model. An arc leading from one module to another module means that the latter module is a submodule of the former module. It can be

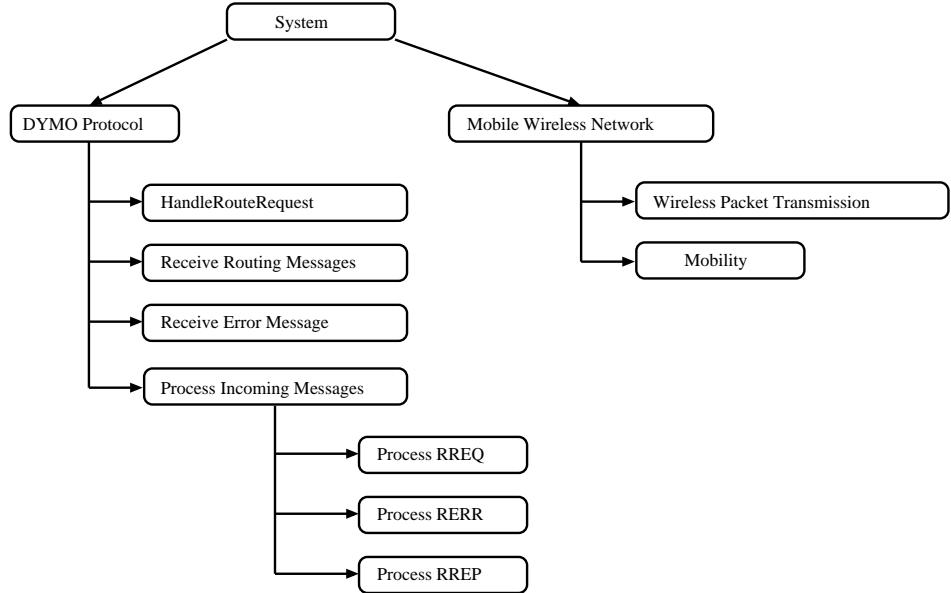


Figure 2. Module hierarchy for the DYMO CPN model.

seen that the model has been organised into two main parts: a **DYMO Protocol** part and a **MobileWirelessNetwork** part. This has been done to separate the parts of the model that are specific to DYMO which are all submodules of the **DYMOProtocol** module and the parts that are independent from the DYMO protocol which are all submodules of the **MobileWirelessNetwork** module. This means that the parts modelling the mobile wireless network over which DYMO operates can be reused for modelling other MANET protocols. We have adopted the convention that a substitution transition and its associated submodule have the same name. Furthermore, to the extent possible we have structured the CPN model into modules such that it corresponds to the structure of the DYMO specification [2]. This makes it easier to understand the relationship between the DYMO specification and the CPN model, and it makes it easier to maintain the CPN model as the DYMO specification is being revised.

The top-level module **System** is shown in Fig. 3 and is used to connect the two main parts of the model. The DYMO protocol logic is modelled in the submodules of the **DYMOProtocol** substitution transition. The submodules of the **MobileWirelessNetwork** substitution transition is an abstract model of the mobile wireless network over which DYMO operates. It models unreliable one-hop wireless transmission of network packets over a network with mobile nodes. It represents operation of the IP network layer down to the physical transmission over the wireless medium.

The two socket places **DYMOToNetwork** and **NetworkToDYMO** are used to model the interaction between the DYMO protocol and the underlying protocol layers as represented by the submodules of the **MobileWirelessNetwork** substitution transition. When DYMO sends a message, it will appear as a token representing a network packet on place **DYMOToNetwork**. Similarly, a network packet to be received by the DYMO protocol module will appear as a token on the **NetworkToDYMO** place. The colour set **NetworkPacket** is defined as follows:

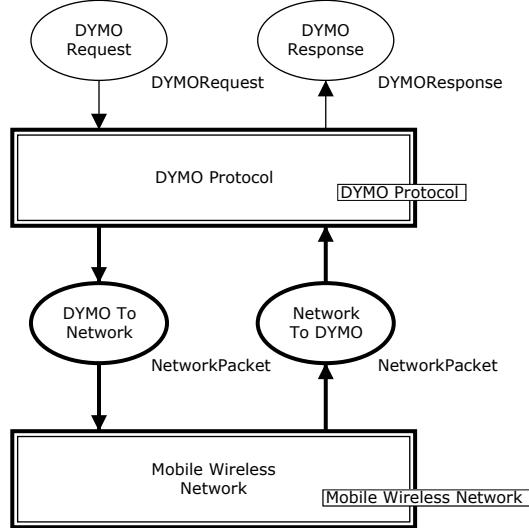


Figure 3. Top-level System module of the CPN model.

```

colset Node = int with 0 .. N;

colset IPAddress = union UNICAST : Node +
                  LL_MANET_ROUTERS;

colset NetworkPacket = record src : IPAddress *
                        dest : IPAddress *
                        data : DYMOMessage;

```

We have used a record colour set for representing the network packets transmitted over the mobile wireless network. A network packet consists of a source, a destination, and some data (payload). The DYMO messages are carried in the data part and will be explained in detail later. DYMO messages are designed to be carried in UDP datagrams transmitted over IP networks. This means that our network packets are abstract representations of IP/UDP datagrams. We have abstracted from all fields in the IP and UDP datagrams (except source and destination fields) as these do not impact the DYMO protocol logic. The source and destination of a network packet are modelled by the `IPAddress` colour set. There are two kinds of IP addresses: `UNICAST` addresses and the `LL_MANET_ROUTERS` multicast address. The multicast address is used, e.g., in route discovery when a node is sending a RREQ to all its neighbouring nodes. Unicast addresses are used as source of network packets and, e.g., as destinations in RREP messages. A unicast address is represented using an integer from the colour set `Node`. Hence, the model abstracts from real IP addresses and identify nodes using integers in the interval $[1; N]$ where `N` is a model parameter specifying the number of nodes in the MANET.

