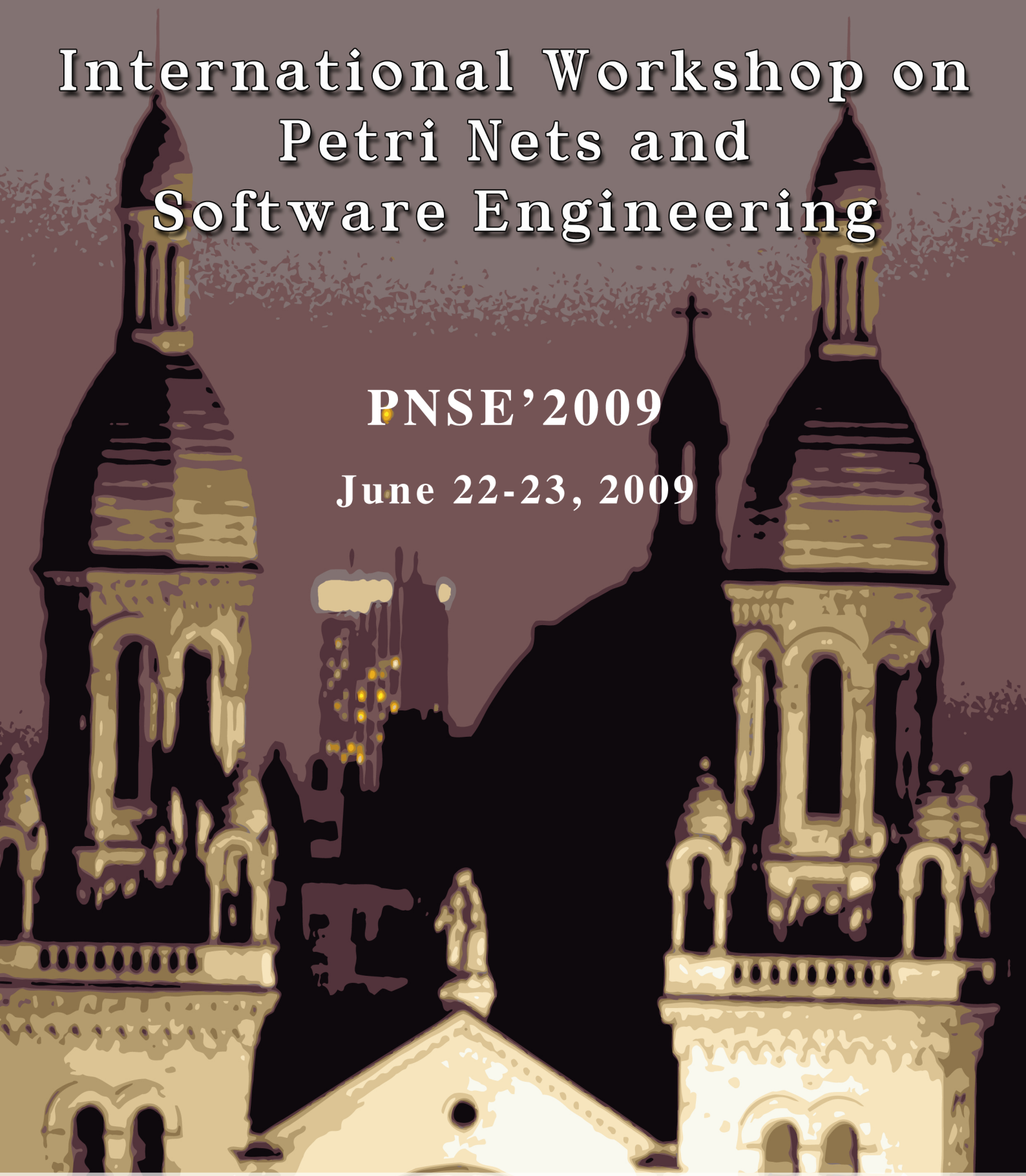
International Workshop on Petri Nets and Software Engineering

PNSE'2009

June 22-23, 2009



Sponsored by:

A satellite event of
Petri Nets'2009
June 22-26, 2009
Paris - France

Preface

This booklet contains the proceedings of the International Workshop on *Petri Nets and Software Engineering* (PNSE'09) in Paris, France, June 22-23, 2009. It is a satellite event of *Petri Nets 2009*, the 30th International Conference on Application and Theory of Petri Nets and Other Models of Concurrency.

More information about the workshop, like online-proceedings, can be found at

`http://www.informatik.uni-hamburg.de/TGI/events/pnse09/`

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

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

The intention of this workshop is to bring together research and application to have a lively mutual exchange of ideas, view points, knowledge, and experience. The submitted papers were evaluated by a programme committee, which was supported by several other international experts resulting in *four* reviews per submitted paper.

The program committee consists of:

Wil van der Aalst (The Netherlands)
Peter Buchholz (Germany)
Didier Buchs (Switzerland)
Piotr Chrzastowski-Wachtel (Poland)
Gianfranco Ciardo (USA)
José-Manuel Colom (Spain)
Raymond Devillers (Belgium)
Zhenhua Duan (China)
Berndt Farwer (United Kingdom)
Jorge C.A. de Figueiredo (Brasil)
Giuliana Franceschinis (Italy)
Luís Gomes (Portugal)
Nicolas Guelfi (Luxembourg)
Guy Gallasch (Australia)
Serge Haddad (France)
Xudong He (USA)
Kees van Hee (The Netherlands)
Monika Heiner (Germany)
Vladimír Janousek (Czech Republic)
Gabriel Juhás (Slovak Republic)
Astrid Kiehn (India)
Ekkart Kindler (Denmark)
Michael Köhler-Bußmeier (Germany)
Gabriele Kotsis (Austria)
Maciej Koutny (UK)
Lars Kristensen (Denmark)
Xiaoou Li Zhang (Mexico)
Johan Lilius (Finland)
Robert Lorenz (Germany)
Daniel Moldt (Germany) (Chair)
Chun Ouyang (Australia)
Wojciech Penczek (Poland)
Laure Petrucci (France)
Heiko Rölke (Germany)
Sol M. Shatz (USA)
Christophe Sibertin-Blanc (France)
Natalia Sidorova (The Netherlands)
Mark-Oliver Stehr (USA)
Harald Störrle (Austria)
Ferucio Laurentiu Tiplea (Rumania)
Ulrich Ultes-Nitsche (Switzerland)
Karsten Wolf (Germany)
Jianli Xu (Finland)
Christian Zirpins (Germany)
Mengchu Zhou (USA)
Wlodek M. Zuberek (Canada)

We received 22 high-quality contributions. The program committee has accepted eight of them for full presentation. Furthermore the committee accepted nine papers as short presentations. At the common poster session with the workshop on Organizational Modeling (OrgMod'09) one poster is presented. The poster session is in addition open to all participants of the whole conference events to present ongoing work and current / future projects without written submissions.

Daniel Moldt

Acknowledgments

The international program committee was supported by the valued work of additional reviewers, listed below:

Florencia Balbastro
Lawrence Cabac
Alfredo Capozucca
Raymond Devillers
Michael Duvigneau
Steve Hostettler
Zhaoxia Hu
Mariusz Jarocki
Fei Liu
Niels Lohmann
Levi Lucio
Olivia Oanea
Martin Schwarick
Marc Voorhoeve
Jan Martijn van der Werf
Matthias Wester-Ebbinghaus

Their work is highly appreciated.

Furthermore, we would like to thank the organizational team of the whole conference setting in Paris for their general support.

The organzational/technical work in Hamburg was supported by Yvonne Küstermann and Dimitri Popov.

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

Contents

Invited Talk

A Petri Net-based Conceptual Meta-model for System Modeling

Christophe Sibertin-Blanc

Université de Toulouse, IRIT
2, rue du Doyen Gabriel Marty, 31042 Toulouse cedex
sibertin@irit.fr

Abstract. The structure of a system may be described with three sorts of related elements: *operations* the performances of which make the system go from a consistent state to another one; active *actors*, or processors, that perform operations; passive *entities* that are involved in operation occurrences. The association of a processor, an operation and a list of entities defines an *action*. Each of these elements may be hierarchically refined and are subject to the type-instance distinction. On the basis of a so defined structure, the behavior of a system is defined by a *control structure* that determines the possibility for an *act* – an action occurrence - to occur.

" *Un système est une unité globale organisée d'interrelations entre des éléments, actions ou individus*[1] ",

Introduction

Thanks to their numerous valuable features, Petri nets are a formalism very well suited to model the behavior of complex, and thus concurrent, systems. However, one may think that they are not used as widely as they could be and, in any case, behavior modeling is a difficult task. This state of affairs is perhaps due to the fact that modeling the behavior of a system requires beforehand to have a model of the structure of the system under consideration: what are its constituting elements and how they are related. To identify and define these elements and relations, a meta-model of the structure of systems is needed.

The concept of system is used in most disciplines, and a review of the huge literature, from (Bertalanffy, 1950) to e.g. (Snooks, 2008) is out of the question. Regarding software systems, meta-modeling is a matter of Model Engineering (Bezivin 2005) in the line of the MOF (Meta Object Facility, (OMG, 2006)) and the UML (OMG, 2004). However, despite its 13 kinds of diagrams and the 1200 pages of its definition, the UML does not propose a model of what is a software system.

[1] *« A system is a global unity organized of interrelations between elements, actions or individuals »* (Morin 1977, p. 102).

In this paper, we propose a meta-model of the structure of systems together with a way to describe their control structure by means of Petri nets; it features the following main properties:

- It is conceptual in as much as it is intended to be used by modelers as well as by domain experts which possess the theoretical or empirical knowledge of the system to be modeled; it can be the language that they need to share in order that the domain expert transfers its knowledge to the modeler and is able to validate the model elaborated by the later.
- It is conceptual also because it is compatible with most Domain Specific Languages (Mernik &al., 2005) that can be used to describe the constitutive elements of a system; it can even be used to structure the description of a system that is expressed in natural language.
- It allows to structure the model of a system. Indeed, a model of a non-trivial system includes a big number of diagrams (or, more generally, of model units) that are related in many ways: a model of a system itself is a system. The effective understanding of each diagram necessitates to know its matter, status and context, that is how it is related to other diagrams, and this requires the model to be organized in a knowledgeable and tractable way.

Briefly speaking (see figure 1), a system includes *Entities*, abstract or concretes, that it handles, processes, exchanges with its environment or uses as resources. An Entity is a passive object mainly composed of attributes and services (or methods). A system also includes *Actors*, active objects that have the same constitutive elements as Entities and in addition consume some energy to store Entities and to perform Operations while carrying their *activity* out. Finally, the structure of a system includes *Operations*, the work units that make the system to change from a coherent state to another one. The *state* of a system is given by the state of all its constitutive elements. All these elements are linked by many *relations*. Entities, actors and operations support the type/instance distinction, that is the actualization in time and space of their definition. They also support the refinement process, which is their breaking into elements of the same nature.

The elements of the structure of a system allow to define its set of *actions*, that is the performance by an actor of an operation involving some entities. The type/instance distinctions may also be applied to actions (the instance, or occurrence of an action is an *act*), and a control structure may be used to break an action into smaller actions. Defining the *Behavior* of a system consists in describing in which cases an action may or must occur and the effect of this occurrence or, in an equivalent way, which are the actions that may occur from a given state.

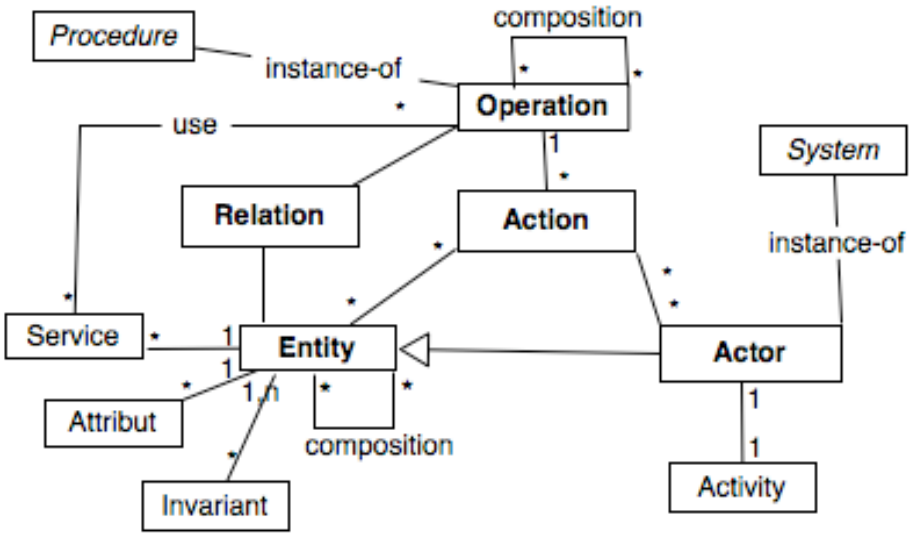


Fig. 1. The meta-model of the structure of a system

2 The structure of a system

All the three sorts of elements, entities, actors and operations, are necessary for a system to feature some activity and be a device useful for its environment. Without entities it provides nothing, without operations there is nothing to be performed, and without actors it is dead, it has no possibility to performed something.

2.1 The Entities

Any system handles some things, items, and objects that we call entities. Some of these entities are permanent and stay in the system as long as the system itself. Others are transient, because they are received from or sent to the system's environment, or because they are created or deleted by the system. In fact, entities may be considered as passive objects in the meaning of the Object-Oriented approach. An entity can be *identified* and distinguished from others. It features a set of *attributes* that have either a constant or a variable value; the *state* of an entity is defined as the value of all its (variable) attributes. To define which are the feasible or consistent states of an entity, it may be associated with some *local state invariants*, that is predicates that must be satisfied by the values of its attributes in any case. An entity also provides some *services*, or methods, that can be invoked by other system's elements. Some services do not change the state of the entities, they are *query-services* that only return an information about the entity; others have to respect the entity's state invariants: if a service is invoked when the entity is in a consistent state, then the entity is still in a consistent state after the execution. Finally, an entity may be associated with some

behavior invariants that define how it can be used, that is in which cases its services may be invoked; the figure 2 shows what could be the behavior invariant of a "gate" entity.

Entities may be linked the ones with the others, and also with actors and operations, by many kinds of relations. This leads to consider *global state invariants*, that is predicates that must be satisfied by the values of the attributes of linked entities; for instance, the date of an invoice must be after the date of the corresponding order.

It is clear that an entity, or an entity type, can be described with the UML notation. However, it is also possible to use any other language more appropriate to the specificities of the considered entities. The above definition just indicates what knowledge it is relevant to capture about an entity, not how this knowledge has to be expressed.

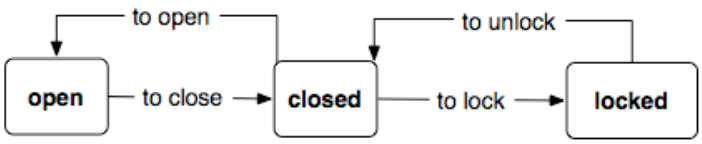


Fig. 2. The behavior invariant of a *gate* entity

2.2 The actors

While the entities are the passive elements that are processed by the system, the *actors* are the active elements, the processors or processing units that perform the work done by the system. Each actor is an engine, a device, in fact a sub-system that consumes some energy to produce some work[2]. Most entities needs to be hosted somewhere, and actors care for them. In any case, an invocation of a service of an entity is performed by an actor that uses the service as a tool. With regard to operations, actors select the ones to be done and carry their occurrence out.

As for entities, some actors are permanent while others are transient. It may be the case that an actor is involved in an operation performed by another actor; as a consequence, all the items that are relevant in the definition of entities are also relevant for actors: an actor *is-a* entity. However, each actor features an additional element, its *activity* that we will introduce in section 3.5.

[2] In many cases there are several ways to design the model of a system, even at a specified level of detail, according to the purpose of the model. Accordingly, there are several ways to identify the actors of a system. The energy consumption criterion does not prevent e. g. to consider that a process running on a computer is an actor that consumes the « computational energy » provided by the processor that consumes the electrical power energy. Where and how energy is consumed still is a modeling choice, the unavoidable fact being that any work requires some energy.

2.3 The operations

While services are the atomic[3] working units at the entity level (as they preserve their consistency), operations are the atomic working units at the level of the whole system and they have to respect all the state and behavior invariants: if the execution of an operation starts when the system is in a consistent state, then the system is still in a consistent state at the completion of this execution (the reader familiar with databases will recognize the similarity with the concept of *transaction* (Gray, 1981)). An operation is defined as a set of entity's service invocations that are gathered according to some control structure, and this control structure may be defined in anyway.

The instances of an operation are its performances, or occurrences. Operations instances are evanescent elements, they have a bounded span life while entities instances last for an undefined period of time. However, we claim that operations are first class elements of the structure of a system (compare the attributes and services of an entity with the entities and operations of the system). The control structure that gathers services into a consistent operation is a steady component of a system.

2.4 The hierarchical breakdown

The model of a system may be drawn at different level of abstraction. At the top level, the model of a system includes a single actor, the system itself; the entities are the ones exchanged with its environment that is other systems with which the one under consideration interact; as for the operations, they are the *procedures* offered by the system to its environment. What about procedures? We are considering systems that have some *raison d'etre*; they are artifacts designed with the purpose to have some utility and thus to provide services to their environment. An offered procedure is launched at the request of the system's environment and it ends when the request has been completely processed (compare with the *use case* concept of the UML). A procedure is a (set of) sequence of treatments that is closed for the causality relation. Besides offered procedures, a system also includes *homeostatic procedures* that are intended to maintain its internal equilibrium and may be viewed as the support procedures necessary for the offered ones to work properly. One may also consider the *require procedures* of a system, the ones that are offered by the systems of this environment.

This top-level model may be decomposed into a lower level one. The system-actor is breakdown into sub-system actors, the procedures are breakdown into operations linked by control structures (some services of the system-actor may become operations), entities exchanged with the environment may be also breakdown into smaller ones and new entities reveal to be considered to describe the operation

---

[3] Atomicity holds in the case of discrete event systems and avoids to consider the state of a service: the state of the system is undefined during the course of the execution of a service. This is the case considered in this paper, but the meta-model could be extended to consider continuous and hybrid systems (David & Alla, 2005).

processing; and so recursively. This is nothing else than the well-known refinement process advocated by the functional approaches, e.g. (SADT, 1993).

3. The behavior of a system

Entities, actors and operations are the necessary and sufficient elements for the system to feature some behavior: they define the space of its potential consistent states and how it can move from one state to another one under the effect of an operation occurrence. Defining the behavior of the system is defining how it operates, that is its possible trajectories within this space of potential states and the states that can be actually reached from an initial state. To this end, the behavior of a system must determine in which cases an operation may or must occur and the effect of this occurrence or, in an equivalent way, which are the operations that may occur from a given state.

3.1 The actions

Any operation occurrence requires some energy, and thus an actor that supplies it, and most often it is applied to entities that have to be processed. This leads to consider the concept of *action* defined as the gathering of one operation, one (or several) actor that performs the operation and a list of entities that are involved in this operation performance (remind that actors are entities, so if an operation invokes a service of an actor, this actor acts as an entity of the action, be it the performer of the action or not). An action may expressed by a sentence where the subject refers to the actor, the verb to the operation and the complements to entities. Thus the structure of a system determines the set of its potential states and it also allows to establish the set of its actions.

Since entities, actors and operations support the type / instance distinction, the same holds for actions: the type of an action is defined by an actor type, an operation type and a list of entity types, while an occurrence of an action, an *act,* is defined by instances of these types. The hierarchical breakdown process may also be applied to actions, according to the decomposition of the operation into several ones.

3.2 Organizing the model of the behavior

Since operations always occur in the context of an action, to model the behavior of a system we have to define in which cases any act may or must take place and the effect of this occurrence.

Due to the number of entities, actors and operation of a system, the definition of its whole behavior yields in a very large model that would be difficultly tractable and overcome the cognitive capacities of the modeler (and thus its validation). So, there is a need to organize the model of the behavior of a system into smaller syntactical units that, if possible, are also semantic units that can be executed and convey some knowledge about the whole system behavior. These units may be determined in

consideration of the structure of the system that is from the points of view of entities, actors and operations.

Now, arises the question of the coordination of these behavior units into a coherent and global model of the whole system's behavior that is how their respective executions are synchronized. Petri nets are very well suited to this end, in addition to their numerous qualities for dealing with complex control structures such as a graphical representation, the possibility to reason about both the states and the operations, a well-defined semantics allowing formal analysis, or the possibility of making simulation or generating code. Indeed, *synchronous* interactions between two nets occur thanks to the fusion of two transitions while an *asynchronous* interactions occur thanks to a communication place that is an *result place* of one net and an *entry place* of the other[4].

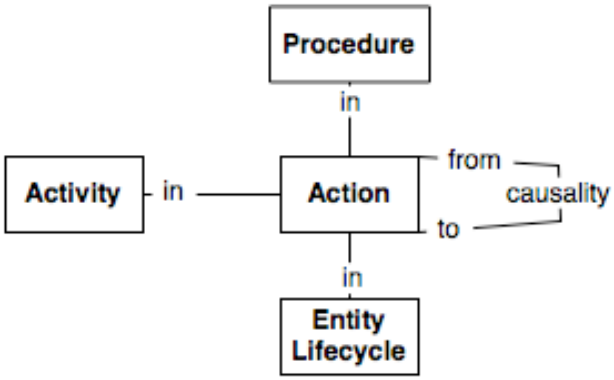


Fig. 3. The different ways to organize the actions of a system

3.3 Which Petri net dialect?

There is a number of Petri net-based formalisms, dedicated to specificities of the behavioral properties of the system under consideration. In our case, we need a formalism allowing to refer to the structure of the system, that is its entities, actors and operations. For this reason, we propose to use the *Petri Net with Objects* formalism (PNO, (Sibertin 1985, Sibertin 2000a), see table 1). There is an implementation of this formalism that supports code generation (Sibertin & al., 1995) and it could be extended to support e.g. stochastic (Ajmone Marsan & al., 1995) or hybrid (David & Alla, 2005) features. In a PNO model of the behavior of a system, places are typed by entity or actor types while transitions are labeled with operations. Thus an action is modeled by a transition labeled by the operation and surrounded by places typed by the corresponding entities and actors, or by several such patterns if the action may occur from states corresponding to different locations of the entity or actor tokens.

[4] Another way to define the interactions between behavioral units is to consider *data flows* and *control flows*. While the former is represented as an asynchronous interaction by token sending, the latter requires a (simple) pattern.

Table 1. The Petri Net with Object formalism

Petri Nets with Objects (**PNO**) are a High-Level Petri Net formalism (Genrich & al. 1981, Reizig 1985, Sibertin 1985, Jensen 1987) allowing to handle tokens that are (lists of) objects.

The type of a *Place* (written in italic characters) is a (list of) type defined in some Object Oriented language, referred to as the *data language*, and the default type of a place is the empty list. The value of a place, or *marking* is the set of tokens it contains. At any moment, the state of the net (or its marking) is defined by the distribution of tokens into places. Objects are accessed by reference that is a *token* that is a reference toward an object matching the type of the place. More generally, a token may be (a list of) either a constant (e. g. 2 or 'Hello'), an instance of a class of the data language, or a reference toward such an instance. Each place typed by an object class corresponds to a possible state for the instances of this class.

A *Transition* is connected to places by oriented arcs as it aims at changing the net state, that is the location and value of tokens.

Each *arc* is labeled with a (list of) variable having the same type that the place it is connected to. The variables of the arc connected to a transition serve as *formal parameters* of that transition and define the flow of objects from input places to output places. A transition *may occur* (or is *enabled)* if there exists a *binding* of its input variables with tokens lying in its input places. The *occurrence* (or the *firing)* of an enabled transition changes the marking of its surrounding places: tokens bound to input variables are removed from input places, and tokens are put into output places according to variables labeling output arcs. Two transitions may occur concurrently if their bindings capture different object references.

A transition may be guarded by a *Precondition*, a side-effect free Boolean expression involving input variables. In this case, the transition is enabled by only if the binding evaluates the Precondition to true. For any transition, the location of objects and the input variables determine the existence of a binding, the values of the bound objects determine the satisfaction of the Precondition.

A transition may also include an *Action* which consists in a piece of code of the data language involving the transition's variables, or a label referring to such a piece of code. This Action is executed at each occurrence of the transition and it allows to process the values of tokens. If an output variable of the transition does not appear on any input arc, the Action must create a new object instance to assign a value to this variable and extend the binding which enabled the transition. Conversely, if an input variable does not appear on any output arc, each occurrence of the transition entails the loss of the reference toward the bound object.

Finally, a transition may be equipped with some *emission rules*, that is side-effect free Boolean expressions involving transition's variables, the satisfaction of which determines the flow of tokens toward the output places (emission rules favor the conciseness of nets but they not extend the expressive power of PNOs).

3.4 Operation based modeling

We assume that each act performed by the system has some utility and thus it appears in the course of the execution of some procedure. Then, if we describe all the procedure of a system and how they interact – by the sending of a (copy of an) entity from a procedure to other ones or by the joining of the transitions of an action common to several procedures -, we have described the whole behavior of the system. A procedure is described by a PNO whose the locations and values of tokens at its initial marking depend on the initial state of the system.

An offered procedure begins either with an entry place in which the environment can put request-tokens or with an initial transition (without input place) and in this case the environment can launch directly executions of the procedure (whenever an appropriate system's actor is available to perform the action), and we may consider that it ends with a terminal place. To distinguish the different occurrences, or executions of a procedure, a new procedure-identifier is generated for each occurrence of the procedure and at least one place is typed by this procedure-identifier on any path from the beginning to the end of the procedure. The possibility to complete each execution of a procedure corresponds to the fact that the terminal place of its PNO model can receive as many tokens as the number of its executions. The net of a procedure includes action-transitions that are connected by places according to the *causality relationship* (the occurrence of one or several actions enables the occurrence of other actions). It may be also convenient to include in the net of a procedure query-transitions that invoke query-services of entities or actors, in case the selection of an alternative among several ones requires some computation, although such selections may be ensured by preconditions of transitions.

The net of an homeostatic procedure is quite different because executions of an homeostatic procedure are not performed upon request of the environment but upon the occurrence of an internal event produced by some procedure(s). Thus the net of an homeostatic procedure must include an *home state* which corresponds to the beginning and the end of each of its executions.

Once the net of a procedure is drawn, we can focus upon the specific contribution of an actor by keeping only the places typed by this actor (as the performer of the action) and avoiding all others. The resulting net describes the *role* of this actor with regard to this procedure.

3.5 Actor based modeling

Each action is performed by (at least) an actor and, as for procedures, if we describe the behavior of each actor, i. e. the sequences of actions that it may perform that we call its *activity*, and how these activities interact, we have described the whole behavior of the system. For the permanent actors, that have the same lifetime as the system, we can have either one net for each actor or one net for each actor type where the types of the places that need to distinguish the concerned actor are stamped by actor identity. The activity net of a actor includes action-transitions and query-transitions in the same way as procedure nets. For a permanent actor, the net includes

a home state, as for homeostatic procedures, while its beginning and end are different for transient actors, as for offered procedures.

If we tag each transition by the procedure in which the corresponding action takes place and keep only the transitions tagged by a procedure, we again obtain the role of this actor within this procedure. Thus, the activity of each actor may be derived from the models of the procedures by gathering its roles with regard to all the procedures and, if needed, synchronizing this partial nets. Conversely, the nets of the procedures may be derived from the actor activities by gathering and synchronizing the roles of all the actors involved in each procedure.

3.6 Entity based modeling

We may consider that each action causes some state change, and then involves some entities. Thus, as for procedures and actors, if we describe the *lifecycle* of each entity (let us remind that an actor can act as an entity in an action and thus it also has a lifecycle), that is all the sequences of operations that invoke its services, and how these lifecycles interact, we have described the whole behavior of the system.

The entity-based model of the behavior of a system is very similar to the one of actors, thus we do not elaborate on this point. The contribution of an entity to a procedure can be derived in the same way as the role of an actor, so that the relationships of the actor-based approach with the procedure-based one also holds for the entity-based approach.

From the lifecycle of an entity, it is possible to keep, in the operations of transitions, the invocations of its services and so to verify that its behavioral invariants are respected.

Conclusion

Let us come back to the epigraph of this paper: *Un système est une unité globale organisée d'interrelations entre des éléments, actions ou individus*. The global unity aspect of a system corresponds to the fact that, among the entities, actors and operations lying in the universe, some are within the system under consideration while others are not. A system is separated from its environment by its border. The elements, actions and individuals are respectively what we have called entities, operations and actors, while the behavior defines the organization of their interrelations.

In any engineering discipline, the structure of the product to be done shapes the organization of the work to be performed. As a consequence, this meta-model may serve as a basis to define some methodologies such as the one proposed in (Sibertin 2000b) in the case of information systems.

Reference

Ajmone Marsan, M., G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley, 1995.

David, R., Alla, H. (2005). *Discrete, Continuous, and Hybrid Petri Nets*, Springer-Verlag.

Bertalanffy, L. von. (1950). "An Outline of General Systems Theory." *British Journal for the Philosophy of Science*, Vol. 1, No. 2.

Bezivin, J. (2005) On the Unification Power of Models, *Software and System Modeling*, 4, 2, p. 171 – 188, may 2005.

Genrich, H., Lautenbach K. (1981). System modelling with High Level Petri Nets. *Theo-retical Computer Science* 13, North-Holland, 1981.

Jensen K. (1987). Coloured Petri Nets. In *Petri Nets: Applications and Relationships to Other Models of Concurrency Part I*, W. Brauer, W. Reisig and G. Rozenberg Eds, Lecture Notes in Computer Science Vol. 254, Springer-Verlag, 1987.

Gray, Jim (1981). The Transaction Concept: Virtues and Limitations. *Proceedings of the 7th International Conference on Very Large Databases*, p. 144–154, Cupertino CA.

OMG (2004). *OMG Unified Modeling Language Specification and Infrastructure*, UML 2.0, 2004/8/01.

OMG (2006). Meta Object Facility (MOF) Core Specification, Version 2.0, 06-01-01.

Marjan Mernik, Jan Heering, and Anthony M. Sloane (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344.

Morin, E. (1977). La Méthode, T.1, *La Nature de La Nature*, Paris, Seuil.

Murata T. (1989) Petri Nets: Properties, Analysis and Applications; In *Proc. of the IEEE*, vol 77, n°84, April 1989.

Reisig, W. (1985). Petri Nets with Individual Tokens. *Theoretical Computer Science* 41, 185-213, North-Holland, 1985.

SADT (1993). Integration definition for function modelling ; *Draft Federal Information Processing Standards Publication 183*, 1993 December 21

Snooks, G.D. (2008). "A general theory of complex living systems: Exploring the demand side of dynamics", *Complexity,13: 12-20*

Sibertin-Blanc, C. (1985). High Level Petri Nets with Data Structure. In Proceedings of the 6th European *Workshop on Application and Theory of Petri Nets*, Espoo (Finlande); K. Jensen Ed., pages 141 - 170, June 1985.

SIBERTIN-BLANC C. (2000a) CoOperative Objects: Principles, Use and Implementation. In *Petri Nets and Object Orientation*, G. Agha & F. De Cindio (Eds)., Lectures Notes in Computer Science vol 2001, Springer-Verlag, 2000.

SIBERTIN-BLANC C. (2000b) Using Petri Nets and Objects: A Formal yet Expressive Approach. In *Software Specification Methods: An Overview Using a Case Study*. Henry Habrias, Marc Frappier (Eds.), Springer-Verlag, p. 259-278, oct. 2000.

Sibertin-Blanc, C., Hameurlain, N. Touzeau P. (1995). SYROCO: A C++ implementation of CoOperative Objects; *Proceedings of the Workshop on Object-Oriented Programming and Models of Concurrency*; Turino (I), June 27 1995.

Full Presentations

Net Components: Concepts, Tool, Praxis

Lawrence Cabac

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

Abstract. The design of complex Petri net models is a challenge with respect to intuition, correctness and efficiency. In the case that systems' models are of large scale and consist of numerous nets, as in the case of Petri net-based software engineering, net components – an implementation of a pattern-based approach – are applied successfully. This paper summarizes our efforts made and experiences with net components. It illustrates the concept, design and usage of net components. It additionally presents the experiences gained from several software development projects over the last few years.
Net components are subnets that are combined with each other to form a Petri net. By this component-based approach of constructing Petri nets, the construction of nets is facilitated as well as accelerated, and the Petri nets are structured in an intuitive, easily readable and unified way. To illustrate the presented concepts, some net components that are frequently used in an agent-oriented approach in the context of Mulan are introduced. Besides the fact that they provide the basic functionality in regard to the communication that is done by protocol nets, also some simple patterns and programming artifacts are introduced into the development process.

Keywords: components, net components, design patterns, workflow patterns, Petri nets, Renew, structured Petri nets, Petri net-based software engineering

1 Introduction

High-level Petri net formalisms such as colored Petri nets [9] or reference nets [13] offer modeling constructs and abstractions comparable to basic programming constructs of high-level programming languages: data types and variables, sequences, branches and iterations and code encapsulation with restricted access. Additionally, Petri nets allow for an elegant and intuitive modeling of concurrency, which is neglected in most widespread programming languages. Therefore, it is possible to use an appropriate Petri net formalism not only for modeling and analyzing of systems – as it is usually done – but also for implementation. This has the additional advantage that the model can be transformed into an implementation without a change of formalism. Thus the gap between model and implementation can be closed. The Petri net formalism serves as a programming

language, and the models at the design stage of the software development process can be directly used – in a refined version – as the implementation of the system.[1] This approach of *implementation through specification* has already been sketched out in previous publications [4,21].

Over the last few years we have used reference nets together with the tool Renew in several advanced student projects (lasting half a year each) that are related to the main topic of Petri net-based software development. This paper summarizes the efforts and advancements of net components and the pattern-based approach for Petri net-based software development. While several details have already been presented to the community in several former publications [3,5,6], this paper gives an overview of the work done and presents some resulting facts as well as experiences gained. We focus on advances of the technique as well as advancements of the supporting tools.

The application domains range from a small *stock exchange game* over our multi-agent system implementation itself to full-scaled multi-agent applications such as an implementation of the *Settler board game* or a *distributed workflow management system* (WFMS). This leads to a high demand of conceptual, methodological and tool support for the implementation process, which is growing continuously. On the conceptual level we are able to employ and build advanced concepts on the foundation of the intuitive and semantically clear concepts of Petri nets. However, when it comes to implementation, Petri net concepts and application can improve from concepts that are common ground in programming languages / techniques. Some of these concepts – among many others – are modularization, information hiding, structuring of code, code reuse and pattern-based approaches.[2] This paper describes the concept and realization of net components, which is a means that accelerates the modeling process through standardized structures of the net models and at the same time allows to introduce a pattern-based approach.[3]

In Section 2, we introduce the concept of net components. An application-specific set of net components – the Mulan components – serves as exemplary implementation of the concept in Section 3 together with the corresponding

[1] In the Petri net formalisms mentioned above, Petri nets are simulated or interpreted, not compiled. Therefore, one has to accept a loss of performance. This is no problem at early stages of a software development process, when rapid prototyping means rapid implementation of a prototype that offers as much of the desired functionality as possible. In later design stages performance plays a more important role. Using, for example, the reference net formalism as a modeling formalism, it is possible to switch over to a Java implementation in an easy and organized way. This, however, shall not be discussed in depth here.

[2] A strong modularization of our code bases are given through the inherent nesting features of the nets-within-nets paradigm, since reference nets are nets-within-nets.

[3] Note that *model* and *implementation* fall together through the *implementation through specification* approach. Thus both concepts are used interchangeably in this work. In the context of Petri nets this is rather unusual. Similar work is done e.g. in the context of CPN tools, where the implementation language is Standard ML (see [9]).

toolset. Section 4 is dedicated to experiences in the usage of the net component concept and its toolset earned after several years of use. We present some related work and other applications of the net components concept in Section 5 and conclude with Section 6, which summarizes our results and gives an outlook on future work.

2 Concept and Design of Net Components

Net components (NC, [3,6]) are subnets, which can be composed or combined to form a large composed net. They should provide general functionality that can be commonly used. However, a set of net components is only meant to serve a special subset of similar Petri nets in a well-defined context of model/software development. Only if many similar nets of the same category are produced, the effort of designing a set of net components is rewarded and the benefits of net components are exploitable.

Net components should not be confused with the software engineering viewpoint on components of large software parts. Instead, the net components presented in this work resemble the control structures of structured programming languages like cycles or conditional statements or design/workflow patterns. In short, a net component is a template implementation (or instantiation) of a pattern.

2.1 Notions

A net component is a set of net elements that fulfills one basic task. The task should be so general that the net component can be applied to a broad variety of nets. Furthermore, the net component can provide additional help, such as a default inscription or comments. One of the used components contains a predefined but adjustable declaration node. In a formal way, net components can be seen as transition-bordered subnets. This suits the notion of net components covering tasks (net components as transition refinement).[4]

Every net component has a unique geometrical form and orientation that results from the arrangement of the net elements. This unique form is intended so that each net component can be easily identified and distinguished from each other. The geometrical figure also has the potential to provide a defined structure for the Petri net.

In the default implementation of the net components used so far in the net components toolset, places are added at the outward connecting transitions (*interface place*) for convenient net component connection.[5] Only one arc has to

[4] Note that the tool does not impose this notion of transition refinement. Instead this notion is a design decision.

[5] In the composed net our approach is similar to Kindler's notion of *Petri net components* [10]. There components are place bounded by interface places that are fused during composition. Here one place exists at the outward connection of the net component, which is connected by an arc to the next net component.

be drawn to connect one net component with another. This simple and efficient method also emphasizes the control-flow through the fact that these simple connecting arcs transport by default only black tokens. The connection of net components is provided by this place, which in the example implementation only holds anonymous tokens.

Direct data exchange between net components is not desired in order to guarantee a simple interface between net components. Instead, we store data objects to data-containing places and access the data via virtual places.[6] When the programmer adds an appropriate virtual place to the net component, data can be transferred separately from the control flow to the transition that uses a variable. Normally, the data is read through an additional test arc. The test arc allows for concurrent read-only access of the data. If the data is to be modified, a reserve arc prohibits concurrent access, and if the data is not needed anymore it can also be consumed by a directed arc. Annotations of the data-containing places should be adjusted to the appropriate name as well as the annotations of the corresponding virtual place. Also coloring the places has established itself by convention as good practice in order to identify corresponding places / virtual places pairs.

2.2 Structure of Net Components

Net components are (transition-bordered[7]) subnets that can be composed to form a larger Petri net. As in the case of design patterns their purpose is to provide pre-manufactured solutions to re-occurring challenges. Moreover, they also impose their structure onto the constructed net. Just like a snowflake's structure is determined through the underlying structure of water molecules, the net is structured by the net components.

Through their geometric form, the net components are easily identified in a larger net. This adds to the readability of the net and to the clarity of the overall structure of the net, which as a composition is an accumulation of substructures.

Jensen [9] describes several design rules for Petri net elements, which are based on work done by Oberquelle [17]. These rules encompass how to draw figures and give general advice for Petri net elements such as places, transitions and arcs. They are also concerned with combinations and arrangement of the elements.

[6] *Virtual places* can be regarded as references to the original places. Another well-known name for this is *fusion-place*. In Renew virtual places can be identified by their doubled outline. They share several attributes with the original place: color, size, form.

[7] Note that, as mentioned above, for easing the practical use each output transition is supplemented with appropriate output places. By doing so, the net components can be connected just by drawing arcs between such an additional output place and an input transition of another net component. Please also note that whether the interface place does or does not belong to the net component is a matter of preference/definition.

Net components extend the rules by giving developer groups the chance to pre-define reusable structures. Within the group of developers, these structures are fixed and well-known, although they are open to improvements. Conventions for the design of the code can be introduced into the development process, and for developers it is easy to apply, adopt and spread these conventions throughout the net component-based construction. Furthermore, the developing process is facilitated and the style of the resulting nets is unified. Once a concrete implementation of net components has been incorporated and accepted by the developers, their arrangements (form) will be recognized as conventional symbols. This makes it easier to read a Petri net that is constructed with these net components. Moreover, to understand a net component-based net it is not necessary to read all its net elements, but it is sufficient to read the substructures. This simplifies the review process as well as the refactoring of net code.

2.3 Net Components versus (Design) Patterns

Patterns and design patterns, also in the form of workflow patterns [24], have been discussed extensively in the past. They are useful elements in software (respectively workflows or business processes) development. They help developers to name and communicate important (abstract) concepts in the development phase of process / workflow / software engineering. Often they are visualized with graphical methods, e.g. UML for design patterns or Petri nets for workflow patterns. Many patterns are simple, some are complex. Many of the common patterns are omnipresent in current development. For instance the *Observer* design pattern is implemented in the Java *Listener* interface. Also other patterns have become so common that they are sometimes not recognized anymore. Nevertheless, there is a widespread agreement that patterns are useful in the development of complex systems.

Net components are instantiations of patterns. They can be instances of commonly known patterns, realizations of trivial patterns or even built from scratch for a special purpose. In addition to the advantages that are offered by patterns and the fact that net components are available as templates, net components have many other advantages. These result in part from the fact that net components are instantiations / implementations of patterns and in part from the fact that they are designed for application in Petri nets. Such advantages are:

Concreteness: In opposition to a pattern a net component is a concrete graphical component that has a fixed graphical representation (*readability, reuse*).

Composability: The net components can be composed with other net components to form a Petri net. This results partly from their concreteness. However, this has also to be taken under consideration during the design phase of the net components (*structured nets, faster development*).

Convention: The analog (geometric) representation of a net component allows the developers to easily recognize the implemented pattern in a net component in different environments (*comprehensibility, conventions*).

Congenerousness: Since the pattern and its implementation are modeled in the same graphical language there exists no breach between model and implementation (*flexibility, correctness*).

From a technical viewpoint, net components also facilitate the generation of code as template implementations together with the predefined layout possibilities and allow – through logical grouping of net elements – to further ease refactorings by simplified graphical reorganization of net code.

Some disadvantages result from the highly conventionalized implementations. Predefined solutions prevent developers from advancing the technology, and even criticism is reduced. Additionally, the generic solutions offered by the set of net components may prevent developers from finding more efficient or more concise implementations.

2.4 Requirements for Net Components

Net components have to be designed for their purpose. In any case, different kinds of nets require different sets of net components. However, within a set of net components that has been designed for a special purpose, the net components should remain as generic as possible. The net components should be easily inter-connectable so that the construction of nets is facilitated. Furthermore, net components should be designed to represent one syntactical entity. This means that a net component should represent one basic task that is decomposable. A net component should be easily identifiable to the reader of the net. This can be achieved by arranging the net elements in a unique (geometrical) form. In the current implementation a *shadow* may emphasize the geometrical form.

A net component should also provide solutions for challenges that frequently re-occur. Functionality that is thus implemented once, can be used again without repeating the process of *low-level* implementation again. Altogether these characteristics for net components are summarized in Table 1:

Characteristic	**Benefit**
Generic character	broad applicability
Interconnectivity	easily composable
Closeness	clear semantics
Unique form	easily identifiable
Located in repository	pre-manufactured but adaptable solutions.

Table 1. Criteria for net components.

Especially when nets are produced in large numbers (while designing Petri net-based applications), the advantages of the component-based approach are unveiled. The net components contribute not only to a clear structure of the nets, but also to a faster development of applications.

3 The Mulan Net Components

Mulan (MULti-Agent Nets, see [12,21]) is a concept model and framework for multi-agent systems designed with reference nets. To build a multi-agent application based on Mulan, numerous protocol nets that implement agent behavior and interactions have to be drawn. A protocol net is an implementation of a part of an agent interaction, which are usually defined as FIPA-compliant agent interaction protocols.

A set of net components for protocol nets exists [3, Chapter 4.3] that has been successfully tested and used in teaching projects (*Settler 2–6*, *WFMS 1,2* [19]) by our group at the University of Hamburg. The set of Mulan net components has been used extensively, and a large number of net component-based protocol nets have been designed during the projects with them.

The Mulan net components provide the basic functionality to construct protocols [11]. Those protocols that are constructed with the help of the Mulan net components are not restricted to the exclusive use of net components; however, it is unnecessary to use non component-based net elements, because the set is self-contained. The set provides structures for control flow management that includes alternatives, concurrency, cycles and sequences. In addition, the functionality for exchanging data is provided that offers receiving or sending of messages. Furthermore, some basic protocol-related structures are provided that handle the starting and the stopping of the protocols.

3.1 Generic Net Components

A selection of Mulan net components is presented in this section. This is done to demonstrate what kind of functionality they provide for protocol nets and their form. In this section, the essential and most frequent net components for message exchange and for basic flow control are presented. Further net components exist that cover sub-calls and manual synchronization.[8]

Control Flow Net Components: Alternatives, Concurrency (Figure 1) The conditional can be used to add an alternative to the protocol. It provides an exclusive or (XOR) situation. To resolve the conflict the `boolean` variable `cond` should be adjusted as desired. As a complement to the *NC if* the *NC ajoin* (alternative join[9]) merges the two alternative lines of the protocol. The *NC psplit* (parallel split) and the *NC pjoin* (parallel join) are provided to enable concurrent processing within a protocol. Note that the forms of these differ significantly from *NC if* and *NC ajoin* to create a clear separation of concurrency and alternatives within the protocols. We also offer a component for the trivial pattern *sequence*, which already contains an inscription that gets the reference to the agent's knowledge base (a net instance), since it is an often used routine.

[8] The full set of net components can be found in [3].

[9] The names of the patterns are inspired by the names of the workflow patterns. See [24] for alternative names of common patterns.

The access to the knowledge base is realized as a synchronous channel. It has to be supplemented with access methods for the data that is stored in or retrieved from the knowledge base. Connectable elements of net components – i.e. the elements that are connected with an arc to an element of another net component – are marked with '>'. Many net components come with predefined text annotation that are intended as in-line comments. This is a good example for the manifestation of conventions. To distinguish between inscriptions and comments, the text color is set to blue and the text is enclosed in square brackets.

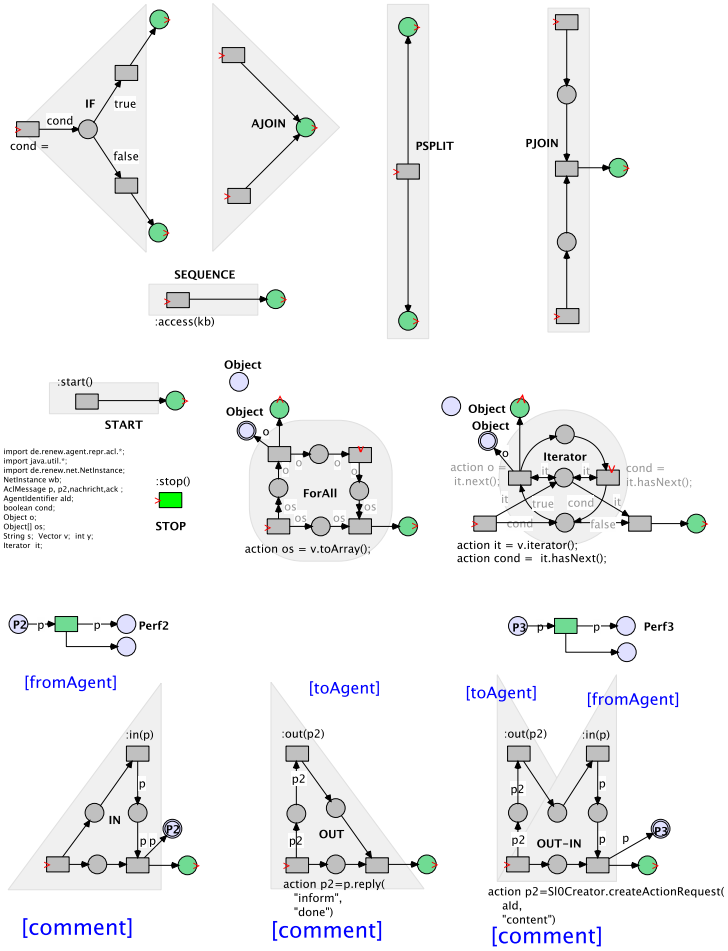


Fig. 1. MULAN Protocol Net Components

Loops These are the equivalent to the basic loops in other programming languages. The *NC iterator* provides a loop through all elements of a set described by the `java.util.Iterator`. It processes the core of the loop in a sequential order. The *NC forall* uses flexible arcs to provide a concurrent processing of all elements of an array. Flexible arcs allow the movement of a flexible number of tokens with one single arc (see [20] and [14]). The number of tokens moved by the flexible arc may vary for each firing, hence its name. In Renew double arrowheads indicate the flexible arcs. A flexible arc puts all elements of an array into the output place and it removes all elements of a pre-known array from the input place. The cores of both loops, *NC iterator* and *NC forall*, are marked with $\wedge$ (beginning) and $\vee$ (ending). Data objects are transferred to the core of the loops via virtual places (marked *Object*) already provided by the net components.

3.2 Mulan Protocol Specific Net Components

Some net components for protocol nets are specialized for the use within Mulan agents. These are net components for protocol management and messaging.

Protocol Management Net Components Beginning (*NC start*) and Ending (*NC stop*) are needed in all protocols. There is exactly one start in every Mulan protocol, but there may be more than one stop. The protocol is started when the transition with the channel inscription `:start()` is fired and stopped when one transition with the inscription `:stop()` is fired. In addition, the *NC start* also provides the declaration of the imports and all variables that are used by the net components. The transitions with the inscriptions `:start()` and `:stop()` are up-links of synchronous channels.

Furthermore, the *NC start* provides a declaration for the net. The declaration already declares the variables that are used in all Mulan net components and the import statements. It can be supplemented with other variables or imports by the developer.

Messaging Net Components These are the net components that provide the means of communication. The *NC in* receives a message, which is handed to the data block of the net component (above the main part of the NC).

Additional data containing places can be added to the data block as desired. These places can contain elements that were extracted from the messages, for example the name of the sender or the type of the performative. The *NC out* provides the outgoing message task. The *NC out-in* is a short implementation for the combination of both *NC out* and *NC in*. It provides a send request and wait-for-answer situation, but does not add functionality other than *NC out* and *NC in*. However, it shortens the protocol significantly.

3.3 Realization

Petri nets can be drawn with Renew in a fast and easy manner. To be able to use net components accordingly, it is desirable to have a seamless integration of net components in Renew. This is provided by a simple palette (Figure 2) that is the usual container for the buttons of all drawing tools for net elements.

Renew supports a highly sophisticated plug-in architecture [22], by which its functionality can be extended through plugins, so that the usual functionality is still available. The net components can be drawn in the same way as simple drawing elements by selecting the tool from a tool palette. Once a palette is loaded into the system, the net components are always available for drawing until the palette is unloaded again.

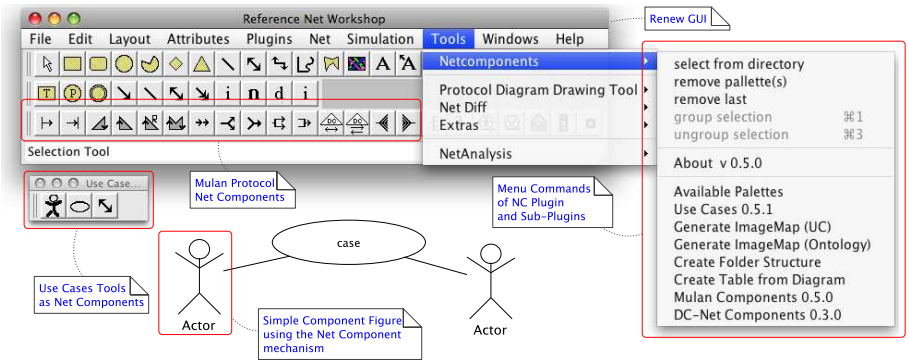


Fig. 2. Palettes and net components for Use Case diagrams in the Renew.

All net components are realized as Renew drawings, so they can easily be adjusted to the need of the programmer by editing within Renew. The net component drawings are held in repositories, thus sets of net components can be shared by a group of programmers. Nevertheless, users can also copy and modify the repository to adjust the net components to their needs, or build new net components with Renew. It is also possible to use multiple palettes of different repositories.

Net components are added to the drawing in the same way as the usual net elements. The mechanism can be compared to typical IDE template mechanisms. As for instance in *Eclipse* it is possible not only to use predefined templates for all kinds of elements (e.g. while loops, for loops, instanceof statements,...) but also to modify them. Renew and the NC Plugin allow for similar handling of net components as templates and in addition, since sets of net components are usually held in a SCM repository, these modifications can easily be spread among the developers.

4 Experiences with Net Components

Net components have been used for about six years in our Petri net-based software development projects. Due to their graphical nature Petri nets allow the creation of *spaghetti code*. Especially during the modification or refactoring of a net model, often code (net elements) is added in areas that do not offer additional space. The rearrangement of all surrounding net elements is a very time-consuming process that is often skipped or the surrounding element are carelessly pushed aside leaving behind a cluttered net structure. With net components, such situations can easily be handled due to several reasons:

- The protocol net components promote a left-to-right net layout, because the components themselves are designed horizontally.
- In most net components there is exactly one element to be customized. Therefore, additional data places are forced to be located near this element.
- Ad-hoc transitions are not needed any more because many net components already come with a skeleton for data flow and manipulation.
- Net components are supported in the tool by a weak and flat grouping mechanism.[10] This improvement eases the insertion of new elements in the middle of an existing net without destroying the layout of individual net components while it retains full modifiability of net component details.
- Documentation of nets is promoted by standardized comment templates that are attached to the net components.

In general, we observed that nets built with net components are tidier than nets without net components. Additionally, the readability of nets is improved significantly. With a little experience the comprehension of the nets is reduced to the reading of two elements per net component: the shape of the component and the customized inscription or comment. Without the need to examine every net element, it is thus possible to understand the described process.

Besides the graphical benefits, the teaching staff observed an increase in code development speed and students' learning curve in Petri net design. There are multiple explanations:

- It is obviously faster to compose a net from larger blocks than to repeat every small step repeatedly (reuse of code); even faster than *copy & paste*.
- The set of net components cover mostly well-known constructs from classic sequential programming languages such as conditionals and loops. So the transition from classical programming to Petri net programming is easier.
- Moreover, net components enable automation of net construction resulting in generation of nets or round-trip engineering techniques.

[10] The weak and flat grouping mechanism allows the movement of all elements of the component as a group while individual net elements of the net component are still accessible to any kind of manipulation. There is no hierarchical grouping, instead grouped groups become fused. The classical grouping feature is more restrictive since single elements of a group can not be manipulated nor selected.

- Existing net components come with transitions and places already inscribed and connected correctly, thus they show examples of correct code and inscriptions.
- For the attending teachers and tutors the results of the students' designs are much easier reviewed due to the structural clarity mentioned above.
- Reuse of concepts and solutions have been intensified. The original set of net components is based on patterns of net elements that have been recognized in protocol nets. Now, they are commonly used and we detect more advanced patterns that can also become net components. Design patterns become graspable for developers (pattern-based approach).
- Reuse of concepts is facilitated. It is easy to cut a recognized pattern from an existing net and turn it into a new net component.
- The overall acceleration of net design leaves more time to discuss e.g. important architectural matters of the software.

These benefits do not only apply to software development. Net components have also been applied to other areas. Besides the workflow patterns presented in [15], also a set of net components for the construction of Petri net-based plans, which are automatically assembled during runtime and executed on the fly, has been developed.

To give an illustration of the qualitative differences we have carried out some internal tests with experienced members of our group and some of our students. The results even for most simple example implementations with a given and detailed specification are already satisfying. Nets drawn without net components tend to be much smaller, due to concise implementation. However, the test persons needed two to five times longer to develop the protocol nets.[11] An immediate repetition of the test also showed that the pure coding is at least twice as fast with the use of net components. If we consider that the use of net components helps the developer avoid many pitfalls in (Petri net) programming / modeling, then the real benefit from using a net component-based approach is obviously much greater.

It has to be admitted that some of the advantages mentioned above entail a trade-off in flexibility of Petri net engineering. The strong form of net components restricts the overall net layout, and – as mentioned already – structured net component-based Petri nets tend to be larger then simple unstructured nets. Moreover, developers tend to stick to the predefined solutions and are very conservative with the introduction of new patterns. Because many net components are oriented along classical sequential programming language constructs, resulting nets sometimes include less concurrency than Petri nets would allow. It could be argued that, depending on the stage of modeling expertise the developers have gained, parts of this flexibility trade-off may also be seen as an advantage rather than a disadvantage. These are very usual and common effects regarded from the perspective of software engineering.

[11] One of our – experienced but sceptical – team members who had claimed to design faster without net components, was quite surprised that he achieved a speedup of factor five with net components.

However, some of the original and some later proposed net components have been used rarely, if at all. This results from numerous reasons, which can not be extensively discussed in this paper. Some of the reasons for rejection of net components are bad integration and a too specialized purpose. Also, frequently the ability of net components to be parameterizable, as for the automation of e.g. inscription adaption or for dynamically adaptable net structures (*switch*), is requested.

5 Related Work

Kindler [10] uses Petri net components to model parts of systems (components). The resulting models are used to verify compositional systems. The notion of Petri net components is similar to ours. In his approach (in which components are place bounded) the interface places are fused, while in our approach only one interface place exists between two connecting points of net components. The resulting nets (or net structures) after composition with Kindlers approach are the same as with the approach presented in this paper. However, no software engineering conceptualizations are discussed in the direction of a set of template implementations.

In [24,23] van der Aalst et al. have presented (advanced) workflow patterns. Although the original intent was directed towards a different goal this work has strongly influenced the net components, especially the set of MULAN protocol components. The workflow patterns show a comprehensive generic catalog of workflow pattern designed with Petri nets. To our knowledge there exists no toolset that allows the application of these. However, they have influenced the YAWL language [7], which supports many of the mentioned patterns through direct notations.

Mulyar and van der Aalst [16] present a realization of workflow patterns in coloured Petri nets with CPN Tools [1] making extensive use of the ML inscription language. By this they offer example implementations of the abstract patterns. Again, they do not discuss the idea of using these patterns as templates in the context of software development.

In the context of net components other implementations have been done. Rölke and Moldt [15] have modified the plugin to design and supply advanced workflow patterns implemented through reference nets. This set is merely an example pattern implementation with no practical purpose, but it has a nice conceptual value offering elegant solutions for challenging problems in workflow design.

Braker [2] has adopted the *workflow pattern* of van der Aalst et al. [24] for reference net-based process definitions in an early modification of the net component plugin. Braker tries to cover a broad variety of advanced workflow patterns in a practical setting. The approach suffers from overloaded, complicated implementations and the weaknesses of the early version of the net component plugin, i.e. the missing possibility to group net components. These net components were

also meant to *dock* on to each other, a feature that was envisioned but not implemented.

Cabac and Knaak [5] have also presented data-flow components in the context of process mining. These differ in the fact that – in opposition to the pure control-flow net components presented in this work – the focus lies on the processing of data, which is handed from one processor, source or sink (which are specialized net components) to the next.

The net component plugin[12] itself has been plugified to respect the fact that several sets of net components have to be supported. Thus the repository can be not only an arbitrary (but suitable) directory but also a plugin that extends the toolset with new sets of net components.

In the context of multi-agent application development with MULAN a new set of net components has been developed for the construction of decision components (DC). These nets implement the internal behavior of agents, especially processes that do not control the communication of an agent and internal long running processes, such as planning or interaction spanning synchronizations. For these nets similar net components as from the set of net components for protocol nets are used (and have accelerated the construction of DCs). However, there are also some differences, such as explicit handling of synchronizations of calls. Each call/return gets its own id, which has to be handled explicitly in the net. Figure 3 shows several proposed components that are currently in use:

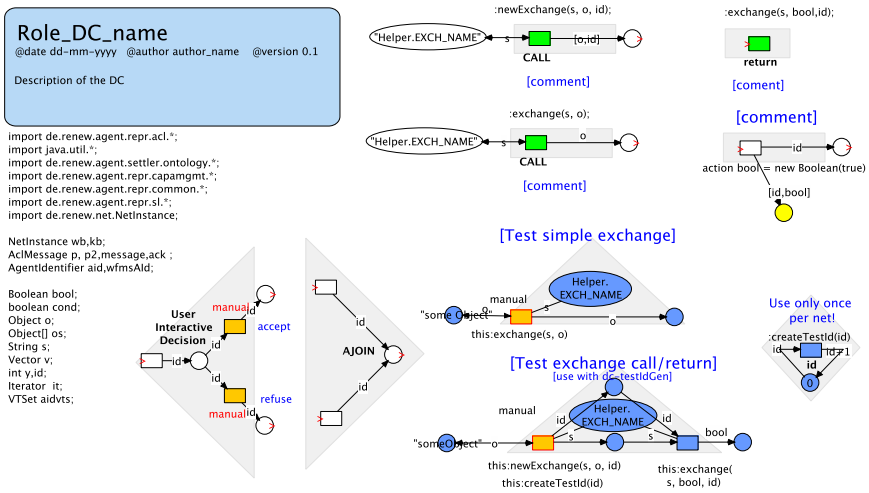


Fig. 3. The decision component net components.

Interface to the protocol nets (*call* (synchronous), *call/return* (asynchronous)),

[12] The net components plugin and some standard components (also as plugin) can be downloaded from the RENEW home page [14].

manual choice and join and some test components with which the functionality of the DC can be tested as a *stand-alone* net instance.

6 Conclusion

For Petri net-based software engineering structuring elements, pattern-based development and efficient implementation of Petri net code is needed. We have shown for the (Petri net-based) agent-oriented paradigm that net components can provide the means for this.

Net components are sub-nets with geometrical arrangements that ease their identification and discrimination. Each net component is bounded by transitions and fulfills one basic task. As a means of structuring, the Mulan net components are capable of accelerating the development of protocol nets. The readability of net component-based protocol nets is increased significantly as well as the speed of construction of nets in comparison to protocol nets without net components. The net components tool implemented as a plugin for Renew enables us to use net components for the fast construction of Petri nets. The net components that are held in a repository are editable in Renew, thus adaptable to the needs of the development team. Moreover, the net components plugin itself is extensible by repository plugins that are dynamically pluggable and unpluggable at runtime.

The Mulan net components are successfully used since the second teaching project of the ongoing series of multi-agent system development projects in our research group and have eased the teaching and development of Petri net-based protocol nets.

We are looking forward to apply the mentioned workflow patterns [15] – and improve the implementation as net components – to the currently developed agent-based distributed workflow engine [18]. This integrates a reference net-based workflow engine [8] into the multi-agent system Mulan. In this context net component-based development can improve the development of workflows. A first design of simple net components has already been integrated into the net component tool set [15].

In the future we want to introduce net components into other areas of net development where extensive coding is undertaken. Net components can be applied to other software development paradigms and besides workflow design they can also improve the construction of large scale Petri net models in other areas.

Currently, we are experimenting with a collapse (and expand) functionality for net components (similar to coarsening/refinement) that could lead to a rapid prototyping of languages such as workflow languages e.g. YAWL [7]. Figure 4 shows an exemplary collapse from a *NC cond* to a YAWL-like notation. New graphical language elements can be defined rapidly and directly executed through the underlying operational Petri net semantics.

References

1. CPN Tools. online, September 2007. `http://wiki.daimi.au.dk/cpntools/`.

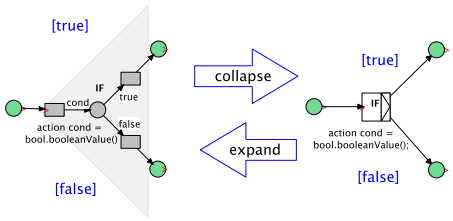


Fig. 4. Collapsing of net components for rapid graphical language prototyping.

2. Marco Braker. Workflowpetrinetze: Hierarchisierung mittels Netzen-in-Netzen. Diploma thesis, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg, March 2004.
3. Lawrence Cabac. Entwicklung von geometrisch unterscheidbaren Komponenten zur Vereinheitlichung von Mulan-Protokollen. Studienarbeit, University of Hamburg, Department of Computer Science, 2002.
4. Lawrence Cabac, Michael Duvigneau, Daniel Moldt, and Heiko Rölke. Modeling dynamic architectures using nets-within-nets. In Gianfranco Ciardo and Philippe Darondeau, editors, *Applications and Theory of Petri Nets 2005. 26th International Conference, ICATPN 2005, Miami, USA, June 2005. Proceedings*, volume 3536 of *Lecture Notes in Computer Science*, pages 148–167, 2005.
5. Lawrence Cabac and Nicolas Knaak. Process mining in Petri net-based agent-oriented software development. In Daniel Moldt, Fabrice Kordon, Kees van Hee, José-Manuel Colom, and Rémi Bastide, editors, *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'07)*, pages 7–21, Siedlce, Poland, June 2007. Akademia Podlaska.
6. Lawrence Cabac, Daniel Moldt, and Heiko Rölke. A proposal for structuring Petri net-based agent interaction protocols. In Wil van der Aalst and E. Best, editors, *24th International Conference on Application and Theory of Petri Nets, Eindhoven, Netherlands, June 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 102–120. Springer-Verlag, June 2003.
7. A. H. M. Ter Hofstede. Yawl: yet another workflow language. *Information Systems*, 30:245–275, 2005.
8. Thomas Jacob, Olaf Kummer, Daniel Moldt, and Ulrich Ultes-Nitsche. Implementation of workflow systems using reference nets – security and operability aspects. In Kurt Jensen, editor, *Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, Ny Munkegade, Bldg. 540, DK-8000 Aarhus C, Denmark, August 2002. University of Aarhus, Department of Computer Science. DAIMI PB: Aarhus, Denmark, August 28–30, number 560.
9. Kurt Jensen. *Coloured Petri Nets*, volume 1. Springer-Verlag, Berlin, 2nd edition, 1996.
10. Ekkart Kindler. A compositional partial order semantics for Petri net components. In *Application and Theory of Petri Nets 1997, Proceedings, volume 1248 of LNCS*, pages 235–252. Springer-Verlag, 1997.
11. Michael Köhler, Daniel Moldt, and Heiko Rölke. Modelling the structure and behaviour of Petri net agents. In J.M. Colom and M. Koutny, editors, *Proceedings of the 22nd Conference on Application and Theory of Petri Nets 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 224–241. Springer-Verlag, 2001.

12. Michael Köhler, Daniel Moldt, and Heiko Rölke. Modelling mobility and mobile agents using nets within nets. In Wil van der Aalst and Eike Best, editors, *Proceedings of the 24th International Conference on Application and Theory of Petri Nets 2003 (ICATPN 2003)*, volume 2679 of *Lecture Notes in Computer Science*, pages 121–139. Springer-Verlag, 2003.
13. Olaf Kummer. *Referenznetze*. Logos Verlag, Berlin, 2002.
14. Olaf Kummer, Frank Wienberg, and Michael Duvigneau. Renew – the Reference Net Workshop. Available at: `http://www.renew.de/`, September 2008. Release 2.1.1.
15. Daniel Moldt and Heiko Rölke. Pattern based workflow design using reference nets. In Wil van der Aalst, Arthur ter Hofstede, and Mathias Weske, editors, *Proceedings of International Conference on Business Process Management, Eindhoven, NL*, volume 2678 of *Lecture Notes in Computer Science*, pages 246–260. Springer-Verlag, 2003.
16. N.A. Mulyar and W.M.P. van der Aalst. Patterns in colored petri nets. BETA Working Paper Series WP 139, Eindhoven University of Technology, Eindhoven, 2005.
17. Horst Oberquelle. Communication by graphic net representations. Technical Report IFI-HH-B-75/81, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg, 1981.
18. Christine Reese, Kolja Markwardt, Sven Offermann, and Daniel Moldt. Distributed business processes in open agent environments. In Yannis Manolopoulos, Joaquim Filipe, Panos Constantopoulos, and José Cordeiro, editors, *ICEIS 2006 - Proceedings of the Eighth International Conference on Enterprise Information Systems: Databases and Information Systems Integration, Paphos, Cyprus, May 2006*, pages 81–86, 2006.
19. Christine Reese, Matthias Wester-Ebbinghaus, Till Dörges, Lawrence Cabac, and Daniel Moldt. A process infrastructure for agent systems. In Mehdi Dastani, Amal El Fallah, Joao Leite, and Paolo Torroni, editors, *MALLOW'007 Proceedings. Workshop LADS'007 Languages, Methodologies and Development Tools for Multi-Agent Systems (LADS)*, pages 97–111, 2007.
20. Wolfgang Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets*. Springer-Verlag New York, October 1997.
21. Heiko Rölke. *Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen*, volume 2 of *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2004.
22. Jörn Schumacher. Eine Plugin-Architektur für Renew – Konzepte, Methoden, Umsetzung. Diploma thesis, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg, October 2003.
23. Wil M. P. van der Aalst, Alistair P. Barros, Arthur H. M. ter Hofstede, and Bartek Kiepuszewski. Advanced workflow patterns. In Opher Etzion and Peter Scheuermann, editors, *Cooperative Information Systems, 7th International Conference, CoopIS 2000, Eilat, Israel, September 6-8, 2000, Proceedings, CoopIS*, volume 1901 of *Lecture Notes in Computer Science*, pages 18–29. Springer-Verlag, 2000.
24. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns, 2000.
`http://tmitwww.tm.tue.nl/research/patterns/wfs-pat-2000.pdf`.

Lets's Play the Token Game – Model Transformations Powered By Transformation Nets*

M. Wimmer[1], A. Kusel[2], J. Schoenboeck[1], T. Reiter[2], W. Retschitzegger[3], and W. Schwinger[2]

[1] Vienna University of Technology, Austria
{wimmer,schoenboeck}@big.tuwien.ac.at
[2] Johannes Kepler University Linz, Austria
kusel@bioinf.jku.at, reiter@ifs.uni-linz.ac.at
wieland.schwinger@jku.ac.at
[3] University of Vienna, Austria
werner.retschitzegger@univie.ac.at

Abstract. Model-Driven Engineering (MDE) is a software engineering paradigm using abstract models to describe systems which are then systematically transformed to concrete implementations. Since model transformations are crucial for the success of MDE, several kinds of dedicated transformation languages have been proposed. Hybrid languages combine the statefulness and the ability to define control flow of imperative approaches with the raised level of abstraction of declarative ones. However the low-level engines employed to execute transformations lead to an impedance mismatch between specification and execution of model transformations which hampers debugging. Additionally, current approaches lack of appropriate reuse mechanisms for resolving recurring transformation problems. Therefore, we propose a process-oriented specification and execution of model transformations based on Transformation Nets, a variant of Colored Petri Nets (CPNs). By using Transformation Nets, the benefits of imperative and declarative approaches are combined, not only for the specification of model transformations, but also for their execution by using CPNs themselves as a transformation engine. Furthermore, Transformation Nets introduce a novel notation for implementing transformation logic within transitions to foster reuse.

Key words: Model-Driven Engineering, Model Transformations, CPN

1 Introduction

Model-Driven Engineering (MDE) [7] is a current trend in software engineering where models are used to describe systems which are then systematically transformed to concrete implementations. Thus, MDE places models as first-class

* This work has been partly funded by the Austrian Science Fund (FWF) under grant P21374-N13.

artifacts throughout the whole software lifecycle [3]. The main promise of MDE is to raise the level of abstraction from technology and platform-specific concepts to platform-independent and computation-independent modelling. To fulfill this promise, the availability of appropriate model transformation languages is *the* crucial factor, since transformation languages are for MDE as important as compilers are for high-level programming languages. Transformation scenarios can be divided into vertical model transformations and horizontal model transformations. Vertical model transformations lower the level of abstraction, e.g., transforming UML class diagrams to relational models, whereas horizontal model transformations transform models between two different representations on the same level of abstraction, which is the focus of our approach, e.g., a UML class model is transformed to an entity relationship diagram.

To specify model transformations, approaches range from purely imperative styles allowing to define how a transformation is carried out to fully declarative transformation definition styles focusing on what a transformation's output should be like according to a certain input. In between this spectrum, hybrid approaches combine the statefulness and the ability to define control flow of imperative approaches with the raised level of abstraction of declarative ones [4]. In general, declarative and hybrid approaches use transformation engines to execute the model transformations operating on a low level of abstraction, e.g., the Atlas Transformation Language (ATL) uses a stack machine [9], shown as black-box to the transformation designer. As a consequence, debugging of model transformations is limited to the information provided by the transformation engine, only, most often consisting of variable values and logging messages, but missing important information, e.g., why a certain part of a transformation is actually executed or not. This is due to the fact that an explicit runtime model [7] for the execution of model transformations is not supported which could be used to observe the runtime behavior of certain transformations.

Another problem current transformation languages suffer from is that no appropriate reuse mechanisms and pre-defined libraries for resolving recurring model transformation problems are provided. Especially, the resolution of structural heterogeneities [10], i.e., the same concept is expressed by different metamodel elements, represents a recurring challenge. Thus, resolving structural heterogeneities requires to manually specify partly tricky model transformations again and again for each scenario.