The two places `DYMOREquest` and `DYMOREsponse` are used to interact with the service provided by the DYMO protocol. A route discovery for a specific destination is requested via the `DYMOREquest` place and a DYMO response to a route discovery is then provided by DYMO via the `DYMOREsponse` place. The colour sets `DYMOREquest` and `DYMOREsponse` are defined as follows:

```

colset RouteRequest = record originator : Node *
                      target      : Node;

```

```

colset DYMOResponse = union ROUTERESPONSE : RouteResponse;

colset RouteResponse = record originator : Node *
                      target      : Node *
                      status       : BOOL;

colset DYMOResponse = union ROUTERESPONSE : RouteResponse;

```

A **DYMOResponse** specifies the identity of the originator node requesting the route and the identity of the target node to which a route is to be discovered. Similarly, a **DYMOResponse** contains a specification of the originator, the target, and a boolean status specifying whether the route discovery was successful. The colour sets **DYMOResponse** and **DYMOResponse** are defined as union types to make it easy to later extend the model with additional request and responses. This will be needed when we later refine the CPN model to more explicitly specify the interaction between the DYMO protocol module and the IP network layer module. By setting the initial marking of the place **DYMOResponse**, it can be controlled which route requests are to be made. We have not modelled the actual transmission of messages containing payload from applications as our focus is on the route establishment and maintenance of the DYMO protocol. In the following we present the submodules of the **DYMOProtocol** and **MobileWirelessNetwork** substitution transitions in more detail.

3.1 The DYMO Protocol Module

The highest abstraction level for the DYMO protocol part of the CPN model is the **DYMOProtocol** module shown in Fig. 4. The module has four substitution transitions modelling the handling of route request from a user (**HandleRouteRequest**), the reception of routing messages **ReceiveRoutingMessages** which are RREQ and RREP messages, the reception of RERRs (**ReceiveErrorMessages**), and the processing of incoming messages (**ProcessIncomingMessages**).

All modules of the substitution transitions in Fig. 4 creates and manipulates DYMO messages which are represented by the colour set **DYMOMessage** defined as follows:

```

colset SeqNum          = int;
colset NodexSeqNum    = product Node * SeqNum;
colset NodexSeqNumList = list NodexSeqNum;

colset RERRMessage = record HopLimit      : NO *
                      UnreachableNodes : NodexSeqNumList;

colset RoutingMessage = record TargetAddr : Nodes   *
                           OrigAddr     : Nodes   *
                           OrigSeqNum  : SeqNum  *
                           HopLimit    : INT     *
                           Dist        : INT;

colset DYMOMessage = union RREQ : RoutingMessage +
                      RREP : RoutingMessage +
                      RERR : RERRMessage;

```

The definition of the colour sets used for modelling the DYMO messages is a direct translation of the description of DYMO messages as found in Sect 4.4.2 and Sect.4.2.3 of [2] modulo the parts of DYMO that we are currently not modelling. In particular we use the same names of

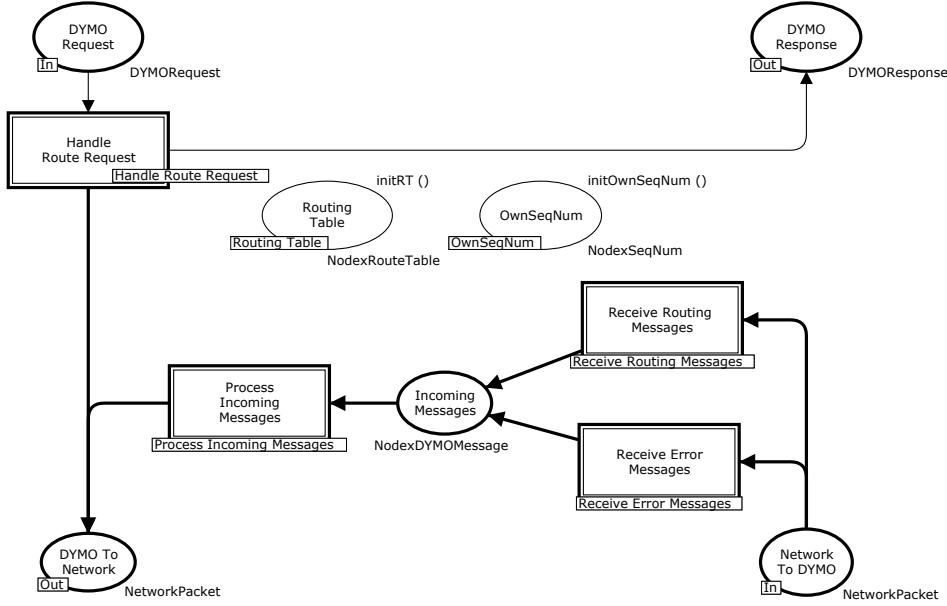


Figure 4. The DYMOProtocol module.

message fields as in [2]. In the modelling of DYMO messages packets, we have have abstracted from the specific layout as specified by the packetbb format [4]. This is done to ease the readability of the CPN model, and the packet layout is not important when considering only the functional operation of the DYMO protocol.

The submodules of the DYMOProtocol module also access the routing table and the sequence number maintained by each mobile node. The routing table and the sequence number are global data structures within each node. To reflect this and reduce the number of arcs in the modules, we decided to represent the routing table and the node sequence numbers using fusion places. The place **OwnSeqNum** contains a token for each node specifying the current sequence number of the node and the place **RoutingTable** contains a token for each node specifying a routing table. The colour set **SeqNum** used to represent the sequence number of a node was defined above and the colour **RouteTable** used to represent the routing table of a node is defined as follows:

```

colset RouteTableEntry = record Address      : IPEndPoint *
                           SeqNum        : SeqNum *
                           NextHopAddress : IPEndPoint *
                           Broken        : BOOLEAN *
                           Dist          : INT;

colset RouteTable      = list RouteTableEntry;
colset NodexRouteTable = product Node * RouteTable;

```

A routing table is represented as a list of **RouteTableEntry**. To allow each node to have its own routing table, we use the colour set **NodexRouteTable** for representing the set of routing tables such that the first component of a pair belonging to this colour set specifies the identity of the node to which the routing table in the second component is associated. The definition of the colour **RouteTableEntry** is a direct translation of routing table entries as described in Sect. 4.1 of [2]. In addition to the mandatory fields, we have included the optional **Dist** field as