The contribution of this paper is a novel approach for defining model transformations called Transformation Nets [12], which use a variant of Colored Petri Nets (CPNs) [8]. Transformation Nets embody a process-oriented view on model transformations, whereby the actual transformation is carried out by reuasable patterns of transformation logic that stream models represented by the net's tokens from source to target. Such a runtime model provides the explicit statefulness of imperative approaches through tokens contained in the net's places. The abstraction of control flow from declarative approaches is achieved as the net's transitions can fire autonomously depending on their environment and effectively make use of implicit, data-driven control flow. Furthermore, Trans-

formation Nets provide a uniform formalism not only for representing the transformation logic together with the metamodels and the models themselves, but also for executing the transformations. Thus, no impedance mismatch between specification and execution occurs which allows for enhanced understandability and debuggability of model transformations. In this paper we present how a variant of CPN is employed for the domain of model transformations. Firstly, we show how model transformations are appropriately represented with a variant of CPNs, and secondly, a novel notion for implementing reusable transformation logic within transitions is proposed.

The remainder of this paper is structured as follows. Section 2 gives an overview of the Transformation Net formalism. While, the subsequent Section 3 describes the static part of Transformation Nets, depicting how metamodels and models are represented within Transformations Nets, Section 4 elaborates on the dynamic part of Transformation Nets, especially how transformation logic is specified. Section 5 reports on lessons learned by the application of Transformation Nets. Related work is discussed in Section 6, and finally, Section 7 gives a conclusion and an outlook on future work.

2 Transformations Nets at a Glance

Within this section the basic concepts of model transformations in general and Transformation Nets in particular are introduced.

2.1 Big Picture of Model Transformations

The general model transformation scenario is illustrated in the upper half of Figure 1 comprising a transformation with one input (source) model and one output (target) model [4]. Both models conform to their respective metamodels which define the abstract syntax of a modeling language. Transformation Nets provide an integrated view on model transformations by homogenously representing metamodels, models, and transformation logic. At the same time Transformation Nets serve also as a runtime model to actually carry out the transformation. One can differentiate between static and dynamic parts of a Transformation Net. The static parts correspond to the transformations' metamodel elements (represented as places), given input model elements and generated output model elements (represented as tokens), whereas the dynamic parts corresponds to the transformation logic itself (represented as transitions) as formalized in a metamodel introduced in the following.

2.2 Transformation Net Metamodel

The abstract syntax of the Transformation Net is formalized by means of a metamodel (cf. Fig. 2) conforming to the OMG's Meta Object Facility (MOF) [1] standard. The Transformation Net metamodel describes appropriately adapted

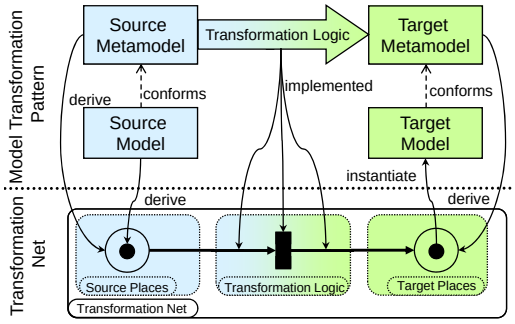


Fig. 1. Conceptual Architecture of Transformation Nets

Colored Petri Net concepts [8] in order to fulfill the special requirements occurring in the domain of model transformations. In particular, in order to be able to encode metamodels and models, we introduce two kinds of places and two kinds of tokens (cf. Sec. 3). The second major adaption concerns the transitions. Since transitions are used to realize the transformation logic, we borrow a well established specification technique from graph transformation formalisms [6], which describe their transformation logic as a set of graphically encoded productions rules (cf. Sec. 4).

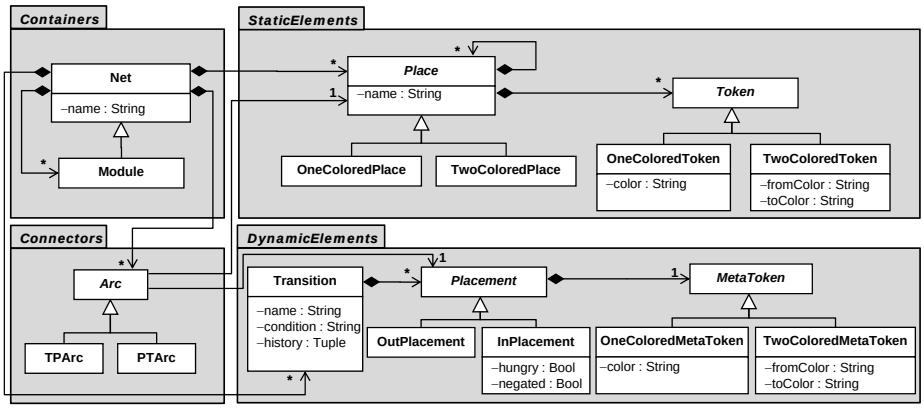


Fig. 2. The Transformation Net Metamodel

The whole Transformation Net metamodel is divided into four subpackages as can be seen in Fig. 2. Thereby, the package `Containers` comprise the modularization concepts. The package `StaticElements` is used to represent the static parts of a model transformation, i.e., metamodels and models. The dynamic elements, i.e., the transformation logic, are represented by concepts of the package `DynamicElements`. The package `Connectors` finally is responsible for connecting the static parts with the dynamic parts. In the following sections, we describe the packages and their contained elements in detail.

3 The Static Part of Model Transformations

To represent metamodels and models in Transformation Nets a translation from the graph-based paradigm underlying MDE to the set-based paradigm underlying Petri Nets is necessary. We rely on the core concepts of an object-oriented meta-metamodel as defined by MOF, being classes, attributes, and references. In the following, we elaborate on how MOF concepts used in metamodels and their respective instances are represented within Transformation Nets (cf. Fig. 3 for a summary).

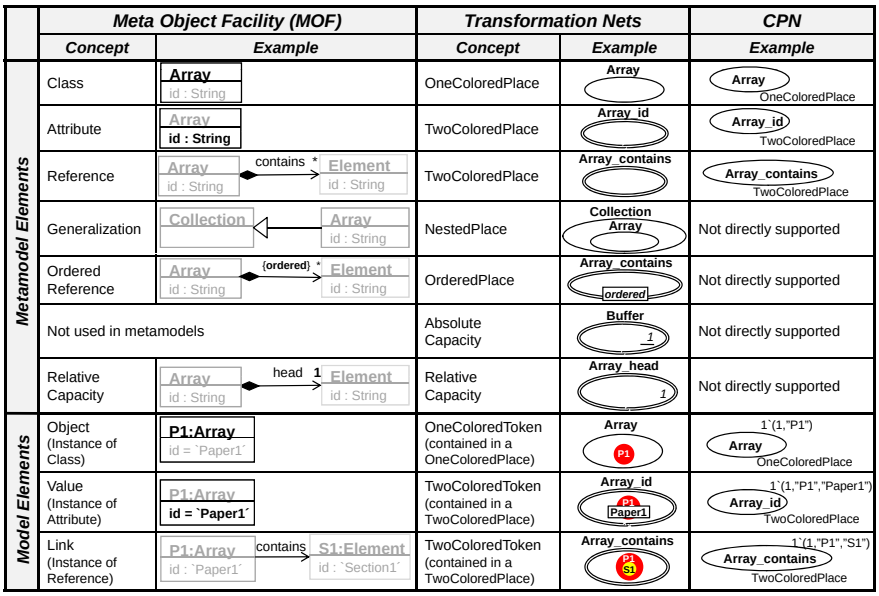


Fig. 3. Representing MOF concepts within Transformation Nets

3.1 Representing Metamodel Elements as Places

A metamodel mainly consists of classes, attributes, and references which are mapped to places in a Transformation Net. We distinguish between two kinds of places, namely `OneColoredPlace` to represent classes, and `TwoColoredPlace` to represent attributes and references. Both types of places can be represented by according data types in CPNs which are assigned to the corresponding places.

Classes represented as One-Colored Places. Abstract and concrete classes are both represented as `OneColoredPlaces`. Although, abstract classes cannot have instances, places created from abstract classes normally contain tokens indirectly due to other places stemming from sub-classes, (cf. below) being contained within them. Furthermore, the name of the class becomes the name of the place (`name`). The notation used to visually represent one-colored places is

an oval as traditionally used to depict places in Petri nets. Subclass relationships are represented by `nestedPlaces` whereby the place corresponding to the subclass is contained within the place corresponding to the superclass. The tokens contained in the"sub-place" are also visible in the "super-place", which means that if a token is contained in a sub-place it may also act as input token for a transition connected to the "super-place".

Attributes and References represented as Two-Colored Places. Attributes and references are represented by `TwoColored Places`, whereby the name of the place consists of the name of the containing class and the name of the attribute or reference itself, separated by an underscore (`ClassName_name`). Notationally, the borders of two-colored places are doubly-lined to indicate that they contain two-colored tokens.

Orderings. References that impose an order on their links, e.g., an array element which has several elements contained in a specific order, require some extensions to normal two-colored places. If in case of such an ordered reference, the content of the two-colored place is internally set up as several ordered sequences (but not explicitly represented). For instance, for each different array represented by different `fromColor`, a sequence of two-colored tokens with that `fromColor` exists. Sequences in ordered places are working according to the FIFO principle in order to facilitate the implementation of transformation logic.

Multiplicities. Places can restrict the amount of tokens they can contain. In particular two-colored places have an absolute capacity (`absCapacity`) to restrict the total number of its tokens and a relative capacity (`relCapacity`) to restrict the maximum length of its sequences. Hence, multiplicities of references can be mapped onto the relative capacity of a two-colored place. For instance, a two-colored place with a capacity of '1' may contain several tokens, but for each token a distinct `fromColor` is mandatory. An absolute capacity would allow only one token irrespective of its color inside the place, e.g., used to ensure a sequential eradication or to represent singleton classes. Capacities are visualized through the respective number inside the place, which is underlined in case of an absolute capacity.

3.2 Representing Model Elements as Tokens

Objects represented as One-Colored Tokens. For every object, i.e., instance of `Class` that occurs in a model a `OneColoredToken` is produced, which is put into a place that corresponds to the respective element in the source metamodel. The "color" is realized through a unique value (`color`) that is derived from the object_id (OID).

Values and Links represented as Two-Colored Tokens. For every value as an instance of an `Attribute`, as well as for every link as an instance of a `Reference`, a `TwoColoredToken` is produced. The `fromColor` attribute refers to the color of the token (thus the OID) that corresponds to the owning object. The `toColor` is given by the color of the token that corresponds to the referenced target object (the OID of the target object). Notationally, a two-colored token

is represented as a ring (depicting the `fromColor`) surrounding an inner circle (depicting the `toColor`).

3.3 Transformation Nets by Example

By making use of an example we show how Transformation Nets can be applied to transform arrays to linked lists. The top of Fig. 4 depicts the source and target metamodels, as well as the input model and the desired, semantically equivalent output model. The mapping of concepts between the array metamodel and the linked list metamodel is achieved by transforming the `Array` class to an equivalent `LinkedList` class, just like the `Element` classes are transformed to `Node` classes. The ordered set of `contains` links needs to be translated into correctly set up `head`, `next` and `prev` links that maintain a semantically equivalent ordering of `Node` objects.

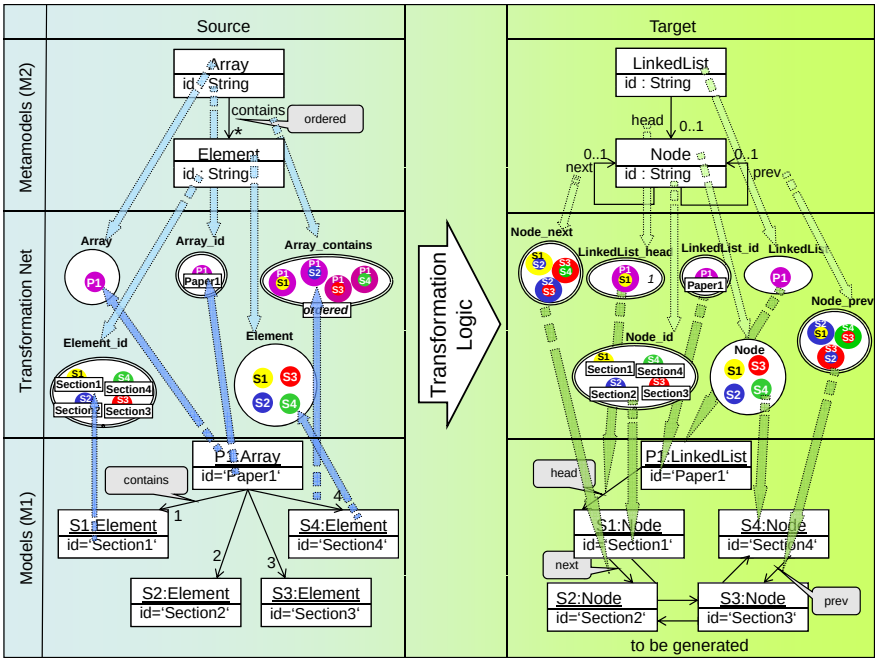


Fig. 4. Motivating Example: Static Part

Fig. 4 depicts the static parts of the Transformation Net for the example. The metamodel elements `Array` and `Element` are represented by two corresponding places in the Transformation Net. Both classes have an `id` property, represented as two-colored places as well as the reference `contains`. The lower part of the figure shows models conforming to the metamodels used to extract tokens which are put into the corresponding places. The array `P1` is indicated by a one-colored token in the `Array` place whereas the `id` of the array is depicted by a two-colored

token in the `Array_id` place. The four elements (`S1` to `S4`) contained in the `Array` are represented by four according tokens in the `Element` place. In addition four two-colored tokens are created representing the `ids` of the attributes and put into the `Element_id` place. The four tokens in the place `Array_contains` represent the links between array (depicted by the outer color) and it's elements (depicted by their inner color). Analogous to the source model, places for the target model are created which are empty until the target tokens are generated by executing the model transformation. The shown target tokens in Fig. 4 are therefore the result of a successfully executed transformation, which are then used to instantiate the target model.

4 The Dynamic Part of Model Transformations

The transformation logic is embodied by a system of Petri Net transitions and additional places which reside in-between those places representing the original input and output metamodels. In this way, tokens are streamed from the source places through the Transformation Net and finally end up in the target places. Hence, when a Transformation Net has been generated in its initial state, i.e., source tokens are already derived from the input model, a specialized Petri Net engine executes the transformation process and streams tokens from source to target places. The resulting tokens in the target places are then used to instantiate an output model that conforms to the target metamodel. In the following it is shown how to actually match for source model elements and how target model elements are produced.

4.1 Matching and producing model elements by firing transitions

An execution of a model transformation has two major phases. The first phase comprises the matching of certain elements of the source model whereas the second phase produces the elements of the output model. This matching and producing of model elements is supported within Transformation Nets by firing transitions. To specify their firing behavior, a mechanism well known from graph transformation systems is used [6]. Transitions consist of input placeholders (LHS of the transition) representing the pre-conditions of a certain transformation, whereas output placeholders (RHS of the transition) depict its post-condition. Those placeholders are expressed by the classes `InPlacement` (LHS) and `OutPlacement` (RHS) in the metamodel as shown in Fig. 2. Every `Placement` is connected to a source or target place using `Arcs`, whereby incoming and outgoing arcs are represented by the classes `PTArc` and `TPArc`, respectively. To express these pre- and post-conditions, so-called meta tokens (cf. class `MetaToken` in the metamodel) are used, prescribing a certain token configuration by means of color patterns which can be used in two different ways, either as Query Token (LHS) or as Production Token (RHS), as shown in Fig. 5.

Query Tokens. Query tokens are meta tokens which are assigned to input placements. Query tokens can either stand for one-colored or two-colored token

configurations, whose colors represent variables that are bound during matching to the color of an actual input token. Note that the colors of query tokens are not the required colors for input tokens, instead they describe configurations that have to be fulfilled by input tokens. Normally, query tokens match for the existence of input tokens but with the concept of negated input placements it is also possible to check for the non-existence of certain tokens (cf. attribute `negated` of class `InPlacement` in Fig. 2).

Production Tokens. Output placements contain so-called production tokens which are equally represented through the class `MetaToken` and its subclasses. For every production token in an output placement, a token is produced in the place that is connected to the output placement via an outgoing arc. The color of the produced token is defined by colors that are bound to the colors of the input query tokens contained in one transition. However, it is also possible to produce a token of a not yet existing color, for instance if the color of the output query token does not fit to any of the input query tokens. With this mechanism, new elements can be created in the target model which do not exist in the source model.

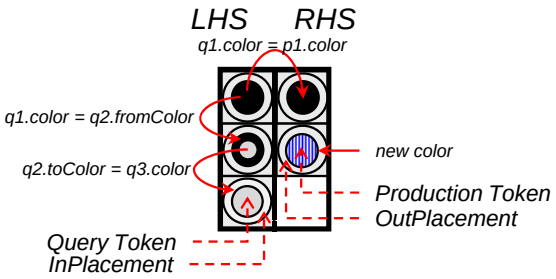


Fig. 5. Example Transition and Color Binding

By matching a certain token configuration from the input places, the transition is ready to fire with the colors of the input tokens bound to the meta tokens residing in the input placements. The production of output tokens once a transition fires is dependent on the matched input tokens. For example, when a transition is simply streaming a certain token, it would produce as an output the exact same token that was matched as the input token (cf. (a) in Figure 6). This form of transition is especially needed for implementing one-to-one correspondences between metamodel elements. Please note that the default firing behavior of a Transformaion Net does not consume the tokens of the input places in order to avoid race conditions as often several transitions make use of one and the same input place. This can be changed by setting the attribute `hungry` of the corresponding `InPlacement` to "true". In order to prevent a transition to fire more than once for a certain token configuration, the already processed configurations are stored in the `history` of a transition.

4.2 Reusable Transformation Logic

Since `InPlacements` as well as `OutPlacements` are just typed to one-colored tokens and two-colored tokens, but not to certain metamodel classes, these transitions can be reused in different scenarios. Different to CPNs which use arc-inscriptions to encode firing behavior as shown in Fig. 6, Transformation Net transitions encapsulate this information. In Transformation Net arcs need no inscriptions and therefore places extracted from a metamodel can be connected directly to predefined transitions. This kind of reuse is not restricted to single transitions only, since through the composition of transitions by sequencing as well as nesting the resulting transformation modules realize complex transformation logic. Furthermore, the graphical representation of pre- and post-conditions by color patterns is similar to graph transformation patterns transformation designers are used to. To exemplify basic firing rules, Fig. 6 shows a series of transitions with their placements containing patterns of one- and two-colored query and production tokens and their histories are shown below the actual transition as well as the equivalent transitions in CPNs. These transitions represent reusable patterns which are applied in the following to solve the example of Sec. 4.3.

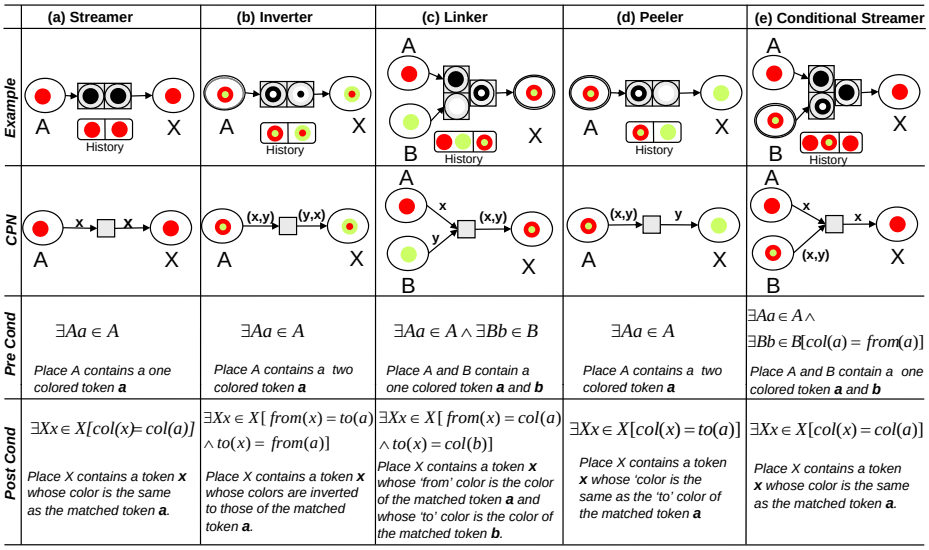


Fig. 6. Reusable Transformation Logic expressed as Transitions

`Transition (a)` in Fig. 6 shows a simple pattern that matches a one-colored token from an input place and streams the exact same token to an output place, therefore this pattern is called `Streamer`. It can be applied in cases of one-to-one mappings, e.g., an array is transformed to a linked list. `Transition (b)` matches a two-colored token from its input place, and produces an inverted token in the output place. This `Inverter` pattern can be applied to set inverse

references (cf. "next" and "prev" references in our example). `Transition (c)` called `Linker` shows two one-colored query tokens on the input side, whose matched colors determine the "from"- and "to"-colors of the produced output token. This pattern is used to introduce new links between objects. `Transition (d)` matches two-colored tokens from input places and peels off the outer color of the token, therefore the name `Peeler`. This pattern is used to get the value of an attribute or the target of a link which is represented by the inner color of the two-colored token. Finally, `transition (e)` represents a variation of the `Streamer` pattern called `ConditionalStreamer` adding additional query tokens to ensure certain preconditions. For example, this pattern may be used to ensure that before a link between two objects is set the objects to be linked have been created.

4.3 Solution for the Motivating Example

Fig. 7 depicts the complete Transformation Net realizing the necessary transformation logic added in between the static parts of source and target places. Note that the shown markings represent the state after execution of the Transformation Net.

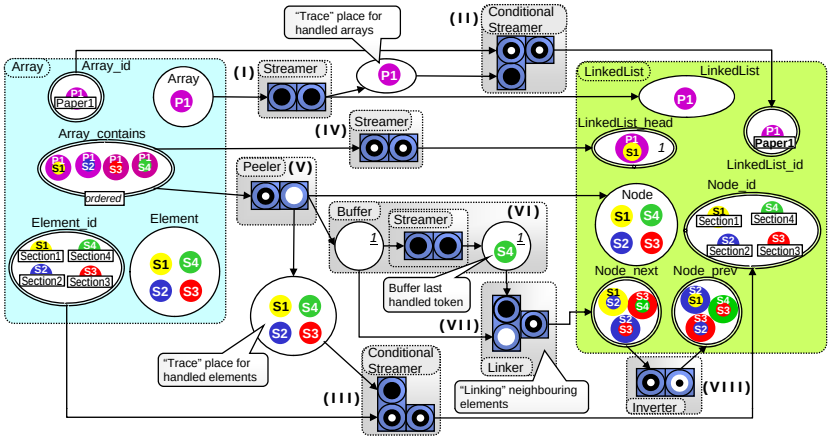


Fig. 7. Motivating Example solved with Transformation Nets

Starting from the top, we see `Transition (I)` that matches one-colored tokens from the "Array" place and propagates them into the "LinkedList" place using the `Streamer` pattern. The "id" attributes are streamed once their owning objects have been transformed, which is the case when the owning objects are present in the nets "trace" places by applying `ConditionalStreamers`, namely `Transition (II)` and `Transition (III)`. Analogously, the `Transition (IV)` matches two-colored tokens from the "contains" place and propagates them to the "head" place. Note that this transition can only fire once, because the "head" place has a relative capacity of one, only. Due to the fact that the input place is

of the ordered kind, the transition matches the tokens starting from the "zero-th" link of the reference. The third and final case where such primitive transition logic suffices is the transformation of all "Element" tokens to the "Node" place. Compared to that, dealing with "contains" tokens is of more interest. Thereby, two-colored tokens are matched from the "contains" place, and the adjacent `Transition (V)` peels out the outer ring, basically producing a duplicate of an "Element" token. The outcome is then put into a place with absolute capacity of one to enforce the ordering of tokens. Only if this place is empty again, the transition matching "Array_contains" tokens can produce the follow-up token therein. That place is cleared by the switching of the adjacent `Transition (VI)`, which consumes its input and moves it to its output with an absolute of capacity one. In case both places are filled with a one-colored token, the `Transition (VII)` is enabled which produces a two-colored token out of its inputs that is streamed into the "next" place. When firing, only the token from the right-most place is consumed, thus freeing the place to be filled again. Hence, this Transformation Net pattern forms a buffer-like structure of two places, which are, once they are filled, partly emptied to make space for successor tokens in the "queue" of places with single capacity. `Transition (VIII)` inverts and copies the created two-colored tokens from the "Node_next" to the "Node_prev" place.

The example has been realized with the help of our prototype tool supporting modeling, executing, and debugging Transformation Nets. Further details about the tool support may be found at our project page[4]. In order to show that Transformations Nets hide complexity in contrast to CPNs Fig. 8 depicts an equivalent CPN executing the same transformation. Firstly, to express the fact that tokens are not consumed per default, tokens consumed from a place are streamed back again. In order to avoid multiple firings every token gets an index and an additional place storing a counter is added to a transition. The transitions uses this counter to match for a specific token which is incremented after firing to match for the next available token (see label (I) in Figure 8). Secondly, standard CPNs offer no built-in concepts to express absolute and relative capacities. For absolute capacities we therefore again add an additional place, cf. CPN patterns presented in [11], holding the required number of tokens (see label (II) in Figure 8). Relative capacities can be expressed using a complex arc inscription, labeled (III) in Figure 8. Thereby it is shown that Transformation Nets can be mapped to standard CPNs in order to make use of already existing, efficient execution engines or to apply formal CPN properties in order to check the specification of the Transformation Net.

5 Lessons Learned

This section presents lessons learned from the running example and thereby discusses key features of the Transformation Net approach.

Representation of model elements by colored tokens reveals traceability. The source model to be transformed is represented by means of one-

[4] http://www.modeltransformation.net

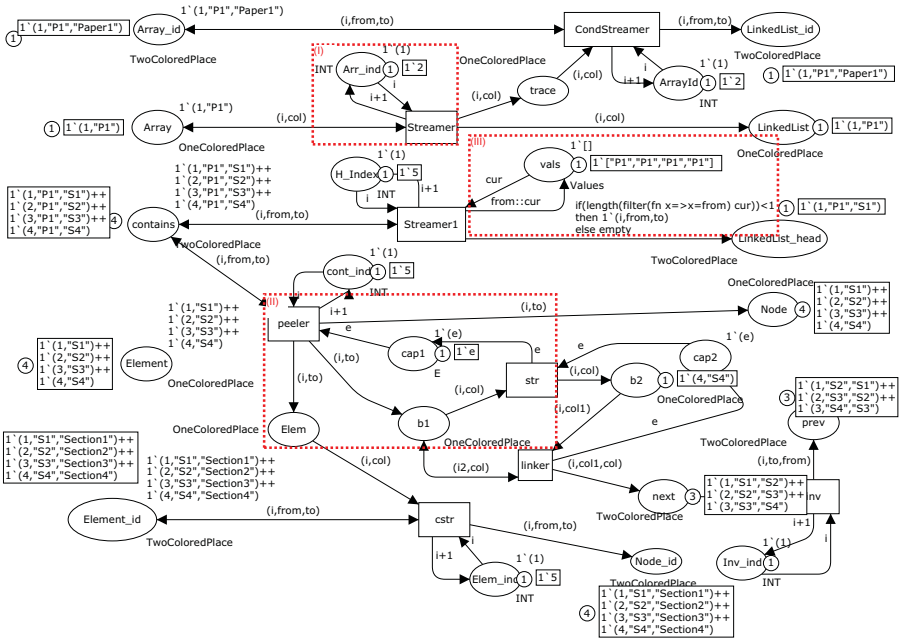


Fig. 8. Array2LinkedList using CPN

colored tokens and two-colored tokens residing in the source places of the Transformation Net whereby the actual transformation is performed by streaming these tokens to the target places. Through this mechanism it is possible to derive the source-target relationship, i.e., traceability, between model elements by searching for same-colored tokens in source places and target places, respectively.

Visual syntax and live programming fosters debugging. Transformation nets represent a visual formalism for defining model transformations which is especially favorable for debugging purposes. This is not least since the flow of model elements undergoing certain transformations can be directly followed by observing the flow of tokens whereby undesired results can be detected easily. Another characteristic of transformation nets, that fosters debuggability, is live programming, i.e., some piece of transformation logic can be executed and thus tested immediately after definition without any further compilation step. Therefore, testing can be done independently of other parts of the Transformation Net by just setting up a suitable token configuration in the input places.

Implicit control flow eases evolution. The control flow in a transformation net is given through data dependencies between various transitions. As a consequence, when changing a transformation, one needs to maintain a single artifact only instead of requiring additional efforts to keep control flow and transformation logic (in the form of rules) synchronized. For instance, when a certain rule would need to be changed to match for additional model objects, one has to explicitly take care to call this rule at a time when the objects to be matched already exist.

Fine-grained model decomposition facilitates resolution of heterogeneities. The chosen representation of models by Transformation Nets lets references as well as attributes become first-class citizens, resulting in a fine-grained decomposition of models. The resulting representation in combination with weak typing turned out to be especially favorable for the resolution of structural heterogeneities. This is since on the one hand there are no restrictions, like a class must be instantiated before an owned attribute and since on the other hand e.g. an attribute in the source model can easily become a class in the target model by just moving the token to the respective place. Due to this fine grained decomposition we can not ensure a target model that is conform to its metamodel during transformations. The conformance to the target metamodel is checked when the target model is instantiated using the tokens in the target places.

Transitions by color-patterns ease development but lower readability. Currently the precondition as well as the postcondition of a transition are just encoded by one-colored as well as two-colored tokens. On the one hand, this mechanism eases development since e.g. for changing the direction of a link it suffices just to swap the respective colors of the query token and the production token. On the other hand, the larger the transformation net grows the less readable this kind of encoding gets. Therefore, it has been proven useful to assign each input as well as each output placement a human-readable label, that describes the kind of input and output, respectively.

6 Related Work

Concerning our Transformation Net approach, we consider two orthogonal threads of related work. First, we discuss current model transformation approaches and point out their debugging support, and second, we elaborate on the usage of Petri Nets for model transformations.

Model Transformation Languages. Model transformation languages in general can be divided into imperative, declarative and hybrid approaches. Basically, imperative approaches are similar to direct manipulation of models through some general purpose programming language API, but offer a dedicated concrete syntax and allow to define the transformation in terms of the models abstract syntax. The operational part of the QVT specification [2] allows to define mappings, which are function-like constructs that can be imperatively called to create target elements. Declarative approaches are typically based on defining rules that are later on interpreted by an execution engine to produce the desired result. Hence, the actual transformation execution as well as the order of rule application generally need not be handled by the user. The way how a transformation is defined, is by specifying rules that constrain under which condition certain elements of the source and target metamodel are related. One part of the QVT specification consists of the Relations language, which allows to define rules in the above described way. However, what a declarative approach gains in abstraction, it loses in flexibility. To alleviate these limitations, hybrid approaches

combine imperative and declarative styles of transformation definition. The Atlas Transformation Language (ATL) [9] is a QVT-like language that distinguishes between declaratively matched and imperatively called rules.

However, the benefits of an explicit runtime model for the execution is not considered by these approaches. Instead low-level execution engines are employed which aggravates debugging and understanding of model transformation. By the process-oriented specification and execution of model transformations using Transformation Nets, we combine the benefits of imperative and declarative approaches not only for the specification of transformations, but also for their execution by using CPNs themselves as a transformation engine, which is currently not supported by hybrid approaches.

Petri Nets employed for Model Transformations. The relatedness of Petri nets and graph rewriting systems has induced some impact in the field of model transformations. Especially in the area of graph transformations some work has been conducted that uses Petri nets to check formal properties of graph production rules. Thereby, the approach proposed in [13] translates individual graph rules into a place/transition net and checks for its termination. Another approach is described in [5], which applies a transition system for modeling the dynamic behavior of a metamodel.

Compared to these two approaches, our intention to use Petri nets is totally different. While these two approaches are using Petri nets as a back-end for automatically analyzing properties of transformations by employing place/transition nets, we are using a variant of CPNs to specify transformations and to foster debuggability and understandability of transformations. In particular, we are focussing on how to represent model transformations as Petri Nets in an intuitive manner. This also covers the compact representation of Petri Nets to eliminate the scalability problem of low-level Petri nets. Finally, we introduce a specific syntax for Petri Nets used for model transformations and integrate several high-level constructs, e.g., inhibitor arcs and pages, into our language.

7 Conclusions and Future Work

In this paper we showed how Transformation Nets, which are a variant of CPNs, can be used to specify and execute model transformations. Thereby we showed how metamodels, models and transformation logic can be expressed in Transformation Nets, providing an integrated view on all transformation artifacts involved as well as a dedicated runtime model to foster debugging. Finally we showed how concepts of Transformations Nets could be expressed in terms of CPNs.

Currently, we are working on a automated translation of Transformation Nets to standard Colored Petri Nets in order to make use of efficient execution engines of third party vendors. This furthermore allows us to use Petri net properties for analyzing and verifying model transformations which is another direction we are going to strive for future work, e.g., liveness properties to check if a transformation finishes.

Acknowledgements

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

References

1. Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Core specification, 2006.
2. Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Final Adopted Specification, 2007.
3. J. Bézivin. On the Unification Power of Models. *Journal on Software and Systems Modeling*, 4(2), 2005.
4. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3), 2006.
5. J. de Lara and H. Vangheluwe. Translating Model Simulators to Analysis Models. *11th Int. Conf. on Fundamental Approaches to Software Engineering*, 2008.
6. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools*. World Scientific Publishing Co., 1999.
7. R. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. *29th Int. Conf. on Software Engineering*, 2007.
8. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Monographs in Theoretical Computer Science, Springer, 1992.
9. F. Jouault and I. Kurtev. Transforming Models with ATL. *Model Transformations in Practice Workshop @ MODELS'05*, 2005.
10. F. Legler and F. Naumann. A Classification of Schema Mappings and Analysis of Mapping Tools. *12. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web (BTW'07)*, 2007.
11. N. Mulyar and W. M. von der Aalst. Towards a pattern language for colored petri nets. In K. Jensen, editor, *Proceedings of Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 39–59, Aarhus, Denmark, 2005.
12. T. Reiter, M. Wimmer, and H. Kargl. Towards a runtime model based on colored Petri-nets for the execution of model transformations. *3rd Workshop on Models and Aspects @ ECOOP'07*, 2007.
13. D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer. Termination Analysis of Model Transformations by Petri Nets. *3rd Int. Conf. on Graph Transformations*, 2006.

Composition of Concerns Using Reference Nets

João Paulo Barros[1,2], Isabel Sofia Brito[1], and Elisabete Soeiro[3]

[1] Instituto Politécnico de Beja,
Escola Superior de Tecnologia e Gestão, Beja, Portugal
[2] UNINOVA-CTS, Monte de Caparica, Portugal
[3] Escola Mário Beirão, Beja, Portugal
jpb@uninova.pt, issb@estig.ipbeja.pt, elisabete.soeiro@gmail.com

Abstract. Crosscutting concerns are responsible for producing tangled representations that are difficult to understand, maintain, and evolve. Aspect-oriented software development aims at addressing those crosscutting concerns, known as aspects, by providing means for their systematic identification, separation, representation, and composition. This paper focuses on the composition activity of the aspect-oriented requirements engineering approaches, by proposing the use of *reference nets*, a class of high-level Petri nets, for the specification of concerns and match points as executable high level models. To that end, we propose a set of reusable reference net models for the automatic composition of concerns, at the requirements level, in order to create executable graphical models amenable to further specification. The resulting models have a precise semantics and can be graphically simulated using the renew tool. This simulation of the composed concerns allows their validation and offers new insights to the modeller, as it offers a readable and detailed view of the multiple inter-dependencies. The proposed approach is applied to a composition of concerns, with four different composition operators, which is translated to an executable Petri net model.
Key words: requirements analysis, aspect-oriented development, net composition, Petri nets, renew tool.

1 Introduction

An effective requirements engineering approach should reconcile the need to integrate functional and non-functional concerns with the need to modularize those that are crosscutting [1, 2]. A concern refers to a property which addresses a certain problem that is of interest to one or more stakeholders and which can be defined by a set of coherent requirements. A concern can be functional or non-functional [3, 4]. This paper builds on the second and third authors previous work on Aspect-Oriented Requirements Engineering (AORE) [3, 5, 6], focusing on the concern composition activity of the model presented therein. There, the composition of concerns was informally defined through a minimum set of operators. The goal is to provide new abstractions and composition mechanisms to modularize and compose such concerns during requirements engineering. This paper extends the work already presented in [3, 5, 6] by proposing a novel process

for the composition of concerns based on a set of operators and their translation to Petri nets.

Petri nets, and especially high-level nets, offer a highly expressive language for the definition of the needed operators, namely parallel, choice, sequence, and break. Here, the preemption semantics of the latter is modelled through the use of transition synchronization among models. Each operator models a way to compose two concerns and is specified as a Petri net model. These models are used as templates whose instances are composed in a scalable, readable, and executable way. This net composition is automatic and uses simple abstract models for specifying each concern. The resulting composed model is already an executable net model, but can also be enhanced with more detailed specifications for each concern.

This paper is organized as follows. Section 2 introduces the background on the AORE model and Petri nets, as well as the the motivation for the presented work. Section 3 presents our proposal, namely the Petri net models for each operator and their application to a case study. Section 4 discusses the related work and Section 5 draws some conclusions and identifies future tasks to be developed.

2 Motivation and background

The present work aims at providing the support for the composition of concerns, as well as for the creation of executable specifications for functional concerns, in a reusable and scalable way. To that end, we propose the use of Petri net models, more specifically reference net models. **Reference nets** have a precise semantics and are supported by a tool — the renew tool [7, 8] — that allows interactive and automatic simulations. Next, we briefly present the AORE approach and reference nets.

2.1 Composition of concerns

The AORE approach is composed of three main tasks, all related to concerns: (1) identification, (2) specification, and (3) composition [5]. In this paper, we focus on the support for the third task: **composition of concerns**. Hence, we assume that crosscutting and non-crosscutting concerns (untangled concerns) were already identified and textually specified. Therefore, we already know which concerns to compose. The result is a Petri net model, composed of several inter-related nets.

In the composition of concerns we incrementally compose a set of concerns until the whole system is obtained. Each composition takes place in a **match point** in the form of a **composition rule**. A match point tells us which concerns should be composed and is identified when one or more required concerns may need to be composed together with a base concern [3]. If a concern is composed in more than one match point, this concern is crosscutting. A composition rule

shows how a set of concerns can be woven together by means of a predefined operator. The rule takes the form `<LeftOperand> <Operator> <RightOperand>`, where the operands can be a concern or a sub-composition (which is another composition rule) and `<Operator>` represents the operator. The four operators we have chosen were inspired by the following LOTOS operators: full synchronization, choice, enabling, and disabling [9]. Here, we represent them by "`|`", "`^`", "`>>`", and "`>=`", respectively. We call them "parallel", "choice", "sequence", and "break". Informally, their semantics are the following:

Parallel (denoted by `T1 | T2`): refers to the parallel operator and means that the behaviour of `T1` and the behaviour of `T2` have synchronous start and end points. It represents concurrent execution of concerns.

Choice (denoted by `T1 ^ T2`): refers to the choice operator and means that only one of the concerns will be satisfied (`T1` or `T2`).

Sequence (denoted by `T1 >> T2`): refers to a sequential composition and means that the behaviour of `T2` begins if and only if `T1` terminates successfully.

Break (denoted by `T1 >= T2`): means that `T2` interrupts the behaviour of `T1` when it starts its own behaviour. This allows the representation of interruptions.

All these four operators are binary and compoundable as each returns a concern. Hence, a composition rule can be defined based on other composition rules, separated by one of the above operators.

Next, we briefly present Petri nets and the specific class of Petri nets we will be using, named *reference nets*.

2.2 Petri nets and *reference nets*

Petri nets (e.g. [10]) are a well-known set of graphical specification languages. A Petri net is a directed graph with two types of nodes named *places* and *transitions*. Places are usually drawn as circles or ellipses, and transitions by simple lines or rectangles. Each arc must connect one place to one transition or one transition to one place. Places can contain one or more tokens. These tokens constitute the place marking. Hence, the system state is modelled by the set of all markings, also named net marking. The marking changes when transitions fire. As the net semantics allows any number of enabled transitions to fire, Petri nets are able to graphically model concurrency and synchronization in a visual way.

A Petri net is said to be low-level if the marking in each place can be either empty or modelled by a positive integer value. If the number of tokens is small, they are typically pictured as black dots inside the place. Then, the net execution, i.e. the sequence of net firings, can be seen as tokens "flowing" around the net, something which is usually called the "token game". Yet, low-level net models can rapidly become too large and too difficult to use. **Coloured Petri nets** [11, 12] allow tokens of complex data types (the "colours") and the specification of functions that operate on the respective token values, thus adding data

processing capabilities to Petri nets. Together with the use of one or more of the known structuring mechanisms for Petri nets (e.g. [13]), it becomes possible to achieve huge reductions in the size of the models. As the definition of functions and data types assume the use of a programming language, the modeller can freely balance the use of textual or graphical elements in reusable and modular models. Hence, *Coloured Petri nets* with structuring mechanisms are a highly expressible graphical specification language, which have already been applied to model large systems (e.g. [12]).

Here, we will be using *reference nets* [7, 8] which are a well-supported example of *Coloured Petri nets* extended with the *nets-within-nets* structuring approach. There, the tokens can be not only data type values but also other nets. Hence, it becomes possible to compose nets in a hierarchical way where each token is an abstraction for a subnet. This possibility is used here to build a hierarchical graphical model that mimics an expression evaluation tree: each node in the tree is a net model. The tree leaves model primitive concerns, while each of the remaining nodes model one operation, i.e. a composition rule. Finally, as presented latter, a top level model is added as the new root of the tree. The modelling of all the presented operators as *reference nets* is presented in Section 3.2.

3 Composition of concerns using Petri nets

This section presents the composition of concerns using an example based on the Washington subway system. The example is first presented using an algebraic specification and then using the proposed Petri net models.

3.1 The EnterSubway case study

The presented `enterSubWay` model is based on the Washington subway system as in [5]:

> "To use the subway, a client has to own a card that must have been credited with some amount of money. A card is bought and credited in special buying machines available in any subway station. A client uses this card in an entering machine to initiate her/his trip. When s/he reaches the destination, the card is used in an exit machine that debits it with an amount that depends on the distance travelled. If the card has not enough credits the gates will not open unless the client adds more money to the card. The client can ask for a refund of the amount in the card by giving it back to a buying machine."

Based on the requirements given above, we identify the following concerns: "buyCard", "enterSubway", "exitSubway", "refundCard", and "creditCard". Functional concerns, as these, are usually not too difficult to identify. A closer look at the requirements can also suggest some non-functional requirements. For example, the text "special buying machines available in any subway station"

suggests that "availability" is important. In fact, from our knowledge of the real world, the system should be available eighteen hours a day. Another concern we should consider is "responseTime", since the system needs to react in a short amount of time to avoid delaying passengers.

Still other concerns can be identified based on some requirements catalogues, such as [4]. For example, as the system can simultaneously be used by many passengers, then "multiaccess" is an issue that the system needs to address. Other concerns identified based on this catalogue are: "accuracy", "security", "compatibility", and "faultTolerance". These concerns are specified text templates according to the AORE approach [3]. After the specification, the concerns are composed using match points, as presented in Section 2.1. This can be better demonstrated by a matrix for the identification of match points, as illustrated in Table 1. A cell is filled with a $\surd$ if a given *required concern* is required by the *base concern*. For example, for the match point **enter**, for the "enterSubway" base concern, the table shows the list of required concerns that must be composed with it, namely "validateCard", "errorCard", "integrity", "availability", "responseTime", "accuracy", "faultTolerance", and "multiaccess".

Match Points	Base Concerns	Required Concerns								
		validateCard	errorCard	integrity	availability	responseTime	accuracy	compatibility	faultTolerance	multiaccess
buy	buyCard			√		√	√		√	
enter	enterSubway	√	√	√	√	√	√		√	√
exit	exitSubway	√	√	√	√	√	√		√	√
refund	refundCard	√		√	√	√	√		√	
credit	creditCard	√		√	√	√	√	√	√	√
validate	validateCard						√			
none	errorCard									
...	...	...	...	...	...	...	...	...	...	...

Table 1. Identification of match points

Next, we exemplify the composition of concerns with two match points: `enter` and `buy`. The compositions rules for each match point are specified using the presented operators as follows, where parentheses are used to specify priorities:

```
enter = ( (availability | multiaccess) >>
          ( (validateCard >> (errorCard ^ enterSubway) ) |
            responseTime | accuracy ) >>
         integrity)
         >= faultTolerance

buy = ( (buyCard | responseTime) >> accuracy >> integrity )
      >= faultTolerance

enterBuy = enter | buy
```

These composition rules express the order in which each concern must be satisfied. For example, in match point `buy`, `buyCard` and `responseTime` are executed in parallel and, only after both have finished, the concerns `accuracy` and `integrity` are executed in sequence. Any of these four concerns can be interrupted by the `faultTolerance` concern. This interruption can also happen in match point `enter`. Finally, both match points are composed in parallel by the composition rule `enterBuy = enter | buy`. The goal of this composition is to illustrate the identification of crosscutting concerns, i.e. when a concern is enabled in more than one match point, for example `integrity` and `responseTime`.

Next, we briefly present *reference nets* together with the proposed net models.

3.2 Graphical and executable models for concerns composition

We now present a reference net model for each of the presented operators, together with a brief overview of the *reference nets* syntax and semantics. Each of these models is designed to be used as a template. Model composition is achieved through the instantiation of the respective template parameters. Neither *reference nets* neither the renew tool provide support for net templates. Hence, templates are in fact invalid *reference nets*. They become valid *reference nets* after text replacement, based on the presented algebraic representation. This is automatically done by a simple script available at http://www.estig.ipbeja.pt/~jpb/aore2renew. The presented match points `buy`, `enter`, and `enterBuy` are examples of inputs to the script. Based on the algebraic specification and using the templates presented next, the script generates a set of *reference nets* in the renew tool format. Together, they form a reference net model.

Now, we present the net templates related to the net operands. Then we present the net templates for each of the four operators.

The operand net templates We start by presenting the `Top` net template in Fig. 1a. This is where the net model simulation begins. First, it creates a set of pairs (*concern_name*, *creator_instance*). To that end transition `init` starts by creating an `HashMap` Java™ object. Then, for each concern name in the model, it generates a creator net instance and stores the respective pair in the `HashMap` object. The code for this operation binds the `START_CREATORS` template parameter. The *(concern name, creator net instance)* pairs will allow the use of the concern creator instance given the respective name, as concern net instances are created by the corresponding creator instance. Then, this set of pairs is passed to the `start` transition, which uses it to get the creator instance (`c = cs.get(CREATOR)`) for the top level concern. Then, it uses the creator instance to create the net instance `x` for the top level concern (`c.create(cs, x)`). Note that the set of creators `cs` is passed to the creator `c` and the instance `x` is returned. Now, the top level concern can be executed by the firing of transition `isEnabled` until it is destroyed by transition `end`.

Note that the `HashMap` object reference is a net token. This token contains pairs of names and references to other net instances. The `enabled` places also

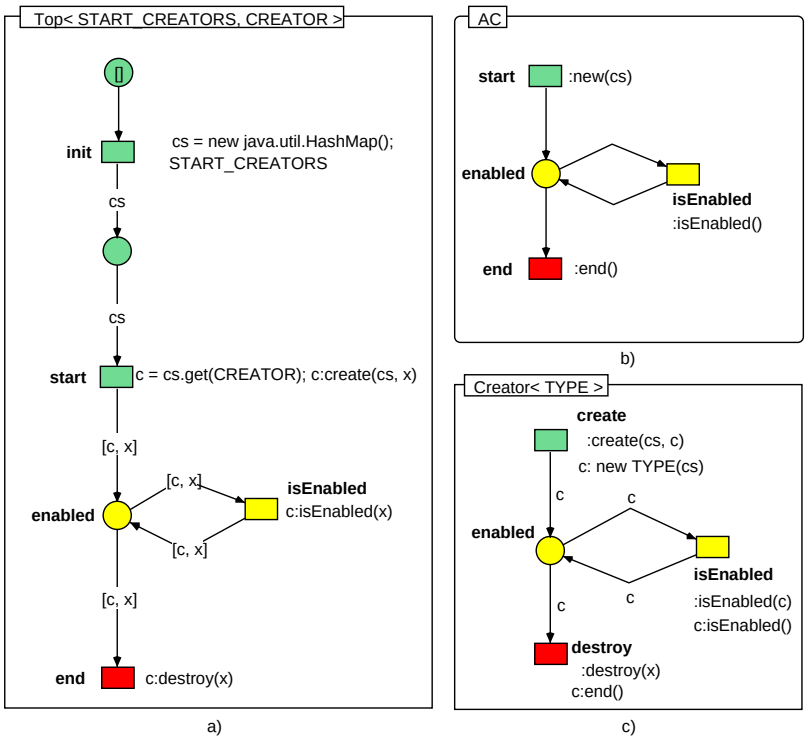


Fig. 1. Reference net models for the a) top level model; b) an abstract concern; and c) a creator model

contain references to other nets. Hence, the model and the respective simulation are strongly based on the *nets-within-nets* structuring approach — the main characteristic of *reference nets*.

The `AC` model in Fig. 1b provides a generic abstract specification for any concern, the **abstract concern**. The *abstract concern* has a `start` and an `end` transition. After started, it can fire the test transition `isEnabled`.

Although each concern is automatically modelled as an abstract concern, it can be specified at the intended level of abstraction by taking advantage of the *reference nets* language, which also allows the use of the Java™ programming language. Hence, for functional concerns, the generated instance of the *abstract concern* model can be replaced by a more detailed one where place `enabled` and transition `isEnabled` are refined. Yet, each of the added transitions should still contain the `:isEnabled` *uplink* which forces the transition enabling by some transition in the upper level concern: the one that has the present concern as an operand. An example of this kind of functional concern model is available, for the `enterSubway` concern, at http://www.estig.ipbeja.pt/~jpb/aore2renew.

For each CN concern we create a net AC_CN and also a net C_AC_CN. The latter is responsible for creating all instances of net AC_CN. The C_AC_CN net is the result of instantiating the creator template in Figure 1c with CN.

Creators are singleton net instances: for each CN concern there is a single C_AC_CN instance that creates all AC_CN instances. Each of these instances is an execution of concern CN. Hence, the level of crosscutting of a given CN concern is given by the quantity of executions of the same CN, i.e. the number of simultaneous executions of enabled AC_CN instances. The references to all AC_CN instances are stored in place enabled in the respective C_AC_CN net instance (see Fig. 1c).

Therefore, to execute a single primitive concern, we generate an instantiation for each template instantiation in Fig. 1. For example, if our concern is named availability we would get three nets each one with one instance, i.e. three singletons:

1. A new template instance of model AC, named AC_availability with one net instance.
2. A new template instance of model creator, named C_AC_availability with one net instance.
3. A new template instance of model Top, named Top_availability with one net instance.

Naturally, to model a single primitive concern, we could also use a single net model. This composition is only interesting when we do specify composition rules, as previously presented for the enterBuy composition. The used operators are also modelled by net templates. These are now presented.

The operator net templates Now we turn to the net templates that model the four operators: *parallel* (|), *choice* (^), *sequence* (>>), and *break* (>=).

Figure 2 shows how to model the parallel (OP_PAR) and choice (OP_CHOICE) operators as reference net models. Both are templates: they become *reference nets* after replacing the C1 and C2 template parameters with the names of the operand concerns. Therefore, a concrete instantiation of the OP_PAR model will replace C1 and C2 with the names of other concerns acting as terms.

Regarding the OP_PAR model, and according to the reference net semantics, the net execution starts by creating a net instance based on the instantiation of the OP_PAR template. Its execution starts with the firing of transition start, which gets two net instances from the creators of each operand concern: one net instance x1 from the creator instance c1 and another net instance x2 from the creator instance c2. The two net instances, x1 and x2, are executed concurrently with the OP_PAR net instance, in particular transitions isEnable1 and isEnable2 can fire in each step in a non-deterministic way. Each transition in a reference net is enabled according to the known *Coloured Petri net* semantics plus the restrictions due to the use of **synchronous channels** between transitions. These are briefly presented in the following paragraphs.

Each *synchronous channel* establishes a synchronous inter-dependency between two or more net instances. It can be seen as a form of transition fusion, where two or more transitions fire together. The *synchronous channels*

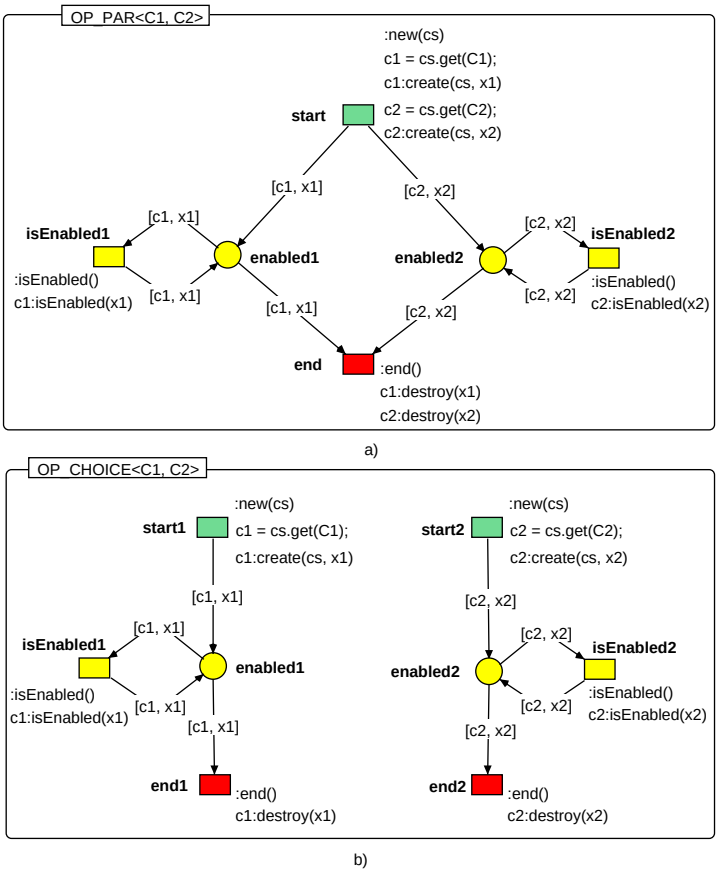


Fig. 2. Reference net models for the parallel and choice operators

in *reference nets* are similar to method calls in classical object-oriented languages. Hence, they have two parts: (1) a *downlink* specifying the net instance and the synchronous channel name together with optional parameters (e.g. `c:destroy(x)`), which is similar to a method call; (2) an *uplink* specifying the synchronous channel name and the optional parameters list (e.g. `:destroy(x)`), which is similar to a method definition. A single transition can have more than one *downlink*. This is the case for transition `end` in the `OP_PAR` model in Fig. 2, which has two: `c1:destroy(x1)` and `c2:destroy(x2)`. This transition can only fire together with the transitions inside the `c1` and `c2` net instances with a `:destroy(x)` *uplink* (see Fig. 1c).

When transition `start` fires, with instantiated `C1` and `C2` concern names, the tuples `[c1, x1]` and `[c2, x2]` are put in places `enabled1` and `enabled2` thus enabling transitions `isEnable1` and `isEnable2`, which model the execution of concern `x1` and concern `x2`, respectively. They are executed while transition `end` does not fire, which can only happen if both concerns (`x1` and `x2`)

can both end. This synchronization is guaranteed by the *synchronous channels* `c1:destroy(x1)` and `c2:destroy(x2)` that synchronize with the respective `destroy` transitions in creators `c1` and `c2` (see Fig. 1c). Then, using the respective destroy transitions, these synchronize with the `end` transitions in the respective concern nets.

Transitions `isEnable1` and `isEnable2`, have an `isEnabled` *uplink*. This guarantees that the concerns in the `OP_PAR` net can only fire if the supermodel (the respective creator instance) allows them to. If those transitions cannot fire, the model will eventually deadlock if transition `end` also cannot fire. At that point, as the supermodel has no references to it, the `OP_PAR` net instance is a target for garbage collection [8].

The `OP_CHOICE` in Fig. 2 is similar regarding the use of composition through net instances and the use of synchronous channels, but with the following important difference: either concern `C1` or `C2` is created and processed, but not both. For this reason, each one must be able to end independently of the other. The choice between the two is non-deterministic, but could easily be made dependent on guards (logical conditions) attached to the respective `start1` and `start2` transitions. This can be added by the modeller when creating a more detailed model.

Figures 3a and 3b show the net model templates for the sequence and break operators, respectively. Their semantics should now be straightforward. The `OP_SEQ` model is a sequential execution of two concerns, where the second starts when the first terminates normally. In the `OP_BREAK` model, transition `end1` ends concern `C1`. Also, when `C1` is being executed, it can be interrupted at any time by the `break` transition, which also starts concern `C2`. In this case, `C1` does not end normally and becomes deactivated as transition `isEnable1` becomes disabled. Hence, it will not enable any *isEnabled* transition, with an `isEnabled` uplink (`:isEnabled()`), in the `C1` net. The `C1` concern will eventually deadlock and, as there will be no more references to it, it will be garbage collected.

3.3 Translation to the executable net model

The translation from the algebraic specification to a Petri net model mimics the construction of an evaluation expression tree. Fig. 4 shows the structure of the model generated from the algebraic specification in Section 3.1, where each rectangle represents a net class (a template instance) in the reference net model and each arrow specifies a "uses" dependency.

In fact, the Petri net model structure is isomorphic to the evaluation expression tree with the difference that it adds a `Top` model instance. More specifically, the following reference net models are generated:

- A top level net, which results from the instantiation of the `Top` template in Fig. 1a with the tree root operator net creator;
- For each distinct leaf node in the tree, two net models are generated: (1) an instance of the creator model template in Fig. 1c; (2) an instance of the abstract concern model template in Fig. 1b; The former will create the net instances of the latter.

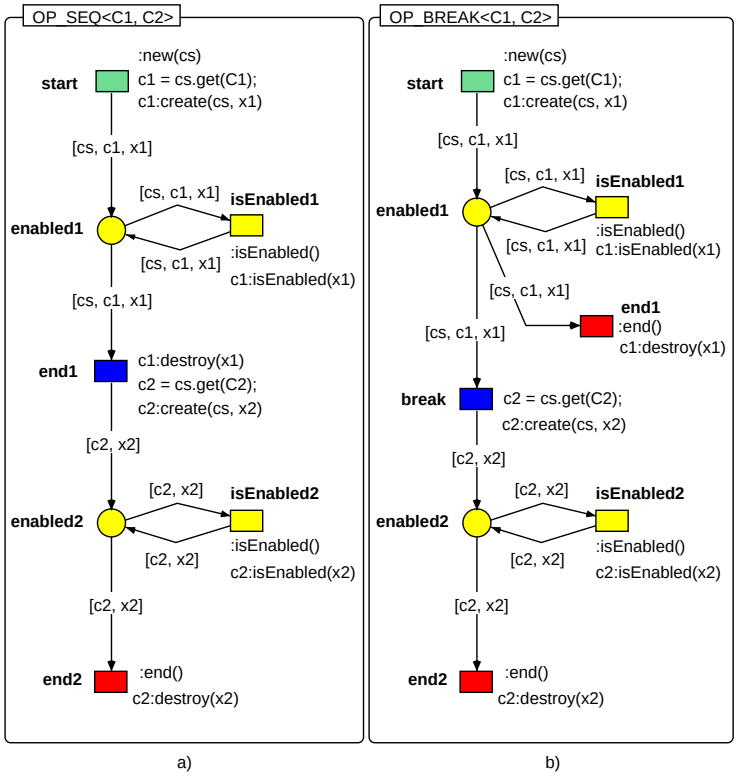


Fig. 3. Reference net model for the sequence operator

– For each non leaf node — the operator nodes — two net models are generated: (1) an instance of the respective operator model template in Figures 2, 3a, and 3b; (2) an instance of the creator model template in Fig. 1c, which is able to create operator net instances.

Basically, two *reference nets* are generated for each node in the tree, except that the repeated occurrence of a given concern (a crosscutting) reuses the same two nets. For example, for the `faultTolerance` concern only two nets are generated: the `C_AC_faultTolerance` and the `AC_faultTolerance`. Hence, the `C_AC_faultTolerance` rectangle has two incoming arrows. The multiple occurrence, specified by multiple tokens in place `enabled`, is modelled by multiple net instances of net `AC_faultTolerance`, one instance for each occurrence.

The `Top` model is used to build all the creator singletons — in transition `init` — putting them all in the `HashMap` object. The respective code text instantiates the `START_CREATORS` parameter of the `Top` net template. Then, transition `start` initiates the top level concern (`c:create(cs, x)`)using the respective creator, obtained from its name `CREATOR` (`c = cs.get(CREATOR)`).

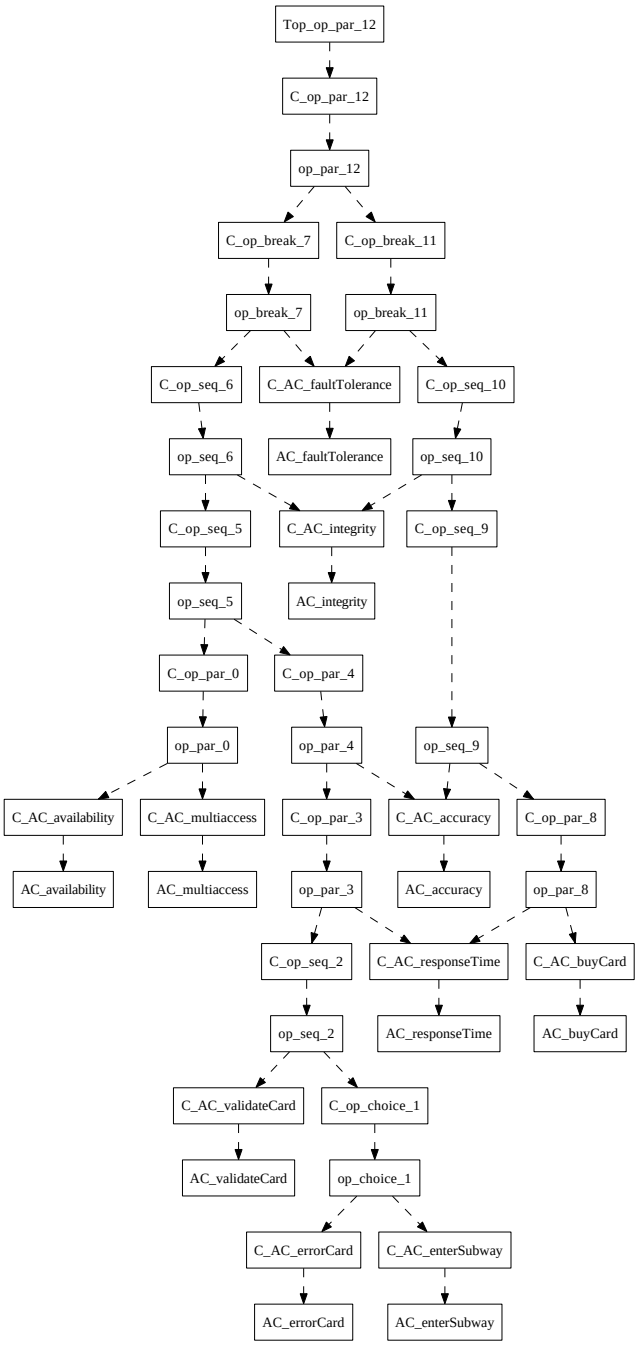


Fig. 4. Structural dependencies in the generated model for the `enterBuy` composition

In the end, we achieve a fully hierarchical and executable model that guarantees an abstract view for all concerns and respective compositions. As already stated, this model provides a composition of all concerns and a starting point for the creation of a more detailed executable specification for functional concerns.

4 Related Work

The work presented in this paper is focused on aspectual composition, i.e. on composition of crosscutting concerns in aspect-oriented requirements engineering. As already presented, the ability to compose crosscutting and non-crosscutting requirements allows the analysis of their interrelationships and, if needed, a more complete view of the requirements. Therefore, this section presents a list of aspect-oriented requirements engineering approaches that support this composition.

The EA-Miner tool [14] supports identification and separation of concerns and crosscutting concerns, i. e. aspects, as well as their relationships at the requirements level, helping to reduce the effort to accomplish these tasks. The tool is web based and utilizes natural language processing (NPL) techniques, such as part-of-speech and semantic tagging, to reason about the properties of the concerns and their relationships. So, these techniques structure the concerns from requirements documents into specific aspect-oriented requirements models, such as viewpoints and aspects. After this, the tool composes aspects and viewpoint requirements using composition rules based on operators. So, a set of composition rules is defined for each aspect, i. e. the approach only proposes composition rules that define how an identified aspect is composed with viewpoints, showing the effect of the aspect on the viewpoint.

Moreira et al. propose a multi-dimensional approach for the separation of concerns in requirements engineering as well as trade-off analysis of the requirements specification from such a multi-dimensional perspective [15]. This approach uses concern projections to specify the influence of a given concern on other concerns. The materialization of these projections is accomplished by defining a set of composition rules (compositional intersections help reduce the number of required compositions), one for each projection. These rules operate at fine granularity, i.e., at individual requirements level and not just at concerns level. After specifying the various projections with the aid of composition rules, identification and resolution of conflicts among the concerns is carried out.

AOV-graph is an extension to V-graph models to avoid the tangling and scattering of crosscutting concerns in requirements models [16]. This approach uses goal models and the concepts defined in aspect-oriented languages to provide separation, composition and visualization of crosscutting concerns to facilitate their modelling and the traceability between them. AOV-Graph wants to model sets of requirements separately and to offer a way to model the relationships between them. Crosscutting relationships are specified using pointcuts and operators, advices and intertype declarations. These crosscutting relationships are identified, either because a requirement impacts on many points, or because it

is important to keep one requirement separate from the others. Furthermore, AOV-Graph offers different views originating from the composite model.

Generally speaking, all the approaches propose an aspectual composition technique, based on the specification of crosscutting relationships or projections, using natural language and composition rules with operators. Our approach differs from the others by offering a set of operators with well-known semantics that can be executed using an established graphical language with a precise semantics. The semantics of the operators, namely, sequence, choice, parallel, and interruption (break) are amenable to a readable graphical specification using Petri nets. In fact, they correspond to well-known Petri net patterns, making Petri nets an obvious choice for the respective specification (see [17] for a systematic review of control-flow patterns).

Additionally, the use of *reference nets*, allows compact and modular graphical representations that can be enriched through the use of the well-known Java™ programming language. The resulting models are supported by the renew tool, available free of charge.

5 Conclusions and Future Work

This paper focuses on the composition activity of the AORE approach, by proposing a Petri net based approach to compose concerns at the requirements level. The concerns composition is specified using a simple textual language that is automatically translated to a skeleton Petri net model. This model offers a starting point for the modeller. From this generated model, the modeller can specify each abstract concern with greater detail or replace it with a previously specified one.

The presented approach allows a separation between the composition of concerns and the specification of each concern, improving understand ability and maintainability. Together with the proposed automatic composition, the presented work can be readily extended with additional operators while guarantying the reuse of existing concerns models. It also provides a basis for the adoption of a model-driven development at the requirements level allowing the creation of readable graphical and executable models with a precise semantics.

As future work it will be useful to enrich the composition rules with the possibility to use operators supporting a finer level of granularity, namely by also allowing structured horizontal compositions together with the presented vertical ones. Regarding the renew tool, a useful addition would be the availability of advanced interactive support for querying the generated log. Currently, this log can be searched and used to ease interactive simulation, but has no associated query language.

References

1. Ossher, H., Tarr, P.: Hyper/J: Multi-Dimensional Separation of Concerns for Java. In: ICSE '00: Proceedings of the 22nd International Conference on Software Engineering, New York, NY, USA, ACM (2000) 734–737

2. Rashid, A., Moreira, A., Araújo, J.: Modularisation and Composition of Aspectual Requirements. In: AOSD '03: Proceedings of the 2nd International Conference on Aspect-oriented Software Development, New York, NY, USA, ACM (2003) 11–20
3. Brito, I.S., Moreira, A.: Towards an Integrated Approach for Aspectual Requirements. In: RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference, Washington, DC, USA, IEEE Computer Society (2006) 334–335
4. Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Volume 5 of The Kluwer International Series in Software Engineering. Springer (October 1999)
5. Soeiro, E., Brito, I.S., Moreira, A.: An XML-Based Language for Specification and Composition of Aspectual Concerns. In Manolopoulos, Y., Filipe, J., Constantopoulos, P., Cordeiro, J., eds.: ICEIS 2006 - Proceedings of the Eighth International Conference on Enterprise Information Systems: Databases and Information Systems Integration. (2006) 410–419
6. Soeiro, E.: Especificação e Composição de Requisitos Aspectuais. Master's thesis, Faculdade de Ciências da Universidade Nova de Lisboa (2007)
7. Kummer, O., Wienberg, F., Duvigneau, M., Schumacher, J., Köhler, M., Moldt, D., Rölke, H., Valk, R.: An Extensible Editor and Simulation Engine for Petri Nets: RENEW. In Cortadella, J., Reisig, W., eds.: Applications and Theory of Petri Nets 2004 25th International Conference, ICATPN 2004, Bologna, Italy, June 21-25, 2004. Volume 3099. Springer (jun 2004) 484–493
8. Kummer, O., Wienberg, F., Duvigneau, M.: RENEW – The Reference Net Workshop. http://www.renew.de/ (2006)
9. International Standards Organization: LOTOS — a Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. ISO 8807:1989 (1989)
10. Girault, C., Valk, R.: Petri Nets for Systems Engineering: A Guide to Modeling, Verification, and Applications. Springer-Verlag (2003)
11. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use - Volume 1 Basic Concepts. Springer-Verlag (1992)
12. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use - Volume 3 Practical Use. Springer-Verlag (1997)
13. Gomes, L., Barros, J.P.: Structuring and Composability Issues in Petri Nets Modeling. IEEE Transactions on Industrial Informatics **1**(2) (Maio 2005) 112–123
14. Sampaio, A., Chitchyan, R., Rashid, A., Rayson, P.: EA-Miner: a Tool for Automating Aspect-Oriented Requirements Identification. In: ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, ACM (2005) 352–355
15. Moreira, A., Rashid, A., Araujo, J.: Multi-Dimensional Separation of Concerns in Requirements Engineering. In: RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering, Washington, DC, USA, IEEE Computer Society (2005) 285–296
16. Silva, L.: An Aspect-Oriented Approach to Model Requirements. In Day, N.A., ed.: Proceedings of the of the Doctoral Consortium at the 13th IEEE International Requirements Engineering Conference, Paris, France, University of Waterloo – School of Computer Science, Technical Report CS-2005-23. (2005) 24–27 Available at http://www.cs.uwaterloo.ca/~nday/re05dc/re05dc-proceedings.pdf.
17. Russell, N., ter Hofstede, A., van der Aalst, W., Mulyar, N.: Workflow Control-Flow Patterns : A Revised View. BPM Center Report BPM-06-22 (2006) Available at http://www.workflowpatterns.com/documentation/documents/BPM-06-22.pdf.

Modeling and Prototyping of Concurrent Software Architectural Designs with Colored Petri Nets

Robert G. Pettit IV[1], Hassan Gomaa[2], and Julie S. Fant[1]

[1] The Aerospace Corporation,
Chantilly, Virginia, USA
{robert.g.pettit, julie.s.fant}@aero.org
[2] George Mason University
Fairfax, Virginia, USA
{hgomaa}@gmu.edu

Abstract. This paper describes an approach for constructing rapid prototypes to assess the behavioral characteristics of concurrent software architecture designs. Starting with a software architecture design nominally developed the using COMET concurrent object-oriented design method, an executable Colored Petri Net (CPN) prototype of the software architecture is developed. This prototype allows an engineer / analyst to explore behavioral and performance properties of a software architecture design prior to implementation. This approach is suitable both for the engineering team developing the software architecture as well as independent assessors responsible for oversight of the software architecture design.

Keywords: UML, rapid prototyping. coloured Petri-nets, concurrent software architecture.

1 Introduction

The increasing complexity of software-intensive systems, particularly with respect to concurrently executing software tasks, requires a thorough understanding of software architecture behavioral properties and tradeoffs among design decisions. Analyzing and understanding the concurrent behavior of a software architecture during the early design stages is imperative to the successful and cost-effective development of the system. To address this issue, we present an approach for constructing rapid prototypes of embedded systems to assess the behavioral characteristics of concurrent software architecture designs. The approach leverages software design nominally developed using the COMET concurrent object-oriented design method [1] and reusable Colored Petri Net (CPN) [2] templates and components to rapidly prototype a concurrent software architecture. The goal of the CPN prototype is to compare and assess concurrent software architecture behavior to determine if the software architecture is feasible before spending valuable resources on hardware purchase, development, testing, etc. This paper expands on previous work [2] by specifically focusing on rapid prototyping executable models using reusable CPN components and templates. The complete set of templates for this approach were defined in [2]. The resulting approach should provide the ability to quickly develop prototypes of software architecture.

1.1 Related Research

Prototyping the concurrent behavior of a system at design time is important to determine whether the system, with its set of concurrent tasks, behaves as desired both in terms of functionality and performance. If potential problems can be detected early in the life cycle, steps can be taken to overcome them.

Typical modeling and analysis methods include event sequence and queuing modeling [1, 3]; simulation modeling [4]; and scheduling analysis [3, 5, 6]. In recent years, there has been an increased effort to construct executable models of software designs and thus allow the logic of the design to be simulated and tested before the design is implemented. Existing modeling tools such as IBM® Rational® Rose® Technical Developer [7] and Ilogix Rhapsody [8] frequently use statecharts [9] as the key underlying mechanism for dynamic model execution. An alternative approach is to model concurrent object behavior using Petri Nets [10-14]. Our efforts [2, 14] have specifically focused on a Colored Petri Net (CPN) approach in which behavioral patterns are identified for objects in the software architecture and then modeled with CPN templates matching the behavioral patterns. We have chosen this approach since CPNs provide excellent modeling, analysis, and simulation capabilities for concurrent systems. Additionally, our approach supports independent assessments of the software architecture without requiring the software architect to adapt to a new paradigm.

2 Rapid Prototyping Approach

The purpose of this paper is to describe an approach leveraging executable CPNs for the rapid prototyping of concurrent software architecture designs. The purpose of the CPN prototypes proposed in this approach is to simulate the concurrently executing software tasks and to enable analysis and understanding of the concurrent behavior of a software architecture during the early design stages.

The proposed rapid prototyping approach has four major steps that are: 1) Develop the platform independent software architecture 2) Create the platform specific software architecture 3) Construct the CPN prototype 4) Execute and analyze the CPN prototype. Each step is described below in more detail.

2.1 Develop the Platform Independent Software Architecture Model

The first step in our approach is to develop the platform independent software architecture model (PIM). The purpose of the PIM is to capture the concurrent object behavior in the form of concurrent behavioral design patterns (BDP), which in subsequent steps will be mapped to CPN templates or components [2]. As discussed in previous work, each BDP represents the behavior of concurrent objects together with associated message communication constructs, and is depicted on a UML concurrent interaction diagram. Each concurrent object is assigned a behavioral role (such as I/O, control, algorithm) which is given by the COMET concurrent object

structuring criteria [1] and depicted by a UML stereotype. An example of a behavioral design pattern for an asynchronous device input concurrent object is given in Figure 1.

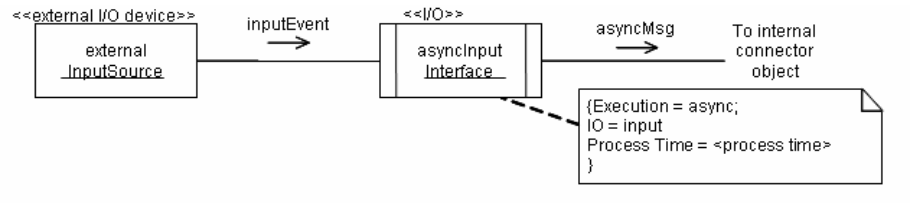


Fig. 1. Asynchronous input concurrent object behavioral design pattern

2.2 Create the Platform Specific Software Architecture Model

The second step in our rapid prototyping approach is to develop the platform specific software architecture model (PSM). The purpose of the PSM is to capture the performance characteristics of how the software architecture will perform if implemented on a specific platform. To enable fast construction of PSMs, the UML PIM model should be annotated with platform specific characteristics. This is quicker than creating a separate or external PSM model.

Platform specific characteristics and values can then be directly added to the UML software architecture model using a UML Profile such as the UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE) or UML Profile for Software Performance, Schedulability, and Time (SPT) [21-22]. The UML SPT Profile and UML MARTE Profile provide the ability to capture non-functional performance characteristics directly in UML models. For example, tagging a message in an interaction diagram with the UML SPT's <<paStep>> stereotype indicates that it is a step in sequence that uses resources. The specific platform specific values, such as execution time, can be captured in the stereotype's tags like PAdemand.

Performance values can be determined from published information about the platform, as well as through measurement. Note that multiple PSMs can be applied to a given architecture, supporting prototypes for tradeoff analyses. These models may also be constructed at varying levels of fidelity depending on available information. As the development efforts mature, so then can the prototypes of the architecture.

2.3 Construct the CPN Prototype

After the PSM is developed, an executable CPN prototype from the PSM can be systematically constructed. For each BDP in the PSM, a self-contained CPN template is required, which by means of its places, transitions, and tokens, models a given concurrent behavioral pattern. A set of existing reusable CPN templates can be found in [2]. For example, Figure 2 is the CPN template for an asynchronous device input concurrent object shown in Figure 1.

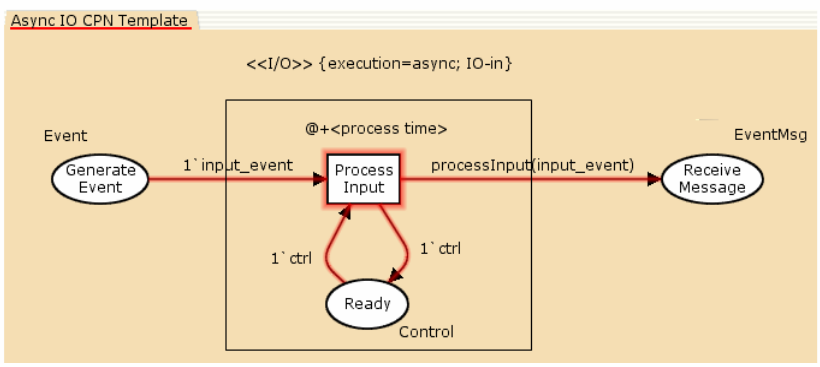


Fig. 2. Asynchronous input concurrent object CPN template

To instantiate the templates for each specific object, an analyst using our approach must provide a certain set of architectural parameters captured by following tagged values:

-Execution Type: passive, asynchronous, or periodic
-IO: input, output, or I/O
-Communication Type: synchronous or asynchronous
-Activation Time: periodic activation rate
-Processing Time: estimated execution time for one cycle
-Operation Type: read or write
-Statechart: for each «state dependent» object.

To illustrate pairing these architectural parameters with BDPs, refer to Figures 1 and 2. Figure 1 is an active object, "asyncInputInterface" that implements the I/O behavioral pattern as indicated by its stereotype. Furthermore, tagged types are used to capture specific architectural properties of the object, namely that it executes asynchronously; handles only input; and has a yet-to-be specified processing time of <process time>. The resulting CPN representation in Figure 2 reflects these parameters with the selection of an asynchronous, input-only CPN template and by setting the time inscription on the Process Input transition to @+<process time>.

This <process time> parameter is an estimate of the time required by the object to complete one activation cycle. This information can be obtained directly from UML MARTE annotations in the PSM. For example, process time can be found UML SPT's <<paStep>> stereotype in the PAdemand tag.

Since CPN templates provide only the basic behavioral pattern and component connections, they must be refined to provide application specific behavior.

To rapidly support construction of the prototype, we recommend using a reuse repository of CPN components. A CPN component is an elaborated CPN template for a commonly used object. For example, if a company commonly uses a specific sensor, a CPN component can be created for the software controller for the particular sensor. This CPN component can then be reused quickly in multiple different prototypes. Reusing CPN components will ultimately reduce the time it takes to construct the CPN prototypes. This is critical in rapid prototyping environments.

After all the BDPs in the PSM have an associated CPN templates or CPN components, the CPN templates and components are then interconnected via

connector templates to create a prototype of the software architecture. The CPN prototype is then executed using a CPN tool, thereby allowing the designer to analyze both the concurrent behavior of the CPN prototype, with a given external workload applied to it.

3. Case Study: Robot Control

We illustrate our rapid prototyping approach using a robot controller case study based on the Lego® Robotics Invention System™ (RIS), commonly known as Mindstorms™ [16]. The RIS platform was chosen based on the embedded nature of the platform with easily reconfigurable sensors and actuators [18].

The robot controller case study is an autonomous rover employing an infrared light sensor and two motors (actuators). The goal of the rover is to search an area for colored discs, while staying within the course boundary and avoiding obstacles. In this case study, the light sensor is the sole input sensor, responsible for detecting boundary markings, obstacle markings, and discs according to different color schemes. This case study was used as a term project for a graduate course on real-time embedded software engineering at George Mason University.

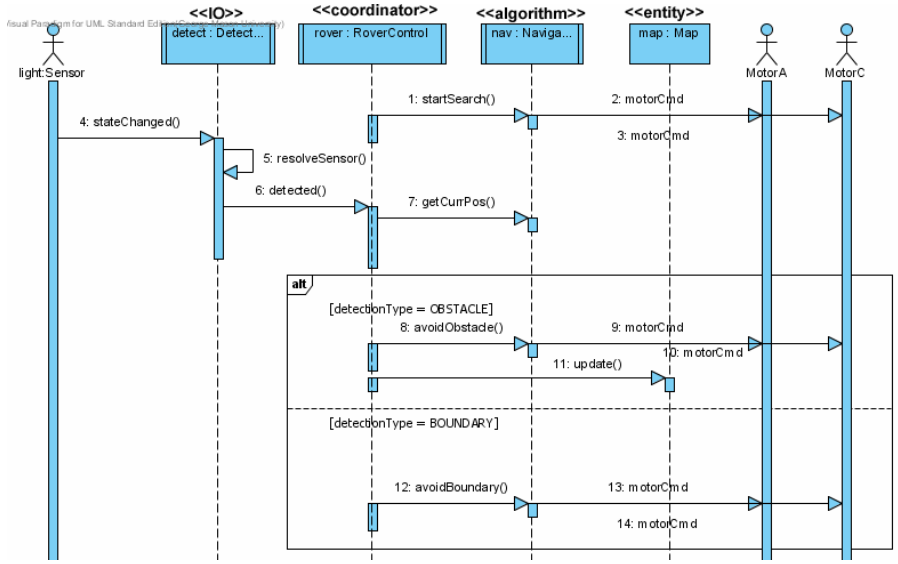


Fig. 3. Robot controller PIM interaction diagram

3.1 Robot Controller PIM

The architecture model for the autonomous rover is illustrated in Figure 3. In this particular scenario, we are interested in navigating the course; responding to changes from the light sensor; and taking the appropriate action based on the detection event.

In this design, there are three active, concurrently executing objects (detect, rover, and nav) and one passive object (map). External I/O objects (depicted as actors in Figure 3) are also shown for receiving light sensor input and for modeling output to the two motors. Following object structuring guidelines from the COMET method, each of the objects in the system is stereotyped according to the hierarchy previously shown in Figure 1. These stereotypes indicate the behavioral design pattern (BDP) implemented by each object. Further details about the behavioral properties are augmented with the architectural parameters as follows:

The detect, rover, and nav objects all operate asynchronously and have an Execution Type tagged value of "async". As the input interface for the light sensor, the detect object has an IO tagged value of "input". All messages between the active objects have a Communication Type tagged value of "synchronous", indicating synchronous, buffered communication. This particular design decision was made to decrease the risk of missing a boundary or obstacle detection event. Other design choices for this system would be to employ FIFO or priority queuing. The affects of these design decisions could also be analyzed using the techniques presented in this paper, but are not shown due to space limitations. Finally, the update() operation on the map object has an Operation Type tagged value of "writer".

Note that values for the Processing Time parameters are left unspecified at this point as we will set these parameters based on the PSM in the next section.

3.2 Robot Controller PSM

The next step in our approach is to create a platform specific model for the target platform. This is shown in Figure 4 using historical data and hardware specifications for the RIS system [18-20].

In this model, our rover is identified as the single node in the system and is based on the Robot Command eXplorer (RCX) platform. The RCX is the central component to any RIS system and houses the Hitachi H8 microcontroller with a 16 MHz CPU. Paired with the leJOS Java environment the execution speed of this CPU is documented to be an average of 1750 floating point operations per second (FLOPS).

Additionally, there are 16 KB of ROM and 28 KB of RAM available on the RCX of which, 17.5 KB of RAM are used by the leJOS operating system. The system clock resolution on the RCX, at 1ms, is longer than the time required for observed context switching between concurrent threads, thus the leJOS.overhead is set to zero. In our system, there are three physical devices attached: one light sensor at port S2 and two motors at ports A and C. Independent control of these motors is used to steer the rover; turning is achieved by rotating the left (Motor A) and right (Motor C) motors in opposite directions. Using historical data, the average detection latency of the light sensor was set at 10.3 ms, while the average output latency of the motors was set at 1 ms, which is set in the PAdelay tag.

It would also be useful to combine the information from the PSM with historical data on software size. This type of information is commonly maintained by software development organizations and, in our case, we will rely on average software sizes across the set of student projects. From this data, we discover the following:

«IO» objects average 19 instructions (in Java bytecode) per execution cycle and have an average size of 1,182 bytes.

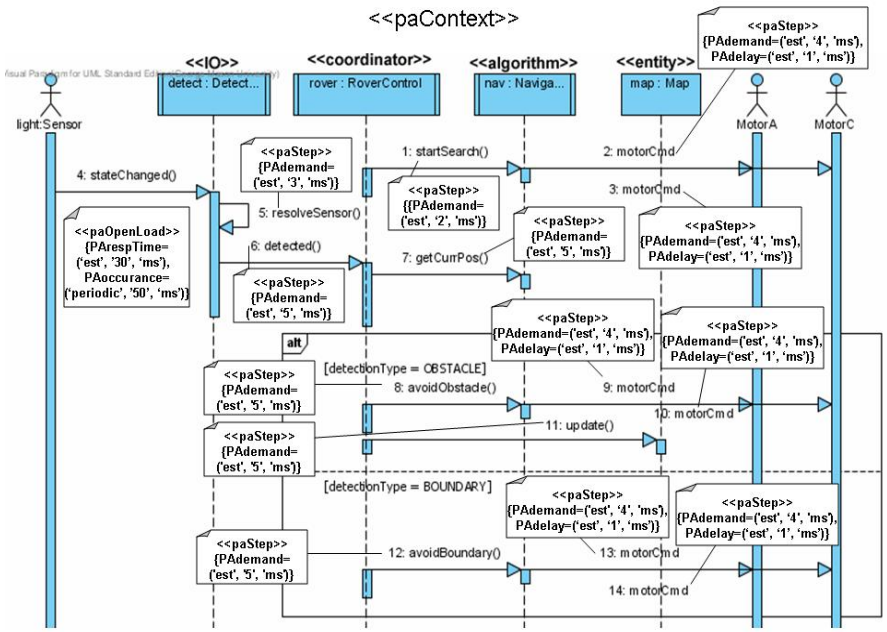


Fig. 4. Robot controller PSM interaction diagram

«coordinator» objects average 27 instructions and 2,722 bytes.
«algorithm» objects average 89 instructions and 1,015 bytes per algorithm.
«entity» objects average 1,400 bytes.

Now, using the combined historical sizing data and information from the rover, we can augment the PSM with this platform specific information. Prior to the CPU being available for performance measurement, an initial estimate of the execution time is computed by multiplying the estimated average number of instructions by the computational speed of the CPU (1750 FLOPS in our PSM example). These estimates are captured in the paDemand tag.

3.3 CPN Prototype

Using the above PSM design information, we can now begin to construct a Colored Petri Net (CPN) prototype of the software architecture [2]. Using our approach, we start with a context level model, capturing the system as a black box (transition) and external sensors and actuators represented as places. This model, allowing us to focus on the highest level of abstraction with observed inputs and outputs is shown in Figure 5.

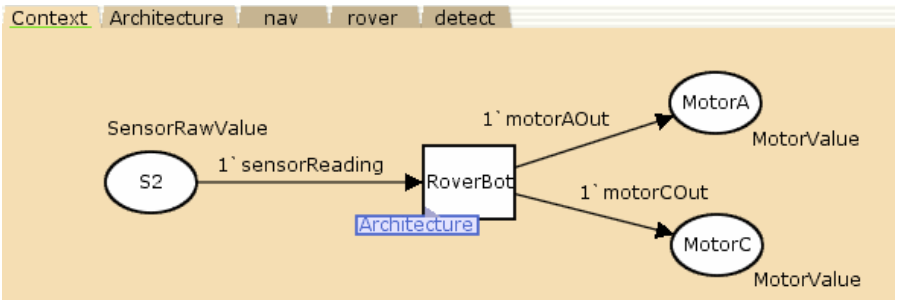


Fig. 5. Robot controller context level CPN

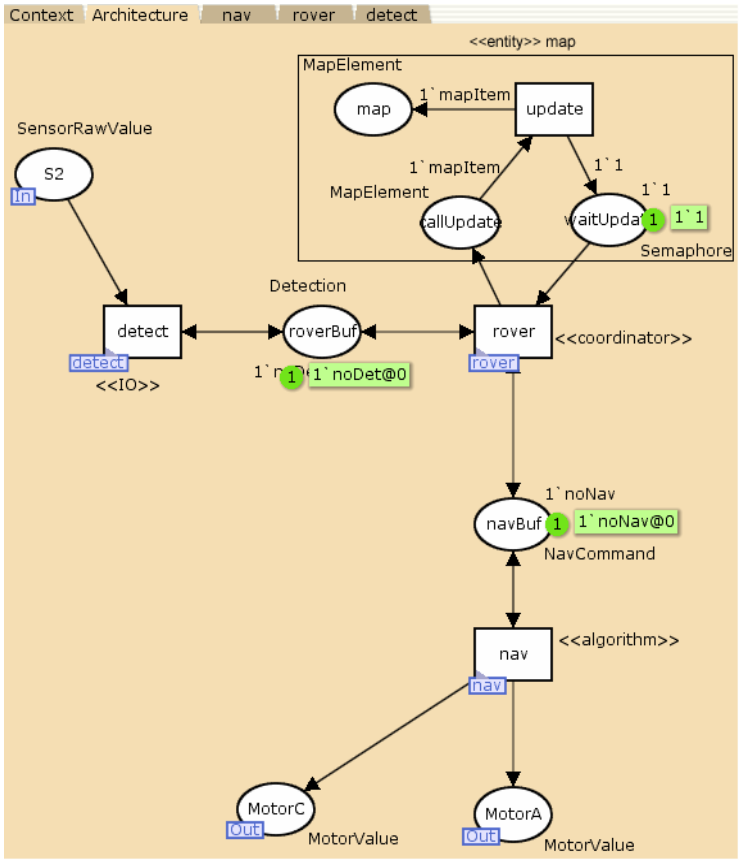


Fig. 6. Robot controller CPN architecture

Moving forward, our second step is to decompose the RoverBot system-level transition into a layer of abstraction representing the concurrent object architecture. This architecture level model is shown in Figure 6. At this level, each of the active objects from is represented as its own transition (box) in the CPN prototype. Each of these will be further decomposed to implement the specific CPN template matching

the objects behavioral design pattern or a CPN component if one exists for the object. We have also included the single «entity» object containing map data and it is represented by a place for the map data to be stored along with a transition and two places representing the behavior for calling the update() operation. Finally, as all message communication between active objects in the RoverBot system is synchronous, there is a CPN place modeling a buffer for the synchronous communication between detect and rover and between rover and nav. Notice that our external input and output places have also been carried down to this level as well.

Once an architecture-level model is established, each of the transitions representing an active object is then decomposed by applying the CPN template associated with the behavioral design pattern of that object. For the asynchronous, input-only «IO» object, "detect", this CPN object-level model is shown in Figure 7. Here, the CPN template has been inserted and instantiated specifically for the detect object by setting the object ID to "1" as seen by the number appended to place and transition names. The specific control token, C1 has also been added as has the function for processing detections, "detection (sensorReading)". To maintain consistency, the main transition of this template, Pin1, has also been connected to the sensor input place and to the roverBuf message buffer place.

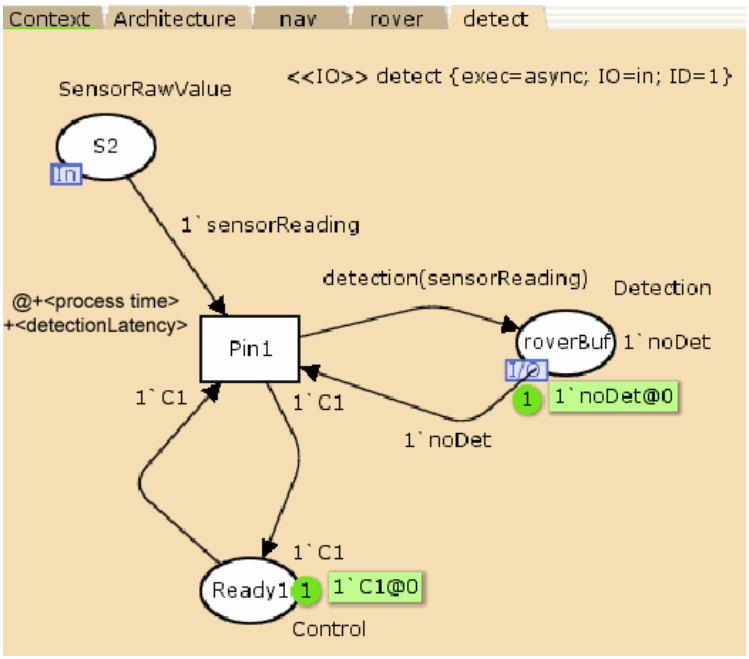


Fig. 7. CPN template for the "detect" object

Now, using the combined historical sizing data and information from the rover PSM, we can augment the architectural parameters within the CPN prototype to obtain further insights as to the behavioral and performance aspects that should be expected when matching the original platform independent design model with the actual platform characteristics of the target implementation.

To begin, we add a place, RAM, to our CPN prototype. This will model the available memory resources measured in bytes. The initial token value for the RAM place is calculated by subtracting the leJOS memory overhead along with the average RAM usage for the objects in our architecture from the total available RAM specified in the PSM.

Table 1. Calculating Memory Availability

Source	Memory (Bytes)
ram.sizeKB	28,672
lejos.kbMemOverhead	17,920
«IO» detect	1,182
«coordinator» rover	2,722
«algorithm» nav	2,030
«entity» map	1,400
Available RAM:	**3,418**

Once the rover system begins execution, the primary consumption of memory occurs when points are added to the map object. For each point added to the map, 16 bytes are used for x and y coordinates; detection event; and timestamp. To prototype this memory consumption, the RAM place from the CPN context level model is attached to the Update transition of the map object's CPN representation on the architecture level model. This is shown in Figure 8. Using this approach, 16 bytes are subtracted from the available RAM each time the update operation is called. If the system reaches a point where less than 16 bytes are available, then the CPN model will be suspended.

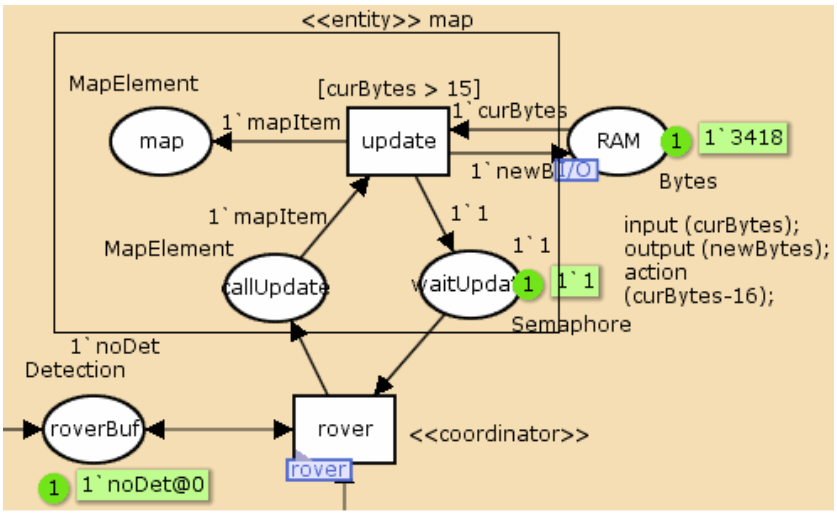


Fig. 8. Consumption of RAM by "map" object

Next, the <process time> parameter will be updated for each active object template. This information can be obtained from PAdemand tag in the PSM's <<psStep>>

stereotype. Additionally, each «IO» object template will add the detection or output latency to the <process time>. For example, the detect object, responsible for interfacing with the light sensor, would have a basic <process time> of 10.8ms. An additional 10.3ms are then added to account for the value of lightSensor.detectionLatency from the PSM, resulting in a total time delay 21ms. Once time values have been allocated to all objects, we can move forward with analyzing the prototype of the architecture as described in the next section.

These initial estimates can eventually be replaced with higher fidelity data as it becomes available, allowing an engineer to refine the behavioral analysis as desired.

3.4 Analyzing the Prototype

Recall from the sequence diagram of that the primary purpose of the autonomous rover system is to navigate an area, mapping objects discovered by the light sensor and taking evasive action when the light sensor detects obstacles or course boundaries. To begin analyzing this behavior with the corresponding CPN prototype, we use a test driver to provide simulated input events at random intervals. One of the first things we want to discover is how quickly the architecture responds to the detection of an obstacle or boundary. This can be analyzed from the context-level model by taking the difference in time stamps from the time an obstacle or boundary event arrives on the light sensor place to the time that a command is issued to the motors. For example, if the first obstacle was detected at time 6459 (all time is in milliseconds in this model). From the timestamps on the Motor places, we can see that from the time an input arrives to the time the system responded, there was an elapsed time of 31ms. This information could then be used, along with the speed of the rover, to determine if the reaction time is sufficient using this software architecture and this particular platform.

Other forms of analysis could include observing the memory usage over time or investigating the interaction among the concurrent objects as the simulated input rate varies. Analysis of physical architectural variations such as different light sensors or motors could also be conducted by applying different PSMs. Analysis of software architecture variations such as the use of different message communication mechanisms (e.g. FIFO or priority queuing) between the active objects could also be explored. These analyses are not shown due to space limitations in this paper.

3.5 Comparing Prototype with Observations

To validate our prototype, the rover design presented above was implemented in leJOS and the code was instrumented to capture timestamps. Execution of the rover when presented with boundaries or obstacles initially identified actual response times of 25-27ms from the point that the light sensor was presented with the boundary or obstacle to the point that the first motor command was output in response to the detection. This is slightly under the 31ms estimated by our analysis in the previous section. Interestingly, though, as we conducted tests with the rover over time, we observed response times increasing as battery power decreased. The above measurements of 25-27ms were observed with fully charged (9.0V) batteries.

However, response times of up to 33ms were observed as the battery power was depleted to 8.2V. These results are summarized in Table 2 below. Replacing the depleted batteries with a fully charged set returned the response times to the initially observed 25-27ms. Thus, we believe that future research should include a power source with the embedded platform specific model.

Table 2. Response Time Results

Team	Response Time in Milliseconds			
	Run 1	Run 2	Run 3	Run 4
1	27	27	29	33
2	25	26	27	27
3	26	25	26	28
4	26	27	29	32
5	25	25	26	26

4 Conclusions and Future Research

In this paper, we have presented an approach to combine information from platform-independent and platform-specific models to construct prototypes of software architectures for embedded systems. This approach allows an engineer / analyst to examine behavioral and performance properties of a software architecture design paired with a candidate implementation architecture. The underlying CPN prototype is particularly useful in modeling concurrent object architectures in event-driven systems. Applying the behavioral design patterns in the UML-based design along with corresponding CPN templates and components, the results from the analyses can be directly mapped back to the original design artifacts. Furthermore, by employing architectural parameters such as processing time, the CPN analysis model can be rapidly modified to account for different candidate architectures.

Future research in this area must continue to examine properties that should be captured and the most effective ways in which to capture them. In comparing our observed results to our analyses, the inclusion of a power model would obviously be desired in an embedded system. Additionally, future work should consider the ability to model distributed software designs configured to execute on multiple distributed embedded nodes and the communication between them.

References

1. H. Gomaa, Designing Concurrent, Distributed, and Real-Time Applications with UML, 1 ed: Addison-Wesley, 2000.
2. R. Pettit and H. Gomaa, "Modeling behavioral design patterns of concurrent objects," presented at ICSE 2006, Shanghai, China, 2006.
3. L. Sha and J. B. Goodenough, "Real-Time Scheduling Theory and Ada," *IEEE Computer*, vol. 23, pp. 53-62, 1990.

4. C. W. Smith, *Performance Engineering of Software Systems*: Addison Wesley, 1990.
5. H. Gomaa and D. Menascé, "Performance Engineering of Component-Based Distributed Software Systems," in *Performance Engineering 2001, LNCS*: Springer, 2001, pp. 40-55.
6. SEI, *A Practioner's Handbook for Real-Time Analysis - Guide to Rate Monotonic Analysis for Real-Time Systems*. Boston: Kluwer, 1993.
7. IBM, "IBM Technical Developer," 2006.
8. Ilogix, "Ilogix Rhapsody," Ilogix, 2006.
9. D. Harel and E. Gery, "Executable Object Modeling with Statecharts," 1996.
10. M. Baldassari, G. Bruno, and A. Castella, "PROTOB: an Object-oriented CASE Tool for Modelling and Prototyping Distributed Systems," *Software-Practice & Experience*, vol. 21, pp. 823-44, 1991.
11. O. Biberstein, D. Buchs, and N. Guelfi, "Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 Formalism," in *COPN, Advances in Petri Nets, LNCS*: Springer-Verlag, 2001, pp. 73-130.
12. L. Baresi and M. Pezze, "On Formalizing UML with High-Level Petri Nets," in *COPN, Advances in Petri Nets, LNCS*. Berlin: Springer-Verlag, 2001, pp. 276-304.
13. K. M. Hansen, "Towards a Coloured Petri Net Profile for the Unified Modeling Language - Issues, Definition, and Implementation," Centre for Object Technology, Aarhus, Denmark, Technical Report COT/2-52-V0.1, 2001.
14. R. Pettit and H. Gomaa, "Modeling Behavioral Patterns of Concurrent Software Architectures Using Petri Nets," presented at 4th WICSA, Oslo, Norway, 2004.
15. B. Huber, R. Obermaisser, P. Peti, and C. E. Salloum, "Resource Specification of the DECOS Integrated Architecture," TU Wien, Vienna, Austria, Technical Report October 12, 2005 2005.
16. Lego, "Lego Mindstorms - http://mindstorms.lego.com."
17. R. Pettit, "SWE 626: Software Project Lab for Real-Time and Embedded Systems," George Mason University, 2006.
18. B. Bagnall, *Core LEGO MINDSTORMS Programming: Unleash the Power of the Java Platform*: Prentice Hall, 2002.
19. K. Proudfoot, "RCX Internals - http://graphics.stanford.edu/~kekoa/rcx/," 1999.
20. N. S. Andersen and M. N. Kjærgaard, "Advanced programming of the LEGO RCX for education," Technical University of Denmark, 2001.
21. UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE) Beta 2, OMG In, June 2008, http://www.omg.org/cgi-bin/doc?ptc/2008-06-08
22. UML Profile for Schedulability, Performance and Time 1.1, February 2005, OMG Inc., http://www.omg.org/technology/documents/formal/schedulability.htm

Schedule-Aware Workflow Management Systems

R.S. Mans[1,2], N.C. Russell[1], W.M.P. van der Aalst[1], A.J. Moleman[2], P.J.M. Bakker[2]

[1] Department of Information Systems, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands.
{r.s.mans,n.c.russell,w.m.p.v.d.aalst}@tue.nl
[2] Academic Medical Center, University of Amsterdam, Department of Quality Assurance and Process Innovation, Amsterdam, The Netherlands.
{a.j.moleman,p.j.bakker}@amc.uva.nl

Abstract. Contemporary workflow management systems offer work-items to users through specific work-lists. Users select the work-items they will perform without having a specific schedule in mind. However, in many environments work needs to be scheduled and performed at particular times. For example, in hospitals many work-items are linked to appointments, e.g., a doctor cannot perform surgery without reserving an operating theater and making sure that the patient is present. One of the problems when applying workflow technology in such domains is the lack of calendar-based scheduling support. In this paper, we present an approach that supports the seamless integration of unscheduled (flow) and scheduled (schedule) tasks. Using CPN Tools we have developed a specification and simulation model. Based on this a system has been realized that uses YAWL, Microsoft Exchange Server 2007, Outlook, and a dedicated scheduling service. The approach is illustrated using a real-life case study at the AMC hospital in the Netherlands.

1 Introduction

Healthcare is a prime example of a domain where the effective execution of tasks is often tied to the availability of multiple scarce resources, e.g. doctors. In order to maximize the effectiveness of individual resources and minimize process throughput times, typically an appointment-based approach is utilized for scheduling the tasks performed by these resources. However, the scheduling of these appointments is often undertaken on a manual basis and its effectiveness is critically dependent on preceding tasks being performed on-time in order to prevent the need for rescheduling.

To illustrate the importance of the afore-mentioned issue, consider a small hospital process for diagnosing a patient, shown in Figure 1. As a first step, the patient is registered. Next, a physical examination (task "physical examination") of the patient takes place which is done by an assistant and a nurse. In parallel, a nurse prepares the documents for the patient (task "make documents"). When these tasks have been completed, a doctor evaluates the result of the test (task "consultation") and decides about the information and brochures that need to

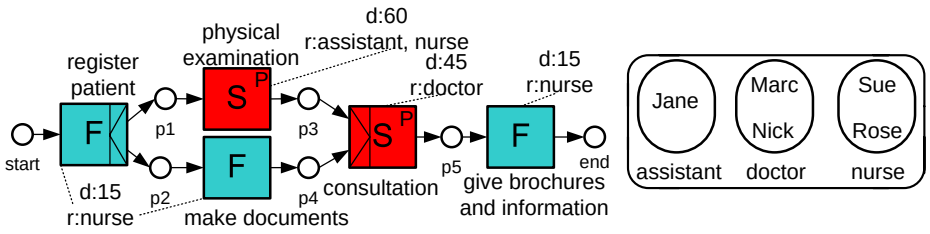


Fig. 1. Running example showing schedule (S) and flow (F) tasks. The prefix "d:" indicates the average time needed for performing the task and prefix "r:" indicates which roles are necessary to perform the task. From each associated role, exactly one person needs to be assigned to the task. For both schedule tasks, indicated by the character "P" in the top-right corner of the task, the patient is also required to be present.

be provided by the nurse (task "give brochures and information"). Figure 1 also shows the corresponding organizational model which specifies the roles being played by people in the organization.

From this example, it can be seen that a distinction can be made between two kinds of tasks. The tasks annotated with an "F" in the figure, can be performed at *an arbitrary point in time when a resource becomes available* and are called *flow tasks*. However, the tasks "physical examination" and "consultation", annotated with an "S" in the figure, can only be performed when the required room is reserved, the patient is present, and the necessary medical staff are present for performing the specific task, i.e. these tasks need to be scheduled and performed at particular times. Therefore, we call these kinds of tasks *schedule tasks* as they are performed *by one or more resources at a specified time*.

For the consultation task in the figure, it is often the case that a doctor finds out at the actual appointment that some results from required diagnostic tests are missing. Consequently, this leads to wasted time for the doctor as a new appointment needs to be scheduled. Therefore, for the effective performance of schedule tasks it is vital that the whole workflow is taken into account in order to guarantee that preceding tasks are performed on-time thereby preventing the need for rescheduling and avoiding unproductive time for resources as a result of canceled appointments.

Workflow technology presents an interesting vehicle with which to support healthcare processes. Based on a corresponding process definition, Workflow Management Systems (WfMSs) support processes by managing the flow of work such that individual work-items are done at the right time by the proper person [2]. Contemporary WfMSs offer work-items through so-called work-lists. At an arbitrary point in time, a user can pick a work-item from this list and perform the associated task.

If we consider the implementation of this process in the context of a WfMS, we find that a significant dichotomy exists in that people are used to working in a scheduled way, but this is not supported by current WfMSs. In contrast

to administrative processes, healthcare processes invoke the coordination of expensive resources which have scarce availability. Therefore, it is of the utmost importance that the scheduling of appointments for these resources is done in an efficient way, that is suitable both for the medical staff and also for the patients being treated. To summarize, there is a *need to integrate workflow management systems with scheduling facilities.*

In this paper, we present the design and implementation of a WfMS supporting both schedule and flow tasks. In addition to the classical work-list functionality generally associated with workflow systems, the concept of a calendar is also introduced in order to present appointments for scheduled work-items to the people involved. Unlike traditional workflow implementations, our focus is on how WfMSs can be *integrated* with scheduling facilities rather than simply extending the functionality of a WfMS or a scheduling system (e.g. a scheduling algorithm). In other words, we investigate how scheduling facilities can be added to workflow systems in general.

An interesting problem in this context lies in the actual development approach taken to extending a WfMS with scheduling facilities. Our strategy for this is based on the use of CPN Tools, a widely used modeling and execution tool for Colored Petri Nets, with which we developed a *comprehensive conceptual model capable of serving both as a specification and simulation model for the application domain.* Formalizing such a system using CP Nets offers several benefits. First of all, building such a net allows for *experimentation.* So, the model or parts of it can be executed, simulated, and analyzed which leads to important insights about the design and implementation of the system. Second, the hierarchical structuring mechanism of CP Nets allows for the modeling of large complex systems at different levels of abstraction. That is, CP Nets can be structured into a set of components which interact with each other through a set of well-defined interfaces, in a similar way to the components in a modular software architecture.

In this way, we were able to use the conceptual model as a *specification* for the subsequent realization of the system. In order to realize the functionality contained in the conceptual model, we *incrementally* mapped it to an operational system based on widely available open-source and commercial-off-the-shelf (COTS) software. Although the conceptual model is detailed, it remains abstract enough, such that its components can be concretized in many different ways. We choose an approach based on the reuse of existing software. In total, the conceptual model consists of 30 nets, 250 transitions, 634 places, and in excess of 1000 lines of ML-code illustrating the overall complexity of the system. For the concrete realization of the system we used the open-source, service-oriented architecture of YAWL and Microsoft Exchange Server 2007 as the implementation platform.

The remainder of the paper is organized as follows. In Section 2 we explain how a workflow language can be augmented with information relevant for scheduling. In Section 3 we present the design of a WfMS integrated with scheduling facilities, together with a concrete implementation. In Section 4 a concrete

application of the realized system is presented. Section 5 discusses related work and finally Section 6 concludes the paper.

2 Flow and Schedule Tasks

In order to allow for the extension of a WfMS with scheduling functionality some concepts need to be introduced. It is assumed that the reader is familiar with basic workflow management concepts, like case, role, and so on [2]. Using the process shown in Figure 1, we will elaborate on how a workflow language can be integrated with scheduling functionality.

2.1 Concepts

We can distinguish between two distinct types of tasks. Flow tasks are performed at an arbitrary point in time when a resource becomes available. As only one resource is needed, it is sufficient to define only *one* role for each of them[3]. Consequently, these tasks can be presented in an ordinary *work-list*. For example, for the flow task "make documents" the work may either be performed by "Sue" or "Rose".

Conversely, schedule tasks are performed by one or more resources at a specified time. As multiple resources can be involved, with different capabilities, it is necessary to specify which kinds of resources are allowed to participate in completing the task. To this end, multiple resources may be defined for a schedule task where for each role specified, only *one* resource may be involved in the actual performance of the task. For example, in Figure 1, the schedule task "physical examination" may be performed by "Jane" and "Rose", but not by "Sue" and "Rose". Note that a resource involved in the performance of a schedule task may also be a physical resource such as medical equipment or a room. Furthermore, for the schedule tasks the patient may also be involved which means that the patient is also a required resource for these tasks. Note that the patient is not involved in the actual execution of the task but is a passive resource who needs to be present whilst it is completed. For this reason, the patient is not added to any of the roles for the task, nor are they defined in terms of a separate role. Instead, it is necessary to identify for which schedule tasks the patient needs to be present.

For presenting the appointments made for schedule tasks to users, the concept of a *calendar* will be used. More specifically, each resource will have its own calendar in which appointments can be booked. Note that each patient also has his / her own calendar. An appointment either refers to a schedule task which needs to be performed for a specific case or to an activity which is not workflow related. So, an appointment appears in the calendars of all resources that are involved in the actual performance of the task. An appointment for a schedule

[3] There also exist approaches for which more roles may be defined, but this is not the focus of our work.

task, for which a work-item does not yet exist, can be booked into the calendar of a resource. However, when the work-item becomes available it has already been determined when it will be performed and by whom. Note that sometimes work-items need to be rescheduled because of anticipated delays in preceding tasks.

In order to be able to determine at runtime the earliest time that a schedule task can be started, information about the duration of every task needs to be known. For example, in Figure 1, for each task the average duration is indicated by prefix "d:". For example, one block represents one minute, which means that the task "physical examination" takes 60 minutes on average.

2.2 Formalization

Based on the informal discussion in the previous section, we now formalize the augmented workflow language. The definition of our language is based on WF-nets [2]. Note that our results are in no way limited to WF-nets and can be applied to more complex notations (BPM, EPCs, BPEL, etc). Note that WF-nets are the most widely used formal representation of workflows. A WF-net is a tuple $N = (P, T, F)$ defined in the following way:

- P is a non-empty finite set of *places*;
- T is a non-empty finite set of *tasks* ($P \cap T = \emptyset$);
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation);
- There is one initial place $i \in P$ and one final place $o \in P$ such that every place or transition is on a directed path from i to o.

A WF-net can be extended in the following way, called a *scheduling WF-net* (sWF-net). A sWF-net is a tuple $N = (P, T_f, T_s, F, CR, Res, Role, R, Rtf, Rts, D)$, where:

- T_f is a finite set of *flow tasks*;
- T_s is a finite set of *schedule tasks*;
- $T_f \cup T_s = T$ and $T_f \cap T_s = \emptyset$, i.e., T_s and T_f partition T. So, a task is either a flow task or a schedule task, but not both;
- (P, T, F) is a WF-net;
- $CR \subseteq T_s$ is the set of schedule tasks for which the human resource for whom the case is being performed is also required to be present.
- Res is a non-empty finite set of *resources*;
- $Role$ is a non-empty finite set of *roles*;
- $R: Res \rightarrow \mathcal{P}(Role)$ is a function which maps resources on to sets of roles;
- $Rtf: T_f \nrightarrow Role$ is a partial function which maps flow tasks on to roles;
- $Rts: T_s \rightarrow \mathcal{P}(Role) \backslash \{\emptyset\}$ is a function which maps schedule tasks on to at least one role;
- $D: T \rightarrow \mathbb{N}_0$ is a function which maps tasks onto the number of blocks that are needed for the execution of the task.

Note that Figure 1 fully defines a particular sWF-net.

3 Design

In this section, we present the design and implementation of a WfMS integrated with scheduling facilities. First of all, the approach followed for doing this is presented. Second, in Section 3.2, we introduce the architecture of the system. Third, for each component identified in Section 3.2, a detailed (functional) description is provided in sections 3.3 to 3.5.

3.1 Approach

Contemporary WfMSs provide a wide range of functions. In order to determine before the implementation phase, how such a system can be integrated with scheduling facilities one needs to identify how the new scheduling functionality being added should be incorporated with existing functionality. To this end, *Colored Petri Nets (CP Nets)* [12] have been chosen as the mechanism to identify and formalize the behavior of the system. CP Nets provide a well-established and well-proven language suitable for describing the behavior of systems exhibiting characteristics such as concurrency, resource sharing, and synchronization.

Formalizing a system using CP Nets offers several benefits. First of all, building such a net allows for *experimentation*. So, the model or parts of it can be executed, simulated and analyzed which leads to insights about the design and implementation of the system. Second, a complete model of the system allows for *testing* parts of the system that are implemented. Given that a CP Net consists of several components, we can "replace" one or more components in the CP Net by the concrete implementation of these components by making connections between the CP Net model and components in the actual system. As the CP Net is an executable model this allows for the testing of numerous scenarios facilitating the discovery of potential flaws in both the architecture and the implementation.

Another important benefit of having a CP Net consisting of several components, is that it provides precise guidance in the configuration of software products, thereby allowing for the use of existing software. As will become clear below, whilst the specification model is detailed, it remains abstract enough, such that it allows components to be concretized in various ways.

3.2 Architecture

In this section, we give a global overview of the architecture of a WfMS integrated with scheduling facilities. The architecture of both the conceptual model of the system and its concrete implementation are shown in Figure 2. Both architectures illustrate the main components and the system is defined in a service oriented way. The components are loosely coupled and the interfaces (shown as clouds) are kept as compact and simple as possible.

In Figure 2b, we see for the actual system implementation how the components have been realized. As the interfaces share the same numbering, it is easy to compare both sets of interfaces.

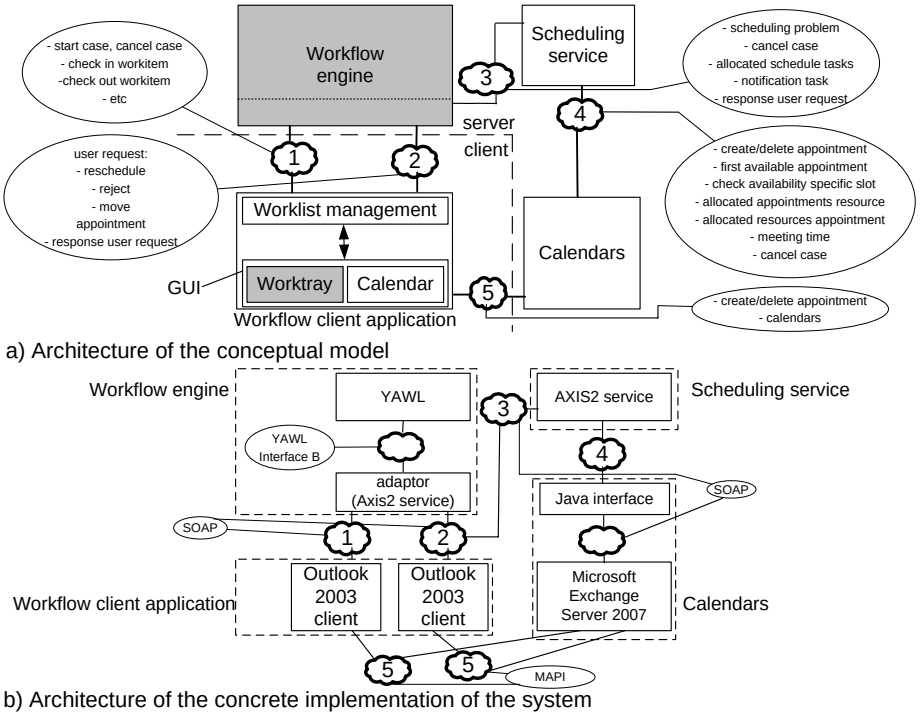


Fig. 2. Architectures of both the conceptual model and the concrete implementation of the system. There are four main components: (I) workflow engine, (II) scheduling service, (III) workflow client application, and (IV) calendars. The distinct interfaces are indicated by numbers.

The architecture consists of four components. First of all, the *workflow engine* routes cases through the organization. Based on the business process definition for a case, tasks are carried out in the right order and by the right people. Once a task in a case becomes available for execution, the corresponding work-item is communicated to users via the *workflow client application* allowing it to be selected and performed by one of them. The scheduling service and the workflow client application communicate with the *Calendar* component in order to obtain a view on users' calendars and to manipulate their contents. Note that users can add /remove appointments that are possibly unrelated to the workflow.

As our focus is on how a WfMS can be *integrated* with scheduling facilities, we want to completely separate the scheduling facilities provided by the system from the engine. As a consequence, we have a separate *scheduling service* component which is responsible for providing scheduling facilities to the system (e.g. (re)scheduling of tasks). In order for the scheduling service to work function correctly, all scheduling constraints imposed by the engine (which might be relevant to a scheduling decision) need to be sent to the scheduling service. To be more precise, the scheduling service receives a *scheduling problem*, which contains all

relevant constraints for one case only. Based on these constraints, the scheduling service makes decisions with regard to the scheduling of schedule tasks for the case.

Informally, the scheduling problem is formulated as a *graph* which has *nodes* and *arcs* between nodes. Nodes, arcs and the graph itself may have properties represented as name-value attributes. The rationale for representing the scheduling problem using this data structure is that any information in the graph can be included which is deemed relevant. For a case, which is in a given state, we map the process definition, defined in terms of the formal definition given in Section 2.2, to the graph (e.g. tasks, duration, split/join semantics of a node, roles). Where a work-item exists for a given node, a property is added to that node indicating the state the work-item is currently in. For a user to reschedule an appointment, additional information is added, such as the name of the requester. Moreover, if the human resource for which the case is being performed is also required in order to perform any task, then the name of the calendar for this resource is included together with the names of the relevant schedule tasks.

An example of a scheduling graph is given in Figure 3. In this figure, we see how the process definition shown in Figure 1 is mapped to the graph. In order to simplify the graph, the figure only shows the properties of the "consultation" node. For this node it indicates that the average duration is 45 minutes, only a doctor is allowed to perform the task, the task is a schedule task, the node has XOR-split semantics, AND-join semantics, and a work-item exists for it which is in the enabled state.

In sections 3.3 to 3.6 the individual components are discussed in more detail. For each component a description of the main functionality is provided together with a discussion on its interaction with other components. Note that, due to space limitations, only the most important interface methods will be discussed.

3.3 Workflow Engine

A workflow engine is responsible for the routing of cases. In addition to the standard facilities an engine should provide [2], the following facilities are added in order to integrate scheduling capabilities.

The engine is responsible for sending a scheduling problem to the scheduling service in order to determine whether appointments need to be (re)scheduled, or if limited time remains in which to finish work-items for preceding tasks

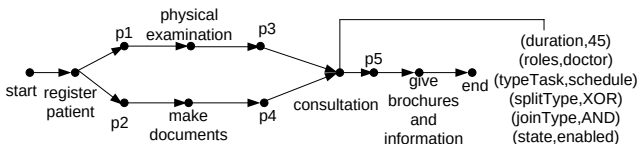


Fig. 3. Scheduling graph for the running example of Figure 1 in which the task "consultation" is enabled.

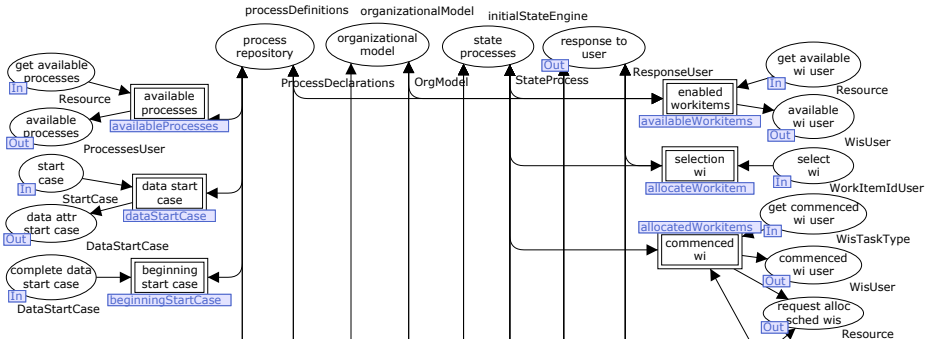


Fig. 4. CP Net component for the workflow engine component.

of an appointment. As a consequence of our choice to completely separate the scheduling facilities from the engine, a scheduling problem for a case is sent when the following situations occur: (1) a case is started; (2) a work-item is finished; (3) a user wants to reschedule an appointment; and (4) at regular time intervals. The fourth option is necessary as it may be the case that no work-items are completed in a given period, but that some appointments need to be rescheduled due to the fact that time has passed. Obviously, the graph is sent the least number of times possible.

As a consequence of the execution of the scheduling service, the engine is informed about appointments for which limited time is left in which to finish work-items of preceding tasks. For these work-items, a warning is sent to the workflow client to indicate that limited time remains in which to finish the work-item.

Model.

A fragment of the CP Net for the workflow engine component is depicted in Figure 4. The places at the far right and far left hand side are part of the interface of the engine with other components. As an indication of the complexity of the engine it is worth mentioning that the flattened substitution transition comprises 54 transitions and 127 places. Moreover, the whole CP Net consists of 217 transitions, 518 places and around 950 lines of ML code. The construction of the whole model required more than three months of work. This underscores the fact that it is a complex system.

Implementation.

The Engine component in the CP Net model can be replaced by a concrete implementation which allows it to be tested. The workflow component is realized (see Figure 2b) using the open-source WfMS YAWL [1] and a service which acts as an adaptor in between YAWL and the workflow client application. The adaptor service communicates with YAWL via “Interface B” [1]. The adaptor also communicates with the scheduling service using SOAP messages. However, the adaptor and the YAWL system are tightly coupled as large volumes of work-item and process related information are exchanged.

3.4 Workflow client application

Users working with the WfMS do so via the workflow client application which delivers the basic facilities that should be provided by this facility [2]. The component consists of a GUI and a work-list management component. The work-list management component serves as a layer between the engine and the GUI and takes care of the communication between them. The GUI component consists of a "worktray" and a "calendar" component where the "worktray" provides the same facilities as a classical worktray. The appointments that are created for schedule tasks are advertised via the calendar. Once a work-item becomes available for such an appointment, it can be performed via the calendar. In our approach, only one user can interact with the WfMS with respect to the completion of the work-item. This prevents concurrency issues where multiple users want to complete the same work-item.

With regard to the appointments that are made for schedule tasks, users can express their dissatisfaction with the nominated scheduling by requesting: (1) the rescheduling of the appointment, (2) the rescheduling of the appointment to a specified date and time, or (3) the reassignment of the appointment to another employee. Such a user request can be done as a single action and is the only supported means for the rescheduling of appointments by users. In addition, the workflow client also indicates whether limited time is left in which to undertake work-items in order to meet the schedule. Moreover, users are also allowed to add appointments to the calendar which are not workflow related (e.g. having dinner with friends).

As can be seen in Figure 2a, two interfaces are defined for the communication between the workflow client application and the engine. The interface with number "1" defines the standard communication that takes place between an engine and a workflow client application. The interface with number "2" defines methods added as a consequence of the scheduling facilities developed for the system. For this interface, nothing is stored in the engine when these methods are called.

Model.

The corresponding CP Net model for the work-list management component is fairly complex (and is not shown here): the component's model contains 104 transitions and 225 places.

Implementation.

Similarly, the workflow client application component of the CP Net can be replaced by a concrete implementation. Once the Exchange Server was in place we could easily use the Microsoft Outlook 2003 client to obtain a view of a user's calendar. Furthermore, the Outlook client can be configured in such a way that it can act as a full workflow client application which can communicate with the WfMS via an adaptor service via the exchange of SOAP messages.

3.5 Scheduling Service

The scheduling service is responsible for providing scheduling facilities to the WfMS. Scheduling is done sequentially on a case-by-case base. Once a scheduling

problem is received, the scheduling service needs to determine whether some of the schedule tasks need to be (re)scheduled. Moreover, several distinct issues need to be addressed of which we mention the most important ones.

First of all, the final scheduling of tasks needs to occur in the same order as the sequence of schedule tasks in the accompanying process definition for the case. Moreover, there should be sufficient time between two scheduled tasks. Also, when rescheduling appointments, any preceding constraints need to be satisfied. For example, in Figure 1, it needs to be guaranteed that first the "physical examination" is scheduled, followed by the "consultation" which needs to occur at a later time.

Second, for the actual scheduling of an appointment multiple roles can be specified for a schedule task. For each role specified a resource needs to be selected, i.e., the number of roles determines the number of resources involved in the actual performance of the task. If the patient for which the case is performed also needs to be present at an appointment, then this is also taken into account. The scheduling service only books an appointment in the calendars of these resources who need to be present during the performance of the task (i.e. the performers of the task and the patient (if needed)).

Third, the scheduling service is also responsible for determining whether limited time is left for performing preceding work-items for scheduled tasks. In such a situation, the engine needs to be informed. Moreover, the scheduling service is also informed about the cancelation of a case, so that all appointments related to the case can be removed. When too little time is left for performing preceding work-items for a scheduled schedule task, the corresponding appointment is automatically rescheduled which in this context can be seen to be the most straightforward recovery action. However, different strategies can also be conceived for dealing with such situations. Potential solutions can be found in [15].

In this paper, we focus on integration aspects instead on devising new scheduling algorithms. Nevertheless, to demonstrate the approach that is used for the scheduling of appointments, we will briefly examine the implemented 'naive' scheduling algorithm. Of course it can be envisaged that more advanced scheduling strategies are possible.

The (re)scheduling of appointments is done automatically, which means that there is no user involvement. Starting with the tasks in the graph for which a work-item exists, it is determined which schedule tasks need to be (re)scheduled. Once we know that tasks are able to be scheduled, they are scheduled. Moreover, these tasks are scheduled on a sequential basis in order to avoid conflicts involving shared resources. However, we do not schedule any tasks which occur after a choice in the process as this can lead to unnecessary usage of available slots in the calendar. Moreover, we do not take loops into account.

For the actual scheduling of an appointment, a search is started for the first opportunity where one of the resources of a role can be booked for the respective work-item. If found, an appointment is booked in the calendar of the resource. If the patient for which the case is performed also needs to be present at the

appointment, then this is also taken into account. For example, for Figure 1, if a case is started, an appointment is created for task "physical examination" in the calendars of "Jane", "Sue", and the patient, or "Jane", "Rose" and the patient.
Model.
The CP Net model which models the scheduling service consists of 48 transitions and 144 places. Moreover, modeling the scheduling behavior necessitated writing many lines of ML code, involving around more than 60 hours of work.
Implementation.
The concrete implementation of this component of the CP Net is shown in Figure 2b. Here we see that the component is implemented in Java as a service which communicates with the WfMS via SOAP messages. However, in order to get a view of and to manipulate the calendar, the service also communicates via a Java interface with the Exchange Server which in turn exchanges information via SOAP messages.

3.6 Calendar

The Calendar component is responsible for providing a view on the calendars of users and for manipulating their contents. It is possible to create / delete appointments or to get information about the appointments that have been made. Moreover, the interface contains some convenience methods for deleting cases and finding the first available slot for a schedule task. Otherwise, large volumes of low-level information need to be exchanged whereas now only one call is necessary.
Model.
The CP Net model which models the scheduling service consists of 11 transitions and 22 places. Note that this model is relatively simple.
Implementation.
For the Calendar component we selected Microsoft Exchange Server 2007 as the system for storing the calendars of users. The big advantages of this system are its widespread use and the fact that it offers several interfaces for viewing and manipulating calendars.

4 Application

In this section, we demonstrate our approach and software in the context of a real-life healthcare scenario. To evaluate our approach, we have taken the diagnostic process of patients visiting the gynecological oncology outpatient clinic at the AMC hospital, a large academic hospital in the Netherlands. This healthcare process deals with the diagnostic process that is followed by a patient who is referred to the AMC hospital for treatment, up to the point where the patient is diagnosed, and consists of around 325 activities. However, for our scenario we will only focus on the initial stages of the process shown in Figure 5.

At the beginning of the process, a doctor in a referring hospital calls a nurse or doctor at the AMC hospital resulting in an appointment being made for

the first visit of the patient. Several administrative tasks need to be requested before the first visit of the patient (e.g. task "first consultation doctor"). At the first consultation, the doctor decides which diagnostic tests are necessary (MRI, CT or pre-assessment) before the next visit of the patient (task "consultation doctor"). Note that for the MRI, CT and pre-assessment tasks we do not show the preceding tasks at the respective departments that need to performed in order to simplify the model presented.

For this scenario, we assume that the task "additional information and brochures" has been performed. Moreover, at the first consultation with the doctor it has been decided that an MRI and a pre-assessment are needed for the patient. So, by looking at the process model it becomes clear that the tasks "MRI", "pre-assessment" and "consultation doctor" need to be scheduled. The result of the scheduling performed by the system for these tasks is shown in Figure 6a. Note that our case has "Oncology" as its process identifier and has "15" as its case identifier. Moreover, for the "consultation doctor", "pre assessment", and "MRI" examination, a doctor, an anaesthetist, and MRI machine are needed respectively. Moreover, the patient is also required to be present.

In Figure 6a we can see that the "MRI" has been scheduled for 10:00 to 10:45 (see first column), the consultation with the doctor has been scheduled for 13:00 to 13:30 in the calendar of doctor "Nick" (see second column), and that the pre-assessment has been scheduled for 11:00 to 11:30 in the calendar of anaesthetist "Jules" (see third column). At the far right, we can see the calendar of patient Anne who also needs to be present for the work-items mentioned, which explains why the previously mentioned appointments are also present in her calendar. For Anne we see that she is not available till 10 'o clock which has influenced the actual scheduling. This is due to the fact that she can not manage to be at the hospital before 10 'o clock by public transport. However, it is important that the "consultation doctor" task is scheduled after the "MRI" and "pre-assessment" task, which is also consistent with the corresponding process definition.

Now, let us assume that unexpectedly some maintenance for the MRI machine is necessary for that day, which will take until 13:30 hours to complete.

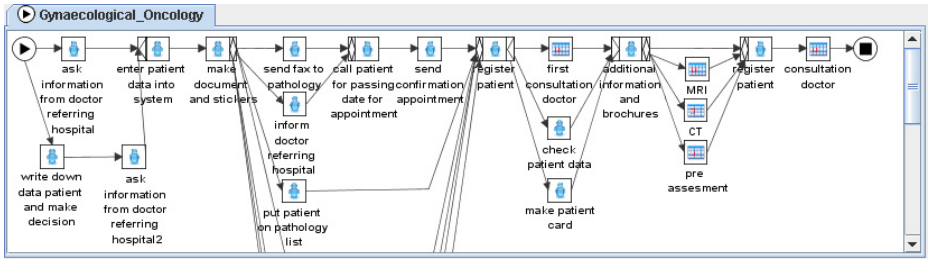


Fig. 5. Screenshot of the YAWL editor showing the initial stages of the gynaecological oncology healthcare process. The flow tasks are indicated by a person icon and the schedule tasks are indicated by a calendar icon.

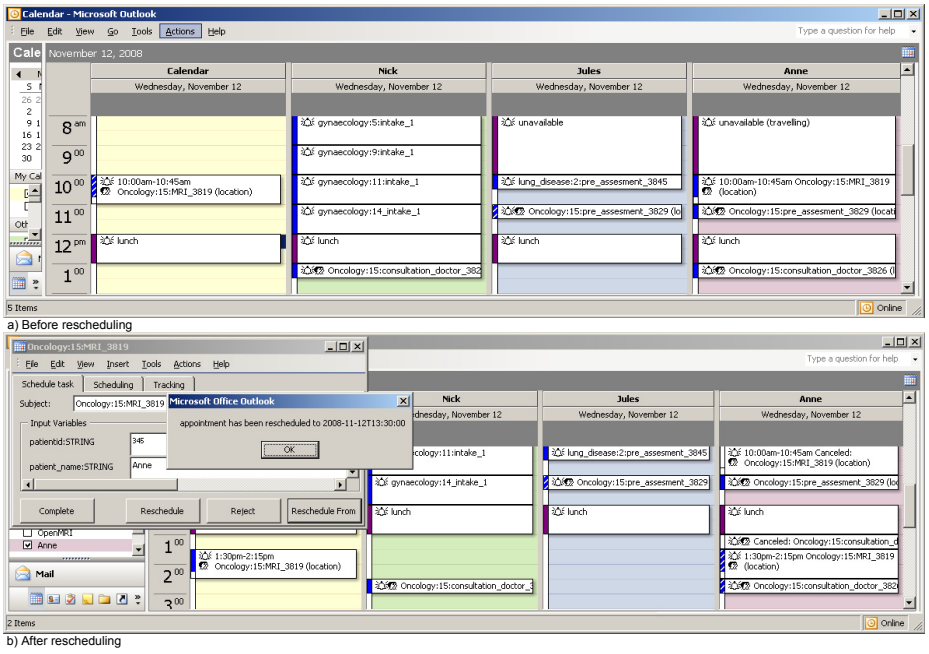


Fig. 6. Screenshot of the calendars for the MRI, consultation with the doctor, and the pre-assessment before and after rescheduling.

.

Consequently, the MRI appointment needs to be rescheduled to 13:30 hours. The effect of this specific rescheduling request can be seen in Figure 6b. In this figure, the message box indicates that the MRI has been successfully rescheduled to the requested time. Moreover, in the calendar of Anne we can see that the MRI now takes place from 13:30 to 14:15. However, it was also necessary to reschedule the appointment with doctor "Nick" which will now take place from 14:30 to 15:00. As can be seen in Figure 5, this rescheduling step is necessary as the task "consultation doctor" occurs after the "MRI" task and the task "register patient" falls in between these two tasks and takes 15 minutes.

5 Related Work

Analysis of the healthcare research shows that significant work has been done on the problem of appointment scheduling. Examples of such research efforts are appointment scheduling for outpatient services [7] and operating room scheduling [6]. However, most of these studies focus on a single unit instead of situations in which a patient may pass through multiple facilities. In our research, we take the scheduling of work-items for the whole workflow into account together with the current state of a case.

Our work is also related to time management in workflows. For example, in [13, 11] the authors focus on the satisfiability of time constraints and the enforcement of these at run-time. In addition, there is also research on the problem of the scheduling of tasks by WfMSs. For example, [4, 9, 14] present algorithms for the scheduling of tasks. In contrast, we focus on the augmentation of a WfMS with scheduling facilities instead of just presenting new scheduling algorithms.

The work presented in [8] is somewhat similar to ours as it presents different architectures for a WfMS in which temporal aspects are explicitly considered. However, the temporal reasoning facilities are added as core functionality to the engine. In this paper, we propose a different approach where this kind of functionality is realized through a separate service in the system. In this way, loose coupling is guaranteed which means that our approach can be generalized to any WfMS (or even to multiple engines at the same time).

Multiple people can be involved in the actual performance of a schedule task. However, in our approach, only one user can interact with the WfMS with respect to the completion of a work-item. In [3, 5, 10] reference models to extend the organizational meta model with a team concept allowing for the distribution of work to teams are proposed. By doing so, advanced mechanisms are offered for the performance of work by such a team. Additionally, in [3, 5] a language is discussed for defining work allocation requirements to people.

6 Conclusions and Future Work

In this paper, we have presented the design and implementation of a WfMS augmented with calendar-based scheduling facilities. Instead of just offering work-items via a work-list, as is the case in most existing WfMSs, they can also be offered as a concrete appointment in a calendar taking into account which preceding tasks are necessary and whether they have been performed.

Our approach demonstrates that the use of CP Nets, for constructing a conceptual model of the system to be realized, provides valuable insights in terms of understanding the problem domain and identifying the behavior of the system. Moreover, the same conceptual model provides a comprehensive specification on which to base the ultimate realization of the required functionality. We have incrementally mapped it to an operational system using widely available open-source and commercial-off-the-shelf (COTS) software. This demonstrates that although the specification model is detailed, it remains at a sufficient level of abstraction to allow its constituent components to be concretized in various ways. Moreover, it also shows that our ideas can, for example, be applied to a variety of WfMSs and scheduling systems.

The resultant system has been tested using several realistic scenarios. We plan to test the components of the system in a more systematic way by incrementally "replacing" components of the CP Net by their concrete implementation. In this way, we can test numerous scenarios facilitating the discovery of flaws both in individual components as well as in the overall architecture of the actual system.

In the design and the implementation of the system, a naive algorithm has been used for the scheduling of appointments. This naive approach can lead to inefficient use of resources. In the future, we plan to use the CP Net for evaluating various scheduling approaches and to investigate the effects of our calendar-based approach on case performance.

Finally, to test the feasibility of our approach, we plan to evaluate the operation of our resultant system in a real-life scenario at the AMC hospital.

References

1. W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede. Design and Implementation of the YAWL System. In *Proceedings of CAiSE'04*, 2004.
2. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, Cambridge, MA, 2002.
3. W.M.P. van der Aalst and A. Kumar. A Reference Model for Team-Enabled Workflow Management Systems. *Data and Knowledge Engineering*, 38(3):335–363, 2001.
4. C. Bettini, X.S. Wang, and S. Jajodia. Temporal Reasoning in Workflow Systems. *Distributed and Parallel Databases*, 11(3):269–306, 2002.
5. J. Cao, S. Zhang, and X. Zhang. Team Work Oriented Flexible Workflow Management System. In X. Meng, J. Su, and Y. Wang, editors, *Advances in Web-Age Information Management*, volume 2419 of *LNCS*, pages 189–200, 2002.
6. B. Cardoen, E. Demeulemeester, and J. Beliën. Operating Room Planning and Scheduling: A Literature Review. FEB Research Report KBI 0807, Katholieke Universiteit Leuven, Leuven, 2008.
7. T. Cayirli and E. Veral. Outpatient Scheduling in Health Care: A Review of Literature. *Product Operations Management*, 12(4):519–549, 2003.
8. C. Combi and G. Pozzi. Architectures for a Temporal Workflow Management System. In H. Haddad, A. Omicini, R.L. Wainwright, and L.M. Liebrock, editors, *Proc. of the 2004 ACM symposium on applied computing*, pages 659–666, 2004.
9. C. Combi and G. Pozzi. Task Scheduling for a Temporal Workflow Management System. In *Thirteenth International Symposium on Temporal Representation and Reasoning (TIME'06)*, pages 61–68, 2006.
10. L. Cui and H. Wang. Research on Cooperative Workflow Management Systems. In W. Shen, Z. Lin, and J.-P.A. Barthès, editors, *Computer Supported Cooperative Work in Design I*, volume 3168 of *LNCS*, pages 359–367, 2005.
11. J. Eder, E. Panagos, and M. Rabinovich. Time Constraints in Workflow Systems. In M. Jarke and A. Oberweis, editors, *Proceedings of CAiSE '99*, volume 1626 of *Lecture Notes in Computer Science*, pages 286–300. Springer-Verlag, Berlin, 1999.
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, 2007.
13. O. Marjanovic and M. Orlowska. On Modeling and Verification of Temporal Constraints in Production Workflows. *Knowledge and Information Systems*, 1(2):157–192, 1999.
14. P. Senkul and I.H. Toroslu. An Architecture for Workflow Scheduling under Resource Allocation constraints. *Information Systems*, 30(5):399–422, 2005.
15. W.M.P. van der Aalst, M. Rosemann, and M. Dumas. Deadline-based Escalation in Process-Aware Information Systems. *Decision Support Systems*, 43(2):492–511, 2007.

Bounded Parametric Model Checking for Elementary Net Systems*

Michał Knapik[1], Maciej Szreter[1], and Wojciech Penczek[1,2]

[1] Institute of Computer Science, PAS, J.K. Ordona 21, 01-237 Warszawa, Poland
{Michal.Knapik,mszreter}@ipipan.waw.pl
[2] Institute of Informatics, Podlasie Academy, Sienkiewicza 51, 08-110 Siedlce, Poland
penczek@ipipan.waw.pl

Abstract. Bounded Model Checking (BMC) is an efficient verification method for reactive systems. BMC has been applied so far to verification of properties expressed in (timed) modal logics, but never to their parametric extensions. In this paper we show, for the first time, that BMC can be extended to PRTECTL – the parametric extension of the existential version of CTL. To this aim we define a bounded semantics and a translation to SAT for PRTECTL. The implementation of the algorithm for Elementary Net Systems is presented together with some experimental results.

1 Introduction

Bounded Model Checking (BMC) [BCCZ99] is a method of performing verification by stepwise unwinding a verified model and translating the resulting fragment, as well as the property in question, to a propositional formula. The resulting formula is then checked by means of efficient external tools, i.e., SAT-solvers. This method is usually incomplete from the practical point of view, but can find counterexamples in systems that appear too large for other approaches.

BMC was invented in late 1990s, and since then has become an established method among verification approaches. BMC is applied to verification of properties specified in temporal, dynamic, epistemic, and timed logics [BCC+99], [BC03], [Hel01], [PWZ02], [Woź03]. In fact, for many system specifications and property languages devised for explicit-state model checking, the BMC counterparts have been developed. In this paper we show how parametric model checking can be performed by means of BMC.

The rest of the paper is organized as follows. In Section 2 we shortly explore the motivations for the choice of parameterized temporal logics vRTCTL and PRTCTL to which the BMC method is applied. Referenced and cited works are mentioned along with an outline of the contents. Section 3 recalls from [ET99] the syntax and semantics of the logics used in this work. In Section 3 we define *existential* fragments of the considered logics – vRTECTL and PRTECTL,

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

respectively. Section 4 introduces k-models together with bounded semantics for vRTECTL and PRTECTL. In Section 5 a translation of a model and a property under investigation is presented together with an algorithm for BMC. Section 6 contains an application of the above method to Elementary Net Systems. We choose two standard problems: the Mutual Exclusion and the Dining Philosophers and test some associated parameterized properties in Section 7. The concluding remarks and an outline of some future work are in Section 8.

2 Related Work

The logics investigated in this paper were introduced in [ET99] while the application of BMC to the existential fragment of the CTL originates from [PWZ02] with a further optimization in [Zbr08]. The work presented in this paper falls into a broad area of the Parametric Model Checking – an ambiguous term which may mean that we deal with the parameters in models (as in [AHV93] and [HRSV01]), in logics (as in [ET99] and [BDR08]) or in both (as in [RB03]). There are two reasons limiting the practical applications of the Parametric Model Checking. The first – computational complexity of the problem – is the result of the presence of satisfiability in the Presburger Arithmetic (PA) as a subproblem. In case of the translation of the existential fragment of TCTL to PA formulae proposed in [BDR08], the joint complexity of the solution is 3EXPTIME. The second – undecidability of the problem for Parametric Timed Automata in general [AHV93] – results in a fact that some of the proposed algorithms need not to stop [HRSV01]. To the best knowledge of the authors, this paper presents the first extension of BMC to parameterized temporal logics.