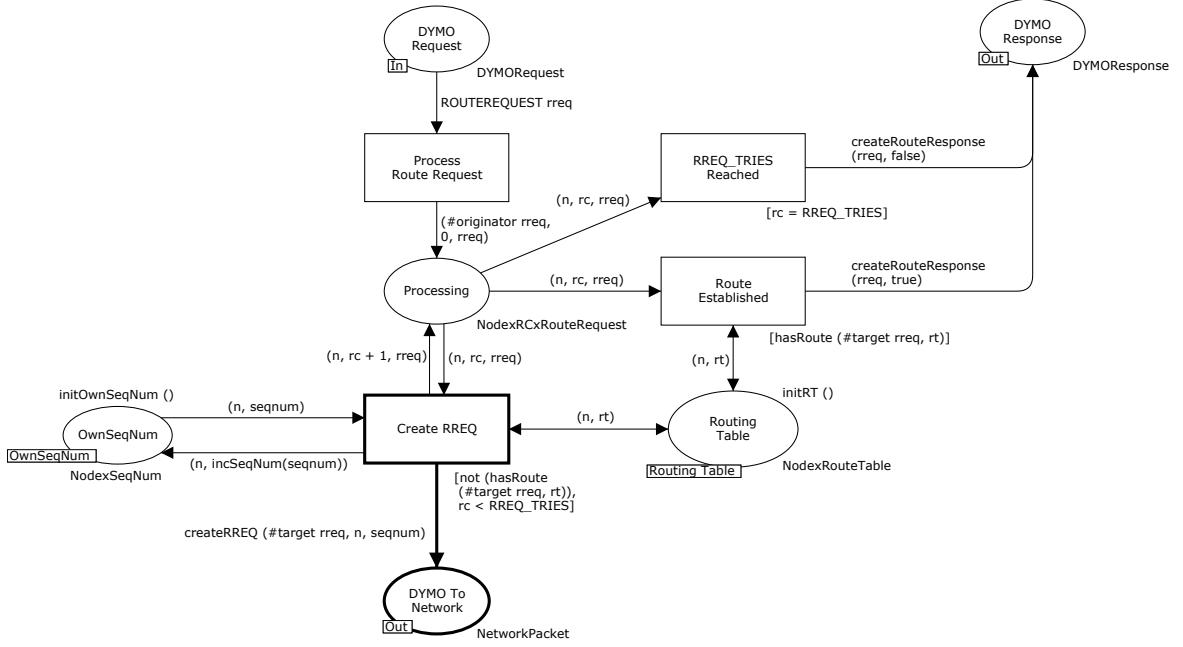


Figure 5. The Handle Route Request module.

this is used in non-trivial ways in the reception of routing messages. We wanted to investigate this operation in more detail as part of the state space analysis and we have therefore included it in the modelling.

Handle Route Request Module. Figure 5 shows the HandleRouteRequest module. When a route request arrives via the DYMORequest input port the ProcessRouteRequest transition is enabled and when it occurs it will initialise the processing of the route request by putting a token on place Processing. A route request being processed is represented by a token over the colour set NodexRCxRouteRequest which is a product type where the first component specifies the node processing the route request (i.e., the originator), the second component specifies how many times the RREQ has been retransmitted, and the third component specifies the route request. If the node does not have a route to the target and the retransmit limit RREQ_TRIES for RREQs has not been reached (as specified by the guard of the CreateRREQ transition), then a RREQ message can be transmitted with the current sequence number of the node. Upon sending a RREQ, the sequence number of the node is incremented and so is the counter specifying how many times the RREQ has been transmitted. If a route becomes established (i.e., the originator receives a RREP on the RREQ), the RouteEstablished transition becomes enabled and a token can be put on place DYMOREsponse indicating that the requested route has been successfully established. If the retransmission limit for RREQs is reached (before a RREP is received), the RREQ_TRIES transition becomes enabled and a token can be put on place DYMOREsponse indicating that the requested route could not be established.

Receive Routing Messages Module. When a routing message arrives at the DYMO protocol module the first task is to compare the routing information in the received message with the information contained in the routing table of the receiving node. Judging the routing

information contained in a routing message is handled by the `ReceiveRoutingMessages` module shown in Fig. 6. The receiver of the message is found in the `dest` field of the incoming network packet bound to `np`. This way we know which node is the current node and thereby which routing table to access. Section 5.2.1 of [2] specifies how routing information is divided into four classes. Each class has a boolean expression specifying when routing information falls into the given class. In the model each class is represented by a accordingly named transition with a guard which is true exactly when the boolean expression corresponding to that class is true.

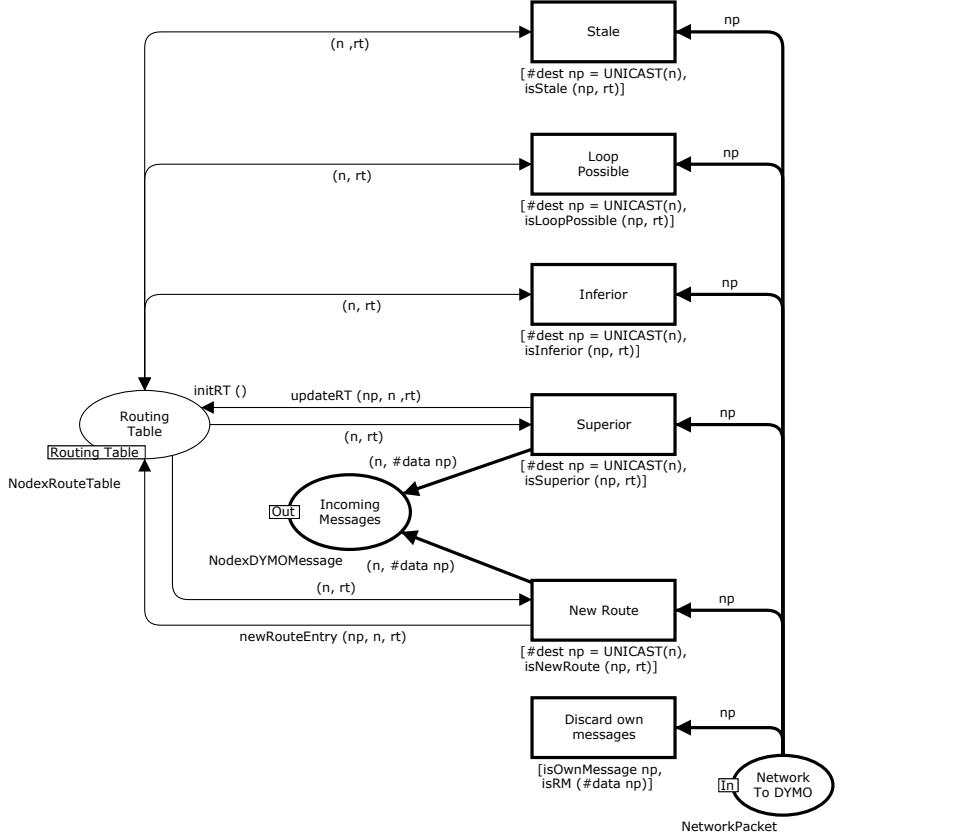


Figure 6. The Receive Routing Message module.