3 Parameterized Temporal Logics

In this section we recall the temporal logics vRTCTL and PRTCTL, first defined in [ET99], both being extensions of Computation Tree Logic (CTL) introduced in [EC82]. The logic vRTCTL allows superscripts of form $^{\leq\eta}$, where η is a linear expression over path quantifiers of CTL. An example of a formula of this logic is $EF^{\leq\Theta_1+\Theta_2}(w_1 \wedge EG\neg c_1)$. The formulae of PRTCTL are built from formulae of vRTCTL by adding additional existential or universal quantifiers which may be restricted or unrestricted. As an example of a PRTCTL formula consider $\exists_{\Theta_1\leq 1}\forall_{\Theta_2\leq 2}EF^{\leq\Theta_1+\Theta_2}(w_1 \wedge EG\neg c_1)$. Following E. A. Emerson's approach [ET99], the formulae are interpreted in standard Kripke structures, which seem to be appropriate for application in many computer science fields, as motivated in [ET99]. The logics mentioned above essentially extend CTL, as they allow to formulate properties involving lengths of paths in a model. We interpret superscripts as time bounds, assuming that a transition in a model takes the unit of time. Throughout this paper by $\mathbb{N}$ we denote the set of all natural numbers (including 0). By a *sentence* of a logic we mean a formula without free variables, and by $\alpha(\Theta_1, \ldots, \Theta_n)$ we point out that the formula α contains free parameters $\Theta_1, \ldots, \Theta_n$.

3.1 Syntax

Let $\Theta_1, \ldots, \Theta_n$ be variables, called here *parameters*. An expression of the form $\eta = \sum_{i=1}^{n} c_i \cdot \Theta_i + c_0$, where $c_0, \ldots, c_n \in \mathbb{N}$, is called a *linear expression*. A function $v : \{\Theta_1, \Theta_2, \ldots, \Theta_n\} \longrightarrow \mathbb{N}$ is called a *parameter valuation*. Let Υ be a set of all the parameter valuations.

Definition 1. *Let $\mathcal{PV}$ be a set of propositional variables containing the symbol* true*. Define inductively the formulae of* vRTCTL :

1. *every member of $\mathcal{PV}$ is a formula,*
2. *if α and β are formulae, then so are $\neg\alpha$, $\alpha \wedge \beta$ and $\alpha \vee \beta$,*
3. *if α and β are formulae, then so are $EX\alpha$, $EG\alpha$, $E\alpha U\beta$,*
4. *if η is a linear expression, α and β are formulae of* vRTCTL*, then so are $EG^{\leq\eta}\alpha$, $E\alpha U^{\leq\eta}\beta$.*

The conditions $1, 2$, and 3 alone define CTL. Notice that η is allowed to be a constant. The logic defined by a modification of the above definition, where $\eta = a$ for $a \in \mathbb{N}$, is called RTCTL in [ET99]. For example $EF^{\leq 3}(w_1 \wedge EG\neg c_1)$ is an RTCTL formula.

Definition 2. *The formulae of* PRTCTL *are defined as follows:*

1. *if $\alpha \in$* vRTCTL*, then $\alpha \in$* PRTCTL*,*
2. *if $\alpha(\Theta) \in$* vRTCTL *or $\alpha(\Theta) \in$* PRTCTL*, where Θ is a free parameter, then $\forall_\Theta\alpha(\Theta), \exists_\Theta\alpha(\Theta), \forall_{\Theta\leq a}\alpha(\Theta), \exists_{\Theta\leq a}\alpha(\Theta) \in$* PRTCTL *for $a \in \mathbb{N}$.*

The following inclusions hold: CTL $\subseteq$ RTCTL $\subseteq$ vRTCTL $\subseteq$ PRTCTL. In this paper we consider only sentences of PRTCTL.
Additionally we use the derived modalities: $EF\alpha \stackrel{def}{=} E(trueU\alpha)$, $AF\alpha \stackrel{def}{=} \neg EG\neg\alpha$, $AX\alpha \stackrel{def}{=} \neg EX\neg\alpha$, $AG\alpha \stackrel{def}{=} \neg EF\neg\alpha$ (CTL modalities) and $EF^{\leq\eta}\alpha \stackrel{def}{=} E(trueU^{\leq\eta}\alpha)$, $AF^{\leq\eta}\alpha \stackrel{def}{=} \neg EG^{\leq\eta}\neg\alpha$, $AG^{\leq\eta}\alpha \stackrel{def}{=} \neg EF^{\leq\eta}\neg\alpha$. Each modality of CTL has an intuitive meaning. The path quantifier A stands for "on every path" and E means "there exists a path". The modality X means "in the next state", G stands for "in the all states", F means "in some state", and U has a meaning of "until".
The introduced superscripts will become clear when the semantics of vRTCTL is presented. As to give an example of the intuitive meaning of an RTCTL formula, $EG^{\leq 3}p$ may be perceived as the statement "there exists a path such that in the first four states of this path p holds". The logic vRTCTL adds a possibility of expressing similar properties under parameter valuations, and PRTCTL allows for stating that some property holds in a model under some class of parameter valuations.

Definition 3. *The logics* vRTECTL, RTECTL, *and* PRTECTL *are defined as the restrictions of, respectively,* vRTCTL, RTCTL, *and the set of sentences of* PRTCTL *such that the negation can be applied to the propositions only.*

3.2 Semantics

We evaluate the truth of the sentences and the formulae accompanied with parameter valuations in Kripke structures.

Definition 4. *Let $\mathcal{PV}$ be a set of propositional variables containing the symbol* true*. A Kripke structure (a* model*) is defined as a tuple $(S, \rightarrow, \mathcal{L})$ where:*

1. *S is a finite set of* states*,*
2. *$\rightarrow \subseteq S \times S$ is a transition relation such that for every $s \in S$ there exists $s' \in S$ with $s \rightarrow s'$ (i.e., the relation is total),*
3. *$\mathcal{L} : S \longrightarrow 2^{\mathcal{PV}}$ is a labelling function satisfying true $\in \mathcal{L}(s)$ for $s \in S$.*

The labelling function assigns to an each state s a set of propositions which are assumed to be true at s. An infinite sequence $\pi = (s_0, s_1, \ldots)$ of states of a model such that $s_i \rightarrow s_{i+1}$ for $i \in \mathbb{N}$ is called a *path*. By $\pi(i)$ we denote the i–th position on a path π. The number of the states of M is called the size of M and denoted by $|M|$. For a parameter valuation υ and a linear expression η, by $\upsilon(\eta)$ we mean the evaluation of η under υ.

Definition 5. (Semantics of vRTCTL**)**
Let M *be a model, s – a state, α, β – formulae of* vRTCTL*. $M, s \models_\upsilon \alpha$ denotes that α is true at the state s in the model M under the parameter valuation υ. We omit M where it is implicitly understood. The relation $\models_\upsilon$ is defined inductively as follows:*

1. $s \models_\upsilon p \iff p \in \mathcal{L}(s)$
2. $s \models_\upsilon \neg p \iff p \notin \mathcal{L}(s)$,
3. $s \models_\upsilon \alpha \wedge \beta \iff s \models_\upsilon \alpha \wedge s \models_\upsilon \beta$,
4. $s \models_\upsilon \alpha \vee \beta \iff s \models_\upsilon \alpha \vee s \models_\upsilon \beta$,
5. $s \models_\upsilon EX\alpha \iff \exists_\pi \big(\pi(0) = s \wedge \pi(1) \models_\upsilon \alpha\big)$,
6. $s \models_\upsilon EG\alpha \iff \exists_\pi \big(\pi(0) = s \wedge \forall_{i \geq 0} \pi(i) \models_\upsilon \alpha\big)$,
7. $s \models_\upsilon E\alpha U\beta \iff \exists_\pi \big(\pi(0) = s \wedge \exists_{i \geq 0} \big[\pi(i) \models_\upsilon \beta \wedge \forall_{j<i} \pi(j) \models_\upsilon \alpha\big]\big)$,
8. $s \models_\upsilon EG^{\leq \eta}\alpha \iff \exists_\pi \big(\pi(0) = s \wedge \forall_{0 \leq i \leq \upsilon(\eta)} \pi(i) \models_\upsilon \alpha\big)$,
9. $s \models_\upsilon E\alpha U^{\leq \eta}\beta \iff \exists_\pi \big(\pi(0) = s \wedge \exists_{0 \leq i \leq \upsilon(\eta)} \big[\pi(i) \models_\upsilon \beta \wedge \forall_{j<i} \pi(j) \models_\upsilon \alpha\big]\big)$.

If α is a formula of RTCTL, then the validity of $s \models_\upsilon \alpha$ does not depend on the parameter valuation υ, as there are no parameters in the formula. In this case we write $M, s \models \alpha$ omitting the parameter valuation subscript.
Observe that for every formula α of RTCTL there exists a formula β of vRTCTL and a parameter valuation υ such, that $\alpha = \upsilon(\beta)$, where $\upsilon(\beta)$ denotes the formula obtained by substituting all the linear expressions with their evaluations under υ. For example the formula $EF^{\leq 5}(w_1 \wedge EG\neg c_1)$ can be obtained from $EF^{\leq \Theta_1}(w_1 \wedge EG\neg c_1)$ by valuation υ such that $\upsilon(\Theta_1) = 5$ or from $EF^{\leq \Theta_1 + \Theta_2}(w_1 \wedge EG\neg c_1)$ by valuation υ' such that $\upsilon'(\Theta_1) = 3$ and $\upsilon'(\Theta_2) = 2$.

The semantics of PRTCTL is defined in such a way that by eliminating the quantifiers we eventually arrive at a sequence of conjunctions and/or disjunctions of RTCTL formulae. By a *fresh (integer) variable* we mean a new variable which is not a parameter and is not present in the considered formula.

Definition 6. (Semantics of PRTCTL**)**
Let M *be a model,* s *– a state, and* α *– a formula of* PRTCTL. $M, s \models \alpha$ *denotes that* α *holds at the state* s *in the model* M. *The relation* $\models$ *is defined inductively as follows:*

1. $s \models \forall_\Theta \alpha(\Theta)$ *iff* $\bigwedge_{i_\Theta \geq 0} s \models \alpha(i_\Theta)$,
2. $s \models \forall_{\Theta \leq a} \alpha(\Theta)$ *iff* $\bigwedge_{0 \leq i_\Theta \leq a} s \models \alpha(i_\Theta)$,
3. $s \models \exists_\Theta \alpha(\Theta)$ *iff* $\bigvee_{i_\Theta \geq 0} s \models \alpha(i_\Theta)$,
4. $s \models \exists_{\Theta \leq a} \alpha(\Theta)$ *iff* $\bigvee_{0 \leq i_\Theta \leq a} s \models \alpha(i_\Theta)$,

where i_Θ *is a fresh integer variable.*

For example:

$$M, s \models \forall_{\Theta_1 \leq 1} \exists_{\Theta_2 \leq 2} EF^{\leq \Theta_1 + \Theta_2}(w_1 \wedge EG\neg c_1)$$

$$\iff \bigwedge_{0 \leq i_{\Theta_1} \leq 1} \bigvee_{0 \leq i_{\Theta_2} \leq 2} M, s \models EF^{\leq i_{\Theta_1} + i_{\Theta_2}}(w_1 \wedge EG\neg c_1).$$

It is straightforward to check that for a model M and a state s, $M, s \models_v EG\alpha \iff M, s \models_v EG^{\leq |M|}\alpha$ and $M, s \models_v E\alpha U\beta \iff M, s \models_v E\alpha U^{\leq |M|}\beta$. The proof of this fact is based on the observation that in every path a prefix of length greater or equal than $|M|$ contains a loop.
Recall Theorem 1 from [ET99]:

Theorem 1. *Let* M *be a model and* $Q_{1\Theta_1} \ldots Q_{n\Theta_n} \alpha(\Theta_1, \ldots, \Theta_n)$ *where* $Q_i \in \{\forall, \exists\}$ *and* $\alpha(\Theta_1, \ldots, \Theta_n) \in$ vRTCTL, *be a* PRTCTL *sentence. Then* $M, s \models Q_{1\Theta_1} \ldots Q_{n\Theta_n} \alpha(\Theta_1, \ldots, \Theta_n)$ *iff* $M, s \models Q_{1\Theta_1 \leq |M|} \ldots Q_{n\Theta_n \leq |M|} \alpha(\Theta_1, \ldots, \Theta_n)$.

In this paper we enhance the above theorem by the following lemma.

Lemma 1. *Let* M *be a model and* $Q_{1\Theta_1 \leq c_1} \ldots Q_{n\Theta_n \leq c_n} \alpha(\Theta_1, \ldots, \Theta_n)$ *where* $Q_i \in \{\forall, \exists\}$ *and* $\alpha(\Theta_1, \ldots, \Theta_n) \in$ vRTCTL *be a sentence of* PRTCTL. *Then* $M, s \models Q_{1\Theta_1 \leq c_1} \ldots Q_{n\Theta_n \leq c_n} \alpha(\Theta_1, \ldots, \Theta_n)$ *iff*
$M, s \models Q_{1\Theta_1 \leq min(c_1, |M|)} \ldots Q_{n\Theta_n \leq min(c_n, |M|)} \alpha(\Theta_1, \ldots, \Theta_n)$.

Proof. See the Appendix.

Basically, Theorem 1 allows for replacing the unrestricted quantifiers with their versions bounded with the size of the model and Lemma 1 states that it suffices to consider the bounds not greater that $|M|$. Therefore, in the rest of this paper we restrict our research to vRTCTL and PRTCTL formulae with superscripted modalities and restricted quantifiers.

3.3 Example

In Figure 1 the states of the model M are drawn as circles, whereas the values of the labelling function (a set of propositions assumed to be true) are rendered inside. The transitions are drawn as arrows connecting states. The presented Kripke structure is induced by the Petri net modelling the classical problem of

Mutual Exclusion for 3 processes (see Subsection 7.1). It is straightforward to check that:

$$M, start \models \forall_{\Theta_1 \leq 1} \exists_{\Theta_2 \leq 2} EF^{\leq \Theta_1 + \Theta_2}(w_1 \wedge EG \neg c_1),$$

$$M, start \models \exists_{\Theta_1 \leq 3} \forall_{\Theta_2} E\big(w_1 U^{\leq \Theta_1} EG^{\leq \Theta_2} r_2\big).$$

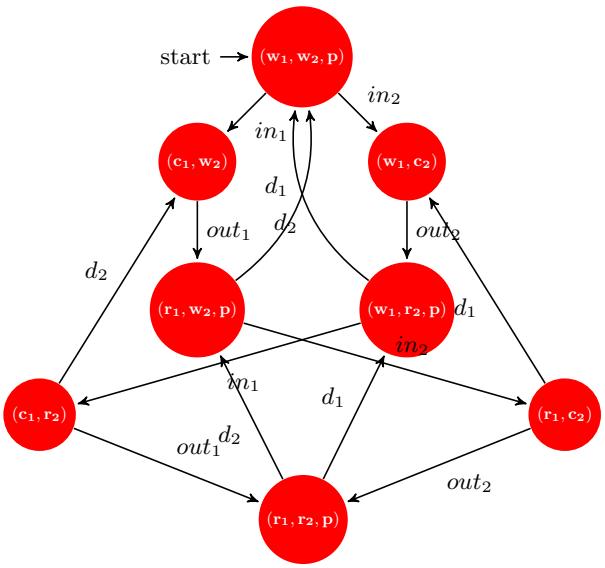
Notice that in the first formula there is no superscript over EG, nevertheless, as we have shown it can be rewritten in the equivalent form:

$$M, start \models \forall_{\Theta_1 \leq 1} \exists_{\Theta_2 \leq 2} EF^{\leq \Theta_1 + \Theta_2}(w_1 \wedge EG^{\leq 8} \neg c_1).$$

Similarly, the second formula can be rewritten in an equivalent form, with the parameter Θ_2 bounded by $|M|$:

$$M, start \models \exists_{\Theta_1 \leq 3} \forall_{\Theta_2 \leq 8} E\big(w_1 U^{\leq \Theta_1} EG^{\leq \Theta_2} r_2\big).$$

Fig. 1.



4 Bounded Semantics

The idea of bounded model checking is based on a concept of unfolding the computation tree of a given model only to a limited depth. In order to make things more clear we need the following definitions.

Definition 7. *Let M be a model and $k \in \mathbb{N}$. Let $Path_k$ be the set of all sequences $(s_0, \ldots, s_k)$ of states of M, where $s_i \rightarrow s_{i+1}$ for each $0 \leq i < k$. The pair $(Path_k, \mathcal{L})$ is called the* k-model *of M and is denoted by M_k.*

An element of $Path_k$ is called a *k-path* and denoted by π_k.

Definition 8. *Let M_k be a k-model of M and $\pi_k \in Path_k$. Define a function $loop : Path_k \longrightarrow 2^{\mathbb{N}}$ as:*

$$loop(\pi_k) = \{l \mid l \leq k \text{ and } \pi_k(k) \rightarrow \pi_k(l)\}.$$

A k-path π_k is called a *loop* if $loop(\pi_k) \neq \emptyset$. Observe that loops are essentially a way of representing some infinite paths in a finite way.

Definition 9. (Bounded semantics for vRTECTL**)**
Let M_k be a k-model, s – a state, $\alpha, \beta \in$ vRTECTL*, p – an atomic proposition, η – a linear expression, and υ – a parameter valuation. By $M_k, s \models_\upsilon \alpha$ let denote that α is true (valid) at the state s of M_k. Again, M_k is omitted if it is implicitly understood. Define the relation $\models_\upsilon$ as follows:*

1. $s \models_\upsilon p$ *iff* $p \in \mathcal{L}(s)$
2. $s \models_\upsilon \neg p$ *iff* $p \notin \mathcal{L}(s)$,
3. $s \models_\upsilon \alpha \wedge \beta$ *iff* $s \models_\upsilon \alpha$ *and* $s \models_\upsilon \beta$,
4. $s \models_\upsilon \alpha \vee \beta$ *iff* $s \models_\upsilon \alpha$ *or* $s \models_\upsilon \beta$,
5. $s \models_\upsilon EX\alpha$ *iff* $\exists_{\pi_k \in Path_k}(\pi_k(0) = s \wedge \pi_k(1) \models_\upsilon \alpha)$,
6. $s \models_\upsilon EG^{\leq \eta}\alpha$ *iff* $\exists_{\pi_k \in Path_k}\big(\pi_k(0) = s \wedge \big[((\upsilon(\eta) \leq k) \wedge \bigwedge_{0 \leq i \leq \upsilon(\eta)} \pi_k(i) \models_\upsilon \alpha) \vee ((\upsilon(\eta) > k) \wedge \bigwedge_{0 \leq i \leq k} \pi_k(i) \models_\upsilon \alpha \wedge loop(\pi_k) \neq \emptyset)\big]\big)$,
7. $s \models_\upsilon E(\alpha U^{\leq \eta}\beta)$ *iff* $\exists_{\pi_k \in Path_k}\big(\pi_k(0) = s \wedge \exists_{0 \leq i \leq min(k, \upsilon(\eta))}\big[\pi_k(i) \models_\upsilon \beta \wedge \bigwedge_{0 \leq j < i} \pi_k(i) \models_\upsilon \alpha\big]\big)$.

The above definition differs from its counterpart for ECTL ([PWZ02]) in the points 6 and 7. In case of the point 6, we need to consider two cases. The first case deals with the situation when α is checked along a finite path of length $\upsilon(\eta)$ smaller or equal than the depth k of the unfolding of the model. Each such a finite path is then a prefix of some k-path. In the second case we deal with the situation when α should be checked along a finite path of length strictly greater than k. Therefore we have to check α along the loop – hence we have the *loop* condition. Both the cases are combined in the disjunction. In case of the point 7, we check the existence of such a k-path π_k that the subformula β is valid on its position $\pi_k(i)$ where $i \leq min(k, \upsilon(\eta))$, and for all positions $\pi_k(j)$ where $j < i$ we have $\pi_k(j) \models_\upsilon \alpha$.

Definition 10. (Bounded semantics for PRTECTL**)**
Let M_k be a k-model of M, s – a state, α – a sentence of PRTECTL *and $a \in \mathbb{N}$. Define the relation $\models$ as follows:*

1. $M_k, s \models \forall_\Theta \alpha(\Theta)$ *iff* $\bigwedge_{i_\Theta \geq 0} M_k, s \models \alpha(i_\Theta)$,
2. $M_k, s \models \forall_{\Theta \leq a} \alpha(\Theta)$ *iff* $\bigwedge_{0 \leq i_\Theta \leq a} M_k, s \models \alpha(i_\Theta)$,
3. $M_k, s \models \exists_\Theta \alpha(\Theta)$ *iff* $\bigvee_{i_\Theta \geq 0} M_k, s \models \alpha(i_\Theta)$,

4. $M_k, s \models \exists_{\Theta \leq a} \alpha(\Theta)$ *iff* $\bigvee_{0 \leq i_\Theta \leq min(a,k)} M_k, s \models \alpha(i_\Theta)$,

where i_Θ *is a fresh integer variable.*

The next two lemmas bring forward the essential properties of bounded semantics. Basically they state that the truth of a formula in some k-model is maintained also in a larger l-model and in the whole model M. Therefore if we prove that a formula holds in the k-model (hopefully k is much smaller than $|M|$), then we obtain also the validity of the formula in the model M. These lemmas form a base for the idea of Bounded Model Checking. Namely, we start the search for a proof in a k-model with $k = 0$, then the length k of the paths is incremented until the proof is found or k reaches $|M|$. Then, the conditions 2 of Lemmas 2 and 3 guard that the property holds also in the model M. On the other hand, the conditions 3 of Lemmas 2 and 3 show, that if $k = |M|$ is reached and no proof was found, the considered property is not valid in M.

Lemma 2. *Let* M_k *be a* k*-model of* M, s *– a state,* υ *– a parameter valuation, and* α *– a formula of* vRTECTL. *Then, the following conditions hold:*

1. $\forall_{l \geq k} \big(M_k, s \models_\upsilon \alpha$ *implies* $M_l, s \models_\upsilon \alpha\big)$,
2. $M_k, s \models_\upsilon \alpha$ *implies* $M, s \models_\upsilon \alpha$,
3. $M, s \models_\upsilon \alpha$ *implies* $M_{|M|}, s \models_\upsilon \alpha$.

Proof. (Sketch) The proof is straightforward. The first and second condition is proved by induction on the length of a formula. In order to prove the third condition notice that each infinite path in the model M contains a looped prefix of length smaller or equal than $|M|$.

Notice that Lemma 2 has its counterpart concerning PRTECTL as stated below.

Lemma 3. *Let* M *be a model,* s *– a state and* α *– a* PRTECTL *sentence. Then, the following conditions hold:*

1. $\forall_{l \geq k} \big(M_k, s \models \alpha$ *implies* $M_l, s \models \alpha\big)$,
2. $M_k, s \models \alpha$ *implies* $M, s \models \alpha$,
3. $M, s \models \alpha$ *implies* $M_{|M|}, s \models \alpha$.

Proof. (Sketch) The proof is based on the observation that the existential and universal quantifiers can be replaced by disjunctions and conjunctions, respectively. Then, the results of Lemma 2 are applied.

4.1 Example

Recall the formulae and the model M from Example 3.3. One can check that

$$M_2, start \models \forall_{\Theta_1 \leq 1} \exists_{\Theta_2 \leq 2} EF^{\leq \Theta_1 + \Theta_2}(w_1 \wedge EG \neg c_1),$$

while this property does not hold in the bounded semantics for the k–models with k strictly smaller than 2. Similarly, we have

$$M_2, start \models \exists_{\Theta_1 \leq 3} \forall_{\Theta_2} E\big(w_1 U^{\leq \Theta_1} EG^{\leq \Theta_2} r_2\big),$$

while this does not hold for the k–models with k strictly smaller than 2.

5 Bounded Model Checking

The algorithm of the bounded model checking is based on the idea of a translation of a part of the model and the formula to a propositional formula. Satisfiability of the result means that the translated formula is true in the model. In the first part of this section we formulate definitions and theorems concerning submodels, the second part presents the rules for the translation, whereas the last part includes the description of the BMC algorithm.

5.1 Submodels

We aim at giving a method of checking the validity of temporal formulae in k-models. In order to obtain the acceptable efficiency, the algorithm works on submodels of the k-model.

Definition 11. *Let $M_k = (Path_k, \mathcal{L})$ be the k-model. A substructure $M'_k = (Path'_k, \mathcal{L}')$, where $Path'_k \subseteq Path_k$ and $\mathcal{L}'$ is the restriction of $\mathcal{L}$ to the states present in the paths of $Path'_k$ is called a* submodel *of M_k.*

The bounded semantics of vRTECTL formulae and PRTECTL sentences over submodels is defined as for k-models. If $M'_k = (Path'_k, \mathcal{L}')$ and $M''_k = (Path''_k, \mathcal{L}'')$ are submodels of some k-model M_k, such that $Path''_k \subseteq Path'_k$, we write $M''_k \subseteq M'_k$.

Lemma 4. *Let M_k be a k-model, M'_k and M''_k – its submodels, such that $M''_k \subseteq M'_k$ and s a state present in some path of M''_k. Then, we have:*

1. $M''_k, s \models_\upsilon \alpha \Rightarrow M'_k, s \models_\upsilon \alpha$ *for* $\alpha \in$ vRTECTL *and any parameter valuation* υ,
2. $M''_k, s \models \alpha \Rightarrow M'_k, s \models \alpha$, *for* $\alpha \in$ PRTECTL.

Proof. (Sketch) The first part of the lemma is easily proved by the structural induction. In order to prove the second part, notice that in the bounded semantics the non-modal quantifiers are rewritten as, respectively, conjunctions or disjunctions, and use the result of the first part.

It was proven in [PWZ02] that in order to determine the truth of an ECTL formula in M_k it is sufficient to consider only submodels of a size given by a special function on the checked formula. We extend these results to vRTECTL and PRTECTL.

Definition 12. *Let $\alpha, \beta \in$ vRTECTL, p – an atomic proposition, η – a linear expression and υ – a parameter valuation. Recall that Υ is the set of all parameter valuations. We define recursively the special function g_k : vRTECTL $\times\, \Upsilon \longrightarrow \mathbb{N}$ as follows:*

1. $g_k(p, \upsilon) = g_k(\neg p, \upsilon) = 0$,
2. $g_k(\alpha \vee \beta, \upsilon) = max(g_k(\alpha, \upsilon), g_k(\beta, \upsilon))$,
3. $g_k(\alpha \wedge \beta, \upsilon) = g_k(\alpha, \upsilon) + g_k(\beta, \upsilon)$,
4. $g_k(EX\alpha, \upsilon) = g_k(\alpha, \upsilon) + 1$,

5. $g_k(EG^{\leq\eta}\alpha, \upsilon) = (min(\upsilon(\eta), k) + 1) \cdot g_k(\alpha, \upsilon) + 1$,
6. $g_k(E\alpha U^{\leq\eta}\beta, \upsilon) = min(\upsilon(\eta), k) \cdot g_k(\alpha, \upsilon) + g_k(\beta, \upsilon) + 1$.

Definition 13. *Let* $\alpha \in$ PRTECTL. *We define recursively the special function* f_k : PRTECTL $\longrightarrow \mathbb{N}$ *as follows:*

1. *if* $\alpha \in$ RTCTL *then* $f_k(\alpha) = g_k(\alpha, \upsilon)$ *for any* υ,
2. *if* $\alpha = \forall_{\Theta \leq c}\beta(\Theta)$ *then* $f_k(\alpha) = \sum_{i_\Theta \leq c} f_k(\beta(i_\Theta))$,
3. *if* $\alpha = \exists_{\Theta \leq c}\beta(\Theta)$ *then* $f_k(\alpha) = max_{i_\Theta \leq min(c,k)}\{f_k(\beta(i_\Theta))\}$

where i_Θ *is a fresh integer variable.*

As the RTCTL formulae considered in the condition 1 of Definition 13 contain no free parameters, the above definition is unambiguous. The following lemmas state that we can determine the truth of vRTECTL and PRTECTL formulae in the k-model using submodels of size bounded by the value of the appropriate function f_k or g_k.

Lemma 5. *Let* $\alpha \in$ vRTECTL, M_k *be the* k*-model and* υ *– a parameter valuation. For any state* s *present in some path of* M_k, $M_k, s \models_\upsilon \alpha$ *if and only if there exists a submodel* M'_k *of* M_k *such that* $M'_k, s \models_\upsilon \alpha$ *and* $|Path'_k| \leq g_k(\alpha, \upsilon)$.

Proof. (Sketch) The "if" part follows directly from Lemma 4. For the "only if" part, use induction on the length of a formula and Lemma 4.

Lemma 6. *Let* β *be a* PRTECTL *sentence and* M_k *be the* k*-model. For any state* s *present in some path of* M_k, $M_k, s \models \beta$ *if and only if there exists a submodel* M'_k *of* M_k *such that* $M'_k, s \models \beta$ *and* $|Path'_k| \leq f_k(\beta)$.

Proof. (Sketch) The proof uses the similar observation as in the proof of Lemma 1 – by recalling the results of Lemma 5 for one-parameter vRTECTL formulae and the structural induction on the number of the nonmodal quantifiers.

From Lemmas 5,6, Lemma 4 (notice that the k-model is also a submodel) and Lemmas 2,3 we obtain that the truth of a formula in some submodel of size bounded by the appropriate g_k or f_k function implies the truth in a model. On the other hand, Lemmas 2,3 state that if a formula is true in a model, then it is also true in some k-model, or equivalently, by Lemmas 2,3 in its submodel of size bounded by the value of appropriately g_k or f_k.

5.2 Translation to SAT

In order to translate the problem of validity of a sentence $\alpha \in$ PRTECTL in the submodel M'_k to the problem of satisfiability of a propositional formula $\left[\alpha\right]_k$ we have to encode M'_k and α, and then combine the results together. We present an adapted version of the efficient translation introduced in [Zbr08].

Consider the model M. As the number of the states of M is finite, they can be perceived as a bit vectors of the length $r = \lceil log|M| \rceil$. Therefore, we can perceive

the states as the valuations of the vector $w = (w_1, \ldots, w_r)$. This vector is called a *global state variable* while each its member w_i is called a *state variable*. Denote by $\mathcal{SV}$ a set of state variables, then a valuation $V : \mathcal{SV} \longrightarrow \{0,1\}$ naturally extends to the valuation of global state variables $\hat{V} : \mathcal{SV}^r \longrightarrow \{0,1\}^r$ in such a way that $\hat{V}(w_1, \ldots, w_r) = (V(w_1), \ldots, V(w_r))$. With a slight notational abuse, we denote by $\hat{V}(w)$ a state encoded by bit vector. The *symbolic k-path* is a vector of global state variables. As we need a number of symbolic k-paths to represent the k-paths in a translated submodel, by $(w_{0,i}, w_{1,i}, \ldots, w_{r,i})$ we denote the i-th symbolic k-path, where $w_{j,i}$ is a global state variable.
Let w, w' be global state variables, s a state and p a proposition. In the rules of the translation the following propositional formulae are used:

1. $p(w)$ denotes a formula such that $V \models p(w)$ iff $p \in \mathcal{L}(\hat{V}(w))$,
2. $T(w, w')$ denotes a formula such that $V \models T(w, w')$ iff $\hat{V}(w) \rightarrow \hat{V}(w')$ (i.e., there exists a transition between $\hat{V}(w)$ and $\hat{V}(w')$ in the model M),
3. $H(w, w')$ is a formula such that $V \models H(w, w')$ iff $\hat{V}(w) = \hat{V}(w')$ (encoding the equality of states),
4. $L_k(j) = \bigvee_{i=0}^{k} T(w_{k,j}, w_{i,j})$ encodes a loop, that is $V \models L_k(j)$ iff $loop((V(w_{0,j}), \ldots, V(w_{k,j}))) \neq \emptyset$,
5. $I_s(w)$ is a formula such that $V \models I_s(w)$ iff $\hat{V}(w) = s$ (encoding the initial state).

Let M be a model and A be a finite subset of $\mathbb{N}$. Then the *unfolding of the transition relation* is defined as

$$[M]_k^A := \bigwedge_{j \in A} \bigwedge_{i=0}^{k-1} T(w_{i,j}, w_{i+1,j}).$$

It is easy to see that $V \models [M]_k^A$ iff for each $j \in A$, $(V(w_{0,j}), \ldots, V(w_{k,j}))$ is a k-path in M. As the translation introduced in [Zbr08] was an essential improvement over the original one of [PWZ02], we follow A. Zbrzezny's approach in our work. We recall the following definitions from [Zbr08].
Let A and B be finite subsets of $\mathbb{N}$. By $A \prec B$ we denote, that $x < y$ for all $x \in A$ and $y \in B$. Let $k, m, p \in \mathbb{N}$ and $m \leq |A|$, then:

1. $\hat{g}_L(A, m)$ is the subset B of A such that $|B| = m$ and $B \prec A \backslash B$,
2. $\hat{g}_R(A, m)$ denotes the subset B of A such that $|B| = m$ and $A \backslash B \prec B$,
3. $h_X(A)$ is the set $A \backslash \{min(A)\}$,
4. if $k + 1$ divides $|A| - 1$ then $h_G(A, k)$ is the sequence of sets $(B_0, \ldots, B_k)$ such that $\bigcup_{i=0}^{k} B_i = A \backslash \{min(A)\}$, $|B_i| = |B_j|$ and $B_i \prec B_j$ for every $0 \leq i < j \leq k$,
5. if k divides $|A| - 1 - p$, then $h_U(A, k, p)$ denotes the sequence of sets $(B_0, \ldots, B_k)$ such that $\bigcup_{i=0}^{k} B_i = A \backslash \{min(A)\}$, $B_i \prec B_j$ for every $0 \leq i < j \leq k$, $|B_0| = \ldots = |B_{k-1}|$ and $|B_k| = p$.

We also need a sequence element selector, that is if $h_G(A, k) = (B_0, \ldots, B_k)$ then define $h_G(A, k)(i) = B_i$ for $0 \leq i \leq k$ and if $h_U(A, k, p) = (B_0, \ldots, B_k)$,

define $h_U(A,k,p)(i) = B_i$ for $0 \leq i \leq k$.
The functions $\hat{g}_L$ and $\hat{g}_R$ are used to divide the set of path indices into the two parts of the sizes sufficient to perform the independent translation of subformulae α and β of formula $\alpha \wedge \beta$. Similarly, the functions h_G and h_U are used to divide the set of path indices into the sequences (hence the use of the selector) of subsets which are of the sizes sufficient to perform the translation of subformulae α and α together with β of, respectively, formulae $EG^{\leq\eta}\alpha$ and $E\alpha U^{\eta}\beta$. For a more in-depth description we refer to [Zbr08].

Definition 14. (Translation of vRTECTL**)**
Let $\alpha, \beta \in$ vRTECTL, p – an atomic proposition, υ – a parameter valuation, η – a linear expression, $(m,n) \in \mathbb{N} \times \mathbb{N}$, and $A \subseteq \mathbb{N}$.

$$\big[p\big]_k^{[m,n,A,\upsilon]} := p(w_{m,n}) \text{ and } \big[\neg p\big]_k^{[m,n,A,\upsilon]} := \neg p(w_{m,n}),$$

$$\big[\alpha \wedge \beta\big]^{[m,n,A,\upsilon]} := \big[\alpha\big]^{[m,n,\hat{g}_L(A,g_k(\alpha,\upsilon)),\upsilon]} \wedge \big[\beta\big]^{[m,n,\hat{g}_R(A,g_k(\beta,\upsilon)),\upsilon]},$$

$$\big[\alpha \vee \beta\big]^{[m,n,A,\upsilon]} := \big[\alpha\big]^{[m,n,\hat{g}_L(A,g_k(\alpha,\upsilon)),\upsilon]} \wedge \big[\beta\big]^{[m,n,\hat{g}_L(A,g_k(\beta,\upsilon)),\upsilon]},$$

$$\big[EX\alpha\big]^{[m,n,A,\upsilon]} := H(w_{m,n}, w_{0,min(A)}) \wedge \big[\alpha\big]_k^{[1,min(A),h_X(A),\upsilon]}.$$

The translation of the formula $EG^{\leq\eta}\alpha$ depends on the value of $\upsilon(\eta)$. If $\upsilon(\eta) > k$, then:

$$\big[EG^{\leq\eta}\alpha\big]^{[m,n,A,\upsilon]} := H(w_{m,n}, w_{0,min(A)}) \wedge L_k(min(A)) \wedge \bigwedge_{j=0}^{k} \big[\alpha\big]_k^{[j,min(A),h_G(A,k)(j),\upsilon]}$$

and if $\upsilon(\eta) \leq k$, then

$$\big[EG^{\leq\eta}\alpha\big]^{[m,n,A,\upsilon]} := H(w_{m,n}, w_{0,min(A)}) \wedge \bigwedge_{j=0}^{\upsilon(\eta)} \big[\alpha\big]_k^{[j,min(A),h_G(A,\upsilon(\eta))(j),\upsilon]}.$$

The translation of $E\alpha U^{\leq\eta}\beta$ is defined as follows:
$\big[E\alpha U^{\leq\eta}\beta\big]^{[m,n,A,\upsilon]} := H(w_{m,n}, w_{0,min(A)})$

$$\wedge \bigvee_{i=0}^{min(\upsilon(\eta),k)} \Big(\big[\beta\big]_k^{[i,min(A),h_U(A,min(\upsilon(\eta),k),g_k(\beta,\upsilon))(min(\upsilon(\eta),k)),\upsilon]}$$

$$\wedge \bigwedge_{j=0}^{min(\upsilon(\eta),k)-1} \big[\alpha\big]_k^{[j,min(A),h_U(A,min(\upsilon(\eta),k),g_k(\beta,\upsilon))(j),\upsilon]}\Big).$$

The above encoding is based on the definition of the bounded semantics for vRTECTL – see the Definition 9 together with the associated comment.

Definition 15. (Translation of PRTECTL**)**
Let $\alpha \in$ PRTECTL, $A \subseteq \mathbb{N}$, $(m,n) \in \mathbb{N} \times \mathbb{N}$, and $c \in \mathbb{N}$. If α contains no quantifiers and no free parameters, then:

$$\big[\alpha\big]_k^{[m,n,A]} := \big[\alpha\big]_k^{[m,n,A,\upsilon]}, \text{ where } \upsilon \text{ is any parameter valuation.}$$

As in the above case $\alpha \in$ vRTECTL *and it contains no free parameters, the choice of* υ *is irrelevant.*

$$\big[\forall_{\Theta\leq c}\alpha(\Theta)\big]_k^{[m,n,A]} := \big[\alpha(c)\big]_k^{[m,n,\hat{g}_L(A,f_k(\alpha(c)))]} \wedge \big[\forall_{\Theta\leq c-1}\alpha(\Theta)\big]_k^{[m,n,\hat{g}_R(A,f_k(\forall_{\Theta\leq c-1}\alpha(\Theta)))]},$$

Let $d = min(c,k)$*, then:*

$$\big[\exists_{\Theta\leq c}\alpha(\Theta)\big]_k^{[m,n,A]} := \big[\alpha(d)\big]_k^{[m,n,\hat{g}_L(A,f_k(\alpha(d)))]} \vee \big[\exists_{\Theta\leq d-1}\alpha(\Theta)\big]_k^{[m,n,\hat{g}_L(A,f_k(\exists_{\Theta\leq d-1}\alpha(\Theta)))]}.$$

Let M_k be the k-model. If $\alpha \in$ vRTECTL and υ is a parameter valuation, then define $G_k(\alpha,\upsilon) := \{i \in \mathbb{N} \mid 1 \leq i \leq g_k(\alpha,\upsilon)\}$. Similarly, if $\beta \in$ PRTECTL then define $F_k(\beta) := \{i \in \mathbb{N} \mid 1 \leq i \leq f_k(\beta)\}$. The sets G_k and F_k contain the indices of symbolic k-paths used to perform the translation. The formulae $\big[M\big]_k^{G_k(\alpha,\upsilon)}$ and $\big[M\big]_k^{F_k(\beta)}$ encode all the M_k submodels of the size not greater than needed to validate the truth of formulae α, β as indicated in Lemmas 5, 6.
Now we are in the position to complete the translation of the problem of validity in vRTECTL and PRTECTL to the problem of satisfiability of propositional formulae. Let M_k be a k-model, $\alpha \in$ vRTECTL and υ be a parameter valuation. Denote

$$\big[M\big]_k^{\alpha,\upsilon} := \big[M\big]_k^{G_k(\alpha,\upsilon)} \wedge I_s(w_{0,0}) \wedge \big[\alpha\big]_k^{[0,0,G_k(\alpha,\upsilon),\upsilon]}.$$

Similarly, let $\beta \in$ PRTECTL, then denote

$$\big[M\big]_k^{\beta} := \big[M\big]_k^{F_k(\beta)} \wedge I_s(w_{0,0}) \wedge \big[\beta\big]_k^{[0,0,F_k(\beta)]}.$$

The following theorems ensure completeness and correctness of the translation.

Theorem 2. *Let* M_k *be a* k*-model of* M*,* υ *– a parameter valuation,* α*– a formula of* vRTECTL *containing at least one modality, and* s *a state. Then, the following equivalence holds:* $M_k, s \models_\upsilon \alpha$ *iff* $\big[M\big]_k^{\alpha,\upsilon}$ *is satisfiable.*

Proof. (Sketch) The modification of the proof of Theorem 3.1 from [Zbr08]. The proof is divided into two parts – the proof of correctness and the proof of completeness of the translation, both obtained by the induction on the length of the formula.

Theorem 3. *Let* M_k *be a* k*-model of* M*,* β *– a sentence of* PRTECTL *containing at least one modality, and* s *– a state. Then, the following equivalence holds:* $M_k, s \models \beta$ *iff* $\big[M\big]_k^{\beta}$ *is satisfiable.*

Proof. (Sketch) Replace the non-modal quantifiers in a formula of PRETCTL with, appropriately, conjunctions or disjunctions. To conclude, use Theorem 2.

5.3 Example

Consider the model M from Example 3.3 and the formula:

$$\alpha = \forall_{\Theta_1\leq 1}\exists_{\Theta_2\leq 2} EF^{\leq\Theta_1+\Theta_2}(w_1 \wedge EG\neg c_1).$$

The number of the paths needed to encode α in the 2–model is computed as following:

$$f_k(\alpha) = \sum_{i_{\Theta_1} \leq 1} max_{i_{\Theta_2} \leq 2} \{ f_k(EF^{\leq i_{\Theta_1} + i_{\Theta_2}}(w_1 \wedge EG\neg c_1)) \}.$$

Let $\beta = EF^{\leq i_{\Theta_1} + i_{\Theta_2}}(w_1 \wedge EG\neg c_1))$, and observe that if $i_{\Theta_1} \leq 1$ and $i_{\Theta_2} \leq 2$ are fixed, then $f_k(\beta) = g_k(\beta, \upsilon)$ where $\upsilon(\Theta_1) = i_{\Theta_1}$ and $\upsilon(\Theta_2) = i_{\Theta_2}$. As $g_k(true, \upsilon) = 0$, we have $g_k(\beta, \upsilon) = g_k(w_1 \wedge EG\neg c_1, \upsilon) + 1 = 2$, therefore $f_k(\alpha) = 4$. Thus, the encoding in the 2–model of M is as follows:

$$\left[\forall_{\Theta_1 \leq 1} \exists_{\Theta_2 \leq 2} EF^{\leq \Theta_1 + \Theta_2}(w_1 \wedge EG\neg c_1)\right]_2^{[0,0,\{1,2,3,4\}]}$$

$$= \left[\exists_{\Theta_2 \leq 2} EF^{\leq \Theta_2}(w_1 \wedge EG\neg c_1)\right]_2^{[0,0,\{1,2\}]} \wedge \left[\exists_{\Theta_2 \leq 2} EF^{\leq 1 + \Theta_2}(w_1 \wedge EG\neg c_1)\right]_2^{[0,0,\{3,4\}]}$$

$$= \bigvee_{i=0}^{2} \left[EF^{\leq i}(w_1 \wedge EG\neg c_1)\right]_2^{[0,0,\{1,2\}]} \wedge \bigvee_{j=1}^{2} \left[EF^{\leq j}(w_1 \wedge EG\neg c_1)\right]_2^{[0,0,\{3,4\}]}.$$

As the illustration of the further steps of the translation, consider:

$$\left[EF^{\leq 2}(w_1 \wedge EG\neg c_1)\right]_2^{[0,0,\{3,4\}]} = H(w_{0,0}, w_{0,3}) \wedge \bigvee_{i=0}^{2} \left[w_1 \wedge EG\neg c_1\right]_2^{[i,3,\{3,4\}]}$$

$$= H(w_{0,0}, w_{0,3}) \wedge \bigvee_{i=0}^{2} \left(\left[w_1\right]^{[i,3,\emptyset]} \wedge \left[EG\neg c_1\right]_2^{[i,3,\{4\}]} \right)$$

$$= H(w_{0,0}, w_{0,3}) \wedge \bigvee_{i=0}^{2} \left(p_{w_1}(w_{i,3}) \wedge H(w_{i,3}, w_{0,4}) \wedge L_2(4) \wedge \bigwedge_{j=0}^{2} \neg p_{c_1}(w_{j,4}) \right).$$

5.4 The BMC algorithm

Let M be a model and $\alpha \in$ PRTECTL.

```
BMCverifyPRTECTL(α)
     for k := 1 to |M|
         compute the translation [M]_k^{α,υ}
         if [M]_k^{α,υ} is satisfiable return true
     end for
     return false
```

Checking the satisfiability of a propositional formula is delegated to an efficient SAT-solver. Obviously the algorithm terminates in a finite number of iterations. By Theorem 2 and Lemma 3 the result is positive (that is – the translation of the formula α is satisfiable) if and only if α is valid in the state s of a model M. It is easy to present similar algorithm for checking the validity of vRTECTL formulae under a parameter valuation υ – the only difference is the choice of the appropriate translation.

6 Implementation of Parametric BMC for Elementary Net Systems

In this section we recall some basic definitions concerning Elementary Net Systems (called also Elementary Petri Nets) and present the implementation of BMC for a model generated by a net. The formulations of this section originate from [PWZ02]. We consider only the *safe* Petri Nets, i.e., each place can be marked with at the most one token.

6.1 Elementary Net Systems

Definition 16. *A* net *is a triple* $N = (B, E, F)$*, where* B *(the* places*) and* E *(the* transitions*) are finite sets satisfying* $B \cap E = \emptyset$*, the relation (called a* flow relation*)* $F \subseteq (B \times E) \cup (E \times B)$ *has the property that for every* $t \in E$ *there exists* $p, q \in B$ *such that* $(p, t), (t, q) \in F$*.*

Let N be a net and $t \in E$, then $\bullet t = \{p \in B \mid (p, t) \in F\}$ is called the *pre-set* of t and $t\bullet = \{p \in B \mid (t, p) \in F\}$ is called the *post-set* of t. A *configuration* of a net $N = (B, E, F)$ is a subset C of B. An usual method of visualisation of nets is where the places are rendered as circles, the transitions as boxes, the elements of flow relation as arrows, and the configuration C is represented by placing a "token" in every circle corresponding to a place in C. A place not marked by a token is called *free*.

Definition 17. *A quadruple* $EN = (B, E, F, C_{in})$*, where* (B, E, F) *is a net and* $C_{in} \subseteq B$ *is the initial configuration, is called an* elementary net system*.*

Definition 18. *Let* $EN = (B, E, F, C_{in})$ *be an elementary net system and* $t \in E$*.*

1. *Let* $C \subseteq B$ *be a configuration. If* t *is a transition,* $\bullet t \subseteq C$*, and* $(t\bullet \setminus \bullet t) \cap C = \emptyset$*, then the transition* t *is* enabled *in* C *(denoted by* $C[t\rangle$*).*
2. *Let* $C, D \subseteq B$ *be configurations. A transition* t fires *from* C *to* D *(denoted by* $C[t\rangle$*) if* $C[t\rangle$ *and* $D = (C \setminus \bullet t) \cup t\bullet$*.*
3. *Let* $t_1, \ldots, t_n \in E$*. A configuration* $C \subseteq B$ *is* reachable *if there are configurations* $C_0, C_1, \ldots, C_n \subseteq B$ *with* $C_0 = C_{in}, C_n = C$ *and* $C_{i-1}[t_i\rangle C_i$ *for all* $1 \leq i \leq n$*. We denote the set of all the reachable configuration by* C_{EN}*.*

Informally, the arrows of the flow relation can be thought of as the directed paths of movement of tokens. If there is an arrow directed from a place b to a transition t, then we say that b *enters* t. If there exists an arrow directed from a transition t to a place b, then we say that t *fills* b. The transition t is enabled if all the places entering t are marked with tokens and all the places filled by t and not entering the transition t are free. If a transition t fires, then the tokens from all the places entering t disappear and the tokens appear in all the places filled by t.

6.2 Implementation

Our goal is to construct a Kripke model reflecting the states (markings) and actions (firings) in an elementary net system. Consider an elementary net system $EN = (B, E, F, C_{in})$ and number the places of the net with integers smaller or equal than $n = |B|$. We use a set $\{p_1, \ldots, p_n\}$ of propositions, where p_i is interpreted as the presence of a token in the place number i. If w is a state, then by $p_i \in w$ we mean that the i-th place is marked in the corresponding configuration.
We define the model $M = (S, \rightarrow, \mathcal{L})$ for EN by placing $S = C_{EN}$ (the reachable configurations are the states), $w \rightarrow v$ iff there exists $t \in E$ such that $w[t\rangle v$ (the transitions model the firings) for $w, v \in S$, and $p_i \in \mathcal{L}(w)$ iff $p_i \in w$ (the labelling models the markings).
It is easy to see, that we can encode the states of S by valuations of a vector of the state variables $w = (w[1], \ldots, w[n])$, where $w[i] = p_i$ for $0 \leq i \leq n$. Moreover, let $P = \{1, \ldots, n\}$ and let $pre(t), post(t) \subseteq P$ be finite sets of the indices of the places of, respectively, $pre - set(t)$ and $post - set(t)$. Denote the initial state C_{in} by s and let $\xi(s) \subseteq P$ be the set of indices of the places in s.
We are in the position to present the definitions:

1. $I_s(w) := \bigwedge_{i \in \xi(s)} w[i] \wedge \bigwedge_{i \in P \setminus \xi(s)} \neg w[i]$,
2. $T(w,v) := \bigvee_{t \in E} \big(\bigwedge_{i \in pre(t)} w[i] \wedge \bigwedge_{i \in (post(t) \setminus pre(t))} \neg w[i] \wedge \bigwedge_{i \in (pre(t) \setminus post(t))} \neg v[i]$
   $\wedge \bigwedge_{i \in post(t)} v[i] \wedge \bigwedge_{i \in (P \setminus (pre(t) \cup post(t))) \cup (pre(t) \cap post(t))} w[i] \iff v[i] \big)$,
3. $p_i(w) := w[i]$,
4. $H(w,v) := \bigwedge_{1 \leq i \leq n} w[i] \iff v[i]$.

7 Experimental Results

We have implemented the presented algorithm on top of the BMC module of Verics model checking tool. The Elementary Net Systems are used as an input specification formalism, and PRTECTL is used as an input logic.

In order to show the performance and present some case studies we use standard scalable benchmarks. The detailed descriptions of these examples can be found in [PWZ02].

The tables with results show the following data in the columns from left to right: the formula verified, the number of processes (denoted by NoP), the depth k of the unfolding of the model, the size of the corresponding propositional formula (numbers of variables and clauses) together with the description of how much resources (time and memory) does the translation take, the time it took for MiniSat SAT solver to check the satisfiability, and finally the SAT? column indicating whether the tested formula is satisfiable (√) or not satisfiable (×).

The experiments were performed on a Linux machine with dual core 1.6 GHz processor. We tested satisfiability using the MiniSAT solver [Min06]. The presented models are relatively simple, yet classical, and the considered formulae were chosen as to show the difference between the expressive power of CTL and

PRECTL. As our work is still in its preliminary stage, we do not include any real-world example, however it should be mentioned that many of problems lead to models similar to presented in Examples 7.1 and 7.2. Tables 1 and 2 show some quantitative details of the experiments.

7.1 Mutual Exclusion

The elementary net system of Figure 2 models the well-known mutual exclusion problem. The system consists of $n+1$ processes (where $n \geq 2$) of which n compete for the access to the shared resource and one, called the permission process, guards so that no two processes use the resource simultaneously. The presence of a token in the place labelled by w_i means that the i-th process is waiting for the access to the critical section while the token in c_i means that the i-th process has acquired the permission and entered the critical section. The place r_i models the unguarded part of the process and the presence of token in place p indicates that the resource is available.

The Kripke structure constructed for 3 processes along the lines of Subsection

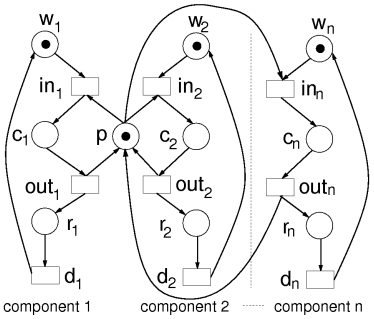


Fig. 2. Mutual exclusion

6.2 is presented in Figure 1. Let us consider the formula $\varphi_1^b = \forall_{\Theta \leq b} EF(\neg p \wedge EG^{\leq \Theta} c_1)$. We explore the validity of this formula with respect to the value of b. We can see that in order for the restricted EG operator to hold we need to have a path on which the first process enters its critical section and then other processes execute their local transitions d_i.

Let us explain how the verification works for this formula. For example, for 3 processes and $b = 2$, first the processes 2 and 3 enter their places r_2 and r_3 resp., then the process 1 enters its place c_1 and then 2 and 3 execute d_2 and d_3 respectively along the path of the length 2 on which c_1 holds. Of course, the order between 2 and 3 may be different. Notice that for $b = 3$ this formula does not hold in this model. Note that the non-parameterized counterpart of the formula φ_1, i.e., $EF(\neg p \wedge EGc_1)$ does not hold in our model, as there is no cycle in which c_1 is true starting in a state where p is false.

formula	NoP	k	PBMC				MiniSAT	SAT?
			vars	clauses	sec	MB	sec	√/×
φ_1^1	3	2	1063	2920	0.01	1.3	0.003	×
φ_1^1	3	3	1505	4164	0.01	1.5	0.008	√
φ_1^2	3	4	2930	8144	0.01	1.5	0.01	×
φ_1^2	3	5	3593	10010	0.01	1.6	0.03	√
φ_1^2	30	4	37825	108371	0.3	7.4	0.2	×
φ_1^2	30	5	46688	133955	0.4	8.9	0.52	√
φ_1^3	4	6	8001	22378	0.06	2.5	0.04	×
φ_1^3	4	7	9244	25886	0.05	2.8	0.05	√

Table 1. Mutual exclusion, testing the formula φ_1^b

7.2 Dining Philosophers

Another benchmark we consider is the Dining Philosophers Problem. Consider n ($n \geq 2$) philosophers sitting around a round table. Each philosopher has a plate in front of him, and between the two neighbouring plates there lies a fork. Whenever a philosopher eats, he uses both the forks from both the sides of his plate. When a philosopher has finished eating, he lays backs both of his forks on the table and starts thinking. The elementary net system modelling the system described above is shown in Fig. 3. The conditions r_i, w_i, s_i denote that i-th philosopher is thinking, waiting for both the forks and eating, respectively; c_i represents that the i-th fork is not taken.

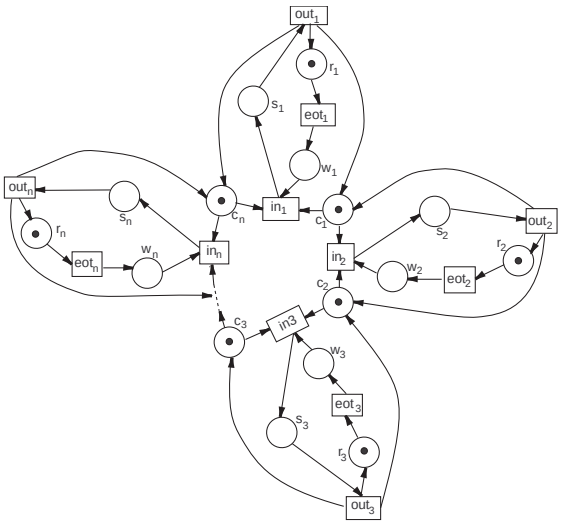


Fig. 3. Dining Philosophers

Let us consider the following properties: $\varphi_2^b = \forall_{\Theta \leq b} EF(s_1 \wedge EG^{\leq \Theta}(\neg c_1 \wedge \neg c_n \wedge \bigwedge_{1<i<n} c_i))$ and $\varphi_3^b = \forall_{\Theta \leq b} EF(s_1 \wedge EG^{\leq \Theta} \bigwedge_{1 \leq i \leq n} \neg c_i)$. The formula φ_2^b expresses that it is possible that in the future there exists a state where for the b time units the first philosopher is eating (therefore his forks are taken) while all the remaining forks are laid on the table. The formula φ_2^b states the similar property, namely that there exists a future state in which for the b time units the first philosopher eats while all the remaining forks are taken.

Note that φ_3^3 does not hold in the model, because there is no path of length 3 along which the first process can stay in the s_1 state.

formula	NoP	k	PBMC				MiniSAT	SAT?
			vars	clauses	sec	MB	sec	√/×
φ_2^1	4	1	1240	3347	0.01	1.5	0.008	×
φ_2^1	4	2	2124	5839	0.02	1.64	0.004	√
φ_2^3	4	1	2518	6821	0.01	1.8	0.004	×
φ_2^3	4	2	4298	11837	0.01	2.01	0.01	√
φ_3^1	4	3	3014	8343	0.02	1.8	0.1	×
φ_3^1	4	4	3898	10385	0.03	1.9	0.2	√
φ_3^2	4	3	4549	12600	0.04	2.07	0.008	×
φ_3^2	4	4	5875	16338	0.06	2.32	0.04	√
φ_3^2	10	9	37981	107724	0.25	7.3	3.78	×
φ_3^2	10	10	42043	119310	0.28	8	8.97	√

Table 2. Dining philosophers, testing the formulae φ_2^b and φ_3^b

8 Conclusions

In this paper we showed how parametric model checking can be performed by means of Bounded Model Checking. We presented an implementation and tested it against some benchmarks. Our work is still in its preliminary phase and can be extended in several directions. One of them is to investigate the remaining parametric logics presented in [ET99], of which General Parametric CTL (GPCTL) seems to be the most interesting. The formulae of GPCTL allow for referring to the number of occurrences of some event. In case of GPCTL, the computational complexity of the model checking problem is at least NP-complete, which is likely to make the BMC approach especially fruitful. Another possibility is to include the parameters to the model. Introducing the real time can also be considered, given that it has been done for non-parametric BMC.

References

[AHV93] R. Alur, T. Henzinger, and M. Vardi, *Parametric real-time reasoning*, Proc. of the 25th Ann. Symp. on Theory of Computing (STOC'93), ACM, 1993, pp. 592–601.

[BC03] M. Benedetti and A. Cimatti, *Bounded model checking for Past LTL*, Proc. of the 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03), LNCS, vol. 2619, Springer-Verlag, 2003, pp. 18–33.

[BCC+99] A. Biere, A. Cimatti, E. Clarke, M.Fujita, and Y. Zhu, *Symbolic model checking using SAT procedures instead of BDDs*, Proc. of the ACM/IEEE Design Automation Conference (DAC'99), 1999, pp. 317–320.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, *Symbolic model checking without BDDs*, Proc. of the 5th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), LNCS, vol. 1579, Springer-Verlag, 1999, pp. 193–207.

[BDR08] V. Bruyére, E. Dall'Olio, and J-F. Raskin, *Durations and parametric model-checking in timed automata*, ACM Transactions on Computational Logic **9(2)** (2008), 1–21.

[EC82] E. A. Emerson and E. Clarke, *Using branching-time temporal logic to synthesize synchronization skeletons*, Science of Computer Programming **2(3)** (1982), 241–266.

[ET99] E. A. Emerson and R. Trefler, *Parametric quantitative temporal reasoning*, Proc. of the 14th Symp. on Logic in Computer Science (LICS'99), IEEE Computer Society, July 1999, pp. 336–343.

[Hel01] K. Heljanko, *Bounded reachability checking with process semantics*, Proc. of the 12th Int. Conf. on Concurrency Theory (CONCUR'01), LNCS, vol. 2154, Springer-Verlag, 2001, pp. 218–232.

[HRSV01] T. Hune, J. Romijn, M. Stoelinga, and F. Vaandrager, *Linear parametric model checking of timed automata*, Proc. of the 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), LNCS, vol. 2031, Springer-Verlag, 2001, pp. 189–203.

[Min06] *MiniSat*, http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat, 2006.

[PWZ02] W. Penczek, B. Woźna, and A. Zbrzezny, *Bounded model checking for the universal fragment of CTL*, Fundamenta Informaticae **51(1-2)** (2002), 135–156.

[RB03] J-F. Raskin and V. Bruyère, *Real-time model checking: Parameters everywhere*, Proc. of the 23rd Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'03), LNCS, vol. 2914, Springer-Verlag, 2003, pp. 100–111.

[Woź03] B. Woźna, *ACTL* properties and bounded model checking*, Proc. of the Int. Workshop on Concurrency, Specification and Programming (CS&P'03) (L. Czaja, ed.), vol. 2, Warsaw University, 2003, pp. 591–605.

[Zbr08] A. Zbrzezny, *Improving the translation from ECTL to SAT*, Fundamenta Informaticae **85(1-4)** (2008), 513–531.

9 Appendix

The proof of Lemma 1:

Proof. Throughout this proof we denote $k = |M|$. We start with formulae ψ of vPRTCTL. Let v be a parameter valuation such that $v(\Theta') = c > k$. Define another valuation

$$v'(\Theta) = \begin{cases} v(\Theta), & \text{for } \Theta \neq \Theta' \\ k, & \text{for } \Theta = \Theta'. \end{cases} \tag{1}$$

We prove that for each state s, $M, s \models_v \psi \iff M, s \models_{v'} \psi$. The proof goes by the structural induction. The cases of $\psi = p$, $\psi = \neg\alpha$, $\psi = \alpha \vee \gamma$, $\psi = \alpha \wedge \gamma$ and

$\psi = EX\alpha$ are easy to prove.
Let us focus on proving $M, s \models_\upsilon EG^{\leq\beta}\alpha \iff M, s \models_{\upsilon'} EG^{\leq\beta}\alpha$.
If $\Theta' \notin Parameters(\beta)$, then the equivalence is valid by $\upsilon(\beta) = \upsilon'(\beta)$ and the inductive assumption. Assume that $\Theta \in Parameters(\beta)$.
If $M, s \models_\upsilon EG^{\leq\beta}\alpha$, then there exists a path π such that $\pi(0) = s$ and $M, \pi(i) \models_\upsilon \alpha$ for all $i \leq \upsilon(\beta)$. Now, from $\upsilon'(\beta) < \upsilon(\beta)$ and the inductive assumption we have $M, s \models_{\upsilon'} EG^{\leq\beta}\alpha$. Similarly, if $M, s \models_{\upsilon'} EG^{\leq\beta}\alpha$, then there exists a path π such that $\pi(0) = s$ and $M, \pi(i) \models_{\upsilon'} \alpha$ for all $i \leq \upsilon'(\beta)$. As $\upsilon'(\beta) \geq k$, there exists a $l \leq \upsilon'(\beta)$ such that $\pi(l) = \pi(n)$ for some $n < l$. Therefore we can define a path π' as follows:

$$\pi'(i) = \begin{cases} \pi(i), & \text{for } i < l \\ \pi(l - i + n), & \text{for } i \geq l. \end{cases} \tag{2}$$

As $\pi'(0) = \pi(0) = s$ and $M, \pi'(i) \models_{\upsilon'} \alpha$ for all $i \in \mathbb{N}$, by the inductive assumption we obtain $M, s \models_\upsilon \alpha$.
Now, let us move to the case of $\psi = E\alpha U^{\leq\beta}\gamma$. We deal with the case of $\Theta' \in Parameters(\beta)$ only. If $M, s \models_\upsilon E\alpha U^{\leq\beta}\gamma$, then there exists a path π having $\pi(0) = s$, such that for some $i \leq \upsilon(\beta)$ it occurs that $M, \pi(i) \models_\upsilon \gamma$ and $M, \pi(j) \models_\upsilon \alpha$ for all $j < i$. If $i \leq \upsilon'(\beta)$, then $M, s \models_{\upsilon'} E\alpha U^{\leq\beta}\gamma$ follows instantly from the inductive assumption. If $i > \upsilon'(\beta)$, notice that from $\upsilon'(\beta) > k$ we get $\pi(i) > k$, thus $\pi(i) = \pi(n)$ for some $n < k < \upsilon'(\beta)$ from which follows $M, s \models_{\upsilon'} E\alpha U^{\leq\beta}\gamma$.
Therefore, by induction on the number of the parameters we get that for formulae $\psi \in$ vPRTCTL, the parameter valuation υ and valuation υ' defined as $\upsilon'(\Theta) = min(\upsilon(\Theta), k)$ we have $M, s \models_\upsilon \psi \iff M, s \models_{\upsilon'} \psi$.
In order to prove the general case, consider a one–parameter vPRTCTL formula $g(\Theta)$. We have

$$M, s \models \forall_{\Theta \leq c} g(\Theta) \iff \bigwedge_{0 \leq i \leq c} M, s \models_{\{\Theta := i\}} g(\Theta).$$

Based on what we have already proven concerning vPRTCTL formulae, we can substitute $\{\Theta := i\}$ by $\{\Theta := min(i, k)\}$ in the right-hand side of the above formula, obtaining:

$$\bigwedge_{0 \leq i \leq c} M, s \models_{\{\Theta := min(i,k)\}} g(\Theta) \iff \bigwedge_{0 \leq i \leq min(c,k)} M, s \models_{\{\Theta := i\}} g(\Theta).$$

Therefore, we have $M, s \models \forall_{\Theta \leq c} g(\Theta) \iff M, s \models \forall_{\Theta \leq min(c,k)} g(\Theta)$. The equivalence $M, s \models \exists_{\Theta \leq c} g(\Theta) \iff M, s \models \exists_{\Theta \leq min(c,k)} g(\Theta)$ is proved in the similar way.
Finally, notice that for the formula $h = Q_{1\Theta_1 \leq min(c_1,k)} \dots Q_{t\Theta_t \leq min(c_t,k)} f$ of PRTCTL where $f \in$ vPRTCTL we can define a one–parameter subformula $\mu(\Theta_1) = Q_{2\Theta_2 \leq min(c_2,k)} \dots Q_{t\Theta_t \leq min(c_t,k)} f(\Theta_1)$. Now, this formula can be rewritten as a vPRTCTL formula $\hat{\mu}(\Theta_1)$ by substituting universal and existential quantifiers with, appropriately, conjunctions and disjunctions. Therefore, by induction on the number of parameters, the thesis of the lemma follows.

VerICS 2008 - a Model Checker for Time Petri Nets and High-Level Languages*

M. Kacprzak[1], W. Nabiałek[2], A. Niewiadomski[2], W. Penczek[2,3], A. Półrola[4], M. Szreter[3], B. Woźna[5], and A. Zbrzezny[5]

[1] Bialystok University of Technology, FCS, Wiejska 45a, 15-351 Bialystok, Poland
[2] University of Podlasie, ICS, Sienkiewicza 51, 08-110 Siedlce, Poland
[3] Polish Academy of Sciences, ICS, Ordona 21, 01-237 Warsaw, Poland
[4] University of Lodz, FMCS, Banacha 22, 90-238 Lodz, Poland
[5] Jan Dlugosz University, IMCS, Armii Krajowej 13/15, 42-200 Czestochowa, Poland
verics@ipipan.waw.pl

Abstract. The paper presents the current stage of the development of VerICS - a model checker for high-level languages, as well as real-time and multi-agent systems. Depending on the type of a system considered, it enables to test various classes of properties - from reachability to temporal, epistemic and deontic formulas. The model checking methods used to this aim include both SAT-based and enumerative ones. In the paper we focus on new features of the verifier: model checking of time Petri nets (TPNs) as well as of high-level languages: UML, Java, and Promela.

1 Introduction

The paper presents the current stage of the development of VerICS, a model checker for high-level languages, as well as real-time and multi-agent systems. Depending on the type of a system considered, the verifier enables to test various classes of properties - from reachability of a state satisfying certain conditions to more complicated features expressed by formulas of (timed) temporal, epistemic, or deontic logics. The model checking methods implemented include both SAT-based and enumerative ones (where by the latter we mean these consisting in generating abstract models for systems). Our previous work [18] presenting VerICS dealt mainly with verification of real-time systems (RTS) and multi-agent systems (MAS). In this paper we focus on VerICS' new features, i.e., SAT-based model checking for time Petri nets and systems specified in UML [28], Java [10], and Promela [12]. Next, we discuss some related verification approaches and tools.

The well-known tools for time Petri nets include the systems discussed below. Tina [2] is a toolbox for analysis of (time) Petri nets, which constructs state class graphs (abstract models) and exploits them for LTL, CTL, or reachability verification. Romeo [34] is a tool for time Petri nets analysis, which provides

* Partly supported by the Ministry of Education and Science under the grant No. N N516 370436 and N N206 258035.

several methods for translating TPNs to timed automata and computation of state class graphs. CPN Tools [4] is a software package for modelling and analysis of both timed and untimed coloured Petri nets, enabling their simulation, generating occurrence (reachability) graph, and analysis by place invariants.

There have been a lot of attempts to verify UML state machines - all of them based on the same idea: translate an UML specification to the input language of some model checker, and then perform verification using the model checker. Some of the approaches [15, 20] translate UML to the language Promela and then make use of the Spin [12] model checker. Other [7, 19] exploit timed automata as an intermediate formalism and exploit UPPAAL [1] for verification. The third group of tools (e.g., [8]) apply the symbolic model checker NuSMV [5] via translating UML to its input language. One of the modules of VeriCS follows this idea. An UML subset is translated to the Intermediate Language (IL) of VeriCS. However, we have developed also a symbolic model checker that deals directly with UML specifications by avoiding any intermediate translations. The method is implemented as the module BMC4UML.

Another situation prevails in the field of Promela verification. There exists only a few model checkers for Promela and its time extensions. SPIN [12] is a model checker for Promela specifications. Correctness properties can be specified as system or process invariants (using assertions), linear temporal logic formulas (LTL), formal Büchi automata, or more broadly as general omega-regular properties in the syntax of never claims. As the first attempt to verification of timed systems, Real Time Promela was developed [37]. This extension of Promela introduces explicit definitions of clocks that can be used in expressions and reset. The other approach, Discrete Time Promela [3], instead of clocks, introduces a new special type - count down timer. Timers can be set to some values and tested if they have expired. We offer a translator of Timed Promela [23] (a large subset of Promela extended by time annotations) to timed automata with discrete data (TADD) [42] as a VeriCS module.

Model checking of Java programs has become increasingly popular during the last decade. However, to the best of our knowledge, there are only two existing model checkers that can verify Java codes: JavaPathFinder (JPF) [11, 29] and Bandera [6]. JPF is a system to verify executable Java bytecode programs and it is one of the backend model-checkers supported by Bandera. Thus, both tools operate on the Java bytecode. On the contrary, we analyse Java programs themselves and translate them to a network of TADDs.

The rest of the paper is organised as follows. In Section 2 we briefly present a theoretical background of the SAT-based verification methods implemented in our tool (i.e., bounded and unbounded model checking). The next section contains a description of the verification system. In Section 4 we provide some experimental results obtained for several typical benchmarks used to test efficiency of model checkers. Finally, Section 5 contains a summary and some concluding remarks.

2 Theoretical Background

A network of communicating (timed) automata is the basic formalism of VerICS for modelling a system to be verified. Timed automata are used to specify RTS (possibly with clock differences expressing constraints on their behaviour), whereas timed or untimed automata are applied to model MAS (possibly extended in a way to handle certain features of interest, like deontic automata in [17]). The current version of VerICS makes extensive use also of timed automata extended by integer variables, called timed automata with discrete data (TADD) [42]. A set (*network*) of timed automata with discrete data consists of n TADDs which run in parallel. The automata communicate with each other via shared (i.e., common for some automata) variables, and perform transitions with shared labels synchronously. We assume the scheme of *multi-synchronisation*, which requires the transitions with a shared label to be executed synchronously by each automaton that contains this label in its set of labels. To obtain a clear semantics of variable updating it is necessary to fix the order of instructions in the case of synchronous execution of transitions. Thus, the transition whose instruction is to be taken first (called the *output transition*) is marked with the symbol !, whereas these which are to be taken later (the *input* ones) are marked with the symbol ?.

The tuples of *local states* of the automata in a network $\mathfrak{A}$ define the *global states* of the system considered. The set of all the possible *runs* (i.e., infinite evolutions from a given initial state) of a system modelled by $\mathfrak{A}$ gives us a *computation tree* which, after labelling the states with propositions from a given set PV which are true at these states (i.e., changing the tree into a *model*), is used to interpret the formulas of timed or untimed temporal logics. These are variants of Computation Tree Logic (CTL) or its timed version (TCTL) expressing properties to be checked. In the case of modelling a MAS we augment the model with *epistemic* or *deontic accessibility relations*. The resulting structure enables us to interpret formulas involving temporal operators, epistemic operators - to reason about *knowledge* of agents [9], and deontic operators - to reason about *correctness* of their behaviour.

The current version of VerICS accepts also an input in the form of *distributed time Petri nets* [32], which are another formalism used to specify RTS. A distributed time Petri net consists of a set of 1-safe sequential[6] TPNs (called *processes*), of pairwise disjoint sets of places, and communicating via joint transitions. Moreover, the processes are required to be *state machines*, which means that each transition has exactly one input place and exactly one output place in each process it belongs to. A state of the system considered is given by a marking of the net and by the values of the clocks associated with the processes[7].

SAT-based verification methods represent the models and properties of systems in the form of boolean formulas in order to reduce the state explosion.

[6] A net is sequential if none of its reachable markings concurrently enables two transitions.

[7] A detailed description of the nets, as well as their semantics, can be found in [33].

These for MAS involve bounded (BMC) and unbounded model checking (UMC). Currently, VerICS implements UMC for CTL_pK (Computation Tree Logic with knowledge and past operators) [16], and BMC for ECTLKD (the existential fragment of CTL extended with knowledge and deontic operators) [17, 30, 38, 39] as well as TECTLK (the existential fragment of timed CTL extended with knowledge operators) [21].

Considering verification of RTS, the current version of VerICS offers BMC for proving (un)reachability [40] (also for timed automata with clock differences [41]), and UMC for proving CTL properties for slightly restricted timed automata [36].

Below we present some intuition behind BMC and UMC.

2.1 Bounded Model Checking

Bounded Model Checking (BMC) is a symbolic method aimed at verification of temporal properties of distributed (timed) systems. It is based on the observation that some properties of a system can be checked over a part of its model only. In the simplest case of reachability analysis, this approach consists in an iterative encoding of a finite symbolic path (computation) as a propositional formula.

In order to apply Bounded Model Checking to testing reachability of a state satisfying a certain (usually undesired) property, we unfold the transition relation of a given automaton/TPN up to some depth k, and encode this unfolding as a propositional formula. Then, the property to be tested is encoded as a propositional formula as well, and satisfiability of the conjunction of these two formulas is checked using a SAT-solver. If the conjunction is satisfiable, one can conclude that a counterexample (a path to an undesirable state) was found. Otherwise, the value of k is incremented. The above process can be terminated when the value of k is equal to the diameter of the system, i.e., to the maximal length of a shortest path between its two arbitrary states.

2.2 Unbounded Model Checking

Unlike BMC, UMC is capable of handling the whole language of the logic. Like any SAT-based method, UMC consists in translating the model checking problem of a CTL_pK formula into the problem of satisfiability of a propositional formula. UMC exploits the characterisation of the basic modalities in terms of Quantified Boolean Formulas (QBF), and the algorithms that translate QBF and fixed point equations over QBF into propositional formulas. In order to adapt UMC for checking CTL_pK, we use three algorithms. The first one, implemented by the procedure "forall" (based on the Davis-Putnam-Logemann-Loveland approach) eliminates the universal quantifier from a QBF formula representing a CTL_pK formula, and returns the result in conjunctive normal form. The remaining algorithms, implemented by the procedures "gfp" and "lfp" calculate the greatest and the least fixed points for the modal formulas in use here. Ultimately, the technique allows for a CTL_pK formula to be translated into a propositional

formula in CNF, which characterises all the states of the model, where the formula holds. Next we apply a SAT-solver to check satisfiability of the obtained propositional formula.

3 Implementation

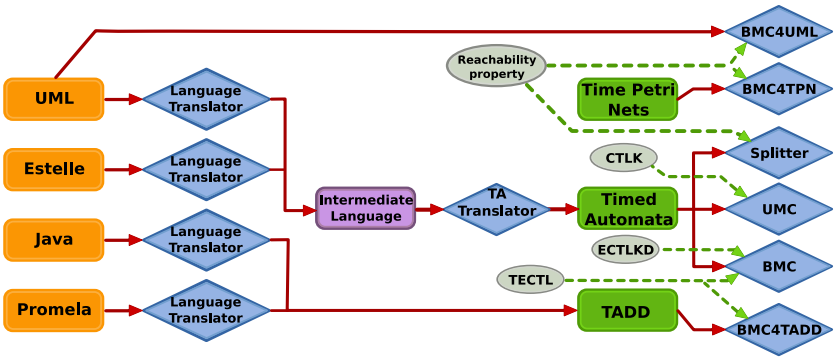


Fig. 1. Architecture of VerICS

The architecture of VerICS is shown in Fig. 1. The new components are:

- **UML to Intermediate Language (IL) translator**, which translates UML specification consisting of class, object and statemachine diagrams to corresponding IL program.
- **Java to TADD translator**, which translates a concurrent multi-threaded Java program to a network of TADDs.
- **Promela to TADD translator**, which generates a network of TADDs corresponding to the given Promela specification, possibly extended by time annotations.
- **BMC4UML module** - a Bounded Model Checker for UML, which applies SAT-based BMC algorithm directly on UML specification, avoiding intermediate translations.
- **BMC4TADD module** - a Bounded Model Checker for a network of TADDs.
- **BMC4TPN module** - a Bounded Model Checker for time Petri nets.

The remaining components, presented in [18], are listed below:

- **Estelle to IL translator**, which enables to handle specifications written in a subset of Estelle [13] (the standardised language for specifying communicating protocols and distributed systems);
- **IL to timed automata translator**, which, given an IL specification, generates the corresponding network of timed automata or the global timed automaton;

- **BMC module**, which implements BMC-based verification for the classes of properties shown in the figure. The SAT-solver used is MiniSat [22] or RSat [35]; the system can be configured to work with other solvers;
- **UMC module**, which provides preliminary implementations of UMC verification methods for properties described above. The module is integrated with a modified version of the SAT-solver ZChaff [44];
- **Splitter module**, which performs reachability verification on abstract models generated for timed automata.

VerICS is implemented in C++ and Java; its internal functionalities are available via a interface written in Java. The demo of current distribution can be accessible from `http://verics.ipipan.waw.pl`. A more detailed description of the tool, and in particular the new high-level languages translators, are presented in the following subsections.

3.1 Model checking of UML

At the moment we deal with model checking of UML in two ways: either translating an input specification to IL, and then use the standard verification path (translation to a network of TA and application of BMC, UMC or Splitting), or use the BMC module, which performs model checking directly, without intermediate translations.

Both the methods require as an input a specification in the XMI format, making use of a similar subset of UML. An input specification should consist of: one class diagram, one object diagram, and one state machine diagram per each class of the class diagram. The class and object diagrams define the static structure of the system, while the state machines determine its behaviour.

Translation of UML to Intermediate Language. In this section we give the main ideas behind our translation from UML to IL. The details can be found in [24]. Objects are mapped onto processes of IL and the number of UML objects corresponds to the number of IL processes. The attributes of objects are translated into process variables. We allow boolean, integer, and object types. The methods are translated into arrays of IL buffers, whereas each method call is realized by placing a special element - *call marker* - in the corresponding buffer, possibly followed by the method's parameters. Each of UML simple- and pseudo-states is mapped onto a state of an IL process. Entry and exit activities are merged with actions of incoming and outgoing transitions. The transitions in State Diagrams are translated directly into transitions of IL processes. A triggered event, a guard, and a sequence of actions can be associated with the transition. The time events in UML are translated into time constraints of IL transitions, using *delay* construction. The latter allows to specify the amount of time that may elapse before certain actions take place. The guards in UML are formed using attributes of objects and parameters of the actions called. These expressions are directly transformed into IL guards, using the variables that correspond to UML attributes and parameters.

BMC4UML - a Bounded Model Checking for UML. In order to perform symbolic model checking directly on an UML specification, an operational semantics of the considered UML subset is defined [25] in terms of a labelled transition system. Then, the transition system is symbolically encoded and the prototype implementation is developed [26].

In general the main ideas of BMC are applied to the transition system representing the executions of UML system. However, very complex elements of the UML state machines semantics (concerning e.g. hierarchy of states and regions, concurrent regions, priorities of transitions and properly handling of completion events and RTC-steps) require numerous non-trivial solutions at the level of symbolic encoding and implementation [27].

3.2 Translation of Java to TADDs

Below we sketch the main ideas behind a translation of a concurrent multi-threaded Java program to a network of TADDs. Each state of TADD is an abstraction of a state of the Java program, and each transition represents the execution of the code transforming this abstract state. The subset of Java that can be translated to TADDs contains: definitions of integer variables, standard programming language constructs like assignments, expressions with most operators, conditional statements and loops (*for*, *while*, *do while*), instructions *break* and *continue* without labels, definitions of classes, objects, constructors and methods, static and non-static methods and synchronisation of methods and blocks. There are recognised standard thread creation constructs as well as special methods: *Object.wait()*, *Object.notify()*, *Thread.sleep(int)*, and *Random.nextInt(int)*.

A theoretical method of constructing a network of TADDs that models a Java program is shown in [43]. To implement the translation we first translate a Java code to an internal assembler. Then, the resulting assembler is translated to timed automata with discrete data.

3.3 Translation of Promela to TADDs

The translation is performed in three stages. The first one consits in a translation of control flow of each Promela process into an automaton structure. The next one concerns representation of Promela data structures and operations on them. Finally, a set of TADDs corresponding to all the instances of the Promela processes is defined. The translation is inductive. The procedure starts with a block (a sequence of statements) representing the behaviour of a whole process and operates in a top-down fashion up to basic statements.

Each Promela process is translated to a TADD, and if it is necessary additional TADDs for *init* and *never-claim* processes are created. The local variables are mapped into global ones, while arrays and channels are translated onto set of global variables. Each Promela statement is represented as a transition, or - in the case of more complex constructions (e.g. loops or selections) - as a set of transitions. The operations on arrays and channels need also more than one

transition. Their number depends on the size of an array or the capacity of a channel.

Our translation covers most of Promela constructs. Moreover it is extended by time expressions, in order to specify real-time systems. The details can be found in [23].

3.4 Bounded Model Checking for TPNs

In order to benefit from the concurrent structure of a system, we consider distributed nets only [31], and assume that all their processes are *state machines*. It is important to mention that a large class of distributed nets can be decomposed to satisfy the above requirement [14]. The interpretation of such a system is a collection of sequential, non-deterministic processes with communication capabilities (via joint transitions). An example of a distributed TPN (Fischer's mutual exclusion protocol) is shown in Fig. 2. The net consists of three communicating processes with the sets of places $P_i = \{idle_i, trying_i, enter_i, critical_i\}$ for $i = 1, 2$, and $P_3 = \{place0, place1, place2\}$. All the transitions of the process N_1 and all the transitions of the process N_2 are joint with the process N_3.

The current implementation supports *reachability* checking, i.e., verification whether the system (net) can ever be in a state satisfying certain properties. The details of the method can be found in [33]. This solution can be also adapted to verification of other classes of properties for which BMC methods exist and is still to be implemented.

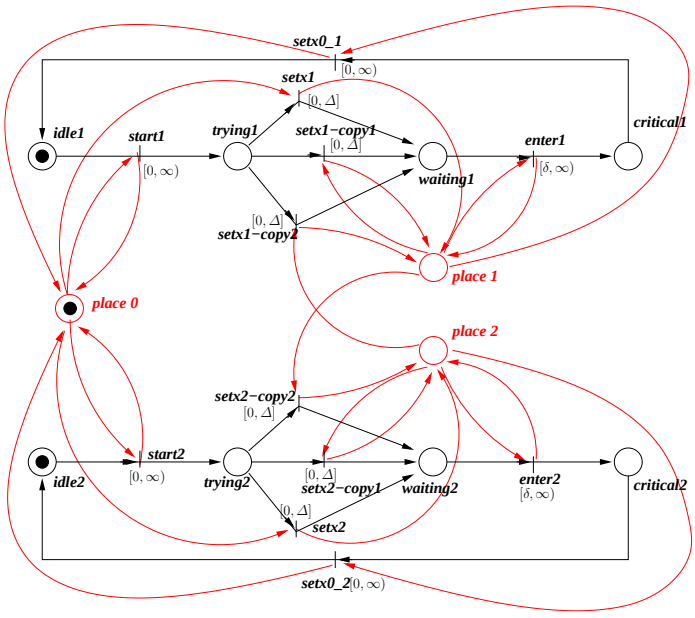
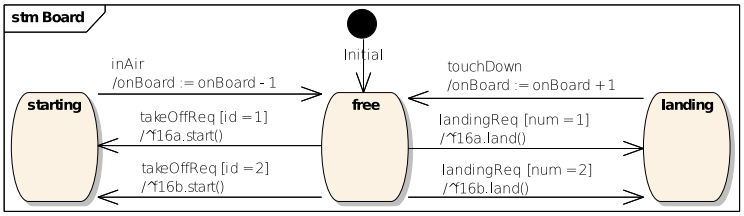


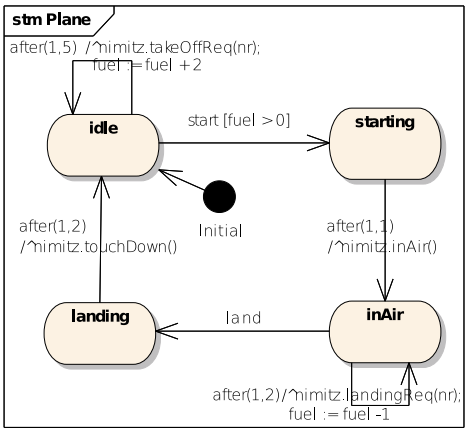
Fig. 2. A net for Fischer's mutual exclusion protocol for $n = 2$

4 Experimental Results

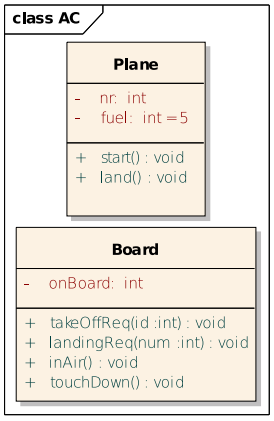
One of the important elements taken into account while rating a model checker is its efficiency. In this section we present some well known benchmarks: Alternating Bit Protocol (ABP) specified in UML and Java and Fischer's mutual exclusion protocol specified in TPNs and Promela, as well as the Aircraft Carrier (AC) UML specification.



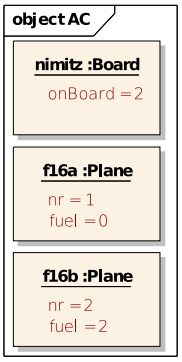
(a) State machine of class Board



(b) State machine of class Plane



(c) Class diagram



(d) Object diagram

Fig. 3. Specification of Aircraft Carrier system

Table 1(a) presents some experimental results of testing reachability of a deadlock state in ABP version written in Java (slightly modified in order to introduce deadlock) and Table 1(b) presents the results of the verification of the negation of the property: *"after the message and an acknowledgement have been received, both (Sender's and Receiver's) internal bits are equal"* for ABP specification in UML.

Table 2 presents the results of verification of the Aircraft Carrier (AC) specification (Fig. 3). AC consists of a ship and a number of aircrafts taking off and landing continuously, after issuing a request being accepted by the controller.

Table 1. Experimental results of verification of ABP

(a) Java via translation to TADDs

k	Clauses	BMC [s]	RSAT [s]	SAT
0	559	0.0	0.0	NO
12	57346	0.5	0.1	NO
24	121540	1.2	0.7	NO
50	**282621**	**2.8**	**82.4**	**YES**
	In total:	**33.2**	**122.9**	

(b) UML via translation to IL and TA

k	Clauses	BMC [s]	zChaff [s]	SAT
1	44098	0.27	0.01	NO
5	214598	1.44	0.40	NO
10	427723	2.84	7.22	NO
13	**555598**	**3.73**	**5.74**	**YES**
	In total:	**25.89**	**35.22**	

The events of answering these requests may be marked as deferred. Each aircraft refills fuel while on board and burns fuel while airborne. We check the property whether an aircraft can run out of fuel during its flight.

Moreover, we have introduced some elements of parametric reachability checking. Using our approach, we are able to verify not only that a property is reachable, but also to find a minimal (integer) time c, *when* this is the case (Table 2, the last column). More examples and a broader comparison with other model checkers for UML can be found in [26, 27].

Table 2. Results of verification of AC system (with/without deferred events)

N	k	Hugo+Uppaal [s]	BMC4UML [s]	Parametric [s], c = 4
3	19	1.32 / 1.25	67.59 / 51.26	31.34 / 22.64
4	20	13.15 / 11.41	101.58 / 81.28	45.44 / 42.38
5	21	147.43 / 95.67	155.63 / 132.34	60.49 / 37.01
6	22	Out of mem	257.08 / 216.42	52.23 / 75.08
7	23	- / -	686.06 / 421.85	101.86 / 199.09

We have also tested the systems modelling the standard Fischer's mutual exclusion protocol (Mutex). In general, the system consists of n processes which run in parallel. *Mutual exclusion* means that no two processes are in their critical sections at the same time. The preservation of this property depends on the relative values of the time-delay constants δ and Δ. In particular, the following holds: "*Fischer's protocol ensures mutual exclusion iff* $\Delta < \delta$".

A TPN model for Mutex consists of n time Petri nets, each one modelling a process, plus one additional net used to coordinate their access to the critical sections. The resulting distributed TPN, for the case of $n = 2$, is shown in Figure 2.

We have checked that if $\Delta \geq \delta$, then the mutual exclusion is violated. We considered the case with $\Delta = 2$ and $\delta = 1$. It turned out that the conjunction of the propositional formulae encoding the k-path and the negation of the mutual exclusion property is unsatisfiable for every $k < 12$. The witness was found for $k = 12$. We were able to test 40 processes. The results are shown in Table 3.

Table 3. Verification of time Petri Nets - Fischer's protocol (40 processes)

		tpnBMC				**RSat**		
k	**n**	**variables**	**clauses**	**sec**	**MB**	**sec**	**MB**	**sat**
0	-	1937	5302	0.2	3.5	0.0	1.7	NO
2	-	36448	107684	1.4	7.9	0.4	9.5	NO
4	-	74338	220335	2.9	12.8	3.3	21.5	NO
6	-	112227	332884	4.2	17.6	14.3	37.3	NO
8	-	156051	463062	6.1	23.3	257.9	218.6	NO
10	-	197566	586144	7.8	28.5	2603.8	1153.2	NO
12	-	240317	712744	9.7	34.0	87.4	140.8	YES
				32.4	34.0	2967.1	1153.2	

Table 4 presents experimental results for a timed Promela version of Fischer's mutual exclusion protocol. The time parameters of the protocol have been set in this way that the protocol is not correct. We have looked for the situation when any pair of processes is in their critical sections at the same time. The tests are done with latest distributions of RTSpin, DTSpin, and VERICS.

Table 4. Experimental results of verification of timed version of Fisher's Mutual Exclusion protocol specified in Promela.

# proc.	Spin				Verics				
	RTSpin		DTSpin			BMC		SAT	
	mem	cpu	mem	cpu	depth	mem	cpu	mem	cpu
8	34.21	5.4	57.86	0.08	12	3.4	0.34	5.83	0.26
80	—	—	146.03	2.49	12	12.0	12.57	93.65	85.67
100	—	—	228.06	4.43	12	16.4	20.21	256.38	339.32
130	—	—	529.19	30.76	12	24.4	32.56	103.91	48.97
135	—	—	—	—	12	25.8	34.91	459.14	1139.66

5 Final Remarks

As it can be seen from the above results, VERICS in many cases is able to handle relatively large examples taken from the standard scalable benchmarks. This allows to expect the same also in the case of "real world" systems. However, it should be said that the size of the system, which can be verified using the BMC method, depends on the formula tested: the more shallow the counterexample and the less paths needed to test the formula, the bigger system can be verified. On the other hand, a strong limitation for both the SAT-based methods we use are the capabilities of the SAT-solvers available, which, in many cases, are not

able to handle the set of clauses generated by the method, or to solve it in a reasonable time. This, however, proves also that the development of solvers can result in an improvement of efficiency of our tool.

References

1. J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, W. Yi, and C. Weise. New generation of UPPAAL. In *Proc. of the Int. Workshop on Software Tools for Technology Transfer*, 1998.
2. B. Berthomieu, P-O. Ribet, and F. Vernadat. The tool TINA - construction of abstract state spaces for Petri nets and time Petri nets. *International Journal of Production Research*, 42(14), 2004.
3. D. Bosnacki and D. Dams. Discrete-time Promela and SPIN. In *Proc. of the 5th Int. Conf. on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT'98)*, volume 1486 of *LNCS*, pages 307–310. Springer-Verlag, 1998.
4. S. Christensen, J. Jørgensen, and L. Kristensen. Design/CPN - a computer tool for coloured Petri nets. In *Proc. of the 3rd Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *LNCS*, pages 209–223. Springer-Verlag, 1997.
5. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NUSMV2: An open-source tool for symbolic model checking. In *Proc. of the 14th Int. Conf. on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 359–364. Springer-Verlag, 2002.
6. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. of the 22nd Int. Conf. on Software Engineering (ICSE 2000)*, pages 439–448. ACM, 2000.
7. K. Diethers, U. Goltz, and M. Huhn. Model checking UML statecharts with time. In *Proc. of the Workshop on Critical Systems Development with UML (CSDUML'02)*, pages 35–52. Technische Universität München, 2002.
8. J. Dubrovin and T. Junttila. Symbolic model checking of hierarchical UML state machines. Technical Report HUT-TCS-B23, Helsinki Institute of Technology, Espoo, Finland, 2007.
9. R. Fagin, J. Y. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge.* MIT Press, Cambridge, 1995.
10. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification. Third Edition.* Addison-Wesley, 2005.
11. K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA PathFinder. *Int. Journal on Software Tools for Technology Transfer*, 2(4):366–381, 1998.
12. G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Eng.*, 23(5):279–295, 1997.
13. ISO/IEC 9074(E), Estelle - a formal description technique based on an extended state-transition model. International Standards Organization, 1997.
14. R. Janicki. Nets, sequential components and concurrency relations. *Theoretical Computer Science*, 29:87–121, 1984.
15. T. Jussila, J. Dubrovin, T. Junttila, T. Latvala, and I. Porres. Model checking dynamic and hierarchical UML state machines. In *Proc. of the 3rd Int. Workshop on Model Design and Validation (MoDeVa 2006)*, pages 94–110. CEA, 2006.

16. M. Kacprzak, A. Lomuscio, T. Łasica, W. Penczek, and M. Szreter. Verifying multiagent systems via unbounded model checking. In *Proc. of the 3rd NASA Workshop on Formal Approaches to Agent-Based Systems (FAABS III)*, volume 3228 of *LNCS*, pages 189–212. Springer-Verlag, 2005.
17. M. Kacprzak, A. Lomuscio, A. Niewiadomski, W. Penczek, F. Raimondi, and M. Szreter. Comparing BDD and SAT based techniques for model checking Chaum's dining cryptographers protocol. *Fundamenta Informaticae*, 72(1-2):215–234, 2006.
18. M. Kacprzak, W. Nabiałek, A. Niewiadomski, W. Penczek, A. Półrola, M. Szreter, B. Woźna, and A. Zbrzezny. VerICS 2007 - a model checker for knowledge and real-time. *Fundamenta Informaticae*, 85(1-4):313–328, 2008.
19. A. Knapp, S. Merz, and C. Rauh. Model checking - timed UML state machines and collaborations. In *Proc. of the 7th Int. Symp. on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT'02)*, volume 2469 of *LNCS*, pages 395–416. Springer-Verlag, 2002.
20. J. Lilius and I. Paltor. vUML: A tool for verifying UML models. In *Proc. of the 14th IEEE Int. Conf. on Automated Software Engineering (ASE'99)*, pages 255–258. IEEE Computer Society, 1999.
21. A. Lomuscio, B. Woźna, and A. Zbrzezny. Bounded model checking real-time multi-agent systems with clock differences: Theory and implementation. In *Proc. of the 4th Int. Workshop on Model Checking and Artificial Intelligence (MoChArt'06)*, pages 62–78. ECCAI, 2006.
22. MiniSat. http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat, 2006.
23. W. Nabiałek, A. Janowska, and P. Janowski. Translation of timed Promela to timed automata with discrete data. *Fundamenta Informaticae*, 85(1-4):409–424, 2008.
24. A. Niewiadomski, W. Penczek, S. Lasota, and J. Kowalski. Weryfikacja UML z wykorzystaniem systemu VerICS. In *Mat. XII Konf. Systemy Czasu Rzeczywistego (SCR'06)*, pages 79–91. Wyd. Komunikacji i Łączności, 2006. In Polish.
25. A. Niewiadomski, W. Penczek, and M. Szreter. Semantyka operacyjna wybranych diagramów UML. Technical Report 1009, ICS PAS, Ordona 21, 01-237 Warsaw, March 2008. In Polish.
26. A. Niewiadomski, W. Penczek, and M. Szreter. Towards bounded model checking of UML. In *Proc. of the Int. Workshop on Concurrency, Specification and Programming (CS&P'08)*, volume 225(3) of *Informatik-Berichte*, pages 386–397. Humboldt University, 2008.
27. A. Niewiadomski, W. Penczek, and M. Szreter. Towards checking parametric reachability for UML state machines. In *Proc. of the 7th Int. Ershov Memorial Conf. 'Perspective of System Informatics' (PSI'09)*, 2009. To appear.
28. OMG. Unified Modeling Language. http://www.omg.org/spec/UML/2.1.2, 2007.
29. C. Pasareanu and W. Visser. Verification of Java progtams using symbolic execution and invariant generation. In *Proc. of the 11th Int. SPIN Workshop (SPIN'04)*, volume 2989 of *LNCS*, pages 164–181. Springer-Verlag, 2004.
30. W. Penczek and A. Lomuscio. Verifying epistemic properties of multi-agent systems via bounded model checking. In *Proc. of the 2nd Int. Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS'03)*, pages 209–216. ACM, 2003.
31. W. Penczek and A. Półrola. *Advances in Verification of Time Petri Nets and Timed Automata: A Temporal Logic Approach*, volume 20 of *Studies in Computational Intelligence*. Springer-Verlag, 2006.

32. W. Penczek, A. Półrola, B. Woźna, and A. Zbrzezny. Bounded model checking for reachability testing in time Petri nets. In *Proc. of the Int. Workshop on Concurrency, Specification and Programming (CS&P'04)*, volume 170(1) of *Informatik-Berichte*, pages 124–135. Humboldt University, 2004.
33. W. Penczek, A. Półrola, and A. Zbrzezny. SAT-based (parametric) reachability for distributed time Petri nets. In *Proc. of the Int. Workshop on Petri Nets and Software Engineering (PNSE'09)*, June 2009. To appear.
34. Romeo: A tool for time Petri net analysis. http://www.irccyn.ec-nantes.fr/irccyn/d/en/equipes/TempsReel/logs, 2000.
35. RSat. http://reasoning.cs.ucla.edu/rsat, 2006.
36. M. Szreter. *SAT-Based Model Checking of Distributed Systems.* PhD thesis, ICS PAS, January 2007.
37. S. Tripakis and C. Courcoubetis. Extending Promela and SPIN to real-time. In *Proc. of the 2nd Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *LNCS*, pages 329–348. Springer-Verlag, 1996.
38. B. Woźna, A. Lomuscio, and W. Penczek. Bounded model checking for deontic interpreted systems. In *Proc. of the 2nd Int. Workshop on Logic and Communication in Multi-Agent Systems (LCMAS'04)*, volume 126 of *ENTCS*, pages 93–114. Elsevier, 2005.
39. B. Woźna, A. Lomuscio, and W. Penczek. Bounded model checking for knowledge and real time. In *Proc. of the 4th Int. Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS'05)*, pages 165–172. ACM, 2005.
40. A. Zbrzezny. Improvements in SAT-based reachability analysis for timed automata. *Fundamenta Informaticae*, 60(1-4):417–434, 2004.
41. A. Zbrzezny. SAT-based reachability checking for timed automata with diagonal constraints. *Fundamenta Informaticae*, 67(1-3):303–322, 2005.
42. A. Zbrzezny and A. Półrola. Sat-based reachability checking for timed automata with discrete data. *Fundam. Inform.*, 79(3-4):579–593, 2007.
43. A. Zbrzezny and B. Woźna. Towards verification of Java programs in VerICS. *Fundamenta Informaticae*, 85(1-4):533–548, 2008.
44. L. Zhang. Zchaff. http://www.ee.princeton.edu/∼chaff/zchaff.php, 2001.

SAT-Based (Parametric) Reachability for Distributed Time Petri Nets ⋆

Wojciech Penczek[1,2], Agata Półrola[3], and Andrzej Zbrzezny[4]

[1] Institute of Computer Science, PAS, Ordona 21, 01-237 Warsaw, Poland
[2] Institute of Informatics, Podlasie Academy, Sienkiewicza 51, 08-110 Siedlce, Poland
penczek@ipipan.waw.pl
[3] University of Łódź, FMCS, Banacha 22, 90-238 Łódź, Poland
polrola@math.uni.lodz.pl
[4] Jan Długosz University, IMCS, Armii Krajowej 13/15, 42-200 Częstochowa, Poland
a.zbrzezny@ajd.czest.pl

Abstract. Formal methods - among them the model checking techniques - play an important role in the design and production of both systems and software. In this paper we deal with an adaptation of the bounded model checking methods for timed systems, developed for timed automata, to the case of time Petri nets. We consider distributed time Petri nets and parametric reachability checking, but the approach can be easily adapted to verification of other kinds of properties for which the bounded model checking methods exist. A theoretical description is supported by some experimental results, generated using an extension of the model checker VerICS.

1 Introduction

The process of design and production of both systems and software – among others, the concurrent ones – involves testing whether the product conforms its specification. To this aim, various kinds of formal methods can be applied. One of the possible approaches, widely used and intensively developed, are *model checking techniques*.

In order to perform a formal verification, the system to be tested is usually modelled using a theoretical formalism, e.g., a version of automata, Petri nets, state diagrams etc. Obviously, the kind of the formalism depends on the features of the system to be described. One of the approaches, used to represent concurrent systems with timing dependencies [10, 11, 20], are *time Petri nets* (TPNs) by Merlin and Farber [21]. After modelling the system in the above way, a suitable verification method is applied.

The main problem to cope with while verifying timed systems is the so-called *state explosion*: in order to check whether the system satisfies a property we usually need to search through its state space, which in most cases is very large due to infinity of the dense time domain. Furthermore, in the case of concurrent systems the size of the state space is likely to grow exponentially when the number of the components increases. So, searching for verification methods which are able to overcome the above problem is an important subject of research.

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

Bounded Model Checking (BMC) is an efficient verification technique whose main idea consists in translating a model checking problem solvable on a fraction of a model into a test of propositional satisfiability, which is then made using a SAT-checker. The method has been successfully applied to verification of both timed and untimed systems [3, 4, 7, 12, 16, 27, 31, 35]. In this paper, we show how to adapt the BMC methods, presented in [27, 34–36] and developed for timed automata, to the case of time Petri nets. The adaptation exploits, in some sense, a method of translating a time Petri net to a timed automaton, described in [28]. However, we perform no structural translation between these two formalisms, but use directly the transition relation defined by the translation. In order to benefit from the concurrent structure of the system, we focus on *distributed* nets (i.e., sets of communicating processes), and exploit a non-standard approach to their concrete semantics, which consists in associating a clock with each of the processes [28]. In this work, we deal with testing whether the system (net) can ever be in a state satisfying certain properties (i.e., with *reachability* checking), but the presented solutions can be also easily adapted to verification of other classes of properties for which BMC methods exist (see [23] for a survey). The algorithm has been implemented as an extension of the model checker VerICS [13]. The next topic we dealt with was searching for bounds on which the property tested can be reached (searching for a value of the parameter c in formulas $\mathrm{EF}^{\sim c}p$, corresponding to these considered in [14]). In the final part of the paper we provide some preliminary experimental results.

To our knowledge, no BMC method for time Petri nets has been defined so far, although some solutions for untimed Petri nets exist [16, 25]. Therefore, the main contribution of this work consists in showing how to apply and implement for TPNs the above technique of verification (a general idea of the approach has been already sketched in [23], but no details are given there). As a result, we obtain an efficient method of checking reachability, as well as searching for counterexamples for the properties expressible by formulas of the logics ACTL^* and TACTL. Although the adaptation of the BMC methods is almost straightforward, the practical consequences seem to be quite useful.

The rest of the paper is organised as follows: in Sect. 3 we introduce time Petri nets, and the abstraction of their state spaces, i.e., an *extended detailed region graphs*. In the further part we sketch the idea of reachability checking using BMC (Sect. 4), and show its implementation for time Petri nets (Sect. 5). Searching for bounds on time at which a state satisfying a property can be reached (parametric reachability) is considered in Sect. 6. Sections 7 and 8 contain experimental results and concluding remarks.

2 Related work

The methods of reachability checking for time Petri nets, mostly consisting in building an *abstract model* of the system, are widely studied in the literature [6, 5, 8, 9, 15, 19]. Detailed region graphs for time Petri nets, based on their standard semantics (i.e., the one associating a clock with each transition of the net) were presented in [22, 33]. Some BMC methods for (untimed) Petri Nets were described in [16, 26]. Parametric verification for time Petri nets was considered in [32].

The current work is a modification and extension of the paper [24] (published in proceedings of a local workshop with the status of a technical report).

3 Time Petri Nets

Let $\mathbb{R}_+$ denote the set of non-negative reals, $\mathbb{Q}$ the set of rationals, and $\mathbb{N}$ ($\mathbb{N}_+$) - the set of (positive) natural numbers. We start with a definition of time Petri nets:

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

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

- a *marking* of $\mathcal{N}$ is any subset $m \subseteq P$;
- a transition $t \in T$ is *enabled* at m ($m[t\rangle$ for short) if $\bullet t \subseteq m$ and $t\bullet \cap (m \setminus \bullet t) = \emptyset$; and *leads from* m *to* m', if it is enabled at m, and $m' = (m \setminus \bullet t) \cup t\bullet$. The marking m' is denoted by $m[t\rangle$ as well, if this does not lead to misunderstanding;
- $en(m) = \{t \in T \mid m[t\rangle\}$ is the set of all the transitions enabled at the marking m of $\mathcal{N}$;
- a marking $m \subseteq P$ is *reachable* if there exists a sequence of transitions $t_1, \ldots, t_l \in T$ and a sequence of markings $m_0, \ldots, m_l$ such that $m_0 = m^0$, $m_l = m$, and for each $i \in \{1, \ldots, l\}$ $t_i \in en(m_{i-1})$ and $m_i = m_{i-1}[t_i\rangle$;
- a marking m *concurrently enables* two transitions $t, t' \in T$ if $t \in en(m)$ and $t' \in en(m \setminus \bullet t)$;
- a net is *sequential* if no reachable marking of $\mathcal{N}$ concurrently enables two transitions.

It should be mentioned that the time Petri nets defined as above are often called *1-safe* in the literature.

Next, we introduce the notion of a *distributed time Petri net*. The definition is an adaptation of the one from [17]:

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

Notice that the function Eft_{i_1} (Lft_{i_1}) coincides with Eft_{i_2} (Lft_{i_2}, resp.) for the joint transitions of each two processes i_1 and i_2. The interpretation of such a system is a collection of sequential, non-deterministic processes with communication capabilities (via joint transitions).

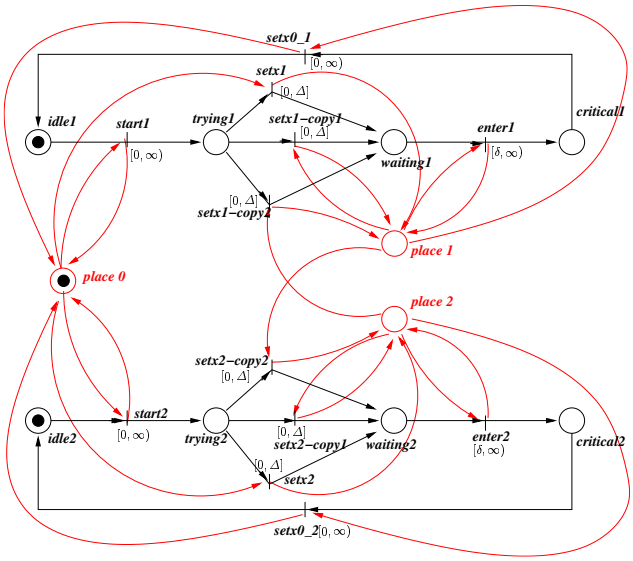


Fig. 1. A net for Fischer's mutual exclusion protocol for $n = 2$

An example of a distributed TPN (Fischer's mutual exclusion protocol) is shown in Fig. 1. The net consists of three communicating processes with the sets of places $P_i = \{idle_i, trying_i, enter_i, critical_i\}$ for $i = 1, 2$, and $P_3 = \{place0, place1, place2\}$. All the transitions of the process N_1 and all the transitions of the process N_2 are joint with the process N_3.

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

3.1 Concrete State Spaces and Models

The current state of the net is given by its marking and the time passed since each of the enabled transitions became enabled (which influences the future behaviour of the net). Thus, a *concrete state* σ of $\mathcal{N}$ can be defined as an ordered pair $(m, clock)$, where m is a marking, and $clock : \mathcal{I} \to \mathbb{R}_+$ is a function which for each index $\mathfrak{i}$ of a process of $\mathcal{N}$ gives the time elapsed since the marked place of this process became marked most recently [28]. The set of all the concrete states is denoted by Σ. The initial state of $\mathcal{N}$ is $\sigma^0 = (m^0, clock^0)$, where m^0 is the initial marking, and $clock^0(\mathfrak{i}) = 0$ for each $\mathfrak{i} \in \mathcal{I}$.

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

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

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

Then, we define a *timed run* of $\mathcal{N}$ starting at a state $\sigma_0 \in \Sigma$ (σ_0-*run*) as a maximal sequence of concrete states, transitions and time passings $\rho = \sigma_0 \xrightarrow{\delta_0}_c \sigma_0 + \delta_0 \xrightarrow{t_0}_c \sigma_1 \xrightarrow{\delta_1}_c \sigma_1 + \delta_1 \xrightarrow{t_1}_c \sigma_2 \xrightarrow{\delta_2}_c \ldots$, where $\sigma_i \in \Sigma$, $t_i \in T$ and $\delta_i \in \mathbb{R}_+$ for all $i \in \mathbb{N}$. A state $\sigma_* \in \Sigma$ is *reachable* if there exists a σ^0-run ρ and $i \in \mathbb{N}$ such that $\sigma_* = \sigma_i + \delta_i$, where $\sigma_i + \delta_i$ is an element of ρ. The set of all the reachable states of $\mathcal{N}$ is denoted by $Reach_{\mathcal{N}}$.

Given a set of propositional variables PV, we introduce a *valuation function* $V_c : \Sigma \to 2^{PV}$ which assigns the same propositions to the states with the same markings. We assume the set PV to be such that each $q \in PV$ corresponds to exactly one place $p \in P$, and use the same names for the propositions and the places. The function V_c is defined by $p \in V_c(\sigma) \Leftrightarrow p \in m$ for each $\sigma = (m, \cdot)$. The structure $M_c(\mathcal{N}) = ((T \cup \mathbb{R}_+, \Sigma, \sigma^0, \to_c), V_c)$ is called a *concrete* (*dense*) *model of* $\mathcal{N}$. It is easy to see that concrete models are usually infinite.

3.2 Extended Detailed Region Graph

In order to deal with countable structures instead of uncountable ones, we introduce *extended detailed region graphs* for distributed TPNs. They correspond to the well-known graphs defined for timed automata in [1] and adapted for time Petri nets [22, 33], but involve disjunctions of constraints, the reflexive transitive closure of the time successor of [1], and make no use of the maximal constant appearing in the invariants and enabling conditions. To do this, we assign a clock to each of the processes of a net.

Given a distributed time Petri net $\mathcal{N}$ whose processes are indexed with a set of indices $\mathcal{I}$ with $|\mathcal{I}| = n$ for some $n \in \mathbb{N}_+$. Let $\mathcal{X} = \{x_1, \ldots, x_n\}$ be a finite set of real-valued variables, called *clocks*. A *clock valuation* on $\mathcal{X}$ is a n-tuple $v \in \mathbb{R}_+^n$. The value of a clock $x_\mathfrak{i}$ in v is denoted by $v(x_\mathfrak{i})$. For a valuation v and a subset of clocks $X \subseteq \mathcal{X}$, by $v[X := 0]$ we denote the valuation v' such that $v'(x) = 0$ for all $x \in X$,

and $v'(x) = v(x)$ for all $x \in \mathcal{X} \setminus X$. Moreover, for some $\delta \in \mathbb{R}_+$, by $v + \delta$ we denote the valuation v' such that $v'(x) = v(x) + \delta$ for all $x \in \mathcal{X}$. The set $\mathcal{C}_\mathcal{X}$ of *clock constraints* over $\mathcal{X}$ is defined by the following grammar:

$$\mathfrak{cc} := \ true \mid x_\mathfrak{i} \sim c \mid \mathfrak{cc} \wedge \mathfrak{cc} \mid \mathfrak{cc} \vee \mathfrak{cc},$$

where $x_\mathfrak{i} \in \mathcal{X}$, $\sim \in \{\leq, <, =, >, \geq\}$ and $c \in \mathbb{N}$. A valuation v *satisfies* a constraint $\mathfrak{cc} \in \mathcal{C}_\mathcal{X}$ (denoted $v \models \mathfrak{cc}$) iff

- $\mathfrak{cc}$ is of the form $true$,
- $v(x_\mathfrak{i}) \sim c$, and $\mathfrak{cc}$ is of the form $x_\mathfrak{i} \sim c$,
- $v \models \mathfrak{cc}_1 \ \wedge \ v \models \mathfrak{cc}_2$, and $\mathfrak{cc}$ is of the form $\mathfrak{cc}_1 \wedge \mathfrak{cc}_2$,
- $v \models \mathfrak{cc}_1 \ \vee \ v \models \mathfrak{cc}_2$, and $\mathfrak{cc}$ is of the form $\mathfrak{cc}_1 \vee \mathfrak{cc}_2$.

The set of clock valuations satisfying a given constraint $\mathfrak{cc}$ is denoted by $[\![\mathfrak{cc}]\!]$ ($[\![\mathfrak{cc}]\!] \subseteq \mathbb{R}_+^n$).

We assume the clock valuations to be such that for any concrete state $\sigma = (m, clock)$, for each $\mathfrak{i} \in \mathcal{I}$ we have $v(x_\mathfrak{i}) = clock(\mathfrak{i})$. Thus, the clock constraint expressing the conditions under which the net can be in a marking m (the *marking invariant*) can be written as

$$inv(m) = \bigwedge_{t \in en(m) \text{ s.t. } Lft(t) < \infty} \ \bigvee_{\mathfrak{i} \in I\mathcal{V}(t)} x_\mathfrak{i} \leq Lft(t),$$

if $\{t \in T \mid t \in en(m) \wedge Lft(t) < \infty\} \neq \emptyset$, and as $inv(m) = true$ otherwise, which intuitively means that staying in m is allowed as long as for each enabled transition t with finite latest firing time there is a process $N_\mathfrak{i}$, involved in this transition, whose clock is not greater than $Lft(t)$ (and therefore t has not been disabled by passage of time). Moreover, for a marking m and a transition $t \in en(m)$ we define the constraint

$$fire_t(m) = \bigwedge_{\mathfrak{i} \in I\mathcal{V}(t)} x_\mathfrak{i} \geq Eft(t)$$

which expresses the condition under which t can be fired at m (note that the marking invariant, which obviously holds if $\mathcal{N}$ is in the marking m, implies that at least one process involved in t has the value of its clock not greater than $Lft(t)$). Given a marking m and $t \in en(m)$, firing t at m results in assigning the value 0 to the clocks belonging to the set

$$reset(m, t) = \{x_\mathfrak{i} \in \mathcal{X} \mid \mathfrak{i} \in I\mathcal{V}(t)\}.$$

Having all the above components, we can introduce the extended detailed region graph for $\mathcal{N}$. Let $\mathcal{C}_\mathcal{N} \subseteq \mathcal{C}_\mathcal{X}$ be a non-empty set of constraints defined by

$$\mathfrak{cc} := \ x_\mathfrak{i} \geq Eft(t) \mid x_\mathfrak{i} \leq Lft(t') \mid \mathfrak{cc} \wedge \mathfrak{cc},$$

where $x_\mathfrak{i} \in \mathcal{X}$, and, for a given $i \in \mathcal{I}$, $t \in T_\mathfrak{i}$ and $t' \in T_\mathfrak{i} \cap \{t \in T \mid Lft(t) < \infty\}$. Moreover, let $frac(a)$ denote the fractional part of a number $a \in \mathbb{R}_+$, and $\lfloor a \rfloor$ denote its integral part. Then, we define equivalence classes of clock valuations [37]:

Definition 3. *For two clock valuations $v, v' \in \mathbb{R}_+^n$, $v \simeq_\mathcal{N} v'$ iff for all $x, x' \in \mathcal{X}$ the following conditions are met:*

1. $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$,
2. $frac(v(x)) = 0$ *iff* $frac(v'(x)) = 0$,
3. $frac(v(x)) < frac(v(x'))$ *iff* $frac(v'(x)) < frac(v'(x'))$.

The last condition implies that $frac(v(x)) = frac(v(x'))$ iff $frac(v'(x)) = frac(v'(x'))$.

We call the equivalence classes of the relation $\simeq_{\mathcal{N}}$ (*extended*) *detailed zones* for $\mathcal{X}$, and denote the set of all of them by $DZ(n)$. It is easy to see from the definition of $\simeq_{\mathcal{N}}$ that the number of extended detailed zones is countable, and that for each $\mathfrak{cc} \in \mathcal{C}_{\mathcal{N}}$ and each $Z \in DZ(n)$ either $v \models \mathfrak{cc}$ for all $v \in Z$, or $v \not\models \mathfrak{cc}$ for all $v \in Z$. We say that $Z \in DZ(n)$ *satisfies a clock constraint* $\mathfrak{cc} \in \mathcal{C}_{\mathcal{X}}$ (denoted by $Z \models \mathfrak{cc}$) iff we have $v \models \mathfrak{cc}$ for each $v \in Z$.

Given an extended detailed zone $Z \in DZ(n)$, we introduce the operation $Z[X := 0] = \{v[X := 0] \mid v \in Z\}$. Moreover, let $Z^0 = \{v \in \mathbb{R}_+^n \mid (\forall x \in \mathcal{X})\ v(x) = 0\}$. Then, we define a successor relation on zones:

Definition 4 (Time successor). *Let Z and Z' be two zones in $DZ(n)$. The zone Z' is said to be the* time successor *of Z, denoted $\tau(Z)$, iff for each $v \in Z$ there exists $\delta \in \mathbb{R}_+$ such that $v + \delta \in Z'$.*

Definition 5 (Action successor). *Let $Z, Z' \in DZ(n)$. The zone Z' is said to be the* action successor *of Z by a transition $t \in T$, denoted $t(Z)$, if there exists a marking $m \subseteq P$ with $t \in en(m)$ such that $Z \models fire_t(m) \wedge inv(m)$ and $Z' = Z[reset(m,t) := 0]$.*

An (*extended detailed*) *region* is a pair (m, Z), where $m \subseteq P$ and $Z \in DZ(n)$. Notice that the set of all the extended detailed regions is countable. Given a concrete state $\sigma = (m', clock')$ we define $\sigma \in (m, Z)$ if $m = m'$ and $v \in Z$, where v is the clock valuation satisfying $v(x_{\mathfrak{i}}) = clock'(\mathfrak{i})$ for all $\mathfrak{i} \in \mathfrak{I}$. Next, we define a countable abstraction of the concrete state space of $\mathcal{N}$ - an *extended detailed region graph*.

Definition 6. *The* extended detailed region graph *for a net $\mathcal{N}$ is a structure $\Gamma(\mathcal{N}) = (T \cup \{\tau\}, W, w^0, \rightarrow)$, where $W = 2^P \times DZ(n)$, $w^0 = (m^0, Z^0)$, and the successor relation $\rightarrow \subseteq W \times (T \cup \{\tau\}) \times W$, where $\tau \notin T$, is defined in the following way:*

- $(m, Z) \xrightarrow{\tau} (m, Z')$ *iff* $Z, Z' \models inv(m)$ *and* $Z' = \tau(Z)$;
- *for* $t \in T$, $(m, Z) \xrightarrow{t} (m', Z)$ *iff* $t \in en(m)$, $m' = m[t\rangle$, $Z' = t(Z)$, $Z \models inv(m)$ *and* $Z' \models inv(m')$.

By an abstract model *based on $\Gamma(\mathcal{N})$ we mean a structure $M_\Gamma(\mathcal{N}) = (\Gamma(\mathcal{N}), V)$, where for each $w \in W$ and each $\sigma \in w$ we have $V(w) = V_c(\sigma)$.*

Notice that the definition of $\xrightarrow{\tau}$ is correct: in spite of a possibly non-convex form of $[\![inv(m)]\!]$, its definition ensures that if $Z, Z' \in DZ(n)$, $Z, Z' \models inv(m)$ and $(m, Z) \xrightarrow{\tau} (m, Z')$, then for any other $Z'' \in DZ(n)$ s.t. $Z'' = \tau(Z)$ and $Z' = \tau(Z'')$ (i.e., for a region (m, Z'') "traversed" when the time passes between (m, Z) and (m, Z')) the condition $Z'' \models inv(m)$ is satisfied as well. This follows from the fact that if in the zone Z some $x_{\mathfrak{i}} \in \mathcal{X}$ satisfies the condition $v(x_{\mathfrak{i}}) > Lft(t)$, then the same holds also for all the time successors of Z, and, on the other hand, if it satisfies $v(x_{\mathfrak{i}}) \leq Lft(t)$

and this condition is violated for some $Z'' = \tau(Z)$, then there is no $Z' = \tau(Z'')$ for which it holds again.

In order to show that the model $M_\Gamma(\mathcal{N})$ preserves the behaviours of the net, we shall prove that it is *bisimulation equivalent* with $M_c(\mathcal{N})$, where the bisimulation equivalence is defined as follows:

Definition 7. *Let $M = ((L, S, s_0, \to), V)$ and $M' = ((L', S', s'_0, \to'), V')$ be two models of a time Petri net $\mathcal{N}$. A relation $\rightsquigarrow_s \subseteq S' \times S$ is a* simulation *from M' to M if the following conditions hold:*

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

The model M' simulates *M ($M' \rightsquigarrow_s M$) if there is a simulation from M' to M. Two models M and M' are called* bisimulation equivalent *if $M' \rightsquigarrow_s M$ and $M(\rightsquigarrow_s)^{-1}M'$, where $(\rightsquigarrow_s)^{-1}$ is the inverse of $\rightsquigarrow_s$.*

Then, we can prove the following lemma:

Lemma 1. *For a given time Petri net $\mathcal{N}$ the models $M_c(\mathcal{N}) = ((T \cup \mathbb{R}_+, \Sigma, \sigma^0, \to_c), V_c)$ and $M_\Gamma(\mathcal{N}) = ((T \cup \{\tau\}, W, w^0, \to), V)$ are bisimulation equivalent.*

The proof can be found in the appendix.

4 Testing Reachability via BMC

The reachability problem for a system S consists in checking, given a property p, whether S can ever be in a state where p holds (which can be described by the CTL formula $\mathrm{EF}p$ - "there exists a path s.t. at that path the property p finally holds"). The property is expressed in terms of propositional variables. In the case the system S is represented by a time Petri net $\mathcal{N}$, the propositions correspond to the set of its places P. Therefore, the reachability verification can be translated to testing whether the set $Reach_{\mathcal{N}}$ contains a state whose marking includes a given subset of P. Checking this can be performed by an explicit exploration of the concrete state space (model), but due to its infinite size such an approach is usually very inefficient in practice.

If a reachable state satisfying the property p exists, this can be usually proven exploiting a part of the model only. This enables us to apply the bounded model checking approach. The basic idea of testing reachability using BMC consists in searching for a *reachability witness* of a bounded length k (i.e., for a path of a length $k \in \mathbb{N}_+$, called a *k-path*, which leads from the initial state to a state satisfying p). Searching for a reachability witness is performed by generating a propositional formula that is satisfiable iff such a witness exists. Satisfiability of this formula is checked using a SAT-solver.

To apply the above procedure, we represent the states of a model $M(\mathcal{N})$ for a given time Petri net $\mathcal{N}$ as vectors of boolean variables, and express the transition relation

of the model in terms of propositional formulas. Then, we *encode* all the k-paths of $M(\mathcal{N})$ starting at its initial state as a propositional formula α_k, and check satisfiability of a formula γ_k which is the conjunction of α_k and a propositional formula expressing that the property p holds at some state of a k-path. The above process is started from $k = 1$, and repeated iteratively up to $k = |M|$. It, however, can be stopped, since if for some k the formula γ_k is satisfiable, then reachability of a state is proven, and no further tests are necessary.

The above method can be inefficient if no state satisfying p exists, since the length of the k-path strongly influences the size of its propositional encoding. Therefore, in order to prove unreachability of a state satisfying p, another solution, shown in [36], is applied. A sketch of the idea is as follows: using the BMC procedures, we search for a longest k-path starting from an arbitrary state of M (a *free path*) such that p holds only in the last state of this path. If such a path π is found, then this means that in order to learn whether a state satisfying p is reachable we need to explore the model only to the depth equal to the length of π.

5 Implementation for Time Petri Nets

In order to apply the above approach to verification of a particular distributed time Petri net $\mathcal{N}$, we deal with a model obtained by a *discretisation* of its extended detailed region graph. The model is of an infinite but countable structure, which, however, is sufficient for BMC (which deals with finite sequences of states only). Below, we show this discretisation, and then encode the transition relation of the model.

5.1 Discretisation of Extended Detailed Region Graphs

Let $\Gamma(\mathcal{N}) = (T \cup \{\tau\}, W, w^0, \rightarrow)$ be the extended detailed region graph for a distributed time Petri net $\mathcal{N}$, and $\mathcal{X}$ be the set of clocks corresponding to its processes. Instead of dealing with the whole extended detailed region graph $\Gamma(\mathcal{N})$, we *discretise* this structure, choosing for each region one or more appropriate representatives. The discretisation scheme is based on the one for timed automata [37], and preserves the qualitative behaviour of the underlying system.

Let n be the number of clocks, and let $c_{max}(\mathcal{N})$ be the largest constant appearing in $\mathcal{C}_{\mathcal{N}}$ (i.e., the greatest finite value of Eft and Lft). For each $m \in \mathbb{N}$, we define

$$\mathbb{D}_m = \{d \in \mathbb{Q} \mid (\exists k \in \mathbb{N})\ d \cdot 2^m = k\},$$

and

$$\mathbb{E}_m = \{e \in \mathbb{Q} \mid (\exists k \in \mathbb{N})\ e \cdot 2^m = k \wedge e \leq c_{max}(\mathcal{N}) + 1\}.$$

The *discretised clock space* is defined as $\mathbb{D}^n$, where $\mathbb{D} = \bigcup_{m=1}^{\infty} \mathbb{D}_m$. Similarly, the set of possible values of time passings is defined as $\mathbb{E} = \bigcup_{m=1}^{\infty} \mathbb{E}_m$. The above definitions give us that the maximal values of time passings are restricted to $c_{max}(\mathcal{N}) + 1$, which is sufficient to express the behaviour of the net. Moreover, such a clock space and the set of lengths of timed steps ensure that for any representative of an extended detailed region there is another representative of this region which can be reached by a time step

of a length $e \in \mathbb{E}$. It should be mentioned that such a solution (different than in [24]) allows us to compute precisely the time passed along a k-path, what is important for the algorithms for parametric verification (and was difficult while using the so-called "adjust transitions" of [24]).

Now, we can introduce discretised region graphs and models:

Definition 8. *The* extended discretised region graph *based on the extended detailed region graph $\Gamma(\mathcal{N})$, is a structure $\widetilde{\Gamma}(\mathcal{N}) = (T \cup \mathbb{E}, \widetilde{W}, w^0, \rightarrow_d)$, where $\widetilde{W} = 2^P \times \mathbb{D}^n$, $w^0 = (m^0, Z^0)$, and the labelled transition relation $\rightarrow_d \subseteq \widetilde{W} \times (T \cup \mathbb{E}) \times \widetilde{W}$ is defined as*

1. *for $t \in T$, $(m, v) \xrightarrow{t}_d (m', v')$ iff $t \in en(m)$, $m' = m[t\rangle$, $v \models fire_t(m) \wedge inv(m)$, $v' = v[reset(m, t) := 0]$, and $v' \models inv(m')$ (*action transition*);*
2. *for $\delta \in \mathbb{E}$, $(m, v) \xrightarrow{\delta}_d (m, v')$ iff $v' = v + \delta$ and $v, v' \models inv(m)$ (*time transition*).*

Given an abstract model $M_\Gamma(\mathcal{N}) = (\Gamma(\mathcal{N}), V)$ based on $\Gamma(\mathcal{N}) = (T \cup \{\tau\}, W, w^0, \rightarrow)$ and the discretised model $\widetilde{\Gamma}(\mathcal{N})$, we can define a *discretised model* based on $\widetilde{\Gamma}(\mathcal{N})$, which is a structure $\widetilde{M}_\Gamma(\mathcal{N}) = (\widetilde{\Gamma}(\mathcal{N}), \widetilde{V})$, where $\widetilde{V} : \widetilde{W} \rightarrow 2^{PV}$ is a valuation function such that for each $\widetilde{w} \in \widetilde{W}$ being a representative of $w \in W$ we have $\widetilde{V}(\widetilde{w}) = V(w)$. This model will be exploited in BMC-based reachability checking.

5.2 Encoding of the Transition Relation of the Discretised Model

In order to apply SAT-based verification methods described in Sec. 4, we need to represent (encode) the discretised model $\widetilde{M}_\Gamma(\mathcal{N})$ as a boolean formula. To do that, we assume that each state $w \in \widetilde{W}$ is given in a unique binary form, i.e., $\widetilde{w} \in \{0,1\}^{h(m)}$, where $h(m)$ is a function of the greatest exponent appearing in the denominators of clock values in $\widetilde{w}$ (see [37] for details). The digits in the binary form of w are denoted by $w(1), \ldots, w(h)$. Therefore, the elements of $\widetilde{W}$ can be "generically" represented by a vector $\mathbf{w} = (\mathrm{w}[1], \ldots, \mathrm{w}[h(m)])$ of propositional variables (called a *symbolic state*), whose valuation (i.e., assignment of values to the variables) represents w iff for each $j \in \{1, \ldots, h(m)\}$ we have $\mathrm{w}[j] = true$ iff $w(j) = 1$, and $\mathrm{w}[j] = false$ otherwise. Moreover, each k-path in $\widetilde{\Gamma}(\mathcal{N})$ can be represented by a finite sequence $\mathbf{w}_0, \ldots, \mathbf{w}_k$ of symbolic states, and again, such a representation is called a *symbolic k-path*.

In what follows, by *state variables* we mean propositional variables used to encode the states of $\widetilde{\Gamma}(\mathcal{N})$. The set of all the state variables, containing the symbols $true$ and $false$, will be denoted by SV, and the set of all the propositional formulas built over SV - by SF. The elements of SF are called *state formulas*.

In order to encode the transition relation of $\widetilde{M}_\Gamma(\mathcal{N})$, we introduce the following functions and propositional formulas:

- $lit : \{0,1\} \times SV \rightarrow SF$, which is defined by $lit(0, p) = \neg p$ and $lit(1, p) = p$;
- $I_w(\mathbf{w}) := \bigwedge_{j=1}^{h} lit(w(j), \mathrm{w}[j])$ which is true iff the vector $\mathbf{w}$ represents the state w;
- $\mathbf{T}(\mathbf{w}, \mathbf{w}')$ which is true iff for the states $w, w' \in \widetilde{W}$, represented by vectors $\mathbf{w}$ and $\mathbf{w}'$, respectively, it holds $w \xrightarrow{e}_d w'$ for some $e \in T \cup \mathbb{E}$.

The formula which encodes all the k-paths in $\widetilde{\Gamma}(\mathcal{N})$ starting at the initial state is of the form

$$\alpha_k := I_{w^0}(\mathbf{w}_0) \ \wedge \ \bigwedge_{j=0}^{k-1} \mathbf{T}(\mathbf{w}_j, \mathbf{w}_{j+1}),$$

where $\mathbf{w}_0, \ldots, \mathbf{w}_k$ is a symbolic k-path. In practice, we consider k-paths with some restrictions on repetition of the action and time transitions, and on lengths of the time steps (see [37] for details). Encoding the fact that a state satisfies a given property is straightforward.

6 Parametric Reachability Checking

Besides testing whether a state satisfying a property p is reachable, one can be interested in finding a minimal time in which a state satisfying p can be reached, or finding a minimal time after which p does not hold. To this aim, *parametric reachability checking* can be used.

In order to be able to perform the above verification, we introduce an additional restriction on the nets under consideration, i.e., require they contain no cycle C of transitions such that for each $t \in C$ we have $Eft(t) = 0$ (which guarantees that the time increases when the net progresses, and is a typical assumption when analysing timed systems). Moreover, we introduce the notations $\mathrm{EF}^{\sim c}p$, with $\sim \in \{\leq, <, >, \geq\}$ and $c \in \mathbb{N}$, which express that a state satisfying p is reached in a time satisfying the constraint in the superscript[1]. The problems intuitively presented at the beginning of the section can be expressed respectively as finding a minimal c such that $\mathrm{EF}^{<c}p$ (or $\mathrm{EF}^{\leq c}p$) holds, and finding a maximal c such that $\mathrm{EF}^{>c}p$ (or $\mathrm{EF}^{\geq c}p$) holds.

An algorithm for finding a minimal c such that $EF^{\leq c}p$ holds looks as follows:

1. Using the standard BMC approach, find a reachability witness of minimal length[2];
2. read from the witness the time required to reach p (denoted x). Now, we know that $c \leq \lceil x \rceil$ (where $\lceil \cdot \rceil$ is the *ceiling* function);
3. extend the verified TPN with a new process N, which is composed of one transition t s.t. $Eft(t) = Lft(t) = n$, and two places p_{in}, p_{out} with $\bullet t = \{p_{in}\}$ and $t\bullet = \{p_{out}\}$ (see Fig. 2(a)),
4. set n to $\lceil x \rceil - 1$,
5. Run BMC to test reachability of a state satisfying $p \wedge p_{in}$ in the extended TPN,
6. if such a state is reachable, set $n := n - 1$ and go to 5,
7. if such a state is unreachable, then $c := n + 1$, STOP.

Some comments on the above algorithm are in place. First of all, it should be explained that the BMC method described in Sec. 4 finds a reachability witness of a shortest length (i.e., involving the shortest possible k-path). However, the shortest path is not necessarily that of minimal time. An example can be seen in Fig. 3, where the short-

[1] The full version of the logic, for a discrete semantics and with $\sim$ restricted to $\leq$ only, can be found in [14].

[2] if we cannot find such a witness, then we try to prove unreachability of p.

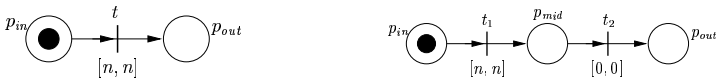


Fig. 2. The processes added to the nets to test parametric reachability

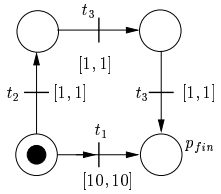


Fig. 3. An example net

est path leading to the place satisfying the property p_{fin} consists of one time step and one action step (i.e., passing 10 time units and then firing t_1), whereas minimal time of reaching such a state is 3, which corresponds to firing t_2, t_3 and t_4, each of them preceded by passing one unit of time. Due to this, after finding a reachability witness for p in Step 1 of the algorithm, we test whether p can be reached in a shorter time. Extending the net with a new process allows us to express the requirement that the time at which p is reached is not greater than n ($n \in \mathbb{N}$), since at time n the transition t has to fire, which unmarks the place p_{in}.

The second comment to the algorithm concerns the possible optimisations. Firstly, the algorithm can be optimized by applying one of the well-known searching algorithms instead of decreasing n by one in each step. Secondly, it is easy to see that if BMC finds a reachability witness for p of length k, then a witness for reaching p in a smaller time cannot be shorter than k (if such a witness existed, it would have been found previously). Thus, in Step 5 of the algorithm the BMC method can start with k equal to the length of the witness found in the previous run, instead of with $k = 1$.

Finally, Step 7 of the algorithm should be explained. In order to decide that no state satisfying $p \wedge p_{in}$ is reachable, we should either prove unreachability of that state using the method of [36], or to find an upper bound on the length of the k-paths such that unreachability of $p \wedge p_{in}$ on the paths up to this length allows us to decide that no state of interest is reachable. We can do the latter in some cases only, i.e., when some restrictions on the nets considered are assumed. This is specified by the following two lemmas:

Lemma 2. *If a net $\mathcal{N}$ contains no transition t with $Eft(t) = 0$, then the length of a reachability witness for $\mathrm{EF}^{\leq c}p$, in which time- and action steps alternate, is bounded by $2 \cdot c$.*

Proof. We make use of the result of [29], which states that each reachable marking of a TPN can be reached on a path whose time steps are of integer values only. Since from the structure of the net and from the structure of the path we have that zero-time

steps are not allowed, the shortest time steps are of length one. The bound $2c$ is then straightforward.

Lemma 3. *Let $\mathcal{N}$ be a distributed net consisting of n processes $N_\mathfrak{i} = (P_\mathfrak{i}, T_\mathfrak{i}, F_\mathfrak{i}, m^0_\mathfrak{i}, Eft_\mathfrak{i}, Lft_\mathfrak{i})$ ($\mathfrak{i} \in \mathcal{I} = \{1, \ldots, n\}$), each of which contains no cycle besides (possibly) being a cycle itself and satisfies the condition $\forall t_1, t_2 \in T_\mathfrak{i}$ ($\bullet t_1 \cap P_\mathfrak{i} = \bullet t_2 \cap P_\mathfrak{i} \iff t_1 \bullet \cap P_\mathfrak{i} = t_2 \bullet \cap P_\mathfrak{i}$). The length of a reachability witness for $\mathrm{EF}^{\leq c} p$, in which time- and action steps alternate, is bounded by $K = 2 \cdot \Sigma_{\mathfrak{i}=1}^{n} z_\mathfrak{i}$, where each $z_\mathfrak{i}$, for $\mathfrak{i} \in \mathcal{I}$, is computed according to the following algorithm:*

1. *set $g := 0$, $time := 0$, and $nextTrans$ to such $t \in T_\mathfrak{i}$ that $\bullet t = m^0_\mathfrak{i}$ and $Eft(t) = min(Eft(t') \mid t' \in T_\mathfrak{i} \wedge \bullet t' = m^0_\mathfrak{i})$,*
2. *do*
   - *$time := time + Eft(nextTrans)$;*
   - *if $time \leq c$ then set $g := g + 1$ and $s_g := nextTrans$;*
   - *set $nextTrans$ to such $t \in T_\mathfrak{i}$ that $\bullet t = s_{g-1}\bullet$ and $Eft(t) = min(Eft(t') \mid t' \in T_\mathfrak{i} \wedge \bullet t' = s_{g-1}\bullet)$,*

   while $time <= c$ and $s_g \bullet \cap P_\mathfrak{i} \neq \emptyset$,
3. *while $Eft(s_g) = 0$ and $(\bullet s_g \cap P_\mathfrak{i}) \notin Prop(p)$, where $Prop(p)$ is the set of propositions occuring in the property p, do $g := g - 1$;*
4. *$setz_\mathfrak{i} := g$.*

Proof. From the structure of a process of $\mathcal{N}$, we have that the algorithm for $z_\mathfrak{i}$ computes first the number of transitions which can be executed in time c provided that $\mathcal{N}_\mathfrak{i}$ proceeds as fast as possible, and then optimises the value obtained by removing a number of final steps which influence neither the time nor reaching the property tested. The length of the path in which time- and action steps alternate is therefore equal to $2z_\mathfrak{i}$. Taking the sum of these values for all the processes corresponds to considering the worst case, in which all the processes proceed independently, performing as many steps as possible.

An algorithm for finding a minimal c such that $\mathrm{EF}^{<c} p$ holds is similar to the previous one:

1. Using the standard BMC approach, find a reachability witness of minimal length[3];
2. read from the witness the time required to reach p (denoted x). Now, we know that $c \leq \lceil x \rceil$;
3. extend the verified TPN with a new process N, which is composed of two transitions t_1, t_2 s.t. $Eft(t_1) = Lft(t_1) = n$, $Eft(t_2) = Lft(t_2) = 0$, $\bullet t_1 = \{p_{in}\}$, $t_1\bullet = \bullet t_2 = \{p_{mid}\}$ and $t_2\bullet = \{p_{out}\}$ (see Fig. 2(b)),
4. set n to $\lceil x \rceil - 1$,
5. run BMC to test reachability of a state satisfying $p \wedge p_{in}$ in the extended TPN,
6. if such a state is reachable, set $n := n - 1$ and go to 5,
7. if such a state is unreachable, set $n := n + 1$ and run BMC to test reachability of a state satisfying $p \wedge p_{mid}$ in the extended TPN,
8. if such a state is reachable, then $c := n + 1$, STOP,
9. if such a state is unreachable, then $c := n$, STOP.

[3] if we cannot find such a witness, then we try to prove unreachability of p.

In this case, the additional process contains the place which can be marked only if the time passed since the net started is equal to n. The algorithm proceeds in the following way: the Steps 1 - 6 (analogous as in the previous algorithm) are aimed at finding a minimal n such that $\mathrm{EF}^{\leq n}p$ holds. Then, it is tested whether p can be reached exactly at time n. Depending on the result of this test, the bound returned is either n or $n+1$ (which follows from the result of [30] stating that the minimal time duration of a transition sequence is an integer value). The improvements to the algorithms, as well as methods of deciding unreachability in Steps 7 and 9, are the same as in the previous case.

The next pair of the algorithms is aimed at finding a minimal time after which no state satisfying p is reachable. This can be done by searching for a maximal c for which $\mathrm{EF}^{\geq c}p$ (or $\mathrm{EF}^{>c}p$) holds. The algorithm for $\mathrm{EF}^{\geq c}p$ is as follows:

1. using a standard BMC approach, test whether there is a k-path π such that p is reachable from its arbitrary state (i.e., whether for π the CTL formula $\mathrm{EGEF}p$ holds),
2. if such a k-path can be found, then no maximal c exists, STOP.
3. if such a k-path cannot be found then, using the standard BMC approach, find a reachability witness for p of a minimal length[4].
4. read from the witness the time x required to reach p,
5. extend the verified TPN with a new process which is composed of one transition t s.t. $Eft(t) = Lft(t) = n$, and two places p_{in}, p_{out} with $t\bullet = \{p_{out}\}$ and $\bullet t = \{p_{in}\}$,
6. set n to $\lceil x \rceil$, and set an upper bound b ($b \geq n$) on c to be searched for[5],
7. run BMC to test reachability of a state satisfying $p \wedge p_{out}$ in the extended TPN,
8. if such a state is reachable and $n + 1 < b$, then set $n := n + 1$ and go to 7,
9. if such a state cannot be found or $n + 1 \geq b$, then set $c := n - 1$, STOP.

Testing whether there is a k-path s.t. p is reachable from its arbitrary state (testing $\mathrm{EGEF}p$) is done by checking whether there is a path which has a loop, and there is a state of this loop at which p holds. In order to ensure that there is no maximal c, we need also the path to be progressive, i.e., such that its loop contains at least one non-zero time step[6].

Again, some optimisations to the algorithm can be introduced. The first one can consist in applying a well-known searching technique instead of increasing n by one in each step. The second is based on an observation that each reachability witness for $\mathrm{EF}^{\geq n}p$ is also a reachability witness for $\mathrm{EF}^{\geq n-1}p$. Thus, no witness for $\mathrm{EF}^{\geq n}p$ can be shorther than the shortest one found for $\mathrm{EF}^{\geq n-1}p$ (if a shorter witness existed, it would have been found while searching for a witness for $\mathrm{EF}^{\geq n-1}p$). Thus, while running

[4] if we cannot find such a k, then we try to prove unreachability of p

[5] the value b can be also a parameter of the algorithm

[6] Formally, let π be a k-path, $\pi(i)$ be the i-th state of the path, $\delta_\pi(i, i+1)$ be the time passed while moving from $\pi(i)$ to $\pi(i+1)$, $loop(\pi) = \{h \mid 0 \leq h \leq k \wedge \pi(k) \rightarrow \pi(h)\}$, and $\Pi_k(s)$ be the set of all the k-paths starting at s. The bounded semantics for $EGEF\alpha$ is as follows: $s \models EGEF\alpha \iff (\exists \pi \in \Pi_k(s))(loop(\pi) \neq \emptyset \wedge (\exists l \in loop(\pi)(\exists l \leq j \leq k)(\pi(j) \models \alpha \wedge \Sigma_{l \leq j < k}\delta_p i(j, j+1) > 0))$.