The first class `Stale` is routing information which is considered outdated and therefore not suitable for updating the routing table. The second class is `LoopPossible` which is routing information where using it to update the routing table may introduce routing loops. New information falls into the class `Inferior` if we already have a better route to the node. Given a network packet `np` and the routing table `rt`, the function `isInferior` returns true if and only if the routing information in the packet is inferior to that in the routing table. Section 5.2.1 in [2] specifies the following boolean expression for routing information to be inferior where `Node` is the new information and `Route` is extracted from the routing table:

```
((Node.SeqNum == Route.SeqNum) AND
((Node.Dist > Route.Dist) OR
((Node.Dist == Route.Dist) AND
(RM is RREQ) AND (Route.Broken == false))))
```

This expression is what is implemented in the function `isInferior`. The function `isInferior` used in the guard of the `Inferior` transition first checks whether the originating node is already known. If so the routing information is extracted from the routing table and compared with the information in the network packet according to the expression shown above.

The last of the four classes is `Superior` routing information. This is routing information which is considered better than the information present in the routing table and is therefore used to update the entry to the originating node using the function `updateRT`. If there is no entry to the originating node, the transition `NewRoute` is enabled and when it occurs adds a new entry is made with the function `newRouteEntry` which conforms to Sect. 5.2.2 of [2]. Network packets originating from the current node are discarded so that only network packets with superior or new routing information are passed from the `ReceiveRoutingMessage` module to the place `IncomingMessages` for further processing.

Process Incoming Messages. The module `ProcessIncomingMessages` shown in Fig. 7(left) is responsible for processing the messages that went through the update routing table module as specified by the `ReceiveIncomingMessages` module described above. The module consists of three substitution transition corresponding to the three types of DYMO messages.

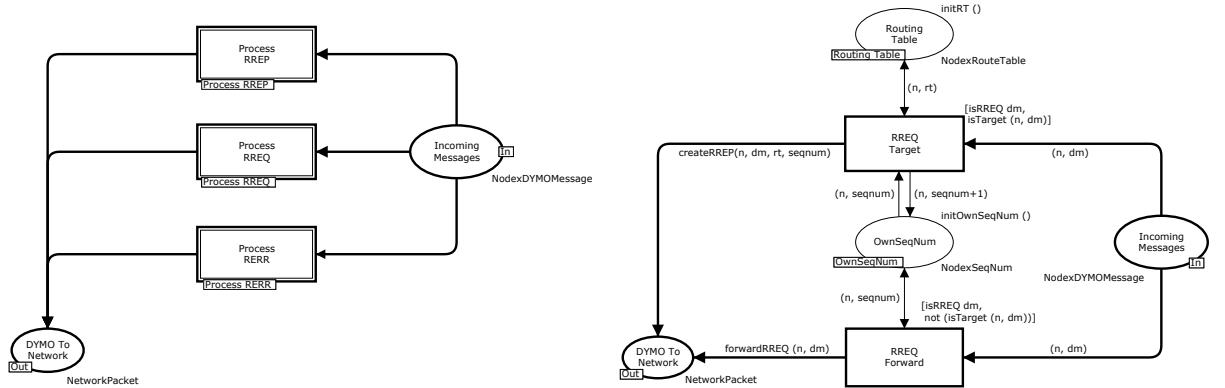


Figure 7. The `ProcessIncomingMessages` module (left) and `ProcessRREQ` module (right).

As a representative example of a submodule associated with these substitution transitions in Fig. 7, we consider the `ProcessRREQ` module shown in Fig. 7(right) which specifies the processing of RREQ messages. The submodules specifying the processing of RREP and RERR are similar.

There are basically two cases in processing a RREQ: either the receiving node is the target for the RREQ or not. If the node is not the target node, the transition `RREQForward` is enabled. The function `forwardRREQ` placed on the outgoing arc conforms to Sect. 5.3.4 of [2], and works in the following way. If the `HopLimit` is greater than or equal to one, the RREQ message has to be forwarded. This is done by creating a new network packet with `dest` set to the `LL_MANET_ROUTERS` multicast address and `src` set to the current node. The `TargetAddr`, `OrigAddr`, and `OrigSeqNum` in the message is not changed but `HopLimit` is decreased by one and `Dist` is increased by one. If the `HopLimit` is one, the function simply returns the empty multi-set, i.e., the message is discarded.

If the current node is the target of the request the transition `RREQTarget` is enabled. The function `createRREP` creates a RREP message where the `dest` field is set to the `nextHopAddress` for the originating node given in the routing table. The `src` is set to the current node, `TargetAddr` is set to the originator of the RREQ and `OrigAddr` is set to the current node.

3.2 Mobile Wireless Network

The `MobileWirelessNetwork` module shown in Fig. 8 is an abstract representation of the MANET that DYMO is designed to operate over. It consists of two parts: a part modelling the transmission of network packets represented by the substitution transition `WirelessPacketTransmission`, and a part modelling the mobility of the nodes represented by the `Mobility` substitution transition. The transmission of network packets is done relative to the current topology of the MANET which are explicitly represented via the current marking of the `Topology` place. The current topology of MANET is represented using the colour set `Topology` defined as follows:

```
colset NodeList = list Nodes;
colset Topology = product Node * NodeList;
```

The idea is that each node has an adjacency list of nodes that it can reach in one hop, i.e., its neighbouring nodes. This adjacency list is then consulted when a network packet is being transmitted from the node to determine the set of nodes that can receive the network packet. In this way, we can model a mobile network where the topology is dynamic by simply adding or removing nodes from the adjacency lists.

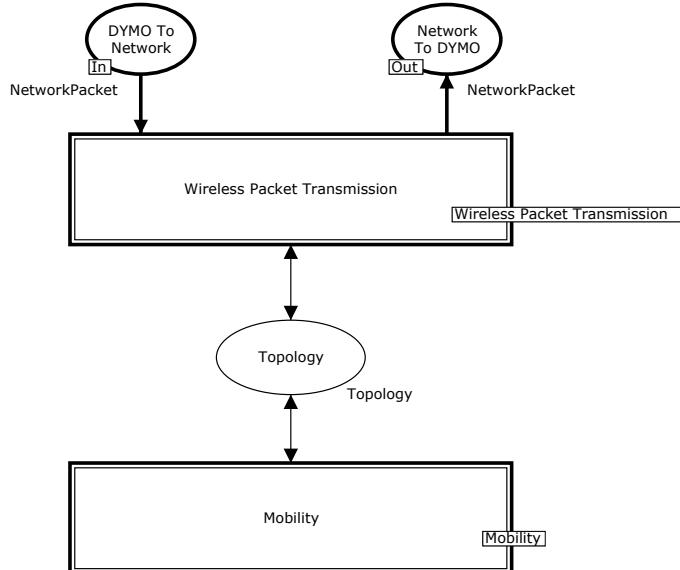


Figure 8. The Mobile Wireless Network module.

The `WirelessPacketTransmission` module models the actual transmission of packets and is shown in Fig. 9. In this module, network packets are transmitted via the physical network from one node to its neighbours. Packets are transmitted over the network using the function `transmit` defined as follows:

```

fun transmit (adjlist, {src, dest, data}, success) =
  if success then
    if (dest = LL_MANET_ROUTERS)
    then List.map (fn n => {src=src, dest=UNICAST(n), data=data}) adjlist
    else
      if (List.exists (fn d => (UNICAST(d) = dest)) adjlist)
      then 1`{src=src, dest=dest, data=data}
      else empty
  else empty;

```

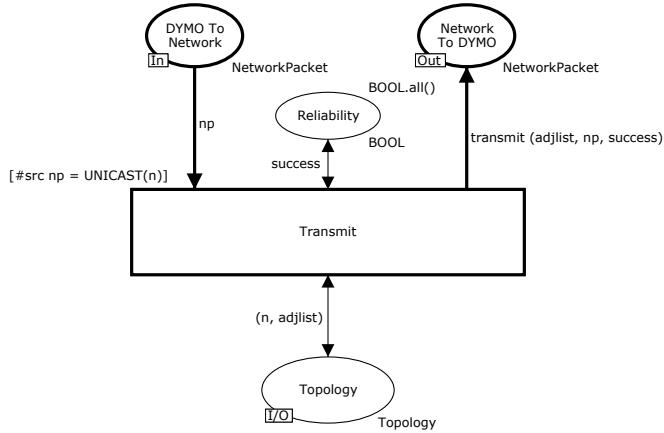


Figure 9. The Wireless Packet Transmission module.

The `transmit` function starts out by checking if the `success` argument is true and if not, the packet is discarded. This corresponds to modelling a simple unreliable network. If the packet is to be successfully sent and the destination address is the multicast address, the packet is sent to each of the nodes in the adjacency list of the transmitting node. If the destination address is a unicast address and the address exists in adjacency list of the transmitting node, i.e., the destination node is within range then the packet is forwarded. It should be noted that in reality a transmission could be received by any subset of the neighbouring nodes because of, e.g., signal interference. We only model that either all of the neighbouring nodes receives the packet or none receives it. This is sufficient because our modelling of the dynamic topology means that a node can move out of reach of the transmitting node immediately before the transmission occurs which has exactly the same effect as a signal interference in that the node does not receive the packet. Hence, signal interference and similar phenomena implying that a node does not receive a packet is in our model equivalent to the node moving out of reach of the transmitting node.

The dynamic changes in the topology of the MANET is modelled in the Mobility module shown in Fig. 10. The module consists of two transitions. The transition `AddLink` creates a link between two nodes `n1` and `n2` in the network. This is done by creating an entry in the adjacency list of the two nodes. The transition `RemoveLink` removes a link between the two nodes. This is done by removing the entry in the adjacency lists of the two nodes that specifies the link.

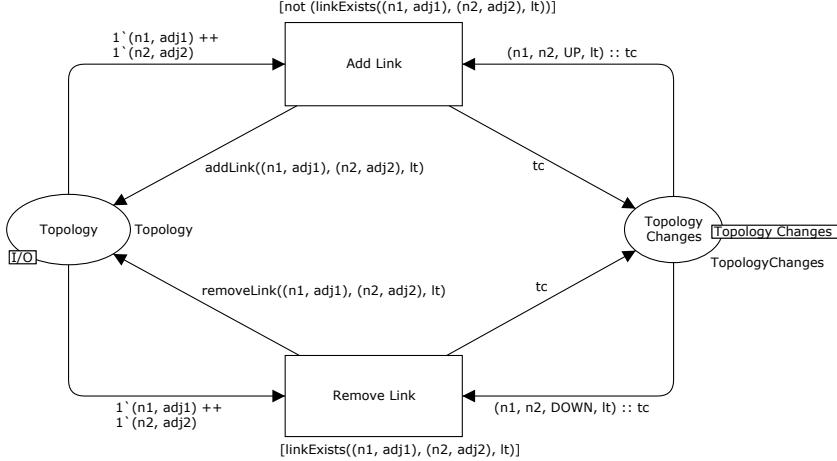


Figure 10. The Mobility module.