Step 7 of the algorithm, we can start with k equal to the length of the witness found in the previous run, instead of with $k = 1$.

It should be noticed that, contrary to the former cases, we cannot set any upper bound on the length of k-paths to be tested in Step 9, besides the one which follows from the value b assumed in the algorithm. In this case, computing the bound is done analogously as we shown in the description of the algorithm for $\mathrm{EF}^{\leq c}p$.

An algorithm for checking $EF^{>c}p$ (and searching for a maximal c) is as follows:

1. using a standard BMC approach, test whether there is a k-path π such that p is reachable from its arbitrary state (i.e., whether for π the CTL formula $\mathrm{EGEF}p$ holds),
2. if such a k-path can be found, then no maximal c exists, STOP.
3. if such a k-path cannot be found then, using the standard BMC approach, find a reachability witness for p of a minimal length[7].
4. read from the witness the time x required to reach p,
5. extend the verified TPN with a new process N, which is composed of two transitions t_1, t_2 s.t. $Eft(t_1) = Lft(t_1) = n$, $Eft(t_2) = Lft(t_2) = 0$, $\bullet t_1 = \{p_{in}\}$, $t_1\bullet = \bullet t_2 = \{p_{mid}\}$ and $t_2\bullet = \{p_{out}\}$,
6. set n to $\lceil x \rceil$, and set an upper bound b ($b \geq n$) on c to be searched for[8],
7. run BMC to test reachability of a state satisfying $p \wedge p_{out}$ in the extended TPN,
8. if such a state is reachable and $n + 1 < b$, then set $n := n + 1$ and go to 7,
9. if such a state is unreachable or $n + 1 > b$, set $n := n - 1$ and run BMC to test reachability of $p \wedge p_{mid}$) in the extended TPN,
10. if such a state is reachable, then $c := n - 1$, STOP;
11. if such a state is unreachable, then $c := n$, STOP.

The idea behind the algorithm is similar to the previous approaches: first a maximal n for which $\mathrm{EF}^{\geq n}p$ is found, then the algorithm tests whether reaching p at time n is possible. The final result depends on the answer to the latter question.

It should be mentioned that in practice all the above methods are not complete (as the BMC itself is not). It can happen that we are not able to prove unreachability of a state, compute an upper bound on the length of a k-path to be tested, or, in spite of finding such an upper bound, are not able to test the paths up to this length using the resources given. However, the preliminary experiments show that the methods can give quite good results.

7 Experimental Results

The experimental results presented below are preliminary, since some methods mentioned in the previous sections are not represented. We have performed our experiments on the computer equipped with Intel Pentium Dual CPU (2.00 GHz), 2 GB main memory and the operating system Linux 2.6.28. We have tested some distributed time Petri nets for the standard *Fischer's mutual exclusion protocol* (mutex) [2]. The system consists of n time Petri nets, each one modelling a process, plus one additional net used to

[7] if we cannot find such a k, then we try to prove unreachability of p

[8] the value b can be also a parameter of the algorithm

coordinate their access to the critical sections. A distributed TPN modelling the system is shown in Figure 1, for the case of $n = 2$. *Mutual exclusion* means that no two processes are in their critical sections at the same time. The preservation of this property depends on the relative values of the time-delay constants δ and Δ. In particular, the following holds: *"Fischer's protocol ensures mutual exclusion iff $\Delta < \delta$"*.

Our first aim was to check that if $\Delta \geq \delta$, then the mutual exclusion is violated. We considered the case with $\Delta = 2$ and $\delta = 1$. It turned out that the conjunction of the propositional formula encoding the k-path and the negation of the mutual exclusion property (denoted p) is unsatisfiable for every $k < 12$. The witness was found for $k = 12$. We were able to test 40 processes. The results are shown in Fig. 4 (left).

Our second aim was to search for a minimal c such that $\mathrm{EF}^{\leq c} p$ holds. The results are presented in Fig. 4 (right). In the case of this net, we are not able to compute an upper bound on the length of the k-path. Unfortunately, we also could not test unreachability, since the method is not implemented yet. Again, we considered the case with $\Delta = 2$ and $\delta = 1$, and the net of 25 processes. The witness was found for $k = 12$, and the time of the path found was between 8 and 9. The column n shows the values of the parameter in the additional component. For $n = 1$ and $k = 12$ unsatisfiability was returned, and testing the property on a longer path could not be completed in a reasonable time.

The next two (scalable) examples were the networks shown in Fig. 5. The net (a) shown in the left-hand side of the figure was scaled by increasing $Eft(t_2)$ and $Lft(t_2)$, according to the schema $A = 2u$, $B = 4u$, for $u = 1, 2, \ldots$. The property tested was $\mathrm{EF}(p_3 \wedge p_6)$. The net (b) shown on the right was scaled by increasing the number of components $\mathcal{N}_\mathrm{i}$ ($\mathrm{i} = 1, 2, \ldots$). In this case, reachability of a state satisfying $p_3 \wedge \bigwedge_{\mathrm{i}=1}^{j} p_3^\mathrm{i}$ was checked (where j is a number of identical processes). For both the nets we searched

		tpnBMC				RSat		
k	**n**	**variables**	**clauses**	**sec**	**MB**	**sec**	**MB**	**sat**
0	-	1937	5302	0.2	3.5	0.0	1.7	NO
2	-	36448	107684	1.4	7.9	0.4	9.5	NO
4	-	74338	220335	2.9	12.8	3.3	21.5	NO
6	-	112227	332884	4.2	17.6	14.3	37.3	NO
8	-	156051	463062	6.1	23.3	257.9	218.6	NO
10	-	197566	586144	7.8	28.5	2603.8	1153.2	NO
12	-	240317	712744	9.7	34.0	87.4	140.8	YES
				32.4	34.0	2967.1	1153.2	

		tpnBMC				RSat		
k	**n**	**variables**	**clauses**	**sec**	**MB**	**sec**	**MB**	**sat**
0	-	840	2194	0.0	3.2	0.0	1.4	NO
2	-	16263	47707	0.5	5.2	0.1	4.9	NO
4	-	33835	99739	1.0	7.3	0.6	9.1	NO
6	-	51406	151699	1.6	9.6	1.8	13.8	NO
8	-	72752	214853	2.4	12.3	20.6	27.7	NO
10	-	92629	273491	3.0	14.8	321.4	200.8	NO
12	-	113292	334357	3.7	17.5	14.3	39.0	YES
12	7	120042	354571	4.1	18.3	45.7	59.3	YES
12	6	120054	354613	4.0	18.3	312.7	206.8	YES
12	5	120102	354763	4.0	18.3	64.0	77.7	YES
12	4	120054	354601	4.1	18.3	8.8	35.0	YES
12	3	115475	340834	3.9	17.7	24.2	45.0	YES
12	2	115481	340852	3.9	17.8	138.7	100.8	YES
12	1	115529	341008	3.9	17.7	2355.4	433.4	NO
				40.1	18.3	3308.3	433.4	

Fig. 4. Results for mutex, $\Delta = 2$, $\delta = 1$, mutual exclusion violated. Left: proving reachability for 40 processes, right: parametric verification for 25 processes. The tpnBMC column shows the results for the part of the tool used to represent the problem as a propositional formula (a set of clauses); the column RSat displays the results of running the RSat solver for the set of clauses obtained from tpnBMC.

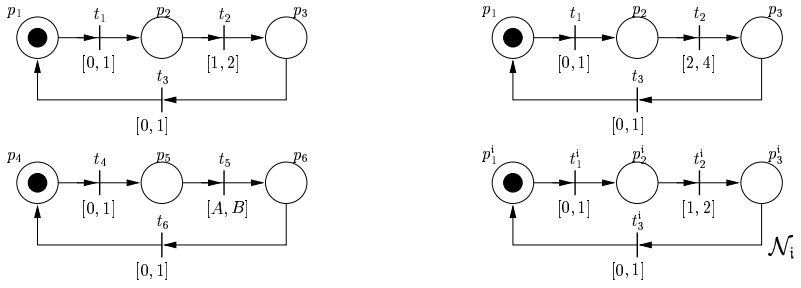


Fig. 5. Time Petri nets tested in experiments

for a minimal time c at which a given property can be reached, and for both of them we were able to compute an upper bound on the length of the k-path to be tested while checking reachability in a time not exceeding n. For the net (a) the bound is $K = 2(z_1 + z_2 + z_3)$, where $z_1 = 3n - 1$, $z_2 = 3\lceil n/(2u)\rceil - 1$ and $z_3 = 1$ (where the third process is that added to test reachability in time n); whereas for the net (b) containing j identical components it is given by $K = z_0 + \Sigma_{i=1}^{j} z_i + z_{j+1}$, where the bound for the first process is $z_1 = 3\lceil n/2\rceil - 1$, the bound for each of the identical processes is $z_i = 3n - 1$ $(i = 1, \ldots, j)$, and the bound for the additional process is $z_{j+1} = 1$. The results for the net (a), with the values of the coefficient u given, are presented in Fig. 6. In the case of $u = 4$ we were able to test the k-paths up to the upper bound $K = 46$, and to show that the parameter searched for is $c = 8$; for $u = 5$ we can only assume that the value of c is 10, since we were not able to test all the k-paths of the lengths up to $K = 58$. Concerning the net (b), we were able to test the net containing 6 identical processes and to show that $c = 2$; the results are given in Fig. 7.

8 Final Remarks

We have shown that the BMC method for checking reachability properties of TPNs is feasible. Our preliminary experimental results prove the efficiency of the method. However, it would be interesting to check practical applicability of BMC for other examples of time Petri nets. On the other hand, it would be also interesting to check efficiency of the above solutions for other (non-distributed) nets (which could be done by applying the translations from [28]).

References

1. R. Alur, C. Courcoubetis, and D. Dill. Model checking for real-time systems. In *Proc. of the 5th Symp. on Logic in Computer Science (LICS'90)*, pages 414–425. IEEE Computer Society, 1990.
2. R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Proc. of the 13th*

		tpnBMC				RSat		
k	**n**	**variables**	**clauses**	**sec**	**MB**	**sec**	**MB**	**sat**
0	-	21	34	0.0	3.0	0.0	1.3	NO
2	-	603	1653	0.0	3.0	0.0	1.4	NO
4	-	1396	3909	0.0	3.1	0.0	1.6	NO
6	-	2174	6097	0.0	3.2	0.0	1.7	NO
8	-	3347	9429	0.1	3.3	0.0	2.0	NO
10	-	4345	12213	0.1	3.5	0.0	2.2	NO
12	-	5413	15175	0.1	3.6	0.0	2.5	NO
14	-	6551	18315	0.1	3.7	0.1	2.8	YES
14	7	8812	24886	0.1	4.0	0.1	3.3	NO
16	7	11299	31987	0.2	4.2	0.1	3.8	NO
18	7	13151	37159	0.2	4.5	0.2	4.2	NO
20	7	15103	42595	0.2	4.8	118.9	17.1	NO
22	7	17155	48295	0.3	5.0	15.6	8.3	NO
24	7	19307	54259	0.3	5.2	18.9	9.7	NO
26	7	21559	60487	0.3	5.5	133.5	19.0	NO
28	7	23911	66979	0.4	5.8	167.1	26.5	NO
30	7	27930	78424	0.4	6.2	96.4	18.5	NO
32	7	30586	85756	0.5	6.5	224.9	32.6	NO
34	7	33342	93352	0.5	6.8	339.8	36.4	NO
36	7	36198	101212	0.5	7.2	549.8	50.2	NO
38	7	39154	109336	0.6	7.5	339.8	50.7	NO
40	7	42210	117724	0.6	7.9	266.5	45.6	NO
42	7	45366	126376	0.7	8.2	1026.9	85.0	NO
44	7	48622	135292	0.7	8.6	558.6	83.3	NO
46	7	51978	144472	0.8	9.0	574.7	75.6	NO
				7.6	9.0	4431.9	85.0	

		tpnBMC				RSat		
k	**n**	**variables**	**clauses**	**sec**	**MB**	**sec**	**MB**	**sat**
0	-	21	34	0.0	3.0	0.0	1.3	NO
2	-	619	1707	0.0	3.0	0.0	1.4	NO
4	-	1428	4017	0.0	3.1	0.0	1.6	NO
6	-	2467	6985	0.0	3.2	0.0	1.8	NO
8	-	3411	9645	0.0	3.3	0.0	2.0	NO
10	-	4425	12483	0.1	3.5	0.0	2.2	NO
12	-	5994	16945	0.1	3.6	0.0	2.6	NO
14	-	7228	20379	0.1	3.7	0.1	2.9	NO
16	-	8532	23991	0.1	3.9	0.1	3.2	NO
18	-	9906	27781	0.1	4.1	0.1	3.5	NO
20	-	11350	31749	0.2	4.2	0.1	3.8	YES
20	10	15223	42995	0.2	4.8	0.2	4.8	YES
20	9	15303	43255	0.2	4.8	5.1	6.0	NO
22	9	17375	49021	0.3	5.0	58.9	10.9	NO
24	9	20802	58804	0.3	5.4	9.1	9.7	NO
26	9	23178	65410	0.3	5.7	73.4	16.4	NO
28	9	25654	72280	0.4	5.9	139.2	21.1	NO
30	9	28230	79414	0.4	6.3	185.2	22.6	NO
32	9	30906	86812	0.5	6.6	1974.1	115.3	NO
34	9	33682	94474	0.5	6.9	566.0	64.9	NO
36	9	36558	102400	0.5	7.2	955.7	67.8	NO
38	9	39534	110590	0.6	7.6	2931.3	160.6	NO
40	9	42610	119044	0.7	8.0	4771.1	187.6	NO
				5.7	8.0	11669.7	187.6	

Fig. 6. Results for net (a). Left: $u = 4$, $K = 46$ (unreachability proven). Right: $u = 5$, $K = 58$ (unreachability not proven)

IEEE Real-Time Systems Symposium (RTSS'92), pages 157–166. IEEE Computer Society, 1992.

3. G. Audemard, A. Cimatti, A. Kornilowicz, and R. Sebastiani. Bounded model checking for timed systems. In *Proc. of the 22nd Int. Conf. on Formal Techniques for Networked and Distributed Systems (FORTE'02)*, volume 2529 of *LNCS*, pages 243–259. Springer-Verlag, 2002.
4. M. Benedetti and A. Cimatti. Bounded model checking for Past LTL. In *Proc. of the 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 18–33. Springer-Verlag, 2003.
5. B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Trans. on Software Eng.*, 17(3):259–273, 1991.
6. B. Berthomieu and M. Menasche. An enumerative approach for analyzing time Petri nets. In *Proc. of the 9th IFIP World Computer Congress*, volume 9 of *Information Processing*, pages 41–46. North Holland/ IFIP, September 1983.
7. A. Biere, A. Cimatti, E. Clarke, M.Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. of the ACM/IEEE Design Automation Conference (DAC'99)*, pages 317–320, 1999.
8. H. Boucheneb and K. Barkaoui. Relevant timed schedules / clock valuations for constructing time Petri net reachability graph. In *Proc. of the 6th Int. Workshop on Formal Analysis and Modeling of Timed Systems (FORMATS'08)*, volume 5215 of *LNCS*, pages 265–279. Springer-Verlag, 2008.

		tpnBMC				RSat		
k	**n**	**variables**	**clauses**	**sec**	**MB**	**sec**	**MB**	**sat**
0	-	71	124	0.0	3.0	0.0	1.3	NO
2	-	1567	4394	0.0	3.1	0.0	1.6	NO
4	-	3754	10753	0.1	3.3	0.0	2.1	NO
6	-	6661	19240	0.1	3.7	0.0	2.8	NO
8	-	9273	26794	0.2	4.0	0.0	3.4	NO
10	-	12105	34956	0.2	4.4	0.1	4.0	NO
12	-	16672	48307	0.3	4.9	0.2	5.1	NO
14	-	20194	58445	0.4	5.4	0.5	5.9	NO
16	-	23936	69191	0.4	5.8	1.4	6.8	NO
18	-	27898	80545	0.5	6.3	2.9	8.0	NO
20	-	32080	92507	0.6	6.8	10.1	10.2	NO
22	-	39247	113458	0.7	7.7	88.9	19.0	NO
24	-	44119	127396	0.8	8.3	341.1	26.9	NO
26	-	49211	141942	0.8	8.9	489.3	42.4	NO
28	-	54523	157096	0.9	9.6	6.2	14.5	YES
28	2	60704	175021	1.0	10.3	40.6	24.0	YES
28	1	60816	175385	1.1	10.4	3027.4	243.0	NO
30	1	67021	193096	1.1	11.1	393.9	75.2	NO
				9.4	11.1	4402.6	243.0	

Fig. 7. Results for net (b) containing 6 identical processes; the bound $K = 30$.

9. H. Boucheneb and G. Berthelot. Towards a simplified building of time Petri nets reachability graph. In *Proc. of the 5th Int. Workshop on Petri Nets and Performance Models*, pages 46–55, October 1993.
10. G. Bucci, A. Fedeli, L. Sassoli, and E. Vicaro. Modeling flexible real time systems with preemptive time Petri nets. In *Proc. of the 15th Euromicro Conference on Real-Time Systems (ECRTS'03)*, pages 279–286. IEEE Computer Society, 2003.
11. G. Bucci and E. Vicaro. Compositional validation of time-critical systems using communicating time Petri nets. *IEEE Trans. on Software Eng.*, 21(12):969–992, 1995.
12. E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
13. P. Dembiński, A. Janowska, P. Janowski, W. Penczek, A. Półrola, M. Szreter, B. Woźna, and A. Zbrzezny. VerICS: A tool for verifying timed automata and Estelle specifications. In *Proc. of the 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 278–283. Springer-Verlag, 2003.
14. E. A. Emerson and R. Trefler. Parametric quantitative temporal reasoning. In *Proc. of the 14th Symp. on Logic in Computer Science (LICS'99)*, pages 336–343. IEEE Computer Society, July 1999.
15. G. Gardey, O. H. Roux, and O. F. Roux. Using zone graph method for computing the state space of a time Petri net. In *Proc. of the 1st Int. Workshop on Formal Analysis and Modeling of Timed Systems (FORMATS'03)*, volume 2791 of *LNCS*, pages 246–259. Springer-Verlag, 2004.
16. K. Heljanko. Bounded reachability checking with process semantics. In *Proc. of the 12th Int. Conf. on Concurrency Theory (CONCUR'01)*, volume 2154 of *LNCS*, pages 218–232. Springer-Verlag, 2001.
17. M. Huhn, P. Niebert, and F. Wallner. Verification based on local states. In *Proc. of the 4th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *LNCS*, pages 36–51. Springer-Verlag, 1998.
18. R. Janicki. Nets, sequential components and concurrency relations. *Theoretical Computer Science*, 29:87–121, 1984.

19. J. Lilius. Efficient state space search for time Petri nets. In *Proc. of MFCS Workshop on Concurrency, Brno'98*, volume 18 of *ENTCS*. Elsevier, 1999.
20. R. Mascarenhas, D. Karumuri, U. Buy, and R. Kenyon. Modeling and analysis of a virtual reality system with time Petri nets. In *Proc. of the 20th Int. Conf. on Software Engineering (ICSE'98)*, pages 33–42. IEEE Computer Society, 1998.
21. P. Merlin and D. J. Farber. Recoverability of communication protocols – implication of a theoretical study. *IEEE Trans. on Communications*, 24(9):1036–1043, 1976.
22. Y. Okawa and T. Yoneda. Symbolic CTL model checking of time Petri nets. *Electronics and Communications in Japan, Scripta Technica*, 80(4):11–20, 1997.
23. W. Penczek and A. Półrola. Specification and model checking of temporal properties in time Petri nets and timed automata. In *Proc. of the 25th Int. Conf. on Applications and Theory of Petri Nets (ICATPN'04)*, volume 3099 of *LNCS*, pages 37–76. Springer-Verlag, 2004.
24. W. Penczek, A. Półrola, B. Woźna, and A. Zbrzezny. Bounded model checking for reachability testing in time Petri nets. In *Proc. of the Int. Workshop on Concurrency, Specification and Programming (CS&P'04)*, volume 170(1) of *Informatik-Berichte*, pages 124–135. Humboldt University, 2004.
25. W. Penczek, B. Woźna, and A. Zbrzezny. Bounded model checking for the universal fragment of CTL. *Fundamenta Informaticae*, 51(1-2):135–156, 2002.
26. W. Penczek, B. Woźna, and A. Zbrzezny. Branching time bounded model checking for elementary net systems. Technical Report 940, ICS PAS, Ordona 21, 01-237 Warsaw, January 2002.
27. W. Penczek, B. Woźna, and A. Zbrzezny. Towards bounded model checking for the universal fragment of TCTL. In *Proc. of the 7th Int. Symp. on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT'02)*, volume 2469 of *LNCS*, pages 265–288. Springer-Verlag, 2002.
28. A. Półrola and W. Penczek. Minimization algorithms for time Petri nets. *Fundamenta Informaticae*, 60(1-4):307–331, 2004.
29. L. Popova. On time Petri nets. *Elektronische Informationsverarbeitung und Kybernetik*, 27(4):227–244, 1991.
30. L. Popova-Zeugmann and D. Schlatter. Analyzing paths in time Petri nets. *Fundamenta Informaticae*, 37(3):311–327, 1999.
31. O. Strichman. Tuning SAT checkers for bounded model checking. In *Proc. of the 12th Int. Conf. on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 480–494. Springer-Verlag, 2000.
32. L-M. Tranouez, D. Lime, and O. H. Roux. Parametric model checking of time Petri nets with stopwatches using the state-class graph. In *Proc. of the 6th Int. Workshop on Formal Analysis and Modeling of Timed Systems (FORMATS'08)*, volume 5215 of *LNCS*, pages 280–294. Springer-Verlag, 2008.
33. I. B. Virbitskaite and E. A. Pokozy. A partial order method for the verification of time Petri nets. In *Fundamental of Computation Theory*, volume 1684 of *LNCS*, pages 547–558. Springer-Verlag, 1999.
34. B. Woźna. ACTL* properties and bounded model checking. *Fundamenta Informaticae*, 63(1):65–87, 2004.
35. B. Woźna, A. Zbrzezny, and W. Penczek. Checking reachability properties for timed automata via SAT. *Fundamenta Informaticae*, 55(2):223–241, 2003.
36. A. Zbrzezny. Improvements in SAT-based reachability analysis for timed automata. *Fundamenta Informaticae*, 60(1-4):417–434, 2004.
37. A. Zbrzezny. SAT-based reachability checking for timed automata with diagonal constraints. *Fundamenta Informaticae*, 67(1-3):303–322, 2005.

9 Appendix

Below, we provide a proof of Lemma 1:

Proof. We shall show that the relation $\mathcal{R} = \{(\sigma, w) \mid \sigma \in w\}$ is a bisimulation. It is easy to see that $\sigma^0 \mathcal{R} w^0$, and that for each $\sigma \in w$ we have $V_c(\sigma) = V(w)$, since the markings of the related states are equal. Thus, consider $\sigma = (m, clock) \in \Sigma$ and $w = (m, Z) \in W$ such that $\sigma \mathcal{R} w$.

- If $w \xrightarrow{\tau} w'$, where $w' = (m', Z') \in W$, then from $Z' = \tau(Z)$ we have that for each $v \in Z$ (and therefore for that given by $v(x_{\mathfrak{i}}) = clock(\mathfrak{i})$ for all $\mathfrak{i} \in \mathcal{I}$) there exists $\delta \in \mathbb{R}_+$ such that $v + \delta \in Z'$. Moreover, the condition $Z' \models inv(m)$ implies that for each $t \in en(m)$ there is $\mathfrak{i} \in I\mathcal{V}(t)$ such that $(v + \delta)(x_{\mathfrak{i}}) \leq Lft(t)$. Thus, there exists a state $\sigma' \in \Sigma$, given by $\sigma' = (m, clock + \delta)$, satisfying $\sigma \xrightarrow{\delta}_c \sigma'$ and $\sigma' \in w'$ (i.e., $\sigma' \mathcal{R} w'$).
- On the other hand, if $\sigma \xrightarrow{\delta} \sigma'$ for some $\sigma' = (m, clock') \in \Sigma$ and $\delta \in \mathbb{R}_+$, then for each $\sigma_1 = (m, clock_1) \in w$ one can find $\delta' \in \mathbb{R}_+$ such that the clock valuation v_1' given by $v_1'(x_{\mathfrak{i}}) = clock_1(\mathfrak{i}) + \delta'$ is equivalent to the clock valuation v' given by $v'(x_{\mathfrak{i}}) = clock'(\mathfrak{i})$ (intuitively, δ' should be chosen such that the increase from $clock_1(\mathfrak{i})$ to $clock_1(\mathfrak{i}) + \delta'$ should "cross" as many integer bounds as the increase from $clock(\mathfrak{i})$ to $clock(\mathfrak{i}) + \delta$, for each $\mathfrak{i} \in \mathcal{I}$). Moreover, from the definition of the time-successor relation we have that for each $t \in en(m)$ there is $\mathfrak{i} \in I\mathcal{V}(t)$ such that $clock'(\mathfrak{i}) \leq Lft(t)$, and therefore from the definition of $\simeq_{\mathcal{N}}$ it holds also $clock_1(\mathfrak{i}) + \delta' \leq Lft(t)$. Thus, for the extended detailed region $w' = (m, Z')$ such that $\sigma' \in w'$ (and therefore $w' \mathcal{R}^{-1} \sigma'$) we have $Z' = \tau(Z)$ and $Z' \models inv(m)$, which implies $w \xrightarrow{\tau} w'$.
- If $w \xrightarrow{t} w'$ for some transition $t \in T$, where $w' = (m[t\rangle, Z') \in W$, then $t \in en(m)$ and $Z \models fire_t(m) \wedge inv(m)$. Thus, it is easy to see that the transition t can be fired also at the state σ, which leads to $\sigma' = (m', clock') \in \Sigma$, with $m' = m[t\rangle$ and $clock'(\mathfrak{i}) = 0$ for $\mathfrak{i} \in I\mathcal{V}(t)$, and $clock'(\mathfrak{i}) = clock(\mathfrak{i})$ otherwise. Therefore, the clock valuation v' given by $v'(x_{\mathfrak{i}}) = clock'(\mathfrak{i})$ belongs to the zone $Z[reset(t, m) := 0]$, which implies $\sigma' \in w'$ (and therefore $\sigma' \mathcal{R} w'$).
- If $\sigma \xrightarrow{t} \sigma'$ for some transition $t \in T$ and $\sigma' = (m', clock') \in \Sigma$, then $t \in en(m)$, $clock(\mathfrak{i}) \geq Eft(t)$ for every $\mathfrak{i} \in I\mathcal{V}(t)$, and there exists $\mathfrak{i} \in I\mathcal{V}(t)$ such that $clock(\mathfrak{i}) \leq Lft(t)$. Thus, from the definition of $\simeq_{\mathcal{N}}$ the zone Z satisfies the constraints $fire_t(m)$ and $inv(m)$. Considering $w' = (m', Z')$ such that $\sigma' \in w'$, it is easy to see from the definition of $\simeq_{\mathcal{N}}$ that $Z' = Z[reset(m, t) := 0]$ (the zone Z collects the clock valuations equivalent to v given by $v(x_{\mathfrak{i}}) = clock(\mathfrak{i})$ for each $\mathfrak{i} \in \mathcal{I}$; therefore from $\sigma \xrightarrow{t} \sigma'$ and from the definition of $\simeq_{\mathcal{N}}$ the zone Z' collects the valuations which are like the elements of Z but with the clocks $x_{\mathfrak{i}}$ with $\mathfrak{i} \in I\mathcal{V}(t)$ set to zero). Thus, $Z' = t(Z)$. Moreover, $Z' \models inv(m')$ in an obvious way (we have $m' = m[t\rangle$; if a transition $t' \in en(m')$ became enabled by firing t then there exists $\mathfrak{i} \in I\mathcal{V}(t')$ such that for all $v' \in Z'$ $v'(x_{\mathfrak{i}}) = 0$ (and therefore $v(x_{\mathfrak{i}}) \leq Lft(t')$), whereas for all the other transitions $t \in en(m')$ the existence of $\mathfrak{i} \in \mathcal{I}$ s.t. $v(x_{\mathfrak{i}}) \leq Lft(t)$ follows from $Z \models inv(m)$, since the values of clocks

have not been increased). Thus, for the detailed region w' such that $\sigma' \in w'$ (and therefore $w' \mathcal{R}^{-1} \sigma'$) we have $w \xrightarrow{t} w'$, which ends the proof.

Short Presentations

Modeling with Net References and Synchronous Channels

Heiko Rölke

German Institute for International Educational Research (DIPF)
Schloßstraße 29, D-60486 Frankfurt
roelke(at)dipf.de

Abstract. Nets-within-nets offer some modeling possibilities which are not available (or only to a limited extent) in classical Petri nets formalisms like P/T nets or colored Petri nets. This holds especially true in the formalism of reference nets. The extended modeling possibilities arise from the newly come concepts, e.g. net instances generated at run-time and synchronization between transitions.
This paper focuses on practical aspects of the new modeling possibilities – rather than theoretical backgrounds – and demonstrates them in practical examples. As a modeling language reference nets [7] are used. This is the only nets-within-nets formalism implemented up to now and with Renew [8] a powerful simulation tool-set is available.
The examples presented throughout the paper can serve as an introduction to advanced modeling using reference nets.

1 Introduction

Petri nets have a long tradition as modeling means, particularly for distributed systems. Such models serve for the purpose of analyzing the modeled systems. However, the models theirselves can also be seen as an implementation of the system (*implementation by modeling*). For both application purposes certain modeling problems are seen as "notoriously difficult" to model with Petri nets [1]. Suggested solutions exist, for instance, in the form of collections of so-called *net patterns* or *net components* which suggest – more or less elegant – solutions for different problems. From this, the collection of patterns for colored Petri nets of Mulyar and Aalst [9] is most extensive. With the study of the patterns introduced there, it strikes the reader's mind that often and – from the point of view of the author – sometimes without need, basic functionality of a pattern is programmed in the respective inscription language. For colored Petri nets the inscription language is ML. An example for this is the use of lists (e.g. pattern "Aggregate Objects"[1] in [9]) and similar data types (containers). This raises the question, whether the use of non-Petri-net constructs is unavoidable. On the other hand, it maybe the case that similar functionality can also be expressed in the net structure itself. This question can be answered of course only for each

[1] The pattern Aggregate Objects uses the built-in list type of CPN Tools.

net formalism individually. Moreover, a profound knowledge of the respective modeling possibilities is necessary. Therefore, the modeling will not be examined here for colored Petri nets [5], but for reference nets [7,8]. Basically, reference nets offer similar expressive power as colored Petri nets. Nevertheless, in detail deviations arise, some of which are exemplarily stated here:

Colored nets (tool Design/CPN or CPN Tools)

- Inscription language ML
- global (static) place fusion
- static hierarchy concept (transition refinement)

Reference nets (tool Renew)

- Inscription language Java
- (global) synchronous channels: dynamic transition fusion
- dynamic hierarchy concept using net instances and references

Therefore, the statements and models of this paper can not or only to a limited extent be directly transferred to colored Petri nets.

From the various modeling possibilities the following are examined in this paper: modeling of data types (tuple, list,...), special arc types, control structures (loops), and recursion. Before doing so, we now take a short look at the formalism of reference nets. More information, especially on the practical application of this net type can be found in the Renew guide [8] that accompanies the Renew modeling tool.

2 Introduction to Reference Nets

Reference nets (defined in [7]) are basically higher-order nets, based on Petri nets whose arcs are annotated by a special inscription language. Java expressions were chosen as the primary inscription language, but with addition of tuples and some other changes. As usual, variables are bound to values, expressions are evaluated, and tokens are moved according to the result of arc inscriptions. Additionally, there are also transition inscriptions.

- Guards, notated as `guard` ***`expr`***, require that the expression evaluates to `true` before the transition may fire.
- ***`expr`***`=`***`expr`*** can be inscribed to a transition, but it does not imply assignment, but rather specification of equality. Variables must be bound to a fixed value during the firing of a transition. This means that modify assignments like `x=x+1` do not make sense.
- Java expressions might be evaluated, even when it turns out that the transition is not enabled and cannot fire. This causes problems for some Java method calls, therefore the notation `action` ***`expr`*** is provided. It guarantees that ***`expr`*** will be evaluated exactly once, a feature that is needed when side effects (e.g. changes to Java objects) come into play.

When a net is constructed, it is merely a net template without any marking. But it is then possible to create a net instance from the template. In fact, an arbitrary number of net instances can be created dynamically during a simulation. Only net instances, not the templates, have got a marking that can change over time.

- A net instance is created by a transition inscription of the form *var*`:new` *netname*. It means that the variable *var* will be assigned a new net instance of the template *netname*. The net name must be uniquely chosen for each template that we specify.

It should be noted that RENEW supports the concepts of NETS-WITHIN-NETS (see [10]) which is a major research topic. In order to exchange information between different net instances, synchronous channels were implemented. They provide greater expressiveness compared to message passing. Unlike the synchronous channels from [3] which are completely symmetric, we will impose a direction of invocation. The invoking side of a channel will be known as the down-link, the invoked side is called the up-link.

- An up-link is specified as a transition inscription `:`*channelname*`(`*expr*`,...)`. It provides a name for the channel and an arbitrary number of parameter expressions.
- A down-link looks like *netexpr*`:`*channelname*`(`*expr*`,...)` where *netexpr* is an expression that must evaluate to a net reference. The syntactic difference reflects the semantic difference that the invoked object must be known before the synchronization starts.

To fire a transition that has a down-link, the referenced net instance must provide an up-link with the same name and parameter count and it must be possible to bind the variables suitably so that the channel expressions evaluate to the same values on both sides. The transitions can then fire simultaneously. Note that channels are bidirectional for all parameters except the down-link's *netexpr*. E.g., a net might pass a value through the first parameter of a down-link and the called net might return a result through the second parameter of the same channel. This is similar to the undirected parameters of Prolog predicates, but different from the invocations of Cooperative Nets by Sibertin-Blanc (see [2]).

A transition may have an arbitrary number of down-links, but at most one up-link. (Again, the similarity with Horn Clauses in Prolog does not occur by chance.) A transition without up-links will be called a spontaneous transition, because it may fire without being invoked by another transition. A transition may have (at most) one up-link and (several) down-links at the same time. A transition may synchronize multiple times with the same net and even with the same transition.

With this short introduction to reference nets in mind we now look at advanced modeling possibilities.

3 Data types

All variants of high-level Petri nets[2] permit the usage of different data types like numbers, characters and strings or logical values. In addition, also composite data types can usually be used. It has to be distinguished between finite and infinite data types. The definition of the data types usually follows that of the respective inscription language of the net formalism, thus, for instance, ML or Java. Both mentioned languages are so called universal computer languages, so that arbitrary data types can be defined – with more or less effort. Nevertheless, the question is more interesting whether the required composite data types can also be provided by expressive means of the net formalism itself.

For all kinds of finite data types the solution is obvious, mostly trivial: Components of fixed length can be modeled using sub-nets or dedicated net instances, that store the data in separate places and allow for suitable access. Examples for such data types are *records* (Pascal) or *structs* (C).

However, such data types always show a fixed upper limit in storage space (the number of places), so that for the case of potentially infinite data types other solutions have to be investigated.

One modeling element for infinite data types is quickly found: Each place can store an arbitrary number of tokens (data). Nevertheless, it is problematic that the order of the data is lost and, even worse, (only) dis-arrayed access is possible. The only data type which can be implemented this way is the multi-set[3].

An additional common restriction is that a so-called *zero test*, the identification of an empty place, is not possible in most common net formalisms[4] . Thus problems arise necessarily, for instance with the calculation of the cardinality of a multi-set. A common and popular solution to this problem is the introduction of an external or integrated counter. However, this solution has the disadvantages of introducing an infinite data type (e.g. of type *natural numbers*) and restricting possible concurrent access to the data place.

Therefore, we use the data type "list" as a starting example for our search for alternative implementation possibilities.[5]

A classic of the list programming is the creation of a list of pairs nested into each other: $(elem_1, (elem_2, (\ldots)))$

[2] The term "colored Petri net" is ambiguous as at refers to the general concept of nets offering structured data types as well as the special formalism of Colored Petri Nets as defined by Jensen [5].

[3] In fact, only a restricted implementation of the multi-set is possible, see the notes on testing for emptiness.

[4] for good reasons

[5] Reference nets not only offer list implementations in the inscription language Java, but also offer a built-in data type list which will be used later in this text. Nevertheless, we explicitly search for alternatives using only net structures or other first-order concepts of the net formalism.

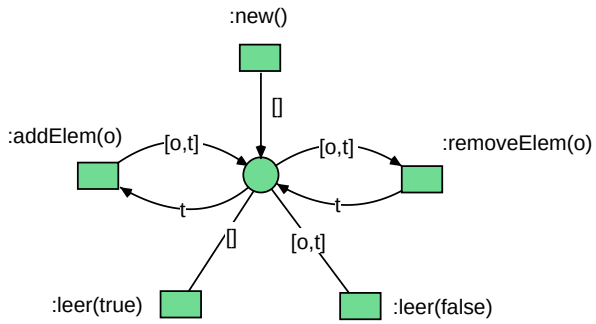


Fig. 1. List implementation using nested tuples

For reference nets the predefined data type "tuple" exists[6]. A tuple has an arbitrary, but fixed arity and, therefore, can not be used directly as a list substitute. However, 2-tuples (pairs) can be nested continuously into each other like in Figure 1 which summarizes typical access operations of a list. A list can be generated from scratch (:new ()) by producing a new net instance (which contains the list). The occurrence of this transition puts an empty tuple on the central place of the net. This empty tuple can be tested by the transition at the left bottom and returns via the channel :empty the value true. From an empty list no element can be removed, so the transition on the right side inscribed with the channel :removeElem is not activated. However, a new element can be put to the list at the head via channel :addElem. Adding a new element adds another nesting layer to the list representation. After that the test for emptiness fails (transition with channel :empty(false)) and the element could be removed.

The claimed goal of this paper was to renounce (predefined) data types as much as possible and whenever possible rely on description means of the net formalism only. For the presented solution attempt, the problem is to substitute the tuples. This is possible in reference nets by the application of net instances. A tuple net instance for the application as a list element is to be found in Figure 2. It allows as a specific feature also for an empty initialization. The list implementation does not differ substantially from the model with tuples. It is to be found in Figure 3.

In analogous ways more complicated list types can also be defined, like queues or priority lists. This is not carried out in detail here and is recommended to the reader as a practical assignment. From now on, in this text it is assumed that lists can be implemented completely with nets. Therefore, the built-in list type which leads to substantially clearer models is used for reasons of simpler representation.

[6] The tuple is not predefined in the inscription language Java, but explicitly has been designed for reference nets. It is supported by a short-hand notation using square brackets.

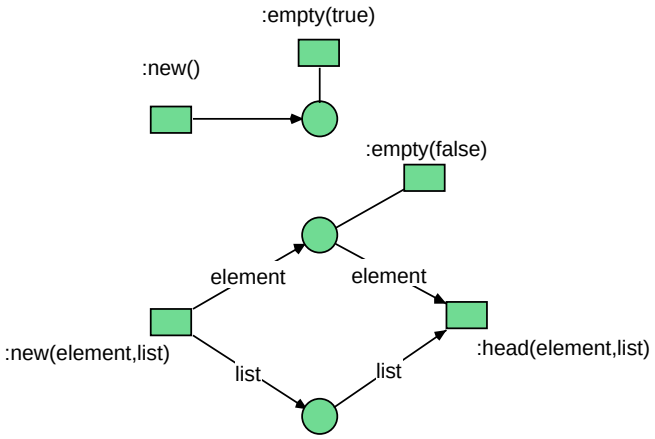


Fig. 2. List element as a net instance

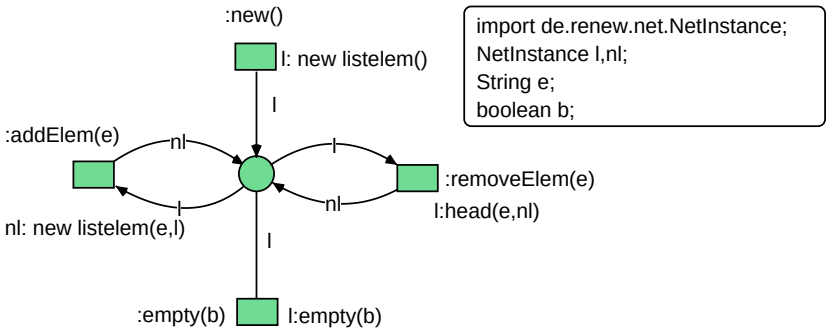


Fig. 3. A list implementation using the net instances of Fig. 2

Nevertheless, it is assured that in each case also the implementation introduced above could be used.

4 Special arcs

Some Petri nets formalisms offer special arcs which go beyond the functionality of usual arcs. These are, for instance, flexible arcs, reset or clear arcs, inhibitor arcs or transfer arcs. For this paper, of particular interest is the reference net's specific feature of *flexible arcs*. The other arc types are not treated here, but bibliographical reference is given.

4.1 Flexible arcs

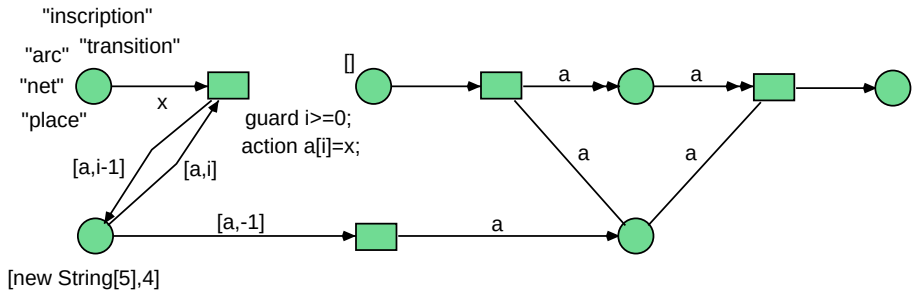


Fig. 4. Example of flexible arcs

Flexible arcs are a specific feature of reference nets. They allow to deposit or remove several tokens from a place at once. However, the amount of tokens has to be specified in advance what is the difference e.g. to reset or transfer arcs. The specification of the amount of tokens is implemented technically in Renew with the help of a *Collection* (Java class, for example, Array, Vector etc.) or a tuple or a list (built-in types). Flexible arcs activate a transition, if all tokens which are included in the inscribed collection can be removed from or deposited on the connected place.

The net from Figure 4, which was taken from the Renew documentation [8] together with this explanation, illustrates the use of flexible arcs. On the left hand side you can see a classical way to remove five tokens from a place by looping with an explicit counter. One after another the tokens are collected and assigned to an array, which results in a rather clumsy net structure. Now we can see the two kinds of flexible arcs in action on the right hand side. One transition puts five tokens onto a place and another transition removes all five tokens atomically.

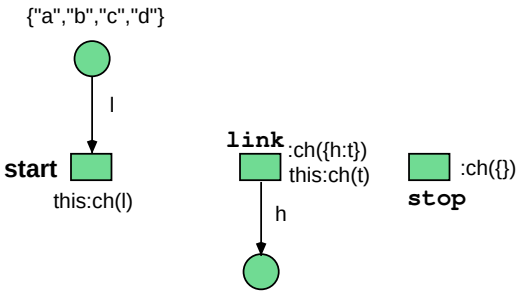


Fig. 5. Simulation of a flexible arc using synchronous channels

The simulation of a flexible arc in Figure 5 shows the case of a flexible arc in the output area of a transition - the reverse case can be constructed analogously. In the case shown the internal list data type is used to store some example values (strings). This is done to keep the example net simple. Nevertheless, it is possible to use an arbitrary Java Collection[7] type containing an arbitrary number of elements.

The simulation of the flexible arc shown here is even more generic than the original flexible arc built-in in Renew, as will be proved in the following section. Simultaneous depositing of several tokens happens by continuous channel calls, by every call the list head is put as a token on the place. The remainder of the list is treated - atomically! - by a (recursive) channel call. The three transitions handle the three distinct cases:

Start of the call chain (left transition `start`), is inscribed with only one downlink. The transition `link` in the middle handles non-empty lists by extracting the list's head `h`, putting it on the place below and recursively call the channel again with the tail `t` of the list as a parameter. The transition `stop` on the right ends the call chain when the list is empty.

The net for simulating a flexible arc serves as a kind of pattern for several other problems as well. This net pattern is suited for the treatment of arbitrary loop constructs what is shown in the next section. Also well-known list operators known from functional programming languages like *fold* or *map* may be implemented using this pattern.

4.2 Transfer, reset, and inhibitor arcs

These arc types can also be simulated by means of net instances as Michael Köhler has already pointed out in [6].

[7] Java class

5 Control structures and recursion

Data types and control structures are closely related, because complex data types are often constructed using (non elementary) controlling structures as for example loops.

5.1 Elementary control structures

The modeling of sequences, branches and concurrency is a trivial task using net formalisms like P/T nets. In addition, reference nets permit a restriction of concurrent structures to real parallelism. This is demonstrated in Figure 6.

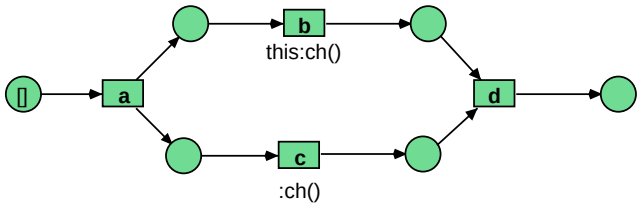


Fig. 6. Forced parallel execution of two concurrent transitions

In the figure both transitions b and c which are – following standard Petri net semantics – concurrent to each other are synchronized by the synchronous channel :ch. They can only occur at the same time. Therefore they are executed in parallel.

5.2 Loops

Petri nets permit the modeling of loops with the basis elements already. The conflict between the continuation of a loop and its termination can be decided, for instance, by a guard inscription.

As a speciality, reference nets permit the possibility of a loop processing that can be carried out even atomically (!) in one step. This is done similar to the simulation of the flexible arc in Figure 5.

An example of this is to be seen in Figure 7. The example net calculates the factorial of an input number - in the figure the number on the place input. The net simulates a *for* loop (loop termination or number of iterations is known in advance). As stated before, the layout of the net is similar to that of Figure 5, because the functionality is similar, too.

Only the case of a loop is demonstrated, where the termination takes place after a fixed number of iterations (for-loop), but also the more general case of a loop with a variable termination condition (while loop) can be implemented with the same means. This means that non-terminating loops are possible – similar to the well-known μ-operator of recursive functions.

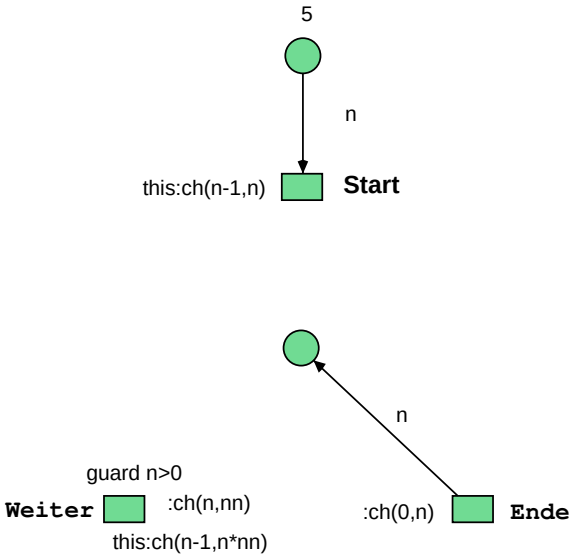


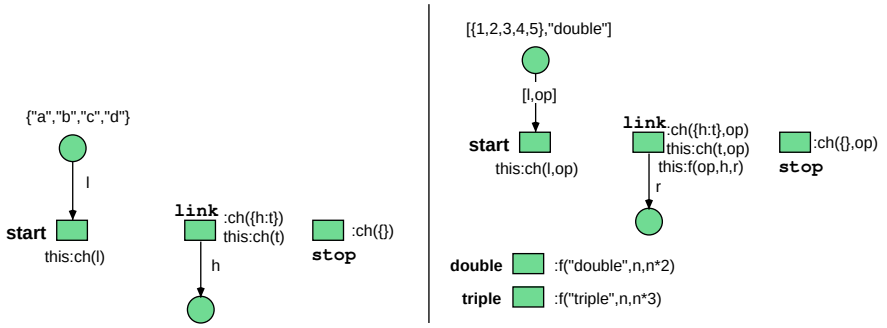
Fig. 7. Calculation of factorials in the style of Figure 5

5.3 List Operators

List operators known from functional programming languages[8] are another application case. The simulation of flexible arcs in Figure 5 can easily be extended with operators inscribed to the output arc that puts the tokens contained in the list to the output place. In this way, a so-called *map* operator is implemented, that applies an operation to all elements of a list. An example can be found in Figure 8. In 8(a) a static operation is performed: Each element of the list is doubled. This can also be done dynamically – see Figure 8(b). The application of the operation on every list element is done via a synchronous channel call to an interchangeable sub-net. The name of the operation is passed as a value. The function call has to be atomic. As an potential alternative, a triple operation is also provided in the example. This simulates so-called *higher-order* functions – functions, that serve as an parameter for other functions, especially list functions.

Numerous other applications in the area of list programming are possible, for instance the separation of the list elements whether they are put on the output place or not – a so-called *filter* operation. Another example is the partition of a list in two halves, as it is used in Figure 10 as part of an implementation of the *Merge sort* algorithm.

[8] For example the functional programming language *Miranda* [4] offers several list operators, but other well-known functional languages like Lisp, ML or Haskell have similar concepts.



(a) Static (b) Dynamic

Fig. 8. Map operator

6 Recursion

Reference nets allow to create net instances at run-time. When creating a net instance, arbitrary parameters can be handed over. Thus net instances can call themselves (create a new instance) in particular in a recursive manner. Therefore they are well suited to implement recursive algorithms.

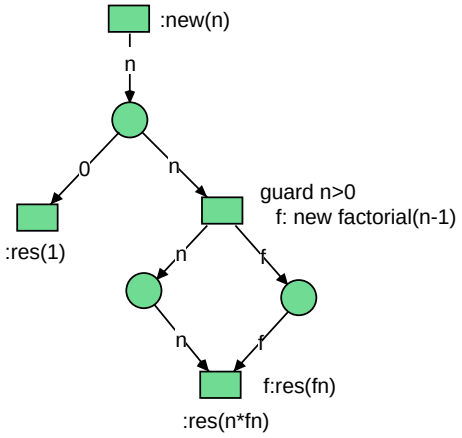


Fig. 9. Recursive net for calculating factorials

As an introducing example of a recursive net, the calculation of the factorials has been implemented once again in Figure 9. After instantiating this net with a number, it is checked whether this number is zero or greater than zero. In case of zero, the result (1) can be handed back. In the other case (guard n>0), a new

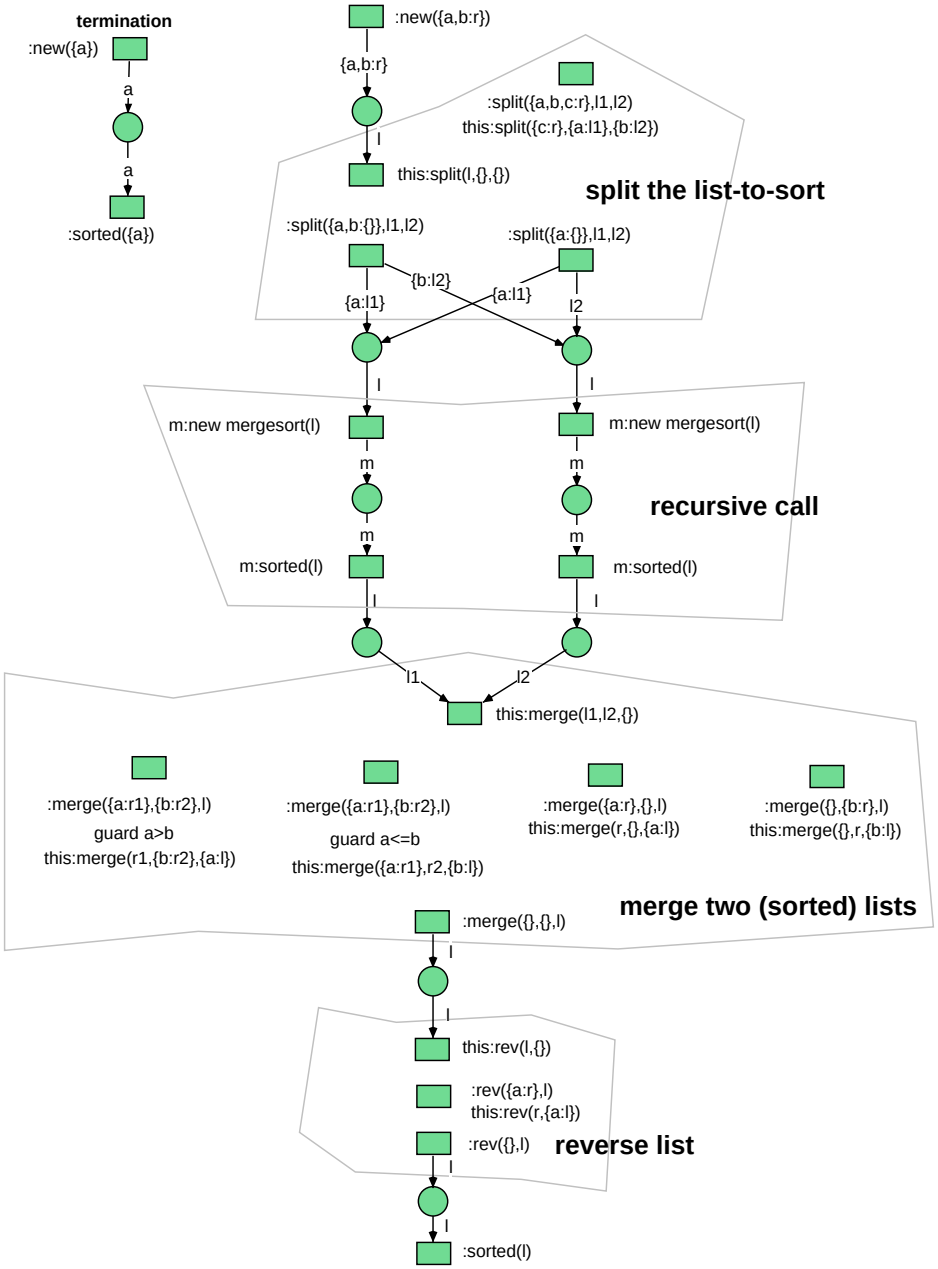


Fig. 10. Merge sort (descending order)

net is instantiated with n-1 for calculating the factorial for n-1. The number n as well as this net instance are stored. Later, the result of the factorial calculation is available. This is multiplied by n to come to the result for the factorial for n.

More complicated recursive algorithms are, for example, the well-known sorting algorithms like Merge sort and Quick sort in the Figures 10 and 11.

Both sorting algorithms work on lists and include some interesting list operations as intermediate steps. The Merge sort algorithm in Figure 10 includes the splitting of the list in two parts (split the list-to-sort), the recursive call of the net, the merge of the sorted list parts and a list reversion (reverse list). These list operations are implemented using the *flexible arc* pattern from Subsection 4.1 and thus demonstrate its usefulness.

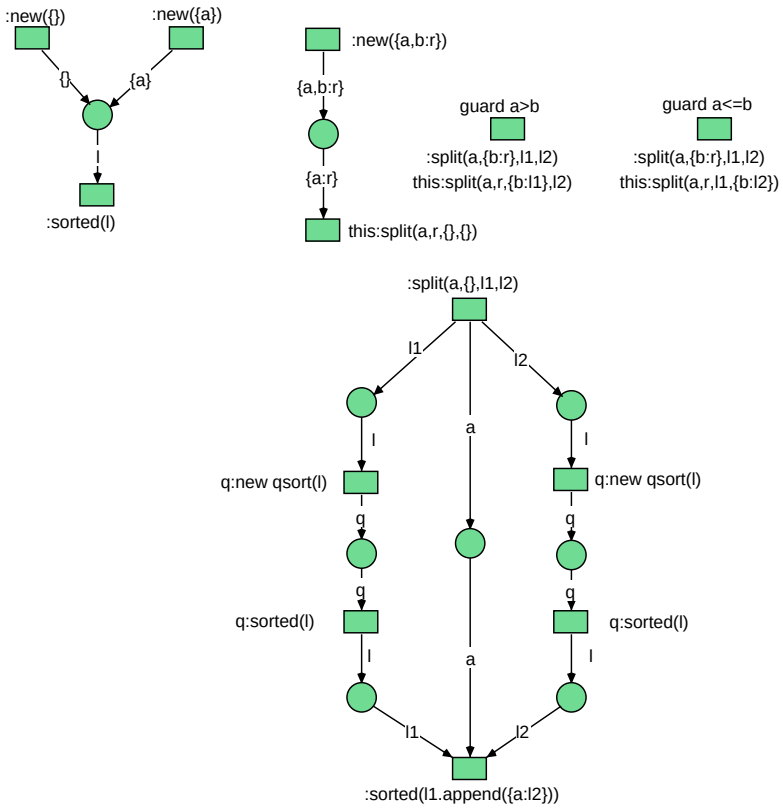


Fig. 11. Recursive net demonstrating the Quick sort algorithm

For didactic reasons the nets are modeled with some additional net elements, so that the steps of the algorithms building up on each other are easily to be seen. Both sorting algorithms could be modeled in a more compact manner.

7 Outlook and Further Work

The modeling possibilities shown here demonstrate only a small amount of what is possible to model and how this could be done. In no way a claim for completeness is raised. In particular, the examples introduced here have been collected not in view to cover, for instance, the collection introduced by Mulyar and Aalst [9]. The purpose was rather to introduce nets and modeling possibilities that appeared to be "elegant" or at least unusual to the author. In this sense the examples sometimes resemble introductory examples from the Renew manual or fulfill similar purposes. A closer look reveals the full power of the concepts shown here, which goes way beyond the manual examples.

The examples shown in this paper can be useful in manifold areas. They were designed with no special application in mind. One possible area of application is education: concepts like recursion are directly shown in the net structure rather than hidden in an inscription language. Another purpose is to show the power of the modeling techniques. This is done by simulating known powerful concepts (e.g., again, recursion). In addition, the net concepts can be used to build elegant models.

Further work could be to examine similar modeling problems in other net formalisms than reference nets. A first candidate would be the formalism of Colored Petri nets as defined by Jensen [5]. A more structured approach to provide solutions to known modeling problems could lead to a complete set of reference net patterns. Such an approach would include a complete discussion of common Petri net elements like variants of arcs. Each net element should be analyzed for modeling possibilities like in this paper.

References

1. W.M.P. van der Aalst and A.H.M. ter Hofstede. Workflow patterns: On the expressive power of (Petri-net-based) workflow languages. pages 1–19, Ny Munkegade, Building 540, DK-8000 Aarhus C, Dänemark, 2002. Computer Science Department, Aarhus University. see also: `http://www.daimi.au.dk/CPnets/workshop02/cpn/papers/`.
2. Sibertin Blanc. Cooperative nets. In Robert Valette, editor, *Application and Theory of Petri Nets 1994, Proc. of 15th Intern. Conf. Zaragoza, Spain*, LNCS, pages 159–178, June 1994.
3. S. Christensen and N.D. Hansen. Coloured Petri nets extended with channels for synchronous communication. In Robert Valette, editor, *Application and Theory of Petri Nets 1994, Proc. of 15th Intern. Conf. Zaragoza, Spain, June 1994*, LNCS, pages 159–178, June 1994.
4. Ian Holyer. *Functional Programming with Miranda*. UCL Press, London, 1991.
5. K. Jensen. *Coloured Petri nets, Basic Methods, Analysis Methods and Practical Use*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1992.
6. Michael Köhler. *Objektnetze: Definition und Eigenschaften*, volume 1 of *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2004.
7. Olaf Kummer. *Referenznetze*. Logos Verlag, Berlin, 2002.

8. Olaf Kummer, Frank Wienberg, and Michael Duvigneau. Renew – the Reference Net Workshop. Available at: http://www.renew.de/, June 2004. Release 2.0.
9. N.A. Mulyar and W.M.P. van der Aalst. Patterns In Colored Petri Nets. Technical report, Eindhoven University of Technology, April 2005.
10. Rüdiger Valk. Petri nets as token objects - an introduction to elementary object nets. In Jörg Desel and Manuel Silva, editors, *19th International Conference on Application and Theory of Petri nets, Lisbon, Portugal*, number 1420, pages 1–25, 1998.

On the Multilevel Petri Nets-Based Models in Project Engineering

Šárka Květoňová and Vladimír Janoušek

Brno University of Technology,
Bozetechova 2, 61266 Brno, Czech Republic
{kvetona,janousek}@fit.vutbr.cz
http://www.fit.vutbr.cz/

Abstract. Project engineering (as a special part of systems engineering) is a domain where suitable formal models and tools are still needed. The approach used for projects and resources modeling affect the way how corresponding computerized tools are designed and implemented. The paper presents a way how the dynamically instantiable, multilevel Petri nets can be applied in all significant processes of project engineering, especially in the project planning, scheduling, monitoring, and analysis. The main emphasis is put on the resources modeling, simulation, and optimization during the project life time. We use the *Object oriented Petri nets* (OOPN) formalism which is a kind of *multi-level Petri nets* with a possibility to synchronize events at different levels. In the case of the project modeling domain, we use two levels. The first level corresponds to the *project plans* and the second level corresponds to the *resources*.

Key words: Object oriented Petri net, modeling, simulation, monitoring, optimization

1 Introduction

Project engineering and management domain is a very specific part of business sphere. It deals with higly dynamic business environment where only a few facts is predictable. That is why we need to search for new methods and tools for projects modeling, control and optimization, preferably on a clear, formal base.

Project engineering concerns conceptualisation, development, integration, implementation and management of projects in a variety of fields. It is a part of the broader domain - systems engineering. Systems engineering focuses on how complex engineering projects should be designed and managed. It deals with work-processes and tools to handle such projects, and it overlaps with both technical and human-centered disciplines such as control engineering and project management. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem, the system lifecycle. Systems engineering encourages the use of tools and methods to better comprehend and manage complexity in systems. These tools include

Modeling and Simulation, Optimization, System dynamics, Systems analysis, Statistical analysis, Reliability analysis, and Decision making [1].

An important part of project engineering is resources management domain on which this paper is focused. This domain is the most crucial part of project management because it affects success and/or failure of the whole project (project portfolio). We keep in mind not only human resources, but material, financial etc., too. Generally, it depends on what/how a resource (or a resources group) is defined and evaluated.

The approach used for projects and resources modeling affect the way how corresponding computerized tools are designed and implemented. We use object-oriented Petri net-based framework. It is described in section 2. Our approach to the projects and resources modeling is decribed in section 3. Before that, we define some basic domain notions and describe motivation and related work.

1.1 Basic Notions

In the following, basic terms and relations of project management and planning/scheduling domains are given.

Project is a temporary effort undertaken to create a unique product or service, or result conforming to certain specifications and applicable standards [4].

Process A process is a series of actions bringing about a result. It is a complex of mutually connected resources and activities, which changes inputs to outputs. At present, activities and resources under the project are managed almost entirely like processes [4].

Project management [33] is a procedure of managing and directing time, material, personnel and costs to complete a particular project in an orderly and economical manner; and to meet established objectives in time, costs, and technical results. Project management is the application of knowledge, skills, tools, and techniques to project activities to meet project requirements. Project management is accomplished through the use of the processes such as: initiating, planning, executing, controlling, and closing [31].

Project Portfolio Management [9, 5, 22] is about more than running multiple projects. Each portfolio of projects needs to be assessed in terms of its business value and adherence to strategy. The portfolio should be designed to achieve a defined business objective or benefit.

Planning and Scheduling [33, 29, 30] are processes dealing with tasks or activities scheduling in time and space. Their main goal is to gain necessary tasks scheduling on limited resources by use of mathematic techniques and heuristic methods. *Planning* is a process of a set of proper activities creation to gain the predefined goals. *Scheduling* is a process of converting a general or outline plan

for a project into a time-based representation given information on available resources and time constraints.

A *task* is an object which is planned. It is characterized by own properties and inherent structure. An operation or subtask, too, is a particular part of the whole task. The task consists of one or more operations which can be realized on one or more resources.

A *resource* processes the individual operations, eventually, it serves as means an operation realization. Resources use is to be limited in a system.

Static (Off-line) scheduling [30, 18] requires a knowledge about all the resources, their parameters, all the requirements, constraints and all criteria for the scheduling process in advance, to complete the schedule before the system starts to run.

On-line scheduling [18] represents a process of creatin a schedule in runtime. The schedule is re-created each time the conditions in the environment are changed or modified. The scheduling method has to be sufficiently fast in this case.

A timetable/schedule is an organized list, usually set out in tabular form, providing information about a series of arranged events; in particular, the time at which it is planned these events will take place.

1.2 Motivation

The main motivation of our underlying research is to simplify the whole process of projects engineering by means of sufficient models, techniques, and tools. Firstly, it is necessary to use an appropriate combination of tools, methods and techniques from different business domains to find the most effective and optimized solutions (economics theory, cybernetics, systems theory etc.). The next important goal is to maximize accuracy of duration estimates and while allowing an adaptation to changing requirements and conditions during project life time. The main accent is put on resources allocation optimization in context of actual demands.

Secondary motivation is to demonstrate possibilities of OOPN use in the project engineering domain, especially in the project scheduling and monitoring.

1.3 Related Work

There are several methods of project analysis already: CPM (Critical Path Methods), PERT, GERT, MPM, and others. These techniques are successful in an offline model of planning – it is quite difficult to monitor and control project (plan) performance and, moreover, to model and detect resource limitations. Thus, it is important to find other ways and possibilities how to monitor and control whole project realization. Especially, if any unexpected events occur is necessary to modify some of project parameters immediately (through the online support of project management).

Nowadays there exist many approaches, methods and formalisms which are using formal models, simulation and optimization in the project management

domain. Manfred Mauerkirchner is focused on human resource allocation, in conrete, Resource Constrained Project Scheduling Problem (RCPSP) and uses a specific non analytical multiobjective function for allocation of qualified human resources [24]. The other approach which he uses is based on mutiagent systems [25]. Kabeh Vaziri focuses on heuristic optimization of the allocation of skilled workers with respect to standard RCPSP problem [36, 35]. He uses simulation-based approach. Simulation-based multiobjective optimization is also demonstrated in [28]. In general, solving this problem has been a challenge for researches for many years. The basic reviews can be found in Morton and Pentico [27], Herroelen et al. [11], Hartmann and Kolish [19], Weglarz [37], Brucker, et al. [7]. In this concepts different methods and techniques for a suitable solution searching are used, for example simulated annealing (SA), genetic algorithms (GA), tabu search and greedy search etc.

Dong-Eun Lee and Jonathan Jingesheng Shi in [23] are targeted on Stochastic Project Scheduling Simulation (SPSS) in combination with statistical analysiss tools. Their approach is based on CPM, PERT, and Discrete Event Simulation (DES) integration into one system.

Several approaches use Petri nets for project modeling, e.g. [17, 38, 12]. Kristensen et al. [20] propose a scheduling tool which is based a model specified by Coloured Petri Nets. It employs state space analysis capability of CPNTools for scheduling. The authors propose two algorithms. Their performance depends on a state space size and structure.

But, an interesting possibility which is still neglected nowadays is object oriented principles together with Petri nets use in the project planning, scheduling and monitoring. It is necessary to implement it by a suitable way to realize not only off-line scheduling, but also on-line optimization or dynamic modification of project parameters depending on the actual external conditions which are evolving in time (changes in resources structure, in project plans etc.).

Actually, Avanes [6] deals with adaptive workflow scheduling under resource allocation constraints and network dynamics. He proposes a dynamic scheduling procedure for distributed workflow managenent. On the other hand, we are focusing rather on basic design patterns for projects and resources modeling by means of a formalism based on Petri nets and objects. A the same time, we take in account model continuity in all phases of project engineering and also adaptation to the changing conditions in the environment.

We use the *Object oriented Petri nets* (OOPN) formalism [14, 8], which is a kind of the *multi-level Petri nets* [34] with a possibility to synchronize events at different levels. The OOPN formalism is very interesting for the project portfolio modelling domain because it offers the concept of dynamically instantiable method nets and shared places belonging to an object net. This feature allows for straightforward modelling of resources shared among a set of running projects (processes). Apart from obvious approaches in the area of project modeling, our model is well structured and allows for dynamic instantiations of project plans or sub-plans.

2 Object Oriented Petri Nets

A lot of attempts to combine Petri nets and object-orientation has been done since 1980s. The probably best known issues have been introduced by Lakos [21], Sibertin-Blanc [32], Moldt [26], and Valk [34]. One of the approaches is a formalism called Object Oriented Petri nets (OOPN) and PNtalk language and system that was developed in 1994 and published in [14, 8]. It combines pure object-orientation inspired by Smalltalk [10] with high-level Petri nets.

Following the Smalltalk-like style, all objects are instances of classes, every computation is realized by message sending, and variables can contain references to objects. A class defines structure and behavior of its instances. A class is defined incrementally, as a subclass of some existing class. In OOPN, this classical kind of object-orientation is enriched by concurrency. Concurrency of OOPN is accomplished by viewing objects as active servers. They offer reentrant services to other objects and at the same time they can perform their own independent activities. Services provided by the objects as well as the independent activities of the objects are described by means of high-level Petri nets - services by method nets, object activities by object nets. Tokens in nets are references to objects. Apart from the concurrency of particular nets, the finest grains of concurrency in OOPN are the transitions themselves (they can represent concurrency inside a method or object net).

An example illustrating the important syntactic elements of the OOPN formalism is shown in Figure 1. In the following, the OOPN syntax and semantics are briefly described.

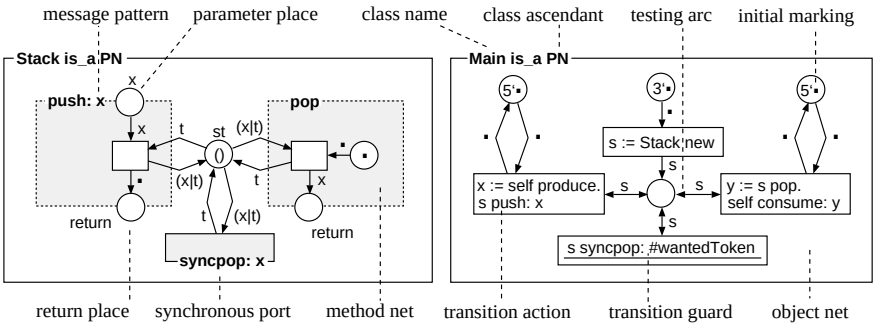


Fig. 1. An OOPN example

An Object Oriented Petri Net (OOPN) consists of Petri nets organized in classes. Each *class* consists of an *object net* and a set of dynamically instantiable *method nets*. Places of the object net are accessible for the transitions of the method nets. Object nets as well as the method nets can be inherited. Inherited transitions and places of the object nets (identified by their names) can be redefined and new places and/or transitions can be added in subclasses. Inher-

ited methods can be redefined and new methods can be added in subclasses. Classes can also define special methods called *synchronous ports*, which allow for synchronous interactions of objects. Message sendings and object creations are specified as actions attached to transitions. The transition execution is polymorphic — the method which has to be invoked is chosen according to the class of the message receiver that is unknown at the compile time. A token in a place represents either a *primitive object* (e.g., a number or a string) or an instance of an OOPN class. The instance consists of an instance of the appropriate object net and possibly several concurrently running instances of the invoked method nets.

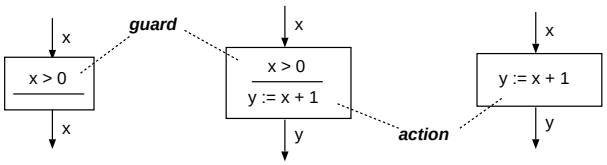


Fig. 2. Transition guard and action.

The transition guards and actions (Figure 2) can send *messages* to objects. The way how transitions are executed depends on the *transition actions*. A message that is sent to a primitive object is evaluated atomically (thus the transition is executed as a single event), contrary to a message that is sent to a non-primitive object. In the latter case, the input part of the transition is performed and, at the same time, the transition sends the message. Then it waits for the result. When the result is available, the output part of the transition can be performed. Each method net has parameter places and a return place. These places are used for passing data (object references) between the calling transition and the method net (Figure 3).

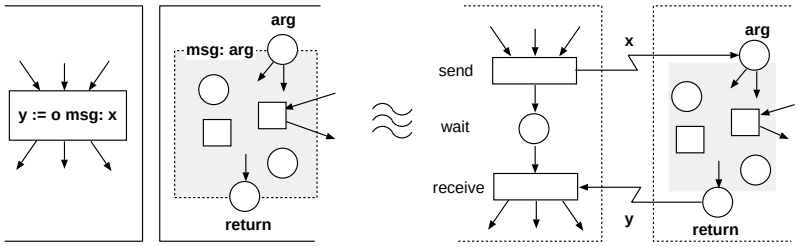


Fig. 3. Client-server interaction.

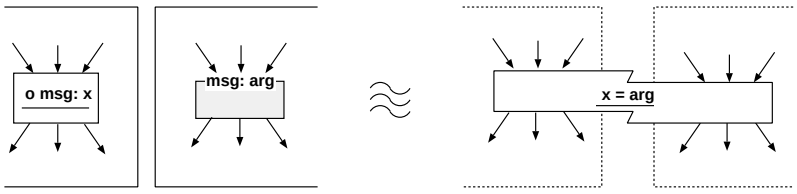


Fig. 4. Atomic synchronous interaction.

In the case of the *transition guard*, the message sending has to be evaluable atomically. Thus, the message sending is restricted only to primitive objects, or to non-primitive objects with appropriate synchronous ports. Synchronous ports allow for *synchronous interactions* of objects. This form of communication (together with execution of the appropriate transition and synchronous port) is possible when the calling transition (which calls a synchronous port from its guard) and the called synchronous port are executable simultaneously (Figure 4). A special variant of the synchronous port concept is *negative predicate*. Its semantics is inverted—the calling transition is firable if the negative predicate is not firable.

2.1 PNtalk

The OOPN formalism is implemented by a tool called PNtalk. PNtalk allows for a specification and simulation of OOPN-based models. In PNtalk, it is possible to specify delayed transition execution (in simulation time). Similarly to Simula-67, the delay is accomplished by sending *hold:* to *self* from a transition action. It is also possible to send the message *hold:* to *self* in a transition guard. In the former case, the execution of the output part of the transition is delayed, in the later case, it specifies the required enabling time for the transition. Note that it is possible to synchronize the simulation time with real time. A more detailed description of the PNtalk system can be found e.g. in [13, 15, 16].

3 Projects and Resources Modeling

The concept of OOPN-based modeling of the projects distinguishes two-levels in the model. The first level corresponds to the *project plans* and the second level corresponds to the *resources*. In the following, we explain the concept in more detail.

Actually, an OOPN object can model multiple projects (project portfolio [22] [5]). Tokens in the object net's places represent the shared resources. They are distributed in the places according to their roles. Method nets correspond to the individual project plan templates. Their instances can be dynamically created and destroyed (it corresponds to a start and a finish of a project) and they share an access to the object net's places containing pointers to the resource

objects. Project start is modelled by the appropriate message sent to the project portfolio object, possibly with parameters. At the same time a new instance of the method net (i.e. project plan) is created and starts to run. Note that it is possible to invoke a method (i.e. instantiate a project template) several times (with specific parameters) and the invocations can run in parallel. It corresponds to the situation where the project templates are instantiated.

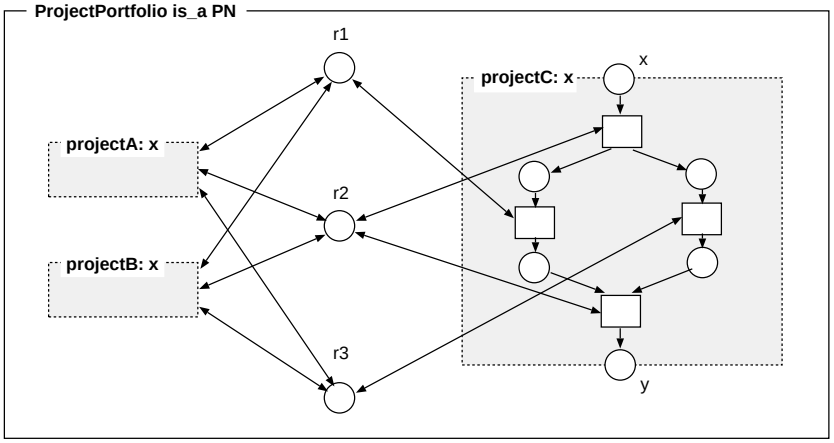


Fig. 5. Basic idea of project portfolio modeling.

An example of the approach is shown in Figure 5. There are described three different project templates (OOPN methods) which share the same resources. Actually, the resource types (their roles) are modelled by places of an object net). Project templates A and B are collapsed (their structure is not shown). Invocation of the methods, i.e. the individual projects creation, is accomplished by sending messages of type *self projectA: x* to the project portfolio object.

Resources are modelled by individual objects which are available as tokens in the places. Actually, the tokens are references to the objects. So it is possible to have a resource available under two and more roles modelled by the places. Class Resource is depicted in Figure 6.

Each *activity* (modeled as a transition of a method net which models a project) attempts to allocate all the resources it needs (by means of the corresponding synchronous ports calling) and if it succeeds, it uses the resources for some time (by means of an invocation of the corresponding method of the resource). Figure 7 depicts a simple example showing how a resource having two roles is used by two activities.

The above sketched model demonstrates only the core idea. Actually, it is necessary to model the resources in the context of some constraints, such as actual availability, skills, compatibility with the activities etc. In the case of multiple resources allocated to an activity, we must take in account also the

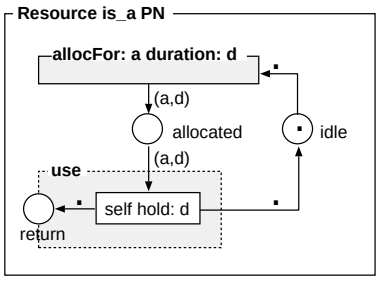


Fig. 6. Resource – base model.

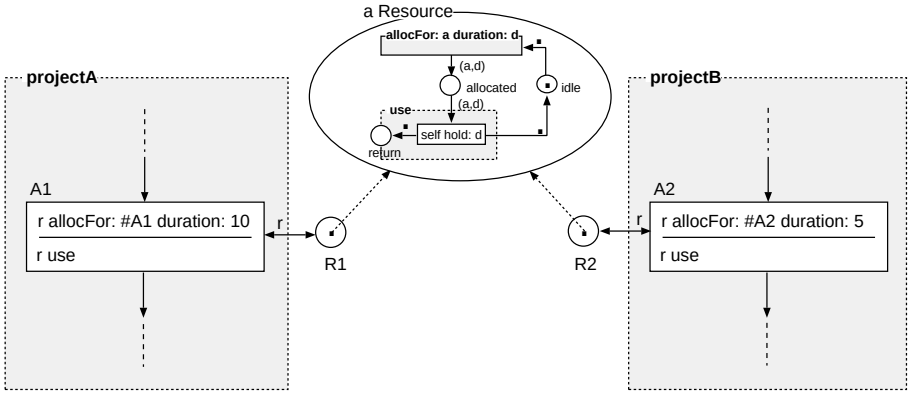


Fig. 7. Conflict of two activities requesting one resource having two roles.

quality of their team collaboration. All these attributes can be expressed using OOPN formalism effectively. Figure 8 depicts a definition of a resource with skills specified in a form of a set of tokens $(activity, skills)$ in the correponding place. Note that the shadow parts of the class specification are inherited from the superclass. Figure 9 depicts a definition of a resource with availability specified in a form of a set of tokens $(time, duration)$ in the correponding place.[1] Figure 10 depicts an example of multiple resources allocation.

3.1 Scheduling and Monitoring

The above sketched model can serve as a basis for scheduling [29]. A *scheduling process* generates a schedule satisfying the specified criteria, e.g. availability, skills, experience, and costs of the resources. We can use genetic algorithm (GA) for scheduling. As part of the fitness function, a simulation is performed in order

[1] It is possible to define a resource with both skills and availablility defined. But, for sake of simplicity, we take in account only availability in the following enhancements of the resource model (as depicted e.g. in Figures 11 and 12).

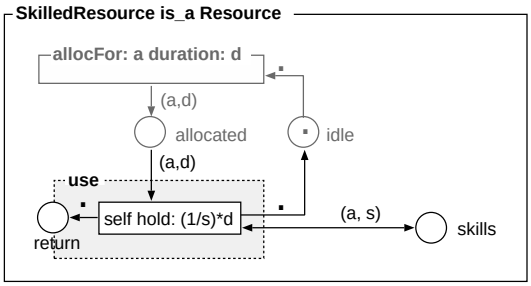


Fig. 8. Resource with skills specified.

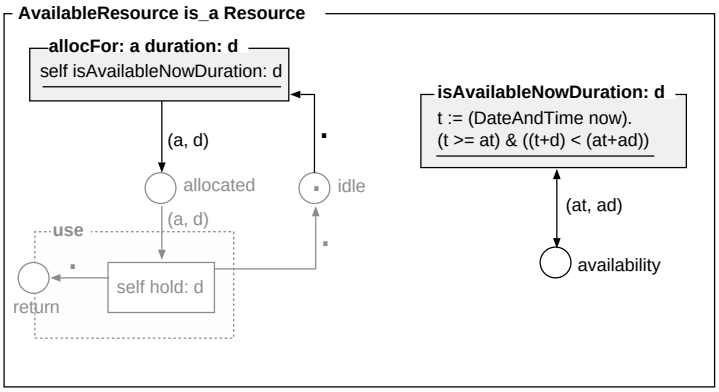


Fig. 9. Resource with availability specified.

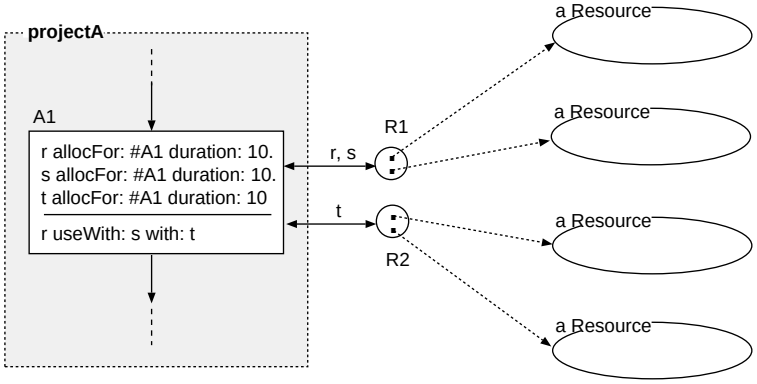


Fig. 10. Multiple resource allocation.

to check the feasibility of each candidate schedule and for gaining the essential performance results such as time and costs.

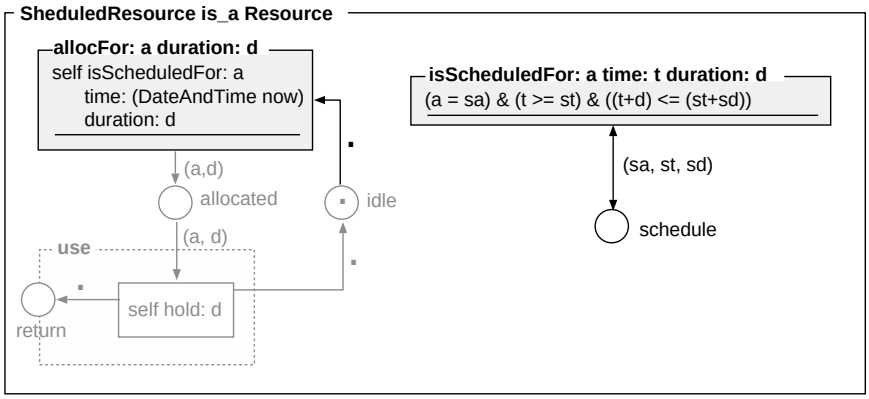


Fig. 11. ScheduledResource.

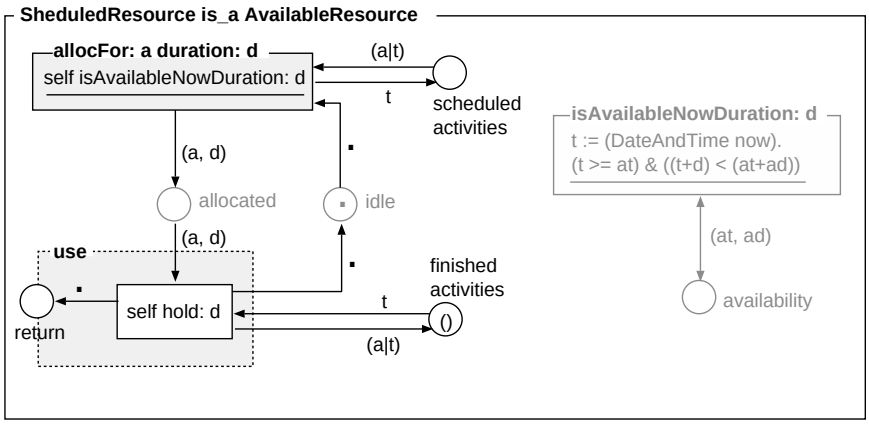


Fig. 12. Resource with another schedule representation (usable by GA).

All the scheduling objectives are weighted and aggregated to a single objective through weighted sum function: $v = \Sigma_{i=1}^{n} w_i u_i$, where u_i is subutility produced through objective i, w_i is relative importance of objective i. The goal of the schedule optimization strategy is to maximize v.

The resulting *schedule* can be attached to the corresponding resource r as a set of triples $(activity, time, duration)$ in a form of tokens in the the place *schedule* (see Fig. 11). When a resource with a schedule is being allocated (by the appropriate synchronous port calling), it checks whether it is scheduled for the requested activity, current time and the requested duration.

Nevertheless, we rather use another implementation of *ScheduledResource* class (see Fig. 12). It is used also for simulations performed as part of fitness function during the scheduling process. It uses a schedule representation which is compatible with the schedule genome used by GA (we use the schedule rep-

resentation from [28]). It is represented as a list of scheduled activities in the appropriate place. During simulation, a resource checks (besides availablilty) whether it is allocatted for the activities in the defined order. In the following, we explain the shedule representation (genome) in more detail.

Let us have activities $A = \{a_1, a_2, ..., a_m\}$ and resources $R = \{r_1, r_2, ...r_n\}$. A candidate schedule representation (a genome) is represented as n lists of variable length $((a_{11}, a_{12}, ...a_{1i_1}), ..., (a_{n1}, a_{n2}, ..., a_{ni_n}))$. A list represents scheduling information for a specific resource. Each list entry represents an activity scheduled on the resource. The genome is a permutation of all activities, an activity is present in only one of the lists. It contains no timing information; timing is generated from the genome using simulation. Simulation also checks feasibility of the schedule, i.e. it answers the question whether the project can reach its end state. It is possible because the model with a schedule is deterministic and the project plans contain no loops.

For a feasible schedule, it is possible to do a stochastic simulation with a probabilistic specification of the durations of the activities. It generates statistics about resource usage, costs, and project duration. Such an informatin can be used as part of fitness function.

A model with the scheduled resources can be used in the *monitoring* process. In that case, the model is simulated in real time[2] and is being dynamically updated according to the actual state of the reality. If necessary, a repeated scheduling is performed on-line. In that case, a clone of the model is used as a basis for the scheduling process. When a new schedule is found, it is incorporated back to the model as tokens in appropriate places in all the resource objects.

3.2 On the Implementation

The OOPN-based model of projects and resources is implemented using PNtalk tool [13, 15, 16].

Allthough PNtalk is a class-based object-oriented language, it is not used here purely as s typical class-based language. Instead, the above shown application rather needs to deal with individual instances. It concentrates more on dynamic evolution of objects in run time than on conceptual design before the system starts tu run.

As PNtalk is a dynamic language and an operating system which follows the Smalltalk tradition, it is possible to inspect and manipulate individual instances in run time in a way that is not known at the time when the instances are created. The inspection and manipulation with individual instances, i.e. objects and method invocations, as well as with their classes and methods is possible thanks to the PNtalk reflective interface (metaobject protocol, MOP) [13]. OOPN/PNtalk MOP allows for inspection and edition of particular nets which define classes, as well as individual net instances implementing the actually living objects and actually running method invocations. It also allows to make a

[2] Since it a a very long and slow simulation, a snapshot ot the model can be stored in a database for most of the time.

clone of the whole running system or its part and to store/restore it to/from a database in a serialized form.

It is possible to use PNtalk directly as part of a tool for decision support in project engineering. As part of future research, we are going to interconnect it with appropritate user interfaces to make usabe i practice. In future we also plan to investigate possibilities of PNML [3] representation of the models and tranformations to/from other tools such as CPNTools [2].

4 Conclusion

The main contribution of the paper is a presentation of a concept of projects and resources modeling by means od object oriented Petri nets. Projects are modeled by dynamically instantiable Petri nets (method nets). Their processes are parts of the project portfolio object. Resources are modeled as objects, which are available as tokens in the project portfolio object net places. The places can represent roles of the resources. Allocation and use of the resources are implemented using OOPN communication mechanisms.

The main advantages of using OOPN in project engineering we see in (1) formal nature of the model enabling potentially an analysis of projects using mathematical methods, (2) intuitively understandable model representation thanks to the visual nature of the formalism, (3) well structured model with clear mapping to the notions of the domain by means of simple design patterns, and (4) *model continuity* – an OOPN-based model is used as the main model which it is being dynamically updated continuously during the monitoring process, while its transformation to/from other views (models) is potentially possible at any time. The model evolution and its use in the dynamic scheduling process is possible thanks to the PNtalk open architecture (meta-object protocol), allowing for cloning, inspection end edition of the model at run time.

Acknowledgments. *This work was supported by Czech Grant Agency and Ministry of Education, Youth and Sports under the contracts GA102/07/0322 and MSM 0021630528.*

References

1. Project Engineering. [online] `http://en.wikipedia.org/wiki/Systems_engineering`.
2. CPNTools. [online] `http://wiki.daimi.au.dk/cpntools`, 2009.
3. Petri Net Markup Language. [online] `http://www2.informatik.hu-berlin.de/top/pnml`, 2009.
4. F. Anbari. *Q & As for the PMBOK Guide*. ISBN 1930699395. Project Management Institute, 2005.
5. Norman P. Archer and Fereidoun Ghasemzadeh. Project portfolio selection and management. In J.K. Pinto P.W.G. Morris, editor, *TheWiley Guide to Managing Projects*. Wiley, New York, 2004.

6. Artin Avanes and Johann-Christoph Freytag. Adaptive workflow scheduling under resource allocation constraints and network dynamics. *Proc. VLDB Endow.*, 1(2):1631–1637, 2008.
7. Peter Brucker, Sigrid Knust, Arno Schoo, and Olaf Thiele. A branch and bound algorithm for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 107(2):272–288, June 1998.
8. M. Ceska, V. Janousek, and T. Vojnar. PNtalk — a computerized tool for object oriented petri nets modelling. In *EUROCAST*, pages 591–610, 1997.
9. S. Rollins G. Kendall. *Advanced Project Portfolio Management and the PMO.* J. Ross Publishing, Boca Raton, FL, 2003.
10. A. Goldberg and D. Robson. *Smalltalk 80: The Language.* Addison-Wesley, 1989.
11. Willy Herroelen, Bert De Reyck, and Erik Demeulemeester. Resource-constrained project scheduling: a survey of recent developments. *Comput. Oper. Res.*, 25(4):279–302, 1998.
12. X.C. Jiao H.S. Yan, Z. Wang. Modeling, scheduling and simulation of product development process by extended stochastic high-level evaluation Petri nets. *Robotics and Computer-Integrated Manufacturing*, 19(4):329342, 2003.
13. V. Janousek and R. Koci. Towards an Open Implementation of the PNtalk System. In *Proceedings of the 5th EUROSIM Congress on Modeling and Simulation.* EUROSIM-FRANCOSIM-ARGESIM, Paris, FR, 2004.
14. V. Janoušek. PNtalk: Object Orientation in Petri nets. In *Proceedings of European Simulation Multiconference ESM'95*, pages 196–200. Prague, CZ, 1995.
15. V. Janoušek and R. Kočí. Towards Model-Based Design with PNtalk. In *Proceedings of the International Workshop MOSMIC'2005.* Faculty of management science and Informatics of Zilina University, SK, 2005.
16. V. Janoušek and R. Kočí. Simulation and design of systems with object oriented petri nets. In *Proceedings of the 6th EUROSIM Congress on Modelling and Simulation*, page 9. ARGE Simulation News, 2007.
17. Jongwook Kim, A.A. Desrochers, and A.C. Sanderson. Task planning and project management using petri nets. *Assembly and Task Planning, IEEE International Symposium on*, 0:0265, 1995.
18. Waldemar Kocjan. Dynamic scheduling state of the art report. Technical Report T2002:28, Dynamic scheduling state of the art report, 2002.
19. Rainer Kolisch and Sönke Hartmann. Experimental investigation of heuristics for resource-constrained project scheduling: An update. *European Journal of Operational Research*, 174(1):23–37, 2006.
20. Lars M. Kristensen, Peter Mechlenborg, Lin Zhang, Brice Mitchell, and Guy E. Gallasch. Model-based development of a course of action scheduling tool. *Int. J. Softw. Tools Technol. Transf.*, 10(1):5–14, 2007.
21. C. A. Lakos. From Coloured Petri Nets to Object Petri Nets. In *Proceedings of the Application and Theory of Petri Nets*, volume 935. Spinger-Verlag, 1995.
22. B. Lee. Multi-project management in software engineering using simulation modeling. *Software Quality Journal*, 12(1):59–82, 2004.
23. Dong-Eun Lee and Jonathan Jingsheng Shi. Statistical analyses for simulating schedule networks. In *WSC '04: Proceedings of the 36th conference on Winter simulation*, pages 1283–1289. Winter Simulation Conference, 2004.
24. Manfred Mauerkirchner. A general planning method for allocation of human resource groups. In Roberto Moreno-Díaz, Bruno Buchberger, and José Luis Freire, editors, *EUROCAST*, volume 2178 of *Lecture Notes in Computer Science*, pages 172–181. Springer, 2001.

25. Manfred Mauerkirchner and Gerhard Hoefer. Towards automated controlling of human projectworking based on multiagent systems. In Roberto Moreno-Díaz, Franz Pichler, and Alexis Quesada-Arencibia, editors, *EUROCAST*, volume 3643 of *Lecture Notes in Computer Science*, pages 281–290. Springer, 2005.
26. D. Moldt. OOA and Petri Nets for System Specification. In *Application and Theory of Petri Nets; Lecture Notes in Computer Science*. Italy, 1995.
27. T.E. Morton and D.W. Pentico. *Heuristic Scheduling Systems*. Wiley, New York, 1993.
28. Anna Persson, Henrik Grimm, Amos Ng, Thomas Lezama, Jonas Ekberg, Stephan Falk, and Peter Stablum. Simulation-based multi-objective optimization of a real-world scheduling problem. In *WSC '06: Proceedings of the 38th conference on Winter simulation*, pages 1757–1764. Winter Simulation Conference, 2006.
29. Michael Pinedo. *Scheduling: Theory, Algorithms, and Systems*. second edition. Prentice Hall, 2002.
30. Michael Pinedo. *Planning and Scheduling in Manufacturing and Services*. Springer, 2005.
31. K. Schwalbe. *Information Technology Project Management, Fourth Edition*. ISBN 1930699395. Course Technology, 2005.
32. C. Sibertin-Blanc. Cooperative Nets. In *Proceedings of Application and Theory of Petri Nets*, volume 815. Springer-Verlag, 1994.
33. M. Spinner. *Elements of Project Management: Plan, Schedule, and Control, 2nd Ed.* Englewood Cliff, NJ, Prentice Hall, 1992.
34. R. Valk. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In *Jorg Desel, Manuel Silva (eds.): Application and Theory of Petri Nets; Lecture Notes in Computer Science*, volume 120. Springer-Verlag, 1998.
35. Kabeh Vaziri, Paul Carr, and Linda K. Nozick. Program planning under uncertainty. In Shane G. Henderson, Bahar Biller, Ming-Hua Hsieh, John Shortle, Jeffrey D. Tew, and Russell R. Barton, editors, *Winter Simulation Conference*, pages 2119–2127. WSC, 2007.
36. Kabeh Vaziri, Linda K. Nozick, and Mark A. Turnquist. Resource allocation and planning for program management. In *Winter Simulation Conference*, pages 2127–2135. ACM, 2005.
37. J. Weglarz. *Project scheduling. Recent models, algorithms and applications.* Kluwer Academic, 1998.
38. Mengchu Zhou. Modeling, analysis, simulation, scheduling, and control of semiconductor manufacturing systems: A petri net approach. *Semiconductor Manufacturing, IEEE Transactions on*, 11(3):333–357, 1998.

On the Dynamic Features of PNtalk

Radek Kočí and Vladimír Janoušek

Brno University of Technology,
Bozetechova 2, 61266 Brno, Czech Republic
{koci,janousek}@fit.vutbr.cz
http://www.fit.vutbr.cz/

Abstract. PNtalk is a tool based on Object Oriented Petri nets. It is intended for systems modeling, simulation and prototyping. In some situations, it is also possible to use it as a programming language and a framework for final implementation of the systems. The paper presents a meta-level architecture of the PNtalk kernel and and demonstrates its reflective features. These features are crucial for the systems development process as well as for the systems maintenance. The usage of the PNtalk metaobjects are demonstrated by examples.

Key words: Object-oriented Petri nets, meta-level architecture, modeling, simulation, rapid prototyping

1 Introduction

PNtalk is a tool for modeling, simulation, and prototyping complex systems. It is based on a formalism called Object Oriented Petri Nets (OOPN). OOPN [8] combine advantages of object orientation and Petri nets. The OOPN formalism is based on high level Petri nets allowing to describe the work-flow and parallelism in the systems. Object orientation of the OOPN formalism allows for better structuring of the models, which is conform with current software development methodologies. OOPN define objects in a very similar way like other object oriented languages but with one important difference – the methods are not described as the sequences of commands but by means of high-level Petri nets. At a method call, a new instance (a copy) of the appropriate net is created and made ready for running.

The current version of the PNtalk tool allows for a higher level of dynamism and a higher level of control over the OOPN interpretation. PNtalk classes are special objects which can be built incrementally during run time (like in Smalltalk [5]). A method is then understood as the least unit constituting the basic block of the class. Along with considering the method to be a pattern and the invoked method to be a copy of that pattern, we can also think about dynamic changes of those structures. By the pattern change, we obtain new behavior of its new copies (invocations). The copies originated till this time are not changed. Nevertheless, the copy itself can be modified according to the same principle.

To achieve features described above, the open implementation issue including the reflective and meta-level architectures was taken into account. The feasibility of the open implementation approach depends on the degree of its implementation (thus on what the designers do consider as a profitable limit of the open implementation degree). The PNtalk system architecture is based on the idea of having the system as open as possible. The paper does not bring complex case study featurign reflection. It rather concentrates on the explanation of main architectural features using simple examples.

The idea of object-oriented computational reflection (including structural and behavioral changes of objects at run time) was proposed in 1970s [5] but roots of this concept are much older (Lisp, Universal Turing machine). As the examples of recent activities the following projects can be cited – Kiczales [12] and Maes [16] introducing aspect-oriented programming, Actalk [3] introducing concurrency via reflection in Smalltalk, Apertos [22] and TUNES [1] are attempts to develop a highly flexible operating system using computational reflection. As to the reflection in modeling and simulation, the most important (from the PNtalk point of view) are Barros [2] and Uhrmacher [20] introducing reflectivity to DEVS [23] in order to allow for structural changes of models. These approaches are important especially for modeling and simulation of intelligent agents.

Our attempts to incorporate reflection to the Petri nets also are not alone. Lakos [15] presents a reflective approach to Object Petri Nets implementation. He emphasizes the advantages of that approach in a clean, flexible and efficient implementation, and also in a possibility to investigate alternative scheduling schemes, interaction policies, etc.

The paper is organized as follows. First, we briefly describe the OOPN formalism and its important elements. The third chapter deals with the basic principle of the PNtalk architecture. The next three chapters describe the main architectural elements and features of the OOPN classes, OOPN objects, including simulation, and the inter-object communication at different levels.

2 Object Oriented Petri Nets

A lot of attempts to combine Petri nets and object-orientation has been done since 1980s. The probably best known issues have been introduced by Lakos [14], Sibertin-Blanc [19], Moldt [17], and Valk [21]. One of the approaches is a formalism called Object Oriented Petri nets (OOPN) and PNtalk language and system that was developed in 1994 and published in [7, 4]. It combines pure object-orientation inspired by Smalltalk [5] with high-level Petri nets.

Following the Smalltalk-like style, all objects are instances of classes, every computation is realized by message sending, and variables can contain references to objects. A class defines structure and behavior of its instances. A class is defined incrementally, as a subclass of some existing class. In OOPN, this classical kind of object-orientation is enriched by concurrency. Concurrency of OOPN is accomplished by viewing objects as active servers. They offer reentrant services to other objects and at the same time they can perform their own independent

activities. Services provided by the objects as well as the independent activities of the objects are described by means of high-level Petri nets - services by method nets, object activities by object nets. Tokens in nets are references to objects. Apart from the concurrency of particular nets, the finest grains of concurrency in OOPN are the transitions themselves (they can represent concurrency inside a method or object net).

An Object Oriented Petri Net (OOPN) consists of Petri nets organized in classes. Each *class* consists of an *object net* and a set of dynamically instantiable *method nets*. Places of the object net are accessible for the transitions of the method nets. Object nets as well as the method nets can be inherited. Inherited transitions and places of the object nets (identified by their names) can be redefined and new places and/or transitions can be added in subclasses. Inherited methods can be redefined and new methods can be added in subclasses. Classes can also define special methods called *synchronous ports*, which allow for synchronous interactions of objects. Message sendings and object creations are specified as actions attached to transitions. The transition execution is polymorphic — the method which has to be invoked is chosen according to the class of the message receiver that is unknown at the compile time. A token in a place represents either a *primitive object* (e.g., a number or a string) or an instance of an OOPN class. The instance consists of an instance of the appropriate object net and possibly several concurrently running instances of the invoked method nets.

The transition guards and actions can send *messages* to objects. The way how transitions are executed depends on the *transition actions*. A message that is sent to a primitive object is evaluated atomically (thus the transition is executed as a single event), contrary to a message that is sent to a non-primitive object. In the latter case, the input part of the transition is performed and, at the same time, the transition sends the message. Then it waits for the result. When the result is available, the output part of the transition can be performed. Each method net has parameter places and a return place. These places are used for passing data (object references) between the calling transition and the method net.

In the case of the *transition guard*, the message sending has to be evaluable atomically. Thus, the message sending is restricted only to primitive objects, or to non-primitive objects with appropriate synchronous ports. Synchronous ports allow for *synchronous interactions* of objects. This form of communication (together with execution of the appropriate transition and synchronous port) is possible when the calling transition (which calls a synchronous port from its guard) and the called synchronous port are executable simultaneously. A special variant of the synchronous port concept is *negative predicate*. Its semantics is inverted—the calling transition is firable if the negative predicate is not firable.

An example illustrating the important elements of the OOPN formalism is shown in Figure 1. Two classes *C0* and *C1* are depicted there. The object net of the class *C0* consists of places *p1* and *p2* and one transition *t1*. The object net

of the class *C1* is empty. The class *C0* has a method *init:*, a synchronous port *get:*, and a negative predicate *empty*. The class *C1* has the method *doFor:*.

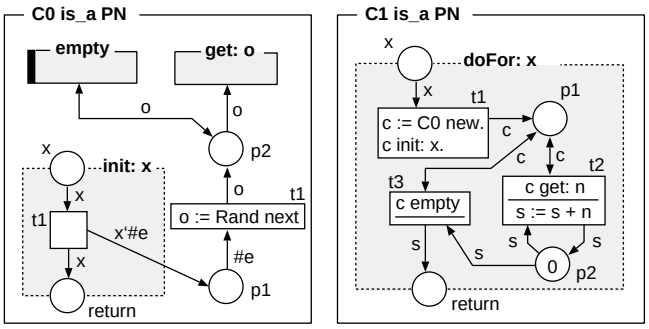


Fig. 1. An OOPN example.

Let us have an expression *C1 new doFor: 3*. Its execution leads to the creation of an instance (object o) of $C1$ and an instance of *doFor:* belonging to the object o. When $t1$ belonging to the instance of *doFor:* inside the object o leads to the instantiation of $C0$ (let the instance be named o') and *init:* inside the object o'. Then, $t1$ inside o' can be executed for three times. Consequently, $t2$ inside the instance of *doFor:* inside o can be executed. Its guard invokes synchronous port *get* : of o'. The variable n is boud to the value of the token from the place $p2$ belonging to o'. When the place $p2$ of o' is emptied, $t3$ belonging to the instance of *doFor* : inside the object o can be fired because the negative predicate *empty* of o' is satisfied.

Each OOPN model has its text representation (source code). It is very important for the meta level manipulation with models. We will demonstrate it on the previous example. The Figure 2 shows the text representation of the class C0, its object net (the keyword `object`), its synchronous port (the keyword `sync`), its negative port (the keyword `negative`), and its method net (the keyword `method`).

Every transition, synchronous port, and negative predicate define its preconditions, conditions, and postconditions. For instance, the synchronous port `get: o` is conditioned by the precondition `precond p2(o)`. It means, that the port is firable if the place contains at least one object and if the port will be fired it will remove this object from the place `p2`. In addition, transitions and ports can have guards (the keyword `guard`) and actions (the keyword `action`) as we can see in the Figure 3 of the class `C1` source code.

```
class C0 is_a PN
  object
    place p1()
    place p2()
    trans t1
      precond p1(#e)
      action{o := Rand next}
      postcond p2(o)
  sync get: o
    precond p2(o)
  negative empty
    cond p2(o)
  method init: x
    place x()
    place return()
    trans t
      precond x(x)
      postcond p1(x‘#e), return(x)
```

Fig. 2. The source code of the class `C0`.

```
class C1 is_a PN
  method doFor: x
    place x()
    place p1()
    place p2(0)
    place return()
    trans t1
      precond x(x)
      action {
        c := C0 new.
        c init: x.}
      postcond p1(c)
    trans t2
      precond p2(s)
      cond p1(c)
      guard {c get: n}
      action {s := s + n.}
      postcond p2(s)
    trans t3
      precond p2(s)
      cond p1(c)
      guard {c empty}
      postcond return(s)
```

Fig. 3. The source code of the class `C1`.

3 PNtalk System Architecture

PNtalk (*P*etri *N*et *talk*) is the tool based on the formalism of OOPN. Its purpose is to make a framework for experiments with simulations as well as formal approaches to the system design [10, 11]. Both OOPN and PNtalk are closely associated with the Smalltalk environment. Smalltalk is the inscription language of the OOPN formalism (actions and guards are described using Smalltalk) and the PNtalk system is implemented in Smalltalk. The PNtalk system is incorporated into the other experimental tool named *SmallDEVS* [6] which is based on the DEVS formalism [23]. PNtalk uses hierarchical repositories of SmallDEVS to store OOPN classes and allows for joining models described by OOPN and DEVS formalisms.

This chapter discusses basic ideas behind the PNtalk system architecture. It is based on the principles of open implementations [9], namely the meta-level architecture. These principles are also shown on the examples. They are written in the Smalltalk environment, but they can be used inside models too.

3.1 Open implementations

The recent systems for complex application support allow the applications not only to use the services offered by the system, but they also offer means to control how these services are provided and processed. The traditional approach (the black-box abstraction) says that some abstraction (object) should expose its functionality but hide its implementation. It has many attractive qualities and brings a possibility of portability, reusing or simplicity of the design process. Nevertheless, it does not allow to adapt parts of the system according to the changing requirements, and/or to develop the applications during their life-time etc. The open implementation principle offers a solution of the problems.

The basic idea of an open implementation is to allow a model to inspect inner aspects of the domain objects (introspection) and to work upon these aspects (reflection). The classic case of an open implementation is the meta-level architecture partitioning a model into two layers – the domain (or basic) level and the meta-level [16]. All the objects describing the domain problem represent the domain level. To each object at the domain level there is a special object (or a set of objects) at the meta-level – metaobject. The meta-level should be understood as a denotation of something what stays behind an object and reflects (or describes) its features and properties—de facto describes information about information. A metaobject offers the metaobject protocol for inspecting and changing the selected aspects of its domain object. The meta-level architecture allows not only to work upon structures of the domain objects but also to modify their computational behavior, e.g. the way how the objects react to messages, what other operations are to be processed in a consequence of sending or receiving messages, etc.

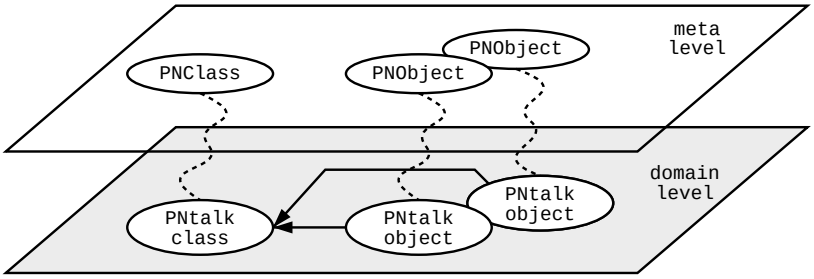


Fig. 4. The PNtalk meta-level architecture – basic overview.

3.2 Meta-level Architecture

The PNtalk architecture introduces a new meta level between the domain objects (i.e., PNtalk classes and PNtalk objects) and Smalltalk. Objects belonging

to the meta-level are implemented by classes of Smalltalk (a basic overview of the PNtalk architecture is presented in Figure 4). While the Smalltalk meta-level architecture is based on classes (i.e., objects are instances of their metaobjects), the PNtalk meta-level architecture is based on objects (i.e., object are not instances of metaobjects, but metaobjects implement the corresponding domain objects).

The PNtalk meta-level comprises metaobjects which control PNtalk classes and PNtalk objects. The metaobjects of the first kind describe *the structure* of PNtalk classes and define the ways of the manipulation with them. The metaobjects of the second kind describe *the computational behavior* of the PNtalk objects (instances of PNtalk classes).

3.3 Metaobjects composition

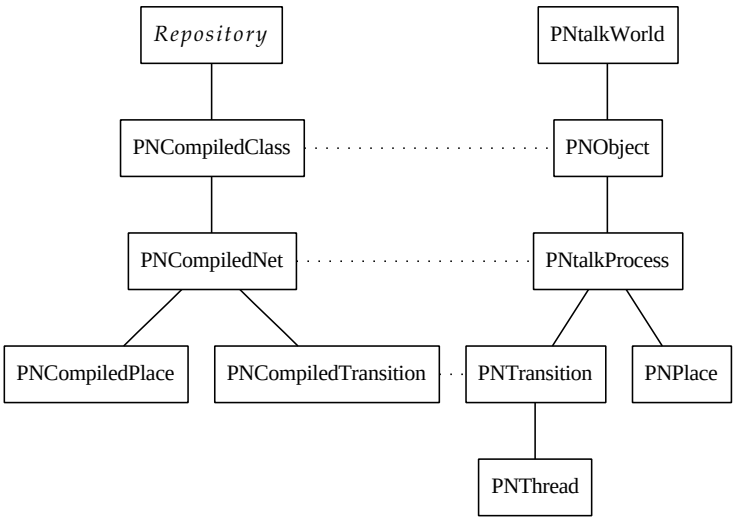


Fig. 5. The basic composition of the metaobjects.

Before we describe the metaobjects and their protocol, we should pay attention to the basic composition of them (see the Figure 5). We will not discuss all parts of the composition, but we will just deal with the most important ones. As we have already said each OOPN class or object is represented by its metaobjects. These metaobjects are instances of the suited classes. These classes are part of the framework implemented, in this case, in Smalltalk. For instance, the OOPN class is represented by the instance of the framework class

`PNCompiledClass` (since the instance is a result of some compilation process we call it with a prefix `Compiled`).[1]

Each OOPN class (the metaobject *PNCompiledClass*) consists of nets (the metaobject *PNCompiledNet*), each net consists of places and transitions, etc. Each OOPN class is placed in a repository which serves as a name space for the domain classes. The metaclass for the repository is specified by italic font in the Figure 5—the repository is a part of the SmallDEVS system to which the PNtalk system is incorporated. This part is not very important for the way how the PNtalk system is explaned here; for more details about this please see [6].

Each OOPN object (the metaobject *PNObject*) is an instance of its OOPN class. In the architecture, the metaobject *PNObject* knows about its class reprtesented by the metaobject *PNCompiledClass*. So that if we send a message to the OOPN object at the domain level, the object looks for the method in the dictionary described by the instance of the class `PNCompiledClass` at the meta level.

Each OOPN object consists of processes (the metaobject *PNProcess* represents an invocaion of a method or an object net), each process consists of transitions (the metaobject *PNTransition*), places (the metaobject *PNPlace*), and thread (the metaobject *PNtalkThread*). Threads represent fired transitions, the transition can be fired for more times simultaneously. The simulation is represented by the metaobject *PNtalkWorld*. Each OOPN object has to be placed into some world in order to be runnable (i.e. can be simulated).

In addition to this, there is the special metaobject accessible via the name *PNtalk* (it is an instance of the class *PNtalkSystem*) supporting special services and requirements (garbage collecting, getting other auxiliary metaobjects etc.). This metaobject is not shown in the presented hierarchy because it is not important for this paper.

4 Representation of the Domain Classes

This section discusses the part of the PNtalk architecture describing the OOPN classes. This part consists of objects (and their classes) representing appropriate elements of OOPN classes (i.e., the *PNCompiledClass* for OOPN class, the *PNCompiledNet* for method net, etc.) and other auxiliary metaclasses. The inheritance hierarchy of classes ot these metaobjects is shown in the Figure 6. The auxiliary class *PNClassComponent* is a root of this inheritance hierarchy and supplies basic services for all other classes of the PNtalk metaobjects. We will deal only with the *PNCompiledClass* in this paper because of two reasons: first, the other classes from this part is not used in our examples, and, second, due to the limited space of this paper.

[1] For the text simplicity, when we will talk about, e.g., *the metaobject `PNCompiledClass`*, we will understand it as an instance of the framework class `PNCompiledClass`.

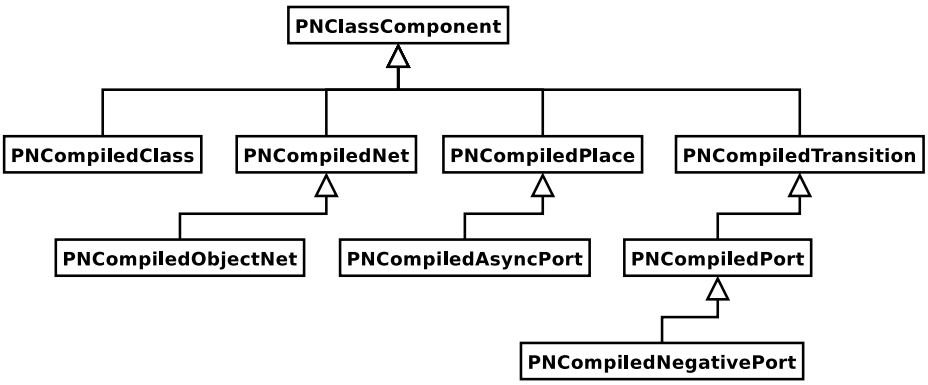


Fig. 6. The inheritance hierarchy of the classes of the OOPN class metaobjects.

PNCompiledClass Instances of *PNCompiledClass* represent the domain classes of the OOPN formalism. The list of selected meta-operations from the metaobject protocol follows:

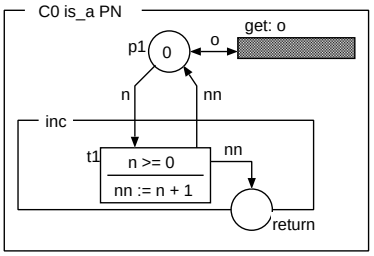
`compile:` compiles a source code of the method (or object) net or synchronous (or negative) port and adds the compiled one (i.e., the instance of the metaclass *PNCompiledNet*, or *PNCompiledPort*, or *PNCompiledNegativePort*) into this PNtalk class
`new` creates the instance of this PNtalk class
`newIn:` creates the instance of this PNtalk class and placed it into the specified simulation space

4.1 Example: Creating the New OOPN Class

Let us have simple example shown in the Figure 7. It demonstrates creating of the new metaobject representing the new PNtalk class `C0`. We can see that OOPN have their text representation (source code) which can be used to the class description and construction. The figure shows the resulting class in graphic notion. This example creates one object net and one method net. The object net consists of the place `p1` and the synchronous port `get:`. The method net consists of the transition `t1` which increments a content of the place `p1` and returns the result (the place `return`).

5 Representation of the Domain Objects and Simulation

This section discusses the part of the PNtalk architecture describing OOPN objects and their simulation. This part consists of the classes of the metaobjects representing appropriate elements of OOPN objects (i.e., the *PNObject* for OOPN object, the *PNProcess* for running method net, etc.) and other auxiliary metaobjects. The inheritance hierarchy of these classes is shown in the Figure




```
cls := PNCompiledClass new.
cls name: 'C0'.
```

```
cls compile: '
object
  place p1(0)
  sync get: o
    cond p1(o)
method inc
  place return()
  trans t1
    precond p1(n)
    guard { n >=0 }
    action { nn := n + 1 }
    postcond p1(nn), return(nn)
'.
```

Fig. 7. The example of creating the new OOPN class.

8. We will deal only with the selected metametaobjects in this paper because of reasons mentioned in the section 4.

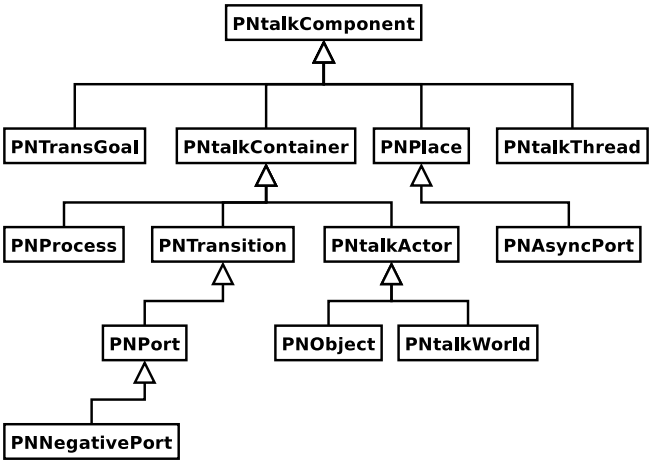


Fig. 8. The inheritance hierarchy of the classes of the metaobjects of the OOPN objects and simulations.

PNtalkComponent The class *PNtalkComponent* is a root of the inheritance hierarchy. It offers basic metaprotocol which is common for all metaobjects. The protocol mainly consists of operations allowing to set or to get the unique object identification (`id`, `id:`), name (`name`, `name:`), and parent object (`parent`,

`parent:`) from the metaobject composition point of view (e.g., a place is a component of some process).

PNPlace Instances of the class *PNPlace* represent places in object or method nets. The list of the selected meta-operations from the metaobject protocol follows:

`add:mult:` adds multiple copies of specified object into the place
`take:mult:` gets and removes multiple copies of specified object from the place
`contains:` tests if the place contains specified object

PNtalkContainer The important metaclass is *PNtalkContainer*. It offers basic metaprotocol for such metaobjects which can store other metaobjects. The metaprotocol allows for adding, removing, and searching of subcomponents. The list of the selected meta-operations from the metaobject protocol follows:

`addComponent:` adds a specified component
`removeComponentNamed:` remove a component identified by the specified name
`componentID:` gets a component identified by the specified id
`componentNamed:` gets a component identified by the specified name
`componentNames` gets a collection of components names

PNProcess Instances of the metaclass *PNProcess* represent evoked method or object nets. The list of selected meta-operations from the metaobject protocol follows:

`placeNamed:` gets a metaobject of *PNPlace* specified by its name
`transitionNamed:` gets a metaobject of *PNTransition* specified by its name

PNtalkWorld The instance of this class represents one simulation space which is called *world*. To be simulated (executed), each OOPN object has to be placed into some world. The simulation algorithms are implemented by this metaobjects in coordination with the metaobject of *PNObject*. The list of selected meta-operations from the metaobject protocol follows:

`start` starts a simulation. This operation is asynchronous and enables inner simulation mechanism. This mechanism calls the *step* operation in the cycle until there is at least one event or if the new one occurs.
`stop` stops a simulation. This operation is asynchronous and disables inner simulation mechanism.
`step` does one simulation step. In each step, it sends the message *step* to the each object having at least one event to perform.
`test` tests transitions and event for firability (needed after every changes from outside).

PNObject Instances of the class *PNObject* represent domain objects, i.e., objects of the OOPN formalism. It offers means for its simulation too. The list of selected meta-operations from the metaobject protocol follows:

`yourClass` gets a metaobject representation (the metaobject *PNCompiledClass*) of the OOPN class.

`compile:` compiles a source code and adds newly created elements (changed object net, method net, ports, etc.) into the object. These changes do not take effect in the OOPN class but only in this instance (an OOPN object).

`performDomainMessage:` performs a domain message. The message is in a special form (special metaobject). The operation is asynchronous—it looks for appropriate method net, creates its instance, i.e., the process as an instance of the metaobject *PNProcess*, and returns. The process execution is then under the control of *PNObject* and is independent of the other processes. At the domain level, the message sending is synchronous, i.e., the calling thread or object waits until this process finished.

`testPort:` tests a port. The argument is a special metaobject representing the name of the port and its arguments. If the port can be fired, this method returns a set of bindings of possible ways of firing.

`performBoundPort:` performs the bound port. This operation is called after the operation *testPort:* and its argument is a special metaobject representing the name of the port and the binding the port should be fired for.

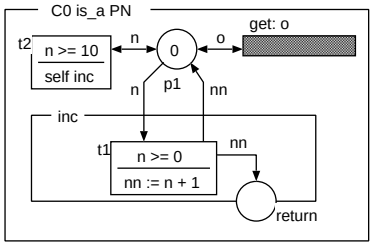
`step` performs one firable event. The event can be

- to fire a transition (creating the thread) including performing its first command
- to perform next command of the thread. The thread is finished along with the last command (it is taken as an atomic event). If the transition has no command, it is fired and finished as one atomic event.
- to finish the message processing. If some object is placed into the place `return` of the called process, it notifies the calling thread (the new event is created). When this event is performed, the thread acquires the return object and destroys called process.

5.1 Example: Simulation

Let us have simple example shown in the Figure 9. It creates new simulation world (`PNtalkWorld new`), new instance of the OOPN class `CO` (the class is shown in picture on the left) and puts this object into the world (`newIn:`). Because this operation returns a special proxy-metaobject (it will be explained in the section 6) we have to get the metaobject `PNObject` by calling `meta`. Then we get metaobjects stepwise: the object net (`componentID: 1`; each object net is always first process so that it has the number 1) and the place named `p1` (`placeNamed: 'p1'`). Now we put the object `11` into the place (`put: 11 mult: 1`) and have to test the changed object (we can test the whole simulation too as it is shown in out example—`world test`). Now we call the operation `step` of the world. This operation is called twice. The first calling causes the transition

`t2` is fired and the operation `inc` is evoked (the process is created). The second calling causes the transition `t1` is fired and the operation `inc` is finished along with the transition `t2` (transition firing and ending as well as process creating and finishing are understood as atomic operations from the one simulation step point of view). At the end, we can test if the place `p1` contains the object `12` (`contains: 12`).




```
world := PNtalkWorld new.
cls := rep componentNamed: 'C0'.
obj := cls newIn: world.
mobj := obj meta.
objnet := mobj componentID: 1.
place := objnet placeNamed: 'p1'.
place put: 11 mult: 1.
world test.
world step. world step.
place contains: 12. "=> true"
```

Fig. 9. The example of the OOPN simulation.

6 Communication mechanisms

As we already said, the formalism of OOPN and PNtalk are closely associated with Smalltalk environment. It implies that there can be native cooperation between OOPN and Smalltalk objects, so that it is possible to transparently access OOPN objects from Smalltalk and vice versa. It is also possible to use Smalltalk objects as the tokens in OOPN and to use PNtalk objects as Smalltalk objects.

To achieve the general communication between objects at different levels, the concept of *proxyobjects* is introduced. The proxyobject does not define the computational behavior of the receiver, but it ensures message passing in the requested way and it conforms the response to sent messages in accordance with the requirements of the sender. Thus, the proxyobject adapts the computational behavior of the receiver to the computational behavior of the sender.

Using proxy is standard Smalltalk technique to control message passing. A proxy is obviously implemented in such a way that it handles the exception *messageNotUnderstood*. The handler (implemented in the proxy) decides how to react to the message. However, the PNtalk proxyobjects have to implement additional properties necessary for the metaobject protocol.

The PNtalk architecture distinguishes several kinds of proxyobjects: a proxy for Smalltalk object, a proxy for PNtalk object (thus a domain level point of view) and a proxy for PNtalk metaobject (thus a meta-level point of view). Each

object is referenced by means of the appropriate proxyobject (mostly the proxyobject is either for PNtalk object or for Smalltalk object) devolving incoming messages on the receiver in the suitable form. The class hierarchy of proxyobjects is shown in the Figure 10.

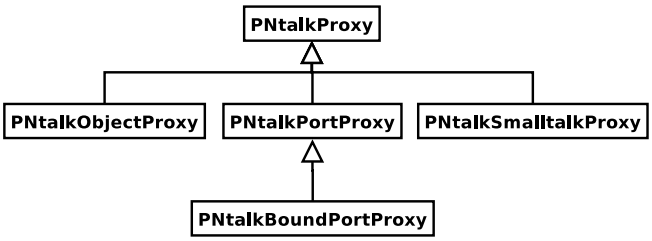


Fig. 10. The composition of the metaobjects *PNClass* and *PNObject*.

PNtalkObjectProxy It serves as a basic proxy-object to the PNtalk object represented by the metaobject *PNObject*. An object (potential sender of a message) always refers to another object (potential receiver) by means of a proxyobject. In spite of the domain level point of view, the actual sender is either the Smalltalk object or the PNtalk object. The receiver is always PNtalk object. Therefore, the metaobject *PNtalkObjectProxy* always refers to the metaobject *PNObject*. The list of selected meta-operations from the metaobject protocol follows:

`performDomainMessage:` performs a domain message. The metaobject *PNObject* should be always the sender, therefore this operation serves for message passing between PNtalk objects. It simply forwards the same message to the receiver (wrapped object).

`doesNotUnderstand:` operates a message unknown for the proxy-object. The message is a domain message of the receiver, the sender is not a metaobject but a Smalltalk object. It sends the caught message via the *performDomainMessage:* operation to the receiver and then waits for its result (when the process is being finished, it notifies this metaobject, see `step` in the *PNObject* metaobject protocol list). This message servers for communication from Smalltalk object to the PNtalk objects.

`asPort` returns a metaobject *PNtalkPortProxy* for the same receiver.

PNtalkPortProxy It servers for accessing PNtalk objects from Smalltalk objects at the level of ports. The list of selected meta-operations from the metaobject protocol follows:

`doesNotUnderstand:` operates a message unknown for the proxy-object. The message is a port calling of the receiver, the sender is not a metaobject but

a Smalltalk object. It sends the caught message via the *testPort:* operation to the receiver. It returns a metaobject *PNtalkBoundPortProxy* which represents the result of the port testing.

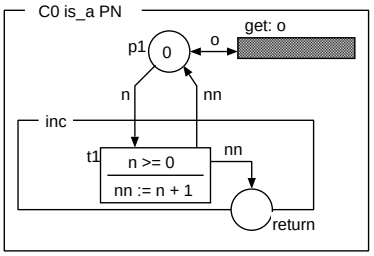
PNtalkBoundPortProxy It serves as an information storage for the bound port. It can be got only via the operation *testPort:* of the metaobject *PNtalkPortProxy*. The list of selected meta-operations from the metaobject protocol follows:

`ifTrue:` performs the specified block of commands if the port testing was successful (there is at least one possible binding)
`ifFalse:` performs the specified block of commands if the port testing was unsuccessful (there is no possible binding)
`collectBindings:` returns all bound variables for specified binding (the number). The operation returns it as an associated array (a.k.a. map or dictionary) of pairs (`name, value`).
`collectBindings` returns a collection of all bindings, for each binding it uses the operation `collectBindings:`.
`collectVariable:` returns a collection of all possible bindings of specified variable.
`variable:` returns a bound value of specified variable for the first binding.
`perform:` performs the port for the specified binding (the number).
`perform` performs the port for the first binding.

6.1 Example: Object Introspection

Let us have the simple example shown in the Figure 11. It supposes that there is a running default world and the PNtalk classes repository is accessible via the variable `rep`. First, we get the metaobject representing the PNtalk class `C0` (see the picture on the left). Then we create an instance of it—in fact, we get a proxy-object of `PNObjectProxy` to the metaobject of `PNObject`. Although the metaobject `PNObject` needs to receive domain message in a special way, we are able to send a message in the ordinary way via the proxy-object (see `obj inc`—the return value will be `1`).

The second part of this example shows how to use synchronous port with free (unbound) variables. We send the message in the same way but we use a special metaobject as the appropriate argument. In our example, we want to get a content of the place `p1` using the port `get:`. By calling `PNtalk variableNamed: #w`, we get the special metaobject representing the variable named `v`. Then we get a special metaobject for communication at the port level (`obj asPort`) and send the message for port named `get:` with free variable named `v`. The result is the metaobject *PNtalkBoundPortProxy* storing information about all possible bindings for variable `v` (it is just one object in our example). If the test is successful (see `port ifTrue:`) we can get a bound value of the variable `v` (see `port variable: #v`).




```
cls := rep componentNamed: 'C0'.
obj := cls new.
res := obj inc. "res is 1"
obj inc.
port := obj asPort get: (
      PNtalk variableNamed: #v).
port ifTrue: [
  res := port variable: #v.
  "res is 2"
].
```

Fig. 11. The example of the object introspection.

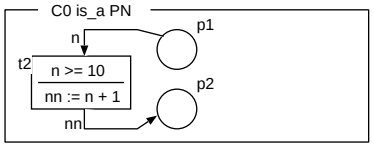
6.2 Example: Object Modification

Let us have the simple example shown in the Figure 12. It supposes that there is running default world and the PNtalk classes repository is accessible via the variable `rep`. First, we get the metaobject representing the PNtalk class `C0` (see the picture on the left). Then we create an instance of it. We can compile new elements into the objects: two synchronous ports `put:` and `get:` (see `obj meta compile: ...`). The resulted object net is shown in the picture on the right. Now, we can communicate with the object using these ports. First, we put a number `20` into the place `p1` (see a sequence of `obj as Port put: 20` and `p perform`). The transition `t1` will be fired because the world is running and the transition becomes firable immediately the port is performed. Second, we put a number `30` into the place `p1` by the same way.

Now, we can get the content of the place `p2` using the synchronous port `get:` (it is the same principle as described in the Example 11). We can get a collection of all bindings of the variable `v` (see `p collectVariable: #v`; the collection will contain numbers `21` and `31`). When we perform this port for the first binding (`v==21`), the number `21` will be removed from the place `p2`.

7 Conclusion

We have presented the basis of the PNtalk system architecture and have demonstrated its features by simple examples. The goal of the PNtalk project is not only to make a tool intended for modeling and simulation but also to make tool allowing the developed model to be integrated into a real environment. Such a model can then serve as a part of the prototype or the target application. When we take in account the reflective features, we can use this system as a framework for interactive application development. The framework allows us to build models and prototypes, to combine different paradigms for the model specification, to experiment with new paradigms, or to allow both interactive and automatic evolution of the models. For instance, the reflection was used for merging OOPN and DEVS formalisms.




```
cls := rep componentNamed: 'C0'.
obj := cls new.
obj meta compile: 'sync put: o
  postcond p1(o)'.
obj meta compile: 'sync get: o
  precond p2(o)'.
p := obj asPort put: 20.
p ifTrue: [ p perform ].
p := obj asPort put: 30.
p ifTrue: [ p perform ].
p := obj asPort get: (
      PNtalk variableNamed: #v).
p ifTrue: [
  res := p collectVariable: #v.
  "res is a collection (21 31)"
  res perform: 1.
  "the place p1 contains 31"
].
```

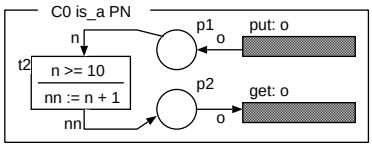


Fig. 12. The example of the object modification.

One of possible application domains is artificial intelligence, especially the area of intelligent multi-agent systems. One of our experimental applications is PNagent [13]. It is a framework for rational agents development. It uses fragments of plans specified by Petri nets and the whole framework is implemented using OOPN in PNtalk. Consequently, it is possible to continually develop both the agents and the agent framework using the same language featuring both visual representation and formal basis. We have experimented with simple case studies [11, 18] in the field of software engineering, too.

Acknowledgments. This work was supported by the Czech Grant Agency under the contracts GA102/07/0322, GP102/07/P306, and Ministry of Education, Youth and Sports under the contract MSM 0021630528.

References

1. The TUNES Project for a Free Reflective Computing System. http://tunes.org, 2006.
2. F. J. Barros. Modeling formalisms for dynamic structure systems. In *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 1997.
3. J. Briot. Actalk : A framework for object oriented concurrent programming - design and experience. In *Object-Based Parallel and Distributed Computing II - Proceedings of the 2nd France-Japan Workshop*. Herms Science, 1999.
4. M. Ceska, V. Janousek, and T. Vojnar. PNtalk — a computerized tool for object oriented petri nets modelling. In *EUROCAST*, pages 591–610, 1997.
5. A. Goldberg and D. Robson. *Smalltalk 80: The Language*. Addison-Wesley, 1989.
6. V. Janousek and E. Kironsky. Reflective Framework for Interactive Modeling and Simulation of Intelligent Systems. In *Proceedings of Nineteenth International Con-*

ference on Systems Engineering 19-21 August 2008 Las Vegas, Nevada, Los Alamitos, US, pages 480–485. IEEE Computer Society, 2008.
7. V. Janoušek. PNtalk: Object Orientation in Petri nets. In *Proc. of European Simulation Multiconference ESM'95*. Prague, CZ, 1995.
8. V. Janoušek. *Modeling objects by Petri nets (in czech)*. PhD thesis, 1998.
9. V. Janoušek and R. Kočí. Towards an Open Implementation of the PNtalk System. In *Proceedings of the 5th EUROSIM Congress on Modeling and Simulation*. EUROSIM-FRANCOSIM-ARGESIM, Paris, FR, 2004.
10. V. Janoušek and R. Kočí. Towards Model-Based Design with PNtalk. In *Proceedings of the International Workshop MOSMIC'2005*. Faculty of management science and Informatics of Zilina University, SK, 2005.
11. V. Janoušek and R. Kočí. Simulation and design of systems with object oriented petri nets. In *Proceedings of the 6th EUROSIM Congress on Modelling and Simulation*, page 9. ARGE Simulation News, 2007.
12. G. Kiczales, J. Lamping, A. Menhdhekar, Ch.Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241 of *1241*. Springer-Verlag, 1997.
13. R. Kočí, Z. Mazal, F. Zbořil, and V. Janoušek. Modeling Deliberative Agents Using Object Oriented Petri Nets. In *Proceedings of the 7th ISDA*, pages 15–20. IEEE Computer Society, 2007.
14. C. A. Lakos. From Coloured Petri Nets to Object Petri Nets. In *Proceedings of the Application and Theory of Petri Nets*, volume 935. Spinger-Verlag, 1995.
15. Ch. Lakos. Towards a Reflective Implementation of Object Petri Nets. In *Proceedings of TOOLS Pacific*. Melbourne, Australia, 1996.
16. P. Maes. Computational reflection. Technical report, Artifical Inteligence Laboratory, Vrije Universiteit Brusel, 1987.
17. D. Moldt. OOA and Petri Nets for System Specification. In *Application and Theory of Petri Nets; Lecture Notes in Computer Science*. Italy, 1995.
18. R. Kočí and V. Janoušek. System Design with Object Oriented Petri Nets Formalism. In *The Third International Conference on Software Engineering Advances Proceedings ICSEA 2008*, pages 421–426. IEEE Computer Society, 2008.
19. C. Sibertin-Blanc. Cooperative Nets. In *Proceedings of Application and Theory of Petri Nets*, volume 815. Springer-Verlag, 1994.
20. A. M. Uhrmacher. Dynamic structures in modeling and simulation: a reflective approach. In *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, volume 11, 2001.
21. R. Valk. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In *Jorg Desel, Manuel Silva (eds.): Application and Theory of Petri Nets; Lecture Notes in Computer Science*, volume 120. Springer-Verlag, 1998.
22. Y. Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 27. ACM Press, New York, 1992.
23. B. Zeigler, T. Kim, and H. Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., London, 2000.

UnitEd

A Petri Net Based Framework for Modeling Complex and Adaptive Systems

Marcin Hewelt and Matthias WesterEbbinghaus

University of Hamburg, Department of Informatics, Vogt-Kölln-Straße 30,
D–22527 Hamburg, http://www.informatik.uni-hamburg.de/TGI/

Abstract. Modeling of systems is an essential part of software engineering, especially if one considers complex, dynamic systems, which can be characterized as systems of systems. Runtime adaptation of the system needs to be considered as a first-class concept and handled in a systematic way throughout the different levels of abstraction.
The formal and intuitive expression of concurrency, the possibility to model nested systems using the nets-in-nets concept and the clearness of the models makes Petri nets a good choice to model and implement the aforementioned systems. However, their static structure does not support adaptation during simulation time directly.
In this paper we present the concept of units as composable building blocks for system modeling and give an informal sketch of Unit Theory, which states how units can be traced back to net foldings of an underlying net. Furthermore we derive an implementation of units as special high-level Petri nets through stepwise refinement of a basic model. Together with a set of basic operations these nets form UnitEd, a modeling framework which provides simulation time adaptation and successive refinements of models.
UnitEd is aimed at modeling and implementing complex systems as dynamic configuration of small building blocks – the units. Units can be recursively nested to form non-disjunct hierarchies, that are ideally suited to model systems of systems at different levels of abstraction. UnitEd is built as a plug-in on top of Renew, a simulation engine for Petri nets and related net formalisms.

1 Introduction

Software development faces an ever increasing complexity, which is reflected by characterizations of modern software systems according to e.g. [1, 2]. Such systems can be characterized as follows. They are very large, measured in lines of code, number of subsystems or people involved (as customers, developers, administrators, programmers, etc.). But one has to go beyond the concept of size alone to get a grasp of the systems' nature. They are systems of systems [3], consisting of inherently distributed and heterogeneous subsystems with potentially loose coupling. Complexity takes on a hierarchic/recursive character as it

is the case for *all* kinds of complex systems [4]: Systems consist of other systems and at the same time contribute to the constitution of even larger systems. Systems of systems often lack a central control facility. Instead, control is horizontally as well as vertically decentralized and subsystems display a great degree of operational and managerial independence. The system as a whole is deeply embedded in a real-world setting, with a dynamic and uncertain environment that comprises a large variety of real-world actors (persons, teams, enterprises, institutions), which all seek to exert their own particular attitudes and interests onto the software system.

These characteristics pose great challenges for information technology. The overhead induced by distribution, decentralization and heterogeneity has to be managed and attenuated in a way that the actual production (sub-)systems still efficiently produce value. One important requirement in this context is adaptivity. Large-scale software systems are not designed and built in a single shot. Neither can they be evaluated and redesigned in a unified manner. Instead, functionality of system parts as well as interactions between system parts have to be added, removed and modified at run time. Given the nature of the systems described above, adaption has to be possible in a modular way and take into consideration different levels of granularity/abstraction.

Information technology has encountered these challenges by developing increasingly powerful abstractions, approaches and paradigms of system engineering over the past decades. Structured analysis, object-orientation, patterns, components, services, agent-orientation all try to structure and cope with complexity. There has been great progress over the last years, and especially agent technology is to some part ideally suited to deal with the above mentioned problems as it embraces the notions of distribution, local autonomy and heterogeneity as core concepts. Organization-oriented approaches to multi-agent systems (MAS) extend the agent paradigm even further by providing system level perspectives.

Nonetheless, this does of course not mean that the problem has been settled. In our opinion, such approaches embody the beginning of a gradual transition from classical agent concepts and metaphors to higher-granular concepts. This might provide the basis for completely new engineering approaches that combines the earnings of the agent paradigm with a much more flexible handling of the actor metaphor, namely in an individual as well as a corporate sense. Consequently, ideas in this context still have to mature and there is still a great need for fresh perspectives and techniques for coping with system of system structures and the accompanying hierarchic nature of complexity. In this paper contribute to this field of research with our proposal of a basic framework for both modeling and simulating/prototyping systems of systems. Our proposal includes a modeling framework, called UnitEd (Unit Editor). Its basic idea is to take a very generic stance. We model *units* and relationships between them. We apply this very basic conception recursively. Units are composed of and constituted by units at each level. This generic approach allows us to systematically derive fundamental building principles and modes of coupling. To put it different, we

take a step back from particular object-, service- or agent-oriented views and adopt a perspective that grants a much more general and universal perspective.

We have chosen Petri nets [5] and especially the high-level Petri net formalism of reference [6] nets as our modeling technique. First of all, Petri nets feature a very restricted set that allows to build very expressive models. This perfectly suits our intention of building universal and generic models that are built up from a restricted set of base concepts. Another advantage of Petri nets is their ability to support rapid system prototyping, as was shown in [7]. UnitEd is built on top of the Renew simulation environment [8]. Renew is a simulator for Petri nets and includes supports for many different Petri net formalisms, like colored nets, timed nets and Java reference nets. UnitEd uses Renew's possibility of extension through plug-ins and is realized as a plug-in. Therefore the modeled systems can be simulated with Renew, which acts as a runtime environment for UnitEd.
This allows us to smoothly transition from abstract unit models to executable unit prototypes ("implementation through specification").

Reference nets in particular add further benefits to the modeling with "ordinary" (colored) Petri nets. They implement the nets-in-nets concept from [9]. This provides us a elegant way of modeling dynamic composition of components on multiple levels. One attribute of Petri nets, which make their direct use (at least in the first instance) problematic for the goal of this paper, is the restriction of dynamic to the flow of tokens – the net structure cannot be changed during simulation. As we described above, modeling and implementing adaptivity is of central importance given the nature of the systems that are our objects of study. It is exactly the reference net formalism that allows us address this obstacle easily. Instead of adjusting the reference nets formalism to allow for structural change of Petri nets during simulation time, UnitEd employs the nets-in-nets concept to work around this characteristic. In reference nets, tokens can be references on other nets and with the concept of synchronized channels communication between different nets can be achieved. Therefore UnitEd allows for a change of the unit system's structure during simulation time without changing the structure of the nets themselves. This way systems can be modeled, simulated and detailed in one application as part of the same process of development.

UnitEd draws inspiration from organization theory, multi agent systems approaches, as well as general systems theory. The biggest advantage is the possibility to successively refine models during simulation and thus allowing rapid prototyping of systems. UnitEd follows the approach of "implementation through specification" that is successfully used at the "Theoretical Foundation Group" at the University of Hamburg (see for example [10],[11]). By successively detailing the specification of a system one arrives at a reference net implementation of the system in question.

UnitEd conceptually builds on unit theory. Unit theory is a "thinking tool" , which conceptualizes the simple idea of systems – which are composed of interacting subsystems – in terms of Petri nets. Unit theory defines a set

of basic operations, which can be used to construct complex, hierarchical unit systems out of atomic units. This basic operations form an axiomatic system of UNIT THEORY. Currently there are eight basic operations: creation and removal, composition and decomposition, connection and disconnection, marking and unmarking.

UNIT THEORY offers principles for structuring complex systems by means of logical units and their interactions. Although other approaches also support structuring systems with some modeling primitives (consider for example the agent metaphor of multi-agent systems or the notion of components) UNITED takes a very generic stance toward structuring. The atomic units are net elements of a C/E-net and more complex units are made up of them through applying the basic operations of UNIT THEORY.
In section 3 we propose an implementation of this logical units as special reference nets – unit nets. This nets are designed to exhibit the properties of logical units corresponding to UNIT THEORY, i.e. dynamic adaptivity, composition and interaction.

UNIT THEORY is a relatively new concept and still work in progress. The first mention of it can be found in [12], another work to refer to is the Thesis of Volker Tell [13]. The framework presented in this paper is part of ongoing work of the authors. We seek to clarify some of the ambiguities of UNIT THEORY and make a proposition for an operational semantic. Thus we supply a technical understanding of the rather abstract concepts of original UNIT THEORY. Due to the space limitations of this paper, only central aspects of this work can be presented. Therefore we will focus on the implementation of units as unit nets. The remaining of the paper is organized as follows. Section 2 presents the UNIT THEORY, which laid the foundations of UNITED. The main part of this paper – section 3 – features a derivation of the unit nets by means of example. They are a proposition for a reference net implementation of the concepts of UNIT THEORY. In the remaining Section 4 the author summarizes and discuses the paper and gives an outlook on what is left to do.