The marking of place **TopologyChange** is used to specify the mobility scenario considered, i.e., the sequences of link changes that are possible in the scenario. The definition of the colour set **TopologyChanges** is as follows:

```

colset LinkType = with SYMMETRIC | ASYMMETRIC;
colset LinkStatus = with UP | DOWN;

colset NodexNodexLinkStatusxLinkType = product Nodes * Nodes *
                                         LinkStatus * LinkType;
colset TopologyChanges = list NodexNodexLinkStatusxLinkType;

```

A change in the topology is modelled by the colour set **NodexNodexLinkStatusxLinkType**. The first two components in a colour of this colour set are the two nodes between which a link change is to occur. The third component specifies whether the link should go UP or DOWN. The fourth component is used to model whether it is a symmetric or asymmetric link change. The initial marking of place **TopologyChanges** then describes possible link changes that can occur, and this allows us to control the mobility scenarios. The reason for having this topology control is that there are $2^{\frac{1}{2}N^2}$ possible topologies in a MANET with N nodes and symmetric links. Hence, this will make state space analysis impossible for dynamic topologies because of state explosion. By having explicit topology control we can limit the number of possible combinations and consider different mobility scenarios one at a time. The model can capture a fully dynamic topology by placing a token on **TopologyChanges** for each pair of nodes, and we can capture the static scenario with no topology changes by not having any tokens in **TopologyChanges**.

4 Initial State Space Analysis

In this section we present our initial state space analysis conducted on the DYMO protocol model in a number of scenarios. A scenario is defined by specifying the route discoveries to be made, the initial topology, and the possible topology changes. In the initial analysis we consider only scenarios with a static topology and with symmetric links. The reason for only considering symmetric links is that DYMO requires symmetric links, and since we do

not yet have link monitoring represented in the CPN model, the protocol cannot distinguish between asymmetric and symmetric links. Furthermore, we consider a reliable network, i.e., a network that cannot lose network packets and we consider only scenarios with one route request. These scenarios allows us to generate state spaces of a reasonable size, but still focus on the correctness of the protocol. By the latter we mean that if some strange behaviour is observed in these simple scenarios, it is due to the protocol and not because of network packet loss or a change in the topology.

To initialise the CPN model according to a specific scenario, we use the symbolic constants `DYMOResquestScenario`, `topologyScenario`, and `RREQ_TRIES`. The scenario in Fig. 11(left) is represented as follows:

```
val DYMOResquestScenario = ROUTERREQUEST {originator=1, target=3}
val topologyScenario    = 1'(1, [2]) ++ 1'(2, [1, 3]) ++ 1'(3, [2]);
val RREQ_TRIES          = 1;
```

The symbolic constant `DYMOResquestScenario` is used as initial marking of the `DYMOResquest` place (see Fig. 3) and hence the above puts a single token on the place `DYMOResquest` with a request which has originator set to node 1 and target set to node 3. The symbolic constant `topologyScenario` is used as initial marking of the `Topology` place (see Fig. 8) and hence in this case we get three tokens on the place `Topology` – one for each node specifying which other nodes this node can reach in one-hop. Setting `RREQ_TRIES` to one has the effect that only a single RREQ will be sent. Since we consider only static topologies, we have no tokens on place `TopologyChanges` (see Fig. 10) in the initial marking.

When considering which scenarios to investigate, we observed that some scenarios can be considered equivalent (symmetric). As an example consider the example scenario depicted in Fig. 11 (left). The figure shows a topology with three nodes where node 1 is connected to node 2 via a symmetric link and similar for node 2 and 3. The arrow represents a request for a route discovery where the source node is the originator node and the destination node of the arrow is the target node. In this case node 1 is requesting a route to node 3. But if we permute the identity of node 1 and 3, we have exactly the same scenario as in Fig. 11 (right). We will call such scenarios equivalent and only explore one representative scenario from each such equivalence class. It is important to notice that equivalence is both with respect to topology and the originator and target of routes request. Hence, two scenarios can be considered equivalent if one can be obtained from the other by a permutation of node identities. In this way, we can reduce the number of scenarios that needs to be considered.



Figure 11. Example scenario with three nodes and one route request.

For scenarios containing two nodes we only have a single equivalence class. Looking at scenarios with three nodes we have four equivalence classes, and with four nodes we have 19 equivalence classes. In the following we consider representatives for each equivalence class in scenarios with two and three nodes. For each representative scenario, we also explore the state space when `RREQ_TRIES` is increased from 1 to 2. Additionally, we look at two representative scenarios for equivalence classes of scenarios with four nodes. All scenarios were analysed using

state space by first generating the full state space, then generating SCC-graph, and finally generating the state space report. The analysis focussed on the dead markings which represents states where the protocol has terminated.

In addition to considering the dead markings, we also investigated the properties of the protocol with respect to judging the incoming routing information against the information in the routing table. Just by looking at the boolean expressions for the four classes (inferior, superior, loop possible, and stale) it is hard to tell if it is always the case that new routing information only falls into one of the classes, i.e., the enabling of the corresponding transition in Fig. 6 is *exclusive* in the sense that only one of them is enabled for a given incoming network packet. Using the state space we want to investigate if there are cases where routing information falls into more than one class. For this purpose we implemented the following query which is explained in more detail below.

```
(* --- get network packet np from a binding element --- *)
fun getnp (Bind.Receive_Routing_Messages'Stale (1,{np,...}))      = SOME np
| getnp (Bind.Receive_Routing_Messages'Loop_Possible (1,{np,...})) = SOME np
| getnp (Bind.Receive_Routing_Messages'Inferior (1,{np,...}))     = SOME np
| getnp (Bind.Receive_Routing_Messages'Superior (1,{np,...}))     = SOME np
| getnp (Bind.Receive_Routing_Messages'New_Route (1,{np,...}))    = SOME np
| getnp _ = NONE;

(* --- get binding in node n related to network packet np --- *)
fun getBindings (n,np) =
  List.mapPartial (fn a => let
    val be = ArcToBE a
    in
    case (getnp be) of
      SOME np' => if np=np' then (SOME be) else NONE
      | NONE => NONE
    end) (OutArcs n);

(* --- check whether bindings of the five transitions in a given node n
   are exclusive for a given packet np --- *)
fun BindingsExclusive (n,np) =
  let
    val bes = getBindings (n,np)
  in
    List.length bes <= 1
  end;

(* --- check whether bindings of the five transition are exclusive --- *)
fun Exclusive n =
  let
    val nps = remDupl (Mark.Receive_Routing_Messages'Network_To_DYMO 1 n)
  in
    List.all (fn np => BindingsExclusive (n,np)) nps
  end;

(* --- search and report nodes where the transitions are not exclusive --- *)
fun CheckExclusive () = PredAllNodes (fn n => not (Exclusive n));
```

The function `CheckExclusive` uses the standard query function `PredAllNodes` to find all nodes `n` in the state space that do not satisfy the `Exclusive` predicate. The function `Exclusive` check whether the transitions are exclusive by considering each network packet

`np` on place `NetworkToDYMO` (see Fig. 6) in turn. The function `remDupl` is a utility function (not shown) which removes duplicates from a list. The function `Exclusive` uses the function `BindingExclusive` which get the enabled bindings of the five transitions in a node `n` for a given network packet `np` and check whether there is at most one such binding. This is done by means of the function `getBindings` and `getnp`.

We have evaluated the function `CheckExclusive` on all the scenarios in the Table 1. On all - except the second from bottom scenario - the function returned the empty list indicating that no marking was found in which one packet could bind in more than one of the five transitions mentioned. But in the second from bottom scenario the function returned a list of markings in which more than one of the transitions was enabled. By manual inspection we could see that it was possible for one packet to be both loop-possible and inferior. Since network packets containing such routing information is discarded this is not a problem. A subject for future work is to construct larger scenarios and try to find other situations where routing information do not fall exclusively into a single class.

Table 1 summaries the state space results for the scenarios considered. In the first column is a graphical representation of the scenario considered. The second column shows the value of `RREQ_TRIES`, and the third and fourth column list the number of nodes and arcs in the state space, respectively. The last column shows the dead markings found in the state space. In the first four scenarios, we can see that with `RREQ_TRIES` set to 1 we get two dead markings. The first dead marking is the state where `RREQ_TRIESReached` (see Fig. 5) has occurred before a `RREP` for the `RREQ` was received and the routing table updated. The second dead marking is the state where `RREQ_TRIESReached` did not occur and `RouteEstablished` occurred after the routing table had been updated. By transferring the dead marking into the simulator, we inspected the marking and observed these are desired terminal state of the protocol i.e. states where the requested route was established.

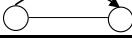
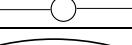
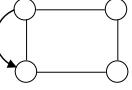
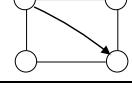
The five dead markings we get when `RREQ_TRIES` is set to 2 is caused by the same sequence of occurrences as above but here we also have overtaking on the network causing routing information to become stale and therefore a different sequence number is put in the routing table which results in a different markings.

Common for all scenarios we have listed in Table 1 is that manual inspection of the dead markings showed that the model had reached a state where the requested route had actually been established. By a route being established we mean that the originator of a request has a route entry to the target node of the request and if we follow the `NextHopAddress` hop-by-hop we can get from the originating node to the target node and vice versa. Another common result for all scenarios is that the SCC graph has the same number of nodes and arcs as the state space graph. This means that there are only trivial SSCs and this implies that there are no cycles in the state space and the protocol will therefore always terminate.

5 Discussion and Future Work

In this paper we have presented an initial CPN model of the DYMO protocol. In the construction of the model, we got familiar with the protocol and how the different mechanisms interact with one another. We have used simulation in CPN Tools to build confidence in the correctness of the model and thereby also in the correctness of the protocol. Using state space analysis we have formally verified properties of the behaviour of the protocol in some specific scenarios. Standard state space analysis was used to verify that the protocol terminates correctly. We have formally verified that in all scenarios with up to three nodes the judging of

Table 1. Summary of state space analysis results.

Scenario	RREQ	TRIES	Nodes	Arcs	Dead markings
	1		18	24	[17,18]
	2		145	307	[50,118,119,142,143]
	1		18	24	[17,18]
	2		145	307	[50,118,119,142,143]
	1		50	90	[49,50]
	2		1,260	3,875	[372,704,705,1219,1220]
	1		74	156	[73,74]
	2		2,785	10,203	[868,2435,2436,2760,2761]
	1		446	1,172	[444,443,404,403,231,...] (6)
	2		166,411	804,394	[8321,69663,69662,69136,69135,...] (23)
	1		1,098	3,976	[852,851,551,550,1096,...] (6)
	1		558	1,606	[555,556,557,558]

routing information in our subset is consistent. We have also analysed some scenarios with four nodes, and found that even with loop possible routes the judging of routing information is consistent.

In the process of constructing the CPN model and simulating in it we have discovered several issues and ambiguities in the specification. The most important ones were:

- When processing a routing message, a DYMO router may respond with a RREQ flood, i.e., a RREQ addressed to oneself, when the node is target for a RREQ message (cf. [2], Sect. 5.3.4). It is not clear which information to put in the RREQ message, i.e., the originator address, hop limit, and sequence number of the RREQ.
- When judging the usefulness of routing information, the target node is not considered. This means that a new request with a higher sequence number can make an older request for an other node stale since the sequence number in the old message is smaller than the sequence number found in the routing table.
- When creating a RREQ message the distance field in the message is set to zero. This means that for a given node n an entry in the routing table of a node n' connected directly to n may have a distance to n which is 0. Distance is a metric indicating the distance traversed before reaching n , and we believe that the distance between two directly connected nodes should be one.
- In the description of the data structure route table entry (cf. [2], Sect. 4.1) it is suggested that the address field can contain more than one node. It is not clear why this is the case.
- When processing RERR messages (cf. [2], Sect. 5.5.4) it is not specified whether hop limit shall be decremented. We believe that it should be decremented to limit the amount of network traffic.
- When retransmitting a RREQ message (cf. [2], Sect. 5.4), it is not explicitly stated whether the node sequence number is increased.
- The DYMO draft 10th revision (which is the version considering in this paper) introduced the concept of distance instead of hop count. The idea is that distance is a more general

metric, but in the routing message processing (cf. [2], Sect. 5.3.4) it is incremented by one. We believe it should be up to the implementers how much distance is incremented depending on the metric used.

As discussed in the introduction, the work presented in this paper is closely related to the work presented in [23]. The work of [23] considered the 5th revision of the DYMO specification which was the most recent at that time. The main differences between our work and the work in [23] is the following:

- [23] models a larger subset of the DYMO protocol than we do, including route maintenance procedures. The work of [23] also considers appending additional routing information on messages, but we do not model this as it is an optional part of the specification. On the other hand, we present some initial state space analysis results of the DYMO protocol in this paper, whereas [23] considers simulation analysis.
- Our modelling approach relies on structuring the CPN model into a set of modules reflecting the DYMO specification, whereas [23] present a highly compact CPN model containing only a single module. A main reason for our choice of a more verbose modelling is that we aim at using the CPN model as a basis for implementation in which case we need a CPN model which is easier to gradually refine. The work of [23] appears to aim at state space analysis, which is why it is beneficial to have a compact CPN model to reduce the effect of state explosion. This is an often encountered trade-off between state space size and compactness of the CPN model.
- A main difference is also in the modelling of the MANET topology. Both papers consider a dynamic topology, but we have the current topology explicitly represented in the marking and have explicit control of the possible topology changes that can occur. The main motivation for this is to make a scenario based state space analysis possible as illustrated in this paper. The approach of [23] relies on an abstract and implicit modelling of the topology where a transmitted network packet is either received by a single node or by no nodes. As the authors state in the conclusion of [23], they should extent this such that any number of nodes can receive a transmitted message. It still remains to be investigated which of the two approaches to modelling the dynamic topology is most appropriate for state space analysis in practice, and perhaps a combination of the two approaches can be developed.

Near future work is to complete the modelling of the DYMO specification such that it covers all non-optional parts of the specification. The most important parts not currently modelled are the timeout mechanisms and the link monitoring to detect broken links. With this included in the CPN model, we can extend our state space analysis to also include route error processing, route maintenance, and dynamic topologies. For the state space analysis, we plan to continue work on the idea of exploiting symmetry in the scenarios to reduce the number of cases that needs to be considered. We hope to be able to apply this approach also for dynamic topologies. This work will include the application of advanced state space methods in order to alleviate the impact of the state explosion problem and extending the set of behavioural properties considered in the analysis.

The next step will then be refine the CPN model such that the interaction between the IP network layer and the DYMO protocol module becomes explicit. Currently, in its only modelled in a very simple way with request and response related to route discovery. In a real implementation interaction between the DYMO module and the IP network layer module is

also required to, e.g., update the IP routing table. This refinement of the CPN model will make the interaction between DYMO and its environment explicit which is required in order to use the CPN model as a basis for implementing the DYMO protocol which is the long term goal of the project.

Acknowledgements. The authors wish to thank the anonymous reviewers for their constructive comments and suggestions that have helped us to improve the CPN model and paper.

References

1. K. Bhargavan, D. Obradovic, and C.A. Gunter. Formal Verification of Standards for Distance Vector Routing Protocols. *Journal of the ACM*, 49(4):538–576, 2002.
2. I.D. Chakeres and C.E. Perkins. Dynamic MANET On-demand (DYMO) Routing. <http://www.ietf.org/internet-drafts/draft-ietf-manet-dymo-10.txt>, July 2007. Internet-Draft. Work in Progress.
3. R. de Renesse and A.H. Aghvami. Formal Verification of Ad-Hoc Routing Protocols using SPIN Model Checker. In *Proc. of IEEE MELECON*, pages 1177–1182, 2005.
4. T. Clausen et. al. Generalized MANET Packet/Message Format. Internet-draft, 2007. Work in Progress.
5. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment fo Higher Order Logic*. Combridge University Press, 1993.
6. S. Hansen. Modelling and Validation of the Dynamic On-Demand Routing (DYMO) Protocol. Master’s thesis, Department of Computer Science, University of Aarhus, February 2007. (in danish).
7. G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.
8. K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3-4):213–254, 2007.
9. Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets Modelling and Validation of Concurrent Systems (draft manuscript)*. Springer, 2007.
10. D. Johnson, Y. Hu, and D. Maltz. The Dynamic Source Routing Protocol (DSR). RFC 4728, 2007.
11. IETF MANET Working Group. <http://www.ietf.org/html.charters/manet-charter.html>.
12. IETF Mobile Ad-hoc Networks Discussion Archive. <http://www1.ietf.org/mail-archive/web/manet/current/index.html>.
13. Wibling O, J. Parrow, and A. Pears. Automatized Verification of Ad Hoc Routing Protocols. In *Proc. of FORTE*, volume 3235 of *LNCS*, pages 343–358. Springer-Verlag, 2004.
14. C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On Demand Distance Vector (AODV) Routing. RFC 3561, 2003.
15. C. E. Perkins. *Ad Hoc Networking*. Addison-Wesley, 2001.
16. K. G. Larsen P. Petterson and W. Yi. UppAal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1+2):134–152, 1997.
17. E.M. Royer and C.-K. Toh. A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks. *IEEE Personal Communication*, pages 46 – 55, April 1999.
18. R. Thorup. Implementation and Evaluation of the Dynamic On-Demand Routing (DYMO) Protocol. Master’s thesis, Department of Computer Science, University of Aarhus, February 2007.
19. R. Thouvenin. Implementation of the Dynamic MANET On-Demand Routing Protocol on the TinyOS Platform. Master’s thesis, Department of Computer Science, University of Aarhus, July 2007.
20. C. Xiong, T. Murata, and J. Leigh. An Approach to Verifying Routing Protocols in Mobile Ad Hoc Networks Using Petri Nets. In *Proceedings. of IEEE 6th CAS Symposium on Emerging Technologies: Frontiers of Mobile and Wireless Communication*, pages 537–540, 2004.
21. C. Xiong, T. Murata, and J. Tsai. Modeling and Simulation of Routing Protocol for Mobile Ad Hoc networks Using Colored Petri Nets. In *Research and Practice in Information Technology*, volume 12, pages 145–153. Australian Computer Society, 2002.
22. C. Yuan and J. Billington. An Abstract Model of Routing in Mobile Ad Hoc Networks. In *Proc. of CPN’05*, pages 137–156. DAIMI PB-576, 2005.
23. C. Yuan and J. Billington. A Coloured Petri Net Model of the Dynamic MANET On-demand Routing Protocol. In *Proc. of CPN’06*, pages 37–56. DAIMI PB-579, 2006.