2 UNIT THEORY

The framework proposed in this paper is based on UNIT THEORY, which will be described in this section. The term "Ansatz" as defined in [12] means a unified approach toward system specification and implementation, consisting of a frame with paradigms, principles and concepts and offering techniques, methods, tools and applications. UNIT THEORY is an "Ansatz" in the broad meaning sketched above.
Until now, there are only very few publications on the subject, see the master thesis of Volker Tell [13] for an elaborated coverage of UNIT THEORY so far. There is also a good report in German from Daniel Moldt, which summarizes the basic ideas behind UNIT THEORY [12]. First we will give an informal outlook at concepts and ideas of UNIT THEORY.

2.1 Informal Introduction to Unit Theory

First of all, unit theory is a "thinking tool" that allows structuring a complex system (e.g. the outside world) under the viewpoint of nested, interacting units. The part of the world that is to be modeled is understood as an infinite branching process net as defined in [14]. This branching process net is called the *underlying executing system (GAS)*[1] and encompasses all possible runs of the considered part of the world. This underlying system is of arbitrary granularity but stays discrete. Because the GAS is very unhandy, it is natural to employ net abstractions to make it more suitable.
An example should make this clearer. Consider a globally operating corporation with dozens of divisions, thousands of employees and millions of interactions now and in the future. Such a corporation is a good example for a (globally) distributed, decentralized, enormous system of systems, which is hard to model. While the corporation's behavior can be comprehended in terms of corresponding branching processes of the GAS, the corporation's structure (i.e. divisions, subsidiaries, departments etc.) is not easily to be identified in the net. Unit Theory proposes to fold appropriate subsets of elements of the branching process net in order to receive a more abstract representation. One would identify units like "the factory in Leipzig", "the marketing department for Europe" or "build a new factory", thus overlaying the GAS with a logical superstructure. The result of such a structuring would be a unit theoretical model consisting of interacting units.

Unit theory is very generic, the concept of a unit applies to a broad spectrum of entities like a molecule, a robot, an organization or the solar system. Not only entities can be comprehended as units, but also processes, transactions, the instance of a running Java application inside a JVM, or even abstract things like e.g. the thought I had last night. However, it is important to note that all units have a representation in the underlying branching process net, thus abstracting away some details of the GAS.
A more formal explanation of units and systems of units is given in the next subsection.

2.2 Units and Unit Systems

At the heart of the unit theory lies the idea that all systems can be described as interacting units, which have their foundations in the GAS. Units are thought of as a subset of all processes that can take place in the GAS. They are conceptualized as net foldings of the GAS. Usually only a small subset of connected net elements will be included in the folding operation that yields a unit. In this way, units abstract from the structure of connected net elements and form identifiable entities. Hence, a unit is at its core a labeled entity – something that can be named and pointed at.

[1] In German it is called *Grundlegendes Ausführendes System*, hence the acronym.

Complex units can be decomposed into subunits, all the way down to the net elements of the GAS – conditions and events, which are the atomic units. This yields the possibility to conceive systems at different levels of abstraction. Returning to the example, we can talk about a corporation at the level of departments and interactions between them, but we can also look in more detail and observe their internals, like bureaus, employees and procedures. In the same way a unit like "build a car" can be divided into the subtasks that are necessary to build a car. In this way units are embedded in other units and form non-disjunct hierarchies, that will be introduced now as *unit systems*. By non-disjunct we mean that a unit may be embedded in more than one other unit, for example a person might be member of two different organizations.

Unit Systems We do not want to talk only about individual units. Instead we want to be able to compose units into a *unit system*. A unit system is conceptualized as a set of units with an embedment relation defined on it.[2] With units being net morphisms of the GAS, embedding of one unit inside another translates into the inverse image of the subunit (which is a set of elements of the GAS) being a subset of the inverse image of the super-unit.
Another relation establishes a connection between individual units and is similar to the arcs of a Petri net. Therefore, a unit system can be considered as a labeled graph, with two relations (embedment and connection) defined on the nodes, that is, the units. Firing activity in the units thus equals transformations of the unit graph.
We want to take a more constructive stance toward unit systems. The creation, connection, marking and composition (and of course the inverse operations) of individual units should be supported by the framework. This is achieved by delivering a set of basic operations to the modeler, that are defined by UNIT THEORY. Those operations can be viewed as interactions between the modeler and UNITED, but they can also be part of complex units that are able to reshape themselves.

Adaptation We now turn to the property of runtime adaption of systems. In the GAS all future runs of the model original are encompassed. This includes runs that represent a restructuring or even vanishing of the system.
In the corporation example the board of directors can decide to relocate a factory. Thus, this possibility is already a branch of the GAS which is entered upon the decision, i.e. upon the occurrence of some event. On an abstract level one event in the GAS can abstract the whole process of negotiating, closing the factory and building and equipping a new one. The view of UNIT THEORY toward adaptation is that new units can enter the stage and others can leave, when the preconditions are met. During simulation conflicts in the GAS are resolved, because decisions are made – of the possibly infinite number of enabled transitions, one is chosen. Hence while the simulation advances, the decisions what

[2] Whether the marking of a unit can be viewed as a special case of this embedment relation or defines an additional marking relation, is a point that needs further discussion.

transitions occur generate a process of the underlying net system.
We propose the following operationalization of this ideas: Units can be created by adding a new net abstraction and can be composed via the composition of net abstractions. This composition works as follows: An additional net abstraction is added, whose range is the combined range of the two units to be composed. The definition of connection is not so straightforward, but the idea is to identify a process of the GAS that lead from one unit to the other.
While the formal foundation of UNIT THEORY is ongoing work, it is still possible to formulate an operationalization of its basic operations. UNITED proposes such an operationalization of UNIT THEORY and backs it up with tool support.

***Modeling in* UNIT THEORY** The task of modeling is to find a representation of a section of the world using the modeling primitives a modeling language offers. Now units are exactly the modeling primitives UNIT THEORY has to offer. In the eyes of UNIT THEORY modeling means to choose the appropriate net foldings and therewith structure the section of the world that is modeled. Since the concept of units is very generic, the task of a modeler employing UNIT THEORY can be viewed as the choice of appropriate units for a certain purpose. To achieve this, he has to create, connect, structure and specialize the units of the model. UNITED offers the tool support the modeler needs to fulfill his task and therefore is a framework for modeling in an unit theoretical manner.

Multiple Views on a System Systems of systems are hard to comprehend. For this reason models of such systems need the possibility of taking different views toward them. As an example imagine an inter-organizational application. On an abstract level organizations can be viewed as units, thus omitting their internals. But because the modeler needs the possibility to alter the internals, the framework must offer a view of the internals and how they relate to peripheral interactions. This causes no problem for UNIT THEORY, as a unit – the organization – is composed of subunits – its departments – which can be focused themselves. Above some examples were given for this shif tof perspective. Hence UNIT THEORY offers the possibility to navigate between different levels of abstraction. To be beneficial for the modeler the concept of embedment must be supported by a tool, which can display the different levels. UNITED is such a tool.

2.3 Axiom System of UNIT THEORY

A set of eight basic operations forms an axiomatic system of UNIT THEORY. All units can be built by applying the axioms. This basic operations are supported by the UNITED framework, so the modeler can use them to build models of unit systems and change them while the simulation is running. The basic operations are divided into four pairs of inverse operations. *Creation* adds a new atomic unit to the model, while *removal* removes a possibly complex unit from the model. Since units can be embedded in more than one unit, the removal operation needs to assert, that all occurrences are removed.

Composition creates a new unit, that is composed of two units that already where part of the model. Those units are preserved. *Decomposition* removes a unit and preserves its subunits.
Connection allows to introduce an interaction between units, while *Disconnection* removes such an interaction.
Marking puts a unit as a subunit on another unit, the identity of the embedded unit is preserved. *Unmarking* is the opposite operations, it removes a subunit out of the embedding unit, but does not delete it.

Those are a the only operations. They are only informally based on net operations on the GAS so far, but a formal foundation of the basic operations is soon to come as part of further work by the authors. Nonetheless, even without formal foundation an operationalization of the basic operations is given by UnitEd and can be applied by the modeler to model unit systems.

3 Unit Nets

This section presents the central concepts of the unit editor tool by means of examples. This is accomplished by consecutive refinements of a very simple Petri net model, which will lead to the proposition of unit nets as fixed building blocks of unit systems.
Unit nets are a implementation of the logical concept of units (which was introduced in section 2) using the Java reference nets formalism, which is supported by Renew for modeling and simulation purposes. The choice of the implementation method was made for particular reasons. Petri nets in general allow to build complex and expressive models based on a restricted and highly universal set of modeling elements. Reference nets especially allow for modeling hierarchies in an elegant way, utilizing the nets-in-nets concept. They a have good tool support by Renew and allow for direct simulation of net models. An introduction to reference nets is given in the next subsection, which shows the main features and differences of this net formalism compared to others. It is expected, that the reader is familiar with "ordinary" colored Petri nets.

3.1 Reference Nets

Reference nets implement the *nets within nets* concept introduced in [9] where a surrounding net - the *system net* - can have nets as tokens - the *object nets*. As hierarchies of net within net relationships are allowed, the denomination of system or object net depends on the beholder's viewpoint. Reference semantics is applied, thus net tokens are *references* to *net instances*. According to objects being instances of classes in object-oriented programming languages, net instances are instantiated copies of template nets. All instances are independent from each other. To facilitate communication between net instances, *synchronous channels* as introduced in [15] permit a fusion of two transitions at a time for the duration of one firing occurrence. In reference nets, a channel is identified by its name and its arguments. Channels are directed, exactly one of the two transitions

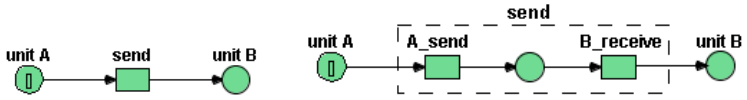
(the one with a *downlink*) indicates the net instance in which the counterpart of the channel (a transition with the corresponding *uplink*) is located. However, information flow between the fused transitions is bi-directional.

As for most other net formalisms there exist tools for the simulation of reference nets. The Renew tool [8] additionally allows for (mostly) arbitrary JAVA inscriptions to transitions. This allows for a powerful approach of "implementation through specification". In order to transform an abstract net model into a specific implementation it is in most cases sufficient to add combined channel/JAVA inscriptions to transitions and to add certain auxiliary net elements.

3.2 Example 1: Message passing

The scenario, which is considered in this subsection is very simple. It consists of only two entities, one of whom sends a message to the other. This is somewhat trivial but can easily be extended to encompass distributed agents communicating over a channel with diverse characteristics.

Basic Model This scenario can be modeled at an abstract level as a Petri net with two places and a transition connecting them, like in Fig. 1(a). The message, which is send will be modeled simply as a black token for a start. Now the



(a) Basic Model (b) Refinement

Fig. 1. Basic Model and first Refinement

behavior of the model will be considered. This can be achieved by looking at its Petri net processes as defined in [16]. The only possible process is isomorphic to the net itself. This will no longer be the case anymore after the model has been refined.

The basic model from Fig. 1(a) is a well-formed Petri net and can be simulated with Renew. Tokens are dynamic during simulation in contrast to changes of the net structure, which can only take place during modeling time. This is a drawback if one considers the net not only as a model, but as an implementation of a system. If such a system is for example critical to some production procedure, it is not convenient to just stop the system, alter it and start it up again, even if it is possible to save the current state of the system and restore it afterward. Supporting structural changes during runtime is a central goal of UnitEd. Therefore, the basic model will now be refined stepwise into a UnitEd model. We state the requirement that this refinements should not alter the behavioral semantics (in terms of Petri net processes).

Refinement: Asynchronism of send In the abstract model, sending and receiving of the message happens in one net step. Now the transition send will be replaced with the two transitions `A_send` and `B_receive` and one place. This net refinement is shown in Fig. 1(b). In the resulting model, *unit A* and *unit B* no longer interact directly. Instead, another unit – modeled by the place in between – mediates. After the coupling of the units has been loosened, parts of the net can be replaced more easily.
The transitions `A_send` and `B_receive` model the behavior of *unit A*, *unit B* respectively. At the same time they stay part of the refinement of the transition send – of the overarching interaction. An intuitive interpretation is obvious: The interaction consists of two interaction shares, one for *unit A* and one for *unit B*. Their shares can in turn be seen as internal interactions in which their (possible) respective subunits are involved.
Accompanying this change, it is worth noting that another change in the ontology occurred: The elements of the model are now called units. This implies all the conceptualization that was introduced in section 2 (like identity, autonomy and embedding, etc.).

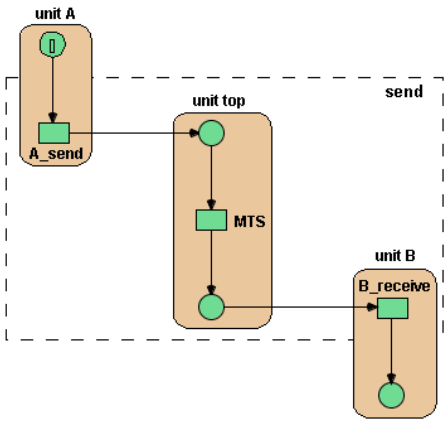


Fig. 2. Refinement of place *unit top* and graphical rearrangement

Graphical Rearrangement The graphical arrangement of the elements of a Petri net does not change the net's semantics as long as the place-transition relationships persist. But an appropriate graphical representation can add a lot of clearness and insight to the model and makes it easier to understand. Therefore, the Petri net from Fig. 1(b) will now be rearranged. The result is shown in Fig. 2. Now the interaction shares are drawn beneath the corresponding units. The three highlighted areas of the net are those, which will be modeled as units in the next step.

Notice also, that the place in between – now called *unit top* – has been subject to refinement and now encompasses the transition `MTS`, which stand for "message transport system".

Refinement: Embedding The simplicity of our sample scenario is ideally suited for illustrating the mechanics behind our next refinement step. Until now our model consists of a flat Petri net. This will now be replaced with a hierarchy of Petri nets. The hierarchy of units is defined by the embedding relation introduced in section 2. *Unit A* and *unit B* will be considered as subunits of *unit top*, as it is shown in Fig. 2. References of the net instances of *unit A* and *unit B* lie as tokens on the place `subunits` of *unit top*. A problem with this net system is

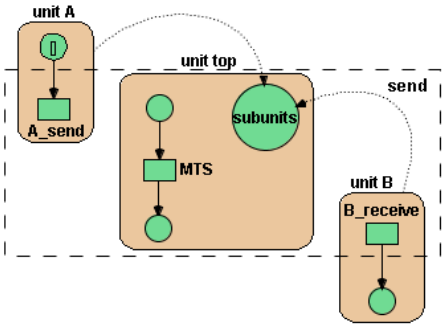
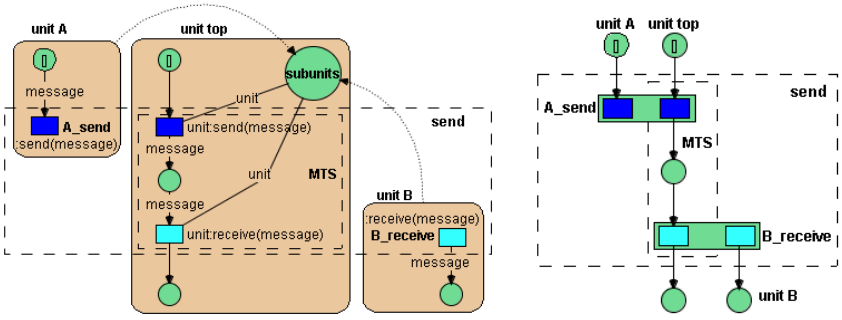


Fig. 3. *Unit A* and *unit B* embedded in *unit top* represented by dotted arrow

the fact that no communication between the units is possible, because they are different nets. The Petri net processes are different from our basic model (only `A_send` can fire). This will be fixed in the next paragraph with the supplement of synchronized channels.

Refinement: Synchronized Channels In UnitEd, different units of a unit system are actually different nets. Each conceptual unit in the unit system is associated with a corresponding unit net, i.e. the net that is derived stepwise in this section. Therefore, no direct arcs can be drawn between net elements in different nets. But to preserve the behavior of the basic model, some sort of communication between the units is needed. To obtain this communication mechanism, we employ the concept of synchronous channels of reference nets.
Two transitions can synchronize and transmit tokens, if they are associated via a pair of uplink and downlink (the downlink also includes the reference to the net where the uplink is located). As *unit top* has references on both *unit A* and *unit B*, a transition of *unit top* should have the downlink and hence the units A and B corresponding uplinks.
The situation is made more clear through Fig. 4(a). Here, two synchronous

channels exist: the send- and the receive-channel. The message transport system (transition `MTS` in the figure) is refined, so that it receives a message from *unit A* over channel send, buffers it in the place in the middle, and then sends it further to *unit B* over the channel receive.[3] The downlink in *unit top* (unit:send(message)) is connected with the place `subunits` over the test arc[4] inscribed with unit. So the variable unit is bound to the token on `subunits` which is tested. Because the tokens are references of net instances, and because the preconditions of both uplink and downlink are satisfied the transitions can fire synchronously. The binding of the two variables message (variables are local to transitions) is unified, so that the message gets stored in the middle place of *unit top*.
The synchronization of the receive-channel happens in an analogous manner.



(a) Embedding with synchronized channels (b) Maximal process of net 4(a)

Fig. 4. Unit system with use of synchronized channels and one of its processes

Processes of the Refined Model Lets now take a look at the processes of the refined model from Fig. 4(a) to determine if the behavior changed compared to the basic model. Therefore, we need to observe processes of a set of nets, because the unit system – our model – consists of more than one net. This raises two questions: How to extend the definition of Petri net processes to sets of reference nets, and how to cope with synchronized channels.
Due to space restrictions only a informal sketch of a solution to this problems can be presented here. The definition of Petri net processes from [16] must be adapted in order to deal with sets of nets. Therefore, the range of the function that maps elements of a causal net onto the Petri net, whose process is considered, needs to

[3] The naming of the channels includes a perspective. Unit A "sends" a message, *unit top* "receives" this message synchronized over the channel send.

[4] The test arc is a special construct of the reference net formalism. It allows to test for a token without removing it from its place. Hence, it is possible for more than one transitions to test the same token. In an extended scenario with more interactions, this means, that several interactions of the same unit can be initiated independently.

be extended in a way that it equals the union of the net elements of all involved nets.
Now we turn to the second question. Since transitions that are connected via a synchronized channel fire synchronously they have only one corresponding preimage in the causal net of the process. The pre-set of such a transition in the causal net consists of all pre-sets of the synchronized transitions. Analogously, the post-set consists of the post-sets of all involved transitions. An algorithm for constructing processes of sets of reference nets is to be found in the doctoral thesis of Christine Reese [17].
Additionally, we do not want to consider all processes of a unit system, but only the "maximal" ones, i.e. those which correspond to a state of the system, where is have no activated transitions.[5] Therefore we need to assert, that our model has only finite processes. This is easily shown, after transition `send` fired the net is dead. The transition with the blue boxes inside symbolizes the synchronization that takes place.

Refinement: Interactions as Subunits In our scenario there is only one interaction, the sending of a message. Generally, it will be the case that a unit system will have a variety of interactions, some of which will be activated, others will be running and still others will be inactive. Drawing them all in one unit net, like we have done it so far, would result in confusion for the modeler and a massive amount of arcs, if some interactions share resources (and therefore cannot take place concurrently). On the other hand, we would like the modeling tool to allow to add, remove and change interactions. Units already have these properties, so it's a natural approach to consider interactions as special subunits. Since interactions are conceptually different from the actor-like units we have considered so far, they will be called interaction units. Until now a synchronization took place between a transition in one unit and a transition in an embedding unit. This refinement extends the need of synchronization to three layers. Now a interaction unit (*unit interaction_ send*) needs to synchronize with its embedding unit (*unit A*), which then synchronizes with its embedding unit (*unit top*), which finally synchronizes with one of its interaction subunits that is responsible for this interaction (*unit interaction_ send/receive*). In this way, four transitions in four different nets fire synchronously. Figure 5 clarifies this situation. The firing of all blue, respectively all turquoise transitions in the figure is synchronized. The original interaction send in the basic model is shown inside the dashed rectangle. All net elements inside it could be folded to receive the send transition from Fig. 1(a).

Refinement: Interaction Channels In the case of many interactions the naming of the synchronized channels needs to be addressed. So far we used different synchronized channels. For example, *unit top* possesses the two channels

[5] Later not only maximal processes need to be considered, but also "multiples" of interactions, i.e. those processes that show the state after a interaction has been finished and not started again – after a complete "interaction cycle". In our scenarios, with the interaction being non-cyclic those two subsets of processes are equal.

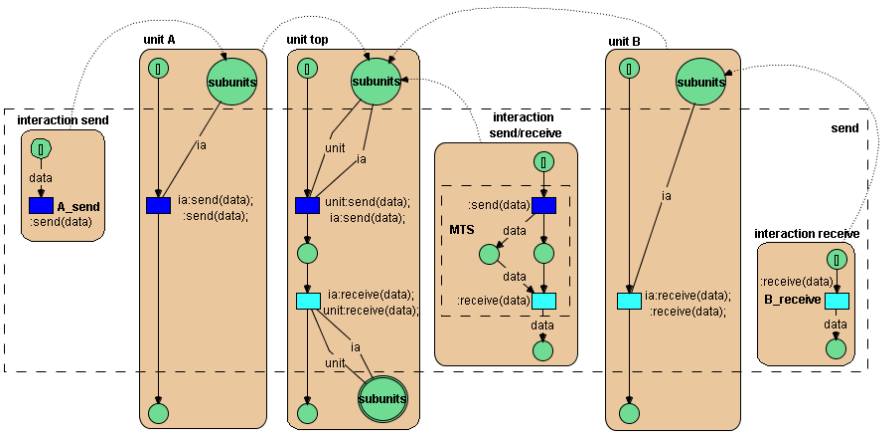


Fig. 5. Interactions are embedded as subunits

send and receive, while *unit A* only has the channel send. This approach will lead to problems, if we imagine a lot of interactions, all of which will need differently named channels. Also the interface of a unit would depend on the interactions in which it is involved. For each of this interactions, the unit net would require to have a transition with the synchronized channel inscription. Therefore we introduce the concept of *interaction channels* in this paragraph.

Instead of using different synchronous channels we take the more general approach of using only one channel, which is parameterized with a string indicating the name of the interaction channel. The generic synchronous channel that realizes the interaction channels is called *interact-channel*. An *interact-channel* realizes different interaction channels. With the introduction of interaction channels, an important feature of unit nets is achieved: the unity of their interface.

There is another point to interaction channels, namely the channel mapping mechanism. This mechanism is responsible for mapping an interaction channel to an associated interaction unit and vice versa. To achieve this mapping unit nets have a place with tuples of the form `[chname, IA ref.]`, chname being a string and IA ref. a reference to an interaction unit.

In Fig. 6 the place for channel mappings of *unit A* is marked with the tuple `[''send'', interaction_send]`. This means, if the *interact-uplink* in *unit interaction_ send* is activated (as is the case) it tries to synchronize with the *interact-downlink* in *unit A*, which require a channel mapping. The variable ia is bound to (a reference to) *unit interaction_ send* and the unification can only take place, if there is a tuple on the channel mapping place of *unit A*, whose second component is identical to *unit interaction_ send*. In this way, the associated chname (bound to "send") is retrieved.

This retrieved string is a parameter to the *interact-uplink* of *unit A*. So "send" together with the message is pushed onto the next-higher level, which in our model is *unit top*, and mapped to *unit interaction_ send/receive*, which finally

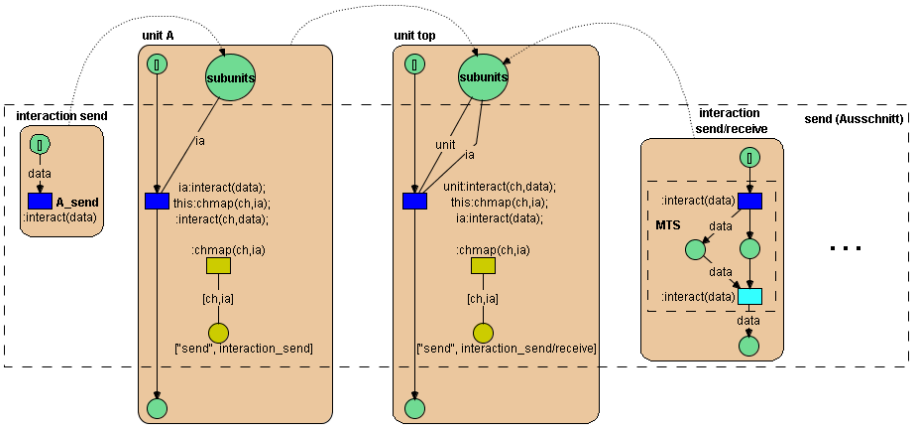


Fig. 6. Mapping of channel names to interactions. Only a part of the interaction is shown

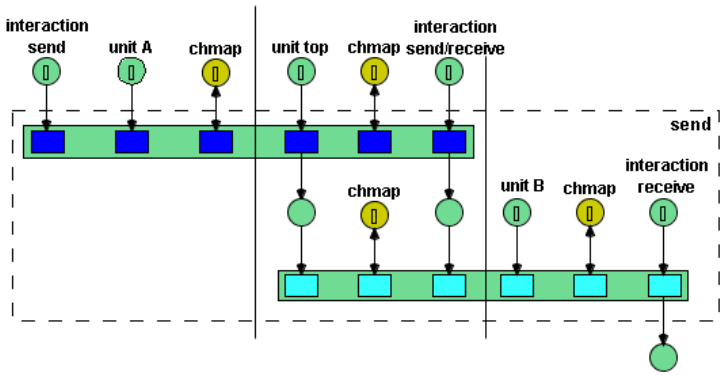


Fig. 7. Maximal process of net 6

gets the message send out by *unit interaction_send*. So far we have described the synchronized firing of the blue transitions, and the golden chmap-transitions. All in all there are now 6 transitions, which need to synchronize, before a firing can occur.
As last part of this paragraph we need to check if and how the Petri net processes of the resulting model changed. The maximal process is shown in Fig. 6. Its causal net consists of two transitions (the long greenish rectangles) and many places. Notice, that the golden places are connected with a reserve arc. The process in 6 is equivalent to the one in 1(a), if fitting channel mapping tuples lie as tokens on the channel mappings places of the involved units.

Refinement: Adding and Removing Subunits So far, the path we followed has led to a model that is several times more complex than the initial model. Instead of one transition firing, now there are twelve, some of which require synchronization. Instead of one net, there are six. But still the resulting model is a behavior-preserving refinement of the original model, like it was shown in 6.
The concept of synchronized channels allowed us to split the nets. With the nets-in-nets concept we were able to embed the nets inside each other. We singled out interactions and realized them as interaction units. The interaction channels supply the design of a uniform interface for units. Now finally we are in the position to cope with changes of the unit system during simulation time. Therefore the unit nets need to be extended with two additional pairs of transitions, which allow the addition and removal of subunits, interaction channels respectively. This extension is shown in 8. The UnitEd framework offers the matching of

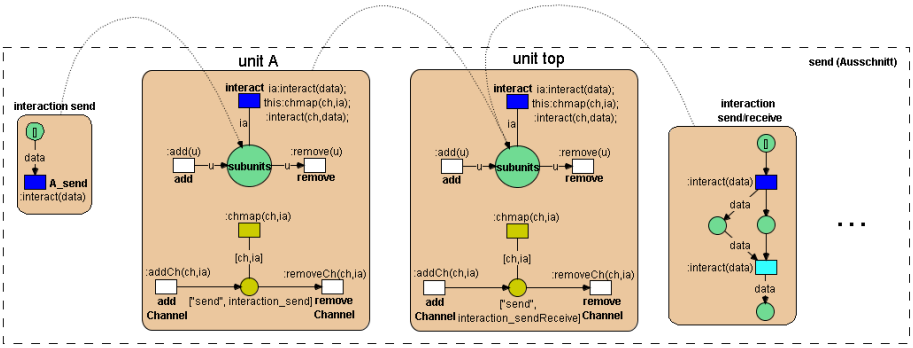


Fig. 8. Extension of units with management transitions that add and remove subunits and interaction channels.

downlinks to uplinks in the unit nets and enables the modeler to add units and interactions during runtime. Also, the modeler can equip some of the units of his unit system with the interaction units for the basic operations of UNIT THEORY, thus enabling the modeled system to adapt itself! So as part of an interaction a

unit may remove some of its parts, or change internal interactions.
As the final point for this paragraph we will once again check the Petri net processes of the final UnitEd model. The causal net for the maximal process is shown in 9.

Discussion In this section a universal unit net model has been derived by means of a simple toy example. This unit net is our basic building block for modeling unit systems and constitutes a fixed component of UnitEd. Its net structure cannot be changed during simulation time, but whole systems of unit nets allow for dynamic composition and adaption through changes of their respective markings. Markings represent other units and interaction channels. So instead of changing the simulation algorithms of the reference net formalism, runtime adaptation is built on top of reference nets.

The overhead that was introduced by our successive refinements is of course inappropriate for the trivial example that we considered here. The example just served its purpose to exemplify our refinement steps in an accessible way. But we claim that the overhead actually does pay off in the context of more complex systems where the distinction of different levels of system abstraction actually becomes a necessity. For these cases, we have provided universal building principles and technical mechanisms for realizing vertical and horizontal coupling in terms of flexible interactions.

The complexity of the resulting unit nets may be staggering not only at first glance. In fact the nets we use for our prototype are even more complex. But this complexity is needed to encompass the possibility of dynamic change, composition and interaction without inventing a new net formalism. A new net formalism would have been an alternative to the proposed approach of implementing unit nets on top of Renew. Because unit nets as presented in this section are not well fitted for direct use by the modeler, an editor that hides the complexity and represents units in a more accessible way to the modeler is also part of the UnitEd framework. This editor however, is out of the scope of this paper.

4 Conclusion

In this paper we have presented our modeling framework UnitEd. It targets at modeling systems that exhibit a systems of systems character. This does not only entail potential loose coupling between system parts but also the necessity to respect different levels of granularity/abstraction when regarding system parts. Consequently, we have laid specific emphasis on modeling vertical as well as horizontal coupling between system parts in an integrated manner. At the heart of our approach lies the modular comprehension of arbitrary complex systems in terms of *units*. We have presented the underlying principles of unit theory. UnitEd builds upon this theory and in particular proposes an operationalization. Each system part is modeled as a unit that is composed of other units and contributes itself to the composition of even higher-granular units. UnitEd models are not just abstract illustrations but executable prototypes. Consequently, they provide a precise technical understanding of the unit theoretical concepts.

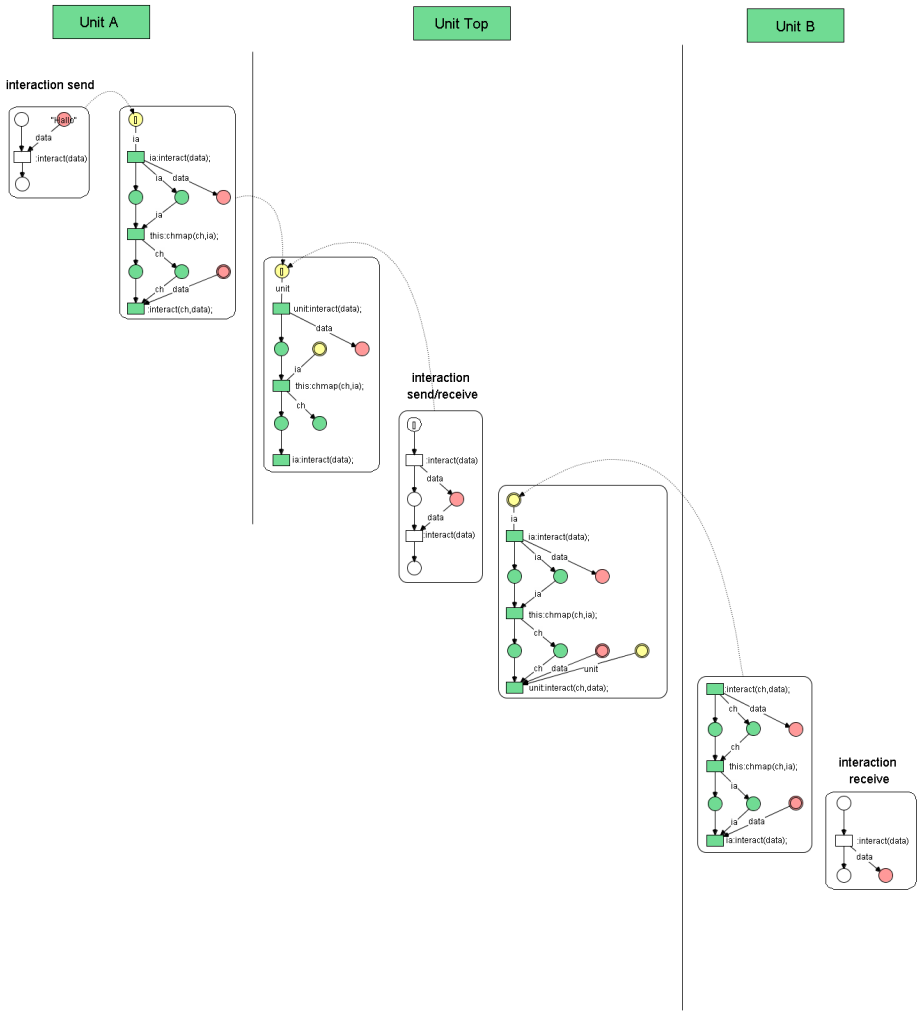


Fig. 9. Maximal process of unit system in Fig. 8. Places for subunits are yellow, data places red. Interaction units are drawn with white net elements, units with green. Dotted Arrows show embedment of units. The channel mapping mechanism is not shown.

We have chosen Petri nets in general and reference nets specifically as our modeling means. Petri nets offer a restricted set of modeling elements from which expressive and complex models can be build. In addition, Petri nets offer a very close link between structure and dynamics of a system. Finally, reference nets offer mechanisms for dynamic composition of (net) components and allow us to circumvent the restriction of static Petri net structures. Thus, we are able to include run-time adaption into our system models.

The work presented here is part of ongoing work at our group. For future work, one important issue is to make our modeling framework UnitEd more accessible, more easy to use. The excessive use of net references and synchronous channels in some way prohibit an intuitive usage of UnitEd. However, they build a lean and effective foundation for an operationalization. We plan to enrich UnitEd with a graphical Editor that relieves the modeler of a great part of the technical overhead that is involved in the specific mechanisms.

References

1. Northrop, L., Feiler, P., Gabriel, R.P., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidtd, D., Sullivan, K., Wallnau, K.: Ultra-large-scale systems: The software challenge of the future. Technical report, Sofwtare Engineering Institute, Carnegie-Mellon (2006)
2. Hess, A., Humm, B., Voss, M., Engels, G.: Structuring software cities a multi-dimensional approach. In: Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International. (2007) 122
3. Maier, M.: Architecturing principles for systems-of-systems. Systems Engineering **1**(4) (1999) 267–284
4. Simon, H.A.: Administrative Behavior. 4. edn. Free Press (March 1976)
5. Girault, C., Valk, R.: Petri nets for systems engineering: a guide to modelling, verification and applications. Springer Verlag (2003)
6. Kummer, O.: Referenznetze. Logos Verlag, Berlin (2002)
7. Moldt, D.: Höhere Petrinetze als Grundlage für Systemspezifikationen. Dissertation, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg (August 1996)
8. Kummer, O., Wienberg, F., Duvigneau, M.: Renew – User Guide. University of Hamburg, Faculty of Informatics, Theoretical Foundations Group, Hamburg. Release 2.1 edn. (May 2006) Available at: http://www.renew.de/.
9. Valk, R.: Petri nets as token objects - an introduction to elementary object nets. In Desel, J., Silva, M., eds.: 19th International Conference on Application and Theory of Petri nets, Lisbon, Portugal. Number 1420 in LNCS, Berlin Heidelberg New York, Springer-Verlag (1998) 1–25
10. Cabac, L., Duvigneau, M., Köhler, M., Lehmann, K., Moldt, D., Offermann, S., Ortmann, J., Reese, C., Rölke, H., Tell, V.: PAOSE Settler demo. In: First Workshop on High-Level Petri Nets and Distributed Systems (PNDS) 2005, Vogt-Kölln Str. 30, D-22527 Hamburg, University of Hamburg, Department of Computer Science (March 2005)
11. Cabac, L., Dörges, T., Duvigneau, M., Reese, C., Wester-Ebbinghaus, M.: Application development with Mulan. In Moldt, D., Kordon, F., van Hee, K., Colom, J.M., Bastide, R., eds.: Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'07). (2007) 145–159

12. Moldt, D.: Petrinetze als Denkzeug. In Farwer, B., Moldt, D., eds.: Report FBI-HH-B-265/05: Object Petri Nets, Process, and Object Calculi, Vogt-Kölln Str. 30, D-22527 Hamburg, University of Hamburg, Department of Computer Science (August 2005) 51–70
13. Tell, V.: Schaffung der Grundlagen für die prototypische Umsetzung eines Multiagentensystem basierten Leitmodells. masters thesis, University of Hamburg, Department of Informatics (March 2005)
14. Rozenberg, G., Engelfriet, J.: Elementary net systems. In Reisig, W., Rozenberg, G., eds.: Lectures on Petri Nets I: Basic Models, Springer (1998) 12—121
15. Christensen, S., Hansen, N.D.: Coloured petri nets extended with channels for synchronous communication. In: Proceedings of the 15th International Conference on Application and Theory of Petri Nets, London, UK, Springer-Verlag (1994) 159–178
16. Reisig, W.: Petri nets: an introduction. Springer-Verlag New York, Inc., New York, NY, USA (1985)
17. Reese, C.: Prozess-Infrastruktur für Agentenanwendungen. Dissertation, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, Germany (2009) unpublished.

A Colored Petri Nets Model for the Diagnosis of semantic faults of BPEL Services

Yingmin LI, Tarek MELLITI, and Philippe DAGUE

LRI, Univ. Paris-Sud, CNRS, and INRIA Saclay-Île de France
Parc Club Orsay Université, 4 rue Jacques Monod, bât G, Orsay, F-91893, France
IBISC, Univ. d'Evry Val d'Essonne, CNRS,
Tour Evry 2, 523 place des Terrasses de l'Agora, Evry, F-91025, France
Yingmin.Li@lri.fr Tarek.Melliti@ibisc.fr Philippe.Dague@lri.fr

Abstract. The paper contributes to modeling an orchestrated complex Web Service (BPEL) with Colored Petri Nets (CPNs) for diagnosis. In the CPNs model, colored tokens are used to represent the faults (either in inputs data or external faulty Web services) in a BPEL process. Three I/O data dependency relations are introduced into the color functions of the CPN models. Thus the transmitting of faults between the process control and internal data, data and data, in a BPEL service can be modeled as the execution of a CPN system. We give a concrete translation from a BPEL service to a CPNs model and define a model-based diagnosis framework. Based on the marking evolution equation in Petri nets theory, we construct an inequations system as a diagnosis problem and solve it with an algebra algorithm. [1]

Key words: Model-based diagnosis, Web service, BPEL, Colored Petri Nets

1 Introduction

Self-healing software is one of the important challenges for Information Society Technologies research. Our paper proposes a centralized diagnosis approach for BPEL ([11]) services, whose goal is to design a framework for self-healing Web services by adopting artificial intelligence methodologies to solve the diagnosis problem by supporting online detection and identification of faults.

A Web service (WS) is a set of distributed message oriented interacting components. We can construct complex WS systems by composing basic WSs in two ways: orchestration and choreography (P2P). An orchestrated BPEL service is a central process to organize (basic or complex) WSs to finish complex tasks. A choreographed WS has not a central process while all the involved WSs are aware of their partners but none has the global view of the whole WS application.

[1] This research has been initiated in the framework of the FP6 IST project 516933, WSDIAMOND (Web Services DIAgnosability, MONitoring and Diagnosis) and continued in the framework of the national ANR project WEBMOV (Web services Modeling and Verification).

Distributed WS applications make B2B engineering more convenient but raise more challenges for handling dysfunctions. For example, how to locate the source and reason of faults when they occur somewhere in a distributed WS application? As orchestration is the basic of the WS composition, we focus on single BPEL service diagnosis based on CPN ([7]) model which can be easily extended to a distributed environment.

During the interaction of distributed WS components, subtle faults can come from corrupted data or some functional errors. Due to the message oriented nature of WS applications, faulty data is propagated through the execution trace and is used to elaborate other faulty data and control decisions. In this way the subtle faults become large ones.

Consider the following example which will be used as an illustration example along this paper: a BPEL service *FlightAgent* calculates a series of business flight costs. The *FlightAgent* starts with a receive activity C to receive a request string of the series of departure cities and dates, for example, from Paris to London on 01/03/2008, and from London to Madrid on 03/03/2008, from Madrid to Rome on 05/03/2008, and from Rome to Paris on 09/03/2008, all the dates are in French format. *FlightAgent* iteratively (by using While activity W) invokes an invoke activity S to split the request string to get the information for one flight: the departure city, a departure date, and an arriving city (which is also the departure city of next flight). Whereafter an invoke activity O reserves the flight tickets and cumulatively calculates the flight fees.

We consider two types of faults: the faulty input data and the bad basic WS which sometimes cannot be tested or detected immediately. For example, bad activity S interprets the date format incorrectly as to different date formats in English and French, 01/03/2008, 03/03/2008, 05/03/2008, and 09/03/2008 are misinterpreted as January 03, 2008, March 03, 2008, May 03, 2008, and September 03, 2008. which is hard to test locally. As to the prizing rules of the airline, at the end of the process, reply activity P returns the total ticket price of the whole trip (in figure 3a) which is unreasonably huge. The client (or other Web service which invokes *FlightAgent*) can arise an exception. These two types of faults both reflect on data within the BPEL process. So we consider both of them as data fault while the latter one is explained as the basic WS fault. Note that we suppose the overall orchestration and choreography schema is correctly designed.

In this paper, we address the data faults by using the model based diagnosis approach, and more specially, the discrete event model based approach ([1], [2], [3], [4], [10], [15], [16], [18]). Among the usual discrete event model, we use colored Petri nets to define the diagnoser. Many works use the Petri nets to do diagnosis ([2], [10], [12], [15], [16]). Some works use high level Petri nets to modeling WS ([4], [17], [18], [3]) as they are more expressive from the aspect of data evolution. While there are few CPN model focus on diagnosis problem.

The main originality of this work is a natural use of the colored Petri nets (supported by CPN Tools [6]); the color domain is used to model data (states) status and transitions are used to define transition status. The model presented

here can be generalized to a very large software domain besides Web services. Another originality is the diagnosis methods: unlike most of other works based on Petri nets, we don't use an unfolding approach ([2], [12], [15], [16]), but use the incidence matrix and the characteristic vector of the observed trace in order to transform the diagnosis problem to an inequations system, and then we propose an algorithm to solve one inequation and then the inequations system.

The paper is organized as follows: in section 2, we introduce CPN model for the BPEL services and define their firing rules. We define CPN model for typical basic activities and structural operators of BPEL in section 3; in section 4, we define the diagnosis problem and its solution and illustrate it with a concrete example; in section 5, we introduce some related work, compare the different methods, and give some directions for future research.

2 Colored Petri Net

A Petri net is a Colored Petri Net if its tokens can be distinguished by colors. Here we restrict the definition of Colored Petri Net that we use in this paper.

Let E be a set, a multiset on E is an application m from E to $\mathbb{Z}$ (a multiset is denoted as $m = q_0e_0 + ... + q_ne_n$ where $q_i = m(e_i)$, $\mathbb{Z}$ is the integer set). We use $\mathcal{M}(E)$ to define the set of finite multisets from E to $\mathbb{Z}$, and $\mathcal{M}^+(E)$ if we restrict it to $\mathbb{N}$. Sum and subtract operators between two multisets are defined as in [9]. For two given value domains D, D', we denote by $[D \to D']$ the set of possible functions from D to D'.

Definition 1 *A Colored Petri Net graph (CPN graph) is a tuple* $N=\langle \Sigma, \mathcal{X}, F, P, T, cd, Pre, Post\rangle$, *where:* Σ *is a set of colors (see [9]).* $\mathcal{X}$ *is set of variables that range over* Σ. F *is a set of color functions,* $F \subseteq \bigcup_n [\Sigma^n \to \Sigma]$. P *is a set of labeled places, and there are two types of places exists:* AP, *the activation places which contains the CPN execution control,* DP, *which contains the data used during the execution of CPN, especially, we denote the constant data places set as* CP; *Formally, this is represented as follows:* $P : AP \cup DP$ *and* $CP \subseteq DP$, $AP \cap DP = \emptyset$. T *is a set of labeled transitions, we denote* $Type : T' \to T''$ *with* $T', T'' \subset T$ *and* $T' \cap T'' = \emptyset$ *is a type function of* T. $Cd : P \to 2^{\Sigma}$, *is a function that associates to each place a color domain*[2]. $Pre, Post$: *are forward and backward matrices such that* $Pre : P \times T \to \mathcal{M}^+(\Sigma \cup \mathcal{X})$, *are input arc expressions. And* $Post : P \times T \to \mathcal{M}^+(\mathcal{E})$, *are output arc expressions[9].*

$\mathcal{E}$ represents a color expression which can be a color constant, a variable, or a color function of F (completely or partially instantiated). Given an expression $e \in \mathcal{E}$, we use $Var(e)$ to denote the set of variables which appear in e, and $Eval(e)$, the evaluation of e in Σ.

We denote ${}^\bullet t$ and $t^\bullet$ as the input and output places set of transition t, ${}^\bullet p$ and $p^\bullet$ as the input and output transitions set of place p.

[2] In this definition, a transition has no color domain. This restriction will be explained in section 3.2.

Definition 2 *A CPN graph* $N = \langle \Sigma, \mathcal{X}, F, P, T, cd, Pre, Post \rangle$ *is well formed iff:* $\forall t \in T, \forall p \in t^\bullet$, *we have* $Var(Post(p,t)) \subseteq Var(Pre(.,t))$ *with* $Var(Pre(.,t)) = \bigcup_{p' \in {}^\bullet t} var(Pre(p',t))$.

In a well formed CPN graph, we restrict that for each transition, the output arc expressions must be composed by the variables which are in the input arcs expressions.

To each CPN graph, we associate its terms incidence Matrix C ($P \times T \rightarrow \mathcal{M}(\mathcal{E})$) with $C = Post - Pre$.

In the following, we define the behaviors (the dynamics) of a CPN System.

Definition 3 *A marking* M *of a CPN graph is a multiset vector indexed by* P, *where* $\forall p \in P, M(p) \in \mathcal{M}^+(cd(p))$.

Operators + and − on multisets are extended to markings in an obvious way.

Definition 4 *A Colored Petri Net system (CPN system) is a pair* $S = \langle N, M_0 \rangle$ *where* N *is a CPN graph and* M_0 *is an initial marking.*

Definition 5 *A transition* t *is enabled in a CPN system* S *with present marking* M, *iff* $\exists u$, *with* $M \geq Pre(.,t)^u$, $Var(Pre(.,t)) \rightarrow \Sigma$, *which is a binding of the input arcs variables.* [3]

We use $M[t\rangle^u$ to denote that t is enabled in M by the use of u, and we use the classic notation $M[t\rangle$ if u is not important (e.g. when u is unique).

Definition 6 *Let* M *be a marking and* t *a transition, with* $M[t\rangle^u$ *for some* u. *The firing of the transition* t *changes the marking of CPN from* M *to* $M' = M + C(.,t)^u$. *We note the firing as* $M[t\rangle^u M'$.

Definition 7 *We extend the definition 6 to a sequence of transitions* $\delta \in T^*$ *as:* $M[\delta\rangle M$ *if* δ *is the empty sequence;* $M[\omega t\rangle M'$ *iff* $\exists M''$ *such that* $M[\omega\rangle M''$ *and* $M''[t\rangle^u M'$.

3 From BPEL to CPN model

There exist already many works dedicated to translate BPEL services into CPN model for verifying ([3], [13]), composing ([18]), supervising ([4]), etc.. In this section, we construct our own CPN model by introducing the faulty behaviors into Petri nets model which is suitable not only for diagnosing BPEL services, but also for diagnosing other large software systems.

A BPEL process consists of basic activities and structured operators. The idea of modeling BPEL to CPN is: to map each primitive data to a place, each basic activity to a transition. To each basic activity, input and output activation

[3] u must respect the color domain of the places, i.e., $\forall p \in^\bullet t$, $x \in var(Pre(p,t))$, we have $u(x) \in cd(p)$.

places $a^{in} \in P$ and $a^{out} \in P$ are associated to identify the execution order. To include the fault model, additional transitions are added to represent the unobservable faulty activities either in basic WSs or in composite BPEL services. The structured operators are modeled as CPNs which sew the structured subprocesses by combining, disjointing, or generating the local activation places $a_i^{in} \in P$ and $a_i^{out} \in P$. Once a red token is generated by a faulty transition in a basic activity, the fault is passed along the execution trace through the arc expressions which are represented in Pre and $Post$ matrices. In the following, we define how to translate the static and dynamic features into CPN models.

3.1 BPEL data Variables and constants

BPEL data variables and constants

To catch maximally the dependency between data (variables, constants, etc.), we decompose the structured data types into their elementary parts, denoted by the *leaves* of their XML tree structure. For a variable X of type m (resp. an Xpath expression), we use x_i to range over the $Leaves(m)$ (resp. $Leaves(X)$) and denote the x_i part of X by a couple (X, x_i). In our mapping, each data variable and constant is represented by a unique place in CPNs.

Color Domain

In our CPN model, three colors are used: red (r) marks a place with faulty data value; black (b), not faulty data value; and unknown color ($*$), unknown correctness of data value.

Data dependency within BPEL v.s. color functions

To specify the effect of each activity on data, we give each activity a data dependency signature in term of three dependency relations ([1]): forward (FW), if the activity just copies the value from the input to the output; source (SRC), if the output data is generated by the activity; and elaboration (EL), if the output data is elaborated from the set of input data. To each of this dependency relation, we associate a color propagation function to represent the data status (faulty, correct, or unknown status) production.

Definition 8 *Given the data relations set* $D = \{FW, SRC, EL\}$, $\forall d \in D$, *the associated color propagation function* d^c *is defined as:* $\forall c,c' \in \Sigma, \forall \mathcal{C} \subseteq \Sigma$,

$$\begin{cases} FW^c \in [\Sigma \to \Sigma],\ FW^c(c)=c \\ SRC^c \in [\emptyset \to \Sigma],\ SRC^c=* \\ EL^c \in [2^{\Sigma} \to \Sigma],\ EL^c(\mathcal{C})=c',\ \text{with } c'= \begin{cases} b,\ \text{iff } \forall c \in \mathcal{C},\ c=b \\ r,\ \text{iff } \exists c \in \mathcal{C},\ c=r \\ *,\ \text{iff } \exists c \in \mathcal{C},\ c=* \wedge \nexists c'' \in \mathcal{C},\ c''=r \end{cases} \end{cases}$$

In the following sections, we model dynamic features, the basic BPEL activities and structured operators with CPNs.

3.2 Translate basic BPEL activities into CPNs

BPEL service is composed with a series of basic activities. We map each basic activity to its CPN model. Due to space limitation, we restrict our definitions to

four main basic activities (*Receive*,*Assign*,*Invoke*, and *Reply*) while the other similar activities can be easily translated in the same way.

The main idea in mapping BPEL basic activities to CPNs is: each primitive data is mapped to a place, each basic activity is mapped to a transition, and *Pre* and *Post* matrices are defined based on data dependency. In order to distinguish the activities execution order and the traces among different branches, to each basic activity, we associate an input activation place a^{in} and an output activation place a^{out}.

As we focus on the data fault diagnosis of one BPEL service, the BPEL service code is assumed to be correct. Possible faults can be faulty data received by *Receive* activities, or faulty activities which come from other WS called by *Invoke* activities. So we must introduce fault models for *Receive* and *Invoke* activities to localize the faulty data or external WS. Our approach is to introduce additional transitions to represent the unobservable faulty activities and to define the color functions in *Pre* and *Post* matrices which represent the propagation of faults.

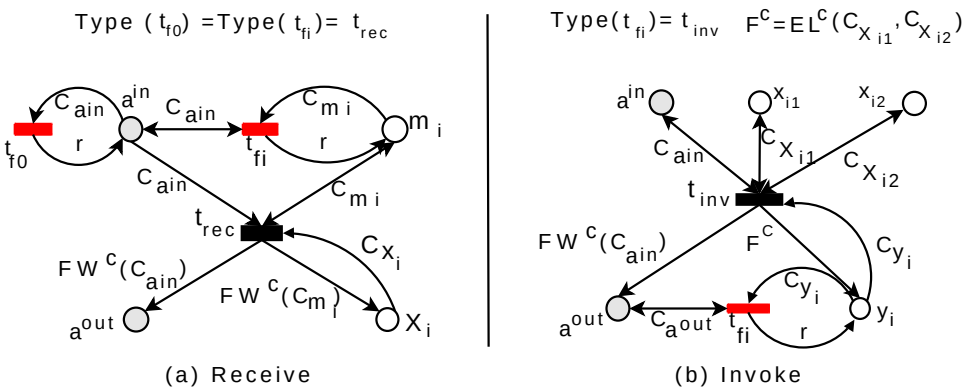


Fig. 1. CPN models of basic activities

Receive(m,X): an activity simply copies the values from a message m to a local variable X. In order to model the receiving of a set of faulty parts from a message value, we add for each part of the message an internal transition (fault) before the firing of the receive transition in figure 1(a). Note that data places $(m, m_i), (x, x_i)$ are simplified as m_i, x_i.

The CPN model of *Receive* contains two kinds of fault transitions: the activation fault transition t_{f_0}, and the fault transitions t_{f_i}, we define their types as: $Type(t_{f_0})$=$Type(t_{f_i})$=t_{rec}. The execution of t_{f_i} is triggered by the consummation of the token in the input activation place. Once t_{f_0} (or t_{f_i}) is executed, we can deduce that there is a faulty control (or data) input. The transmission of the fault (red token) is illustrated on the arc expressions. Each arc expression

represents the colored token consumed (on an arc (p,t)) or produced (on an arc (t,p)). To keep the liveness of the CPNs, we add an arc from the output place x_i to the receive transition t_{rec} and its associated color function C_{x_i} is the color of the output data place x_i.

Reply(Y,m): an activity that copies values from a variable Y to a message m for returning the response of the BPEL service to its invoker. So *Reply* can be the ending of BPEL and simply forwards (FW) values. There is no fault model in its CPN and we simply fill *Post* with FW functions.

Assign(X,Y): an activity that reorganizes local variable parts inside a BPEL process without changing the values. So its model is similar to *Reply* activity. Similar operators: *Throw* and *Rethrow*. The $Wait$, *Empty*, and *Exit* activities do not have relation with the variables, so their CPN model only have the input and output activation places.

Invoke(X,Y): an activity that calls another basic or composite Web service. It takes the value of the variable X as input and stores the output in the variable Y. The data dependency can be FW, EL, and/or SRC. As Y can be infected by external faulty WS which is unobservable, we introduce a series of unobservable faulty transitions after the *invoke* transition to model the faults caused by external WS as is illustrated in figure 1(b).

The CPN model of *Invoke* only contains the fault transitions t_{f_i}, which are triggered by the consummation of the token in the output activation place. Once t_{f_i} is executed, there should be a fault in its output data place and it can be passed to the other activities along the BPEL process execution trace. Again, we define $Type(t_{f_i}) = t_{inv}$.

3.3 Translate structured BPEL activities into CPNs

In this section, we show how to obtain BPEL process CPN by a modular combination of a set of CPNs. We formally define four main structural operators (*Sequence*, *Switch*, $While$, and $Flow$) while the other similar operators can be easily translated in the same way.

Sequence operator $sequence(S_1, S_2)$
Sequence connects different activities, and the execution order of these activities is the same as their appearance order in the constructor. So we can generate the resulting sequence CPN by simply merging the local intermediate output and input activation places of contractive CPNs (in figure 2(a)).

Conditional operator $Switch(\{(con_i(\overline{X_i}, \overline{V_i}),S_i)\}_{i\in I})$
Switch represents an alternative execution of the activities S_i under the conditions $con_i(\overline{X_i}, \overline{V_i})$. $\overline{X_i}$ and $\overline{V_i}$ are respectively the variables and constants. For each subprocess S_i, we add a transition con_i to generate its activation place. Each con_i takes the common activation input place of *Switch*, $\overline{X_i}$, and $\overline{V_i}$ as inputs to **elaborate** an input activation place a_i^{in} for subprocess S_i. So the faults in the relevant data places $\overline{X_i}$, $\overline{V_i}$ cause the faulty choosing of subprocess. That is, one a_i^{in} is activated by mistake. This modeling method allow the diagnosis of the control fault in a BPEL process. At the end of the *Switch* process, a new

a^{out} is added to replace all the a^{out_i} of subprocess S_i (in figure 2(c)). Similar operators: *Link* with its "transitionCondition" ([11]).

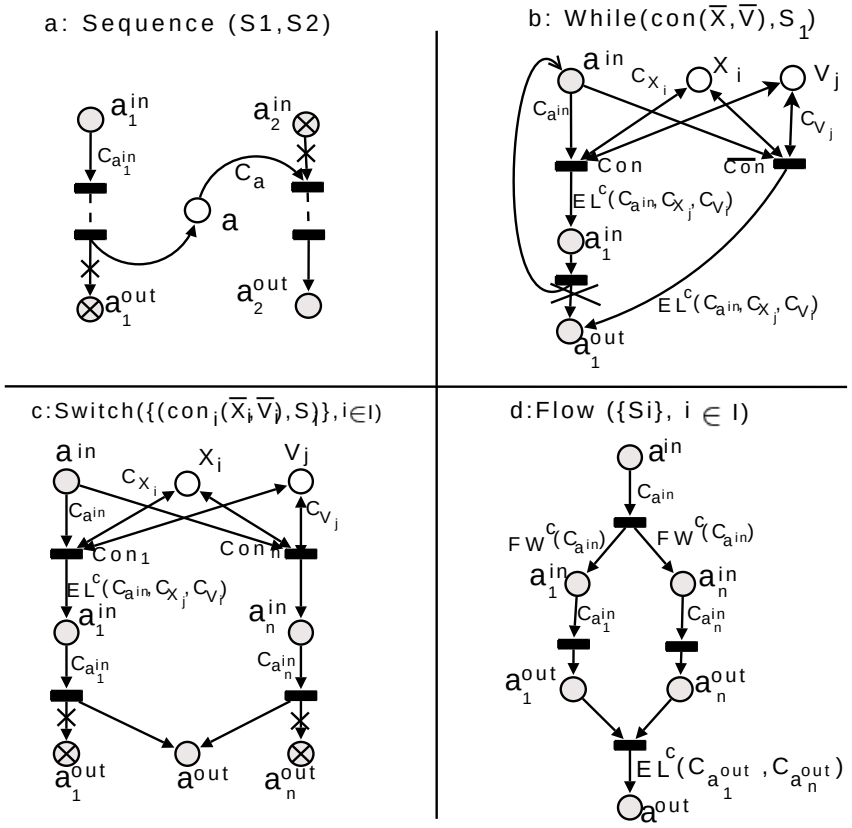


Fig. 2. CPN models of the structural operators

Iterative operator $while(con(\overline{X}, \overline{V}), S_1)$
While iterates the activity S_1 execution until the breaking off of the conditions $con(\overline{X})$. The CPN graph of *While* is similar to *Switch* in which the activation input place of the subprocess S_1 is elaborated by the activation input place of *While*, $\overline{X}$, and $\overline{V}$. But in *While*, the a^{out} of iterative subprocess is also a^{in} of t_{con}. Note that t_{con} represents the transition if condition *con* is true and $t_{\overline{con}}$ represents the transition if condition *con* is false (in figure 2(b)). Similar operators: *RepeatUntil*, *ForEach*.

Parallel operator $flow(\{S_i\}_{i \in I})$
Flow executes the activities S_i in parallel. It terminates when all the activi-

ties are finished (fork-join). So we add a^{in}, a^{out}, t^{in}, and t^{out} to compose the subprocesses together in parallel (in figure 2(d)). Similar operators: *Scope* together with the compensation handlers, event handlers, and fault handlers; *Pick* together with $OnMessage$.

3.4 Some remarks on the BPEL model

Observable vs unobservable transitions
To distinguish the BPEL activities transitions which are observable and the fault transitions which are not, we divide T into observable transitions T_{obs} and fault transitions T_F ($T{=}T_{obs}{\cup}T_F$ and $T_{obs}{\cap}T_F{=}\emptyset$). Note that a type function over faults has been defined in definition 1, which associates a fault to its observable transition Type: $T_F{\rightarrow}T_{obs}$.
Initial and symptom markings of BPEL model net
The initial marking is obtained by marking $P{=}AP{\cup}DP$: constant places ($CP \subseteq DP$) are marked as unknown as they cannot be changed by any transition; other data places ($DP{\setminus}CP$) are marked as black; an activation place $ap \in AP$ which activates the first transition of CPN is marked as black (b) and the other AP are marked as 0. The final marking is retrieved from the thrown exception. When fault(s) occurs, an exception will be thrown to specify on which activity, there is a faulty part(s), which corresponds to the places in DP. Specially, unmatched or uninitiated data (variable) refers the BPEL process may chose fault execution branch. In this case, the input activation place of the activity will be marked as faulty (r). All the other places are marked as unknown ($*$) because there is no information of their marking.
One-boundness of the BPEL model nets
The resulted CPNs are one-bounded (or safe, means one place can at most contain one token). Concerning the data places, the transitions always consume one token in each input places and produce a new one token in each output places. Concerning the activation places, the one-boundness is guaranteed by the fact that a BPEL process does not allow a subprocess call that can lead to more than one token production in the activation places.

3.5 Example (cont'd): CPN model and incidence matrixes

The CPN of the BPEL service *FlightAgent* is constructed as in figure 3. Note that place d_0 represents the request message, d_1 is a null flight schedule message, d_3 and d_2 are respectively the corresponding variables defined in BPEL process. d_4 stock the intermediate single flight information which is generated by invoke activity t_s. Invoke activity t_o continually fills t_2 during the execution of process *FlightAgent*. Place d_5 is the output flight schedule message of d_2. To keep the visibility of the graph, the color functions which do not concern the data dependency are omitted (e.g., color function $C_{a^{in}}$ on the arc (a^{in}, t_c)).

We can see that *FlightAgent* CPN contains 12 places and 11 transitions, in which 5 of them are unobservable, and 6 are observable. Table 2 is the forward matrix, table 1 is the backward matrix, and table 3 is the incidence matrix of

FlightAgent got by $C{=}C^{+}-C^{-}$. As to space limitation, in the incidence matrices, we use the name of places to represent the colors of the places, for example, a^{in} represents $C_{a^{in}}$. Transitions t_{f_a}, t_{f_0}, t_{f_1}, t_{f_2}, and t_{f_3} are unobservable activities (in gray boxes). Especially t_{f_a}, t_{f_0}, and t_{f_1} generate the input fault data of *FlightAgent*, t_{f_3} represents the external fault in the WS which is invoked by t_S, and t_{f_2} represents the external fault in the WS which is invoked by t_O.

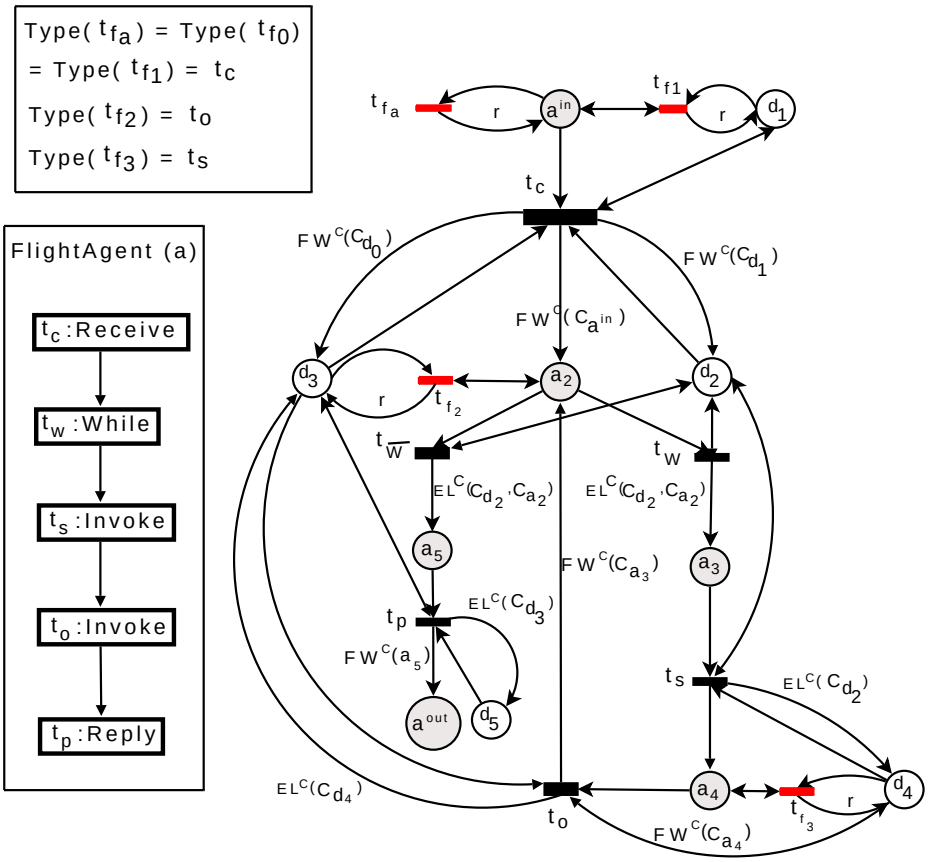


Fig. 3. BPEL(a) and CPN model of *FlightAgent*

4 Diagnosis of BPEL service using CPN

A BPEL process can be considered as a **discrete event system** (DES) of which the classical diagnosis approach is the model based diagnosis ([8]). Given a DES, the diagnosis is to compare the observed behavior of the real system and the

Table 1. C^-: backward matrix of *FlightAgent*

C^-	t_{f_a}	t_{f_0}	t_{f_1}	t_C	t_W	$t_{\overline{W}}$	t_S	t_{f_3}	t_O	t_{f_2}	t_P
a^{in}	a^{in}	a^{in}	a^{in}	a^{in}							
a_2					a_2	a_2					
a_3							a_3			a_3	
a_4								a_4	a_4		
a_5											a_5
a^{out}											
d_0		d_0		d_0							
d_1			d_1	d_1							
d_2				d_2	d_2	d_2	d_2			d_2	
d_3				d_3					d_3		d_3
d_4							d_4	d_4	d_4		
d_5											d_5

Table 2. C^+: forward matrix of *FlightAgent*

C^+	t_{f_a}	t_{f_0}	t_{f_1}	t_C	t_W	$t_{\overline{W}}$	t_S	t_{f_3}	t_O	t_{f_2}	t_P
a^{in}	r	a^{in}	a^{in}								
a_2				$FW^c(a^{in})$					$FW^c(a_4)$		
a_3					$EL^c(a_2, d_2)$					a_3	
a_4							$FW^c(a_3)$	a_4			
a_5						$EL^c(a_2, d_2)$					
a^{out}											$FW^c(a_5)$
d_0		r		d_0							
d_1			r	d_0							
d_2				$FW^c(d_0)$	d_2	d_2	d_2			r	
d_3				$FW^c(d_1)$					$EL^c(d_4)$		d_3
d_4							$EL^c(d_2)$	r	d_4		
d_5											$EL^c(d_3)$

Table 3. $C = C^+ - C^-$: incidence matrix of *FlightAgent*

$C^+ - C^-$	t_{f_a}	t_{f_0}	t_{f_1}	t_C	t_W	$t_{\overline{W}}$	t_S	t_{f_3}	t_O	t_{f_2}	t_P
a^{in}	r-a^{in}			$-a^{in}$							
a_2				$FW^c(a^{in})$	$-a_2$	$-a_2$			$FW^c(a_4)$		
a_3					$EL^c(a_2\ ,\ d_2)$		$-a_3$				
a_4							$FW^c(a_3)$		$-a_4$		
a_5						$EL^c(a_2\ ,\ d_2)$					$-a_5$
a^{out}											$FW^c(a_5)$
d_0		r-d_0									
d_1			r-d_1								
d_2				FW$^c(d_0)$ -d_2						$r - d_2$	
d_3				FW$^c(d_1)$ -d_3					EL$^c(d_4)$ -d_3		$-d_3$
d_4							EL$^c(d_2)$ -d_4	r- d_4			
d_5											EL$^c(d_3)$ -d_5

simulated behavior of its abstract model to isolate, detect, and explain the faults. After defining a CPN model of BPEL process in the former part, we will formally define the CPN diagnose for the data fault in this section.

4.1 Diagnosis problem

During the execution of a BPEL service instance, we can record the sequence of activities executed within this instance, that we call the trace. This trace belongs to $(T_{obs})^*$. When a fault occurs at some moment of the instance execution, an exception is thrown, what we call in diagnosis literature, a symptom. Exceptions are thrown due to some inconsistency of a part of the services state. The inconsistency can concern either data variables values or activation data (e.g receiving a bad message, or not receiving an expected message). In both cases, a thrown exception can be represented as a marking where the faulty data (or activation) places are marked with a red token and the others can be marked either as black or unknown.

Definition 9 *Let M be a marking, M is a symptom (exception) marking iff $\exists p$, $M(p)(r)\neq 0$. We denote the symptom markings by $\hat{M}$.*

So a diagnosis problem is a 3-tuple of the model, observed behavior, and symptom:

Definition 10 *A diagnosis problem is a tuple $\mathcal{D}=< N, \delta_o, \hat{M} >$:*

- *N is a CPN system that represents the model of a BPEL service;*
- *δ_o is an observable trace $\delta_o \in (T_{obs})^*$;*
- *$\hat{M}$ is a symptom marking.*

As fault transitions are unobservable, an observation trace always corresponds to multiple characteristic vectors. That is, different intermediate markings can lead to the same symptom marking. We say the symptom marking "covers" all its former markings. We introduce a covering relation as follows:

Definition 11 *A covering relation $\preceq$ between colors of $\Sigma=\{r, b, *\}$ is a partial ordered relation where any color covers itself and the $*$ color covers all colors (i.e $\preceq=\{(r,r),\ (b,b),(*,*),(r,*),(b,*)\}$). We extend the color covering relation to multisets and markings as follows:*

- *let $m, m' \in \mathcal{M}^+(\Sigma)$, we have $m \preceq m'$ iff $\sum_{c\in\Sigma} m(c) = \sum_{c\in\Sigma} m'(c) \wedge \forall c \neq *$, $m'(c) > 0 \Rightarrow m(c) \geq m'(c)$*
- *let M, M' be two markings, we have $M \preceq M'$ iff $\forall p \in P$, $M(p) \preceq M'(p)$*

We give now a definition of a diagnosis:

Definition 12 *Let $\mathcal{D}=\langle N, \delta_o, \hat{M}\rangle$ be a diagnosis problem, a diagnosis $Sol \subseteq T_F$ and $Sol\neq\emptyset$ such that: $M_0 + C\times \overrightarrow{\delta} \preceq \hat{M}$ with $\overrightarrow{\delta}$ is a characteristic vector defined as follows:*

- $\forall t \in T_{obs}$, $\overrightarrow{\delta}(t)=\overrightarrow{\delta_o}(t)$, *where* $\overrightarrow{\delta_o}(t)$ *is the occurrence number of* t *in* δ_o*;*
- $\forall t_f \in Sol$, $\overrightarrow{\delta}(t_f)=1$*;*
- $\forall t_f \in (T_F \setminus Sol)$, $\overrightarrow{\delta}(t_f)=0$.

Note that we restrict the value of a fault transition to 1. This is due to the fact that a fault transition only changes the color of token to red and has no effect on the activation places marking. Even if a fault happens more than once we consider only the occurrence of the fault transition that can explain the symptom (the red token). Thus we restrict the value of the characteristic vector of a fault transition to one or zero (happened and explains the symptom or did not happen).

Definition 13 *Let* $\mathcal{D}=\langle N,\delta_o,\hat{M}\rangle$ *be a diagnosis problem and* Sol *be a diagnosis,* Sol *is minimal iff* $\forall Sol' \subset Sol$, Sol' *is not a diagnosis.*

Definition 14 *Let* $\mathcal{D}=\langle N,\delta_o,\hat{M}\rangle$ *be a diagnosis problem, the diagnosis solution* $\mathcal{DS} \subseteq 2^F$ *is the set of all possible minimal diagnoses.*

4.2 Diagnosis of CPN by inequations system solving

The aim of diagnosis is to explain the symptoms with minimum set of fault. That is, to find the set of traces in the CPN model all of which cover the same observed trace, the faults that appear in those traces represents the diagnosis.

In our model, the symptom is a reachable marking of the CPN model. While we assume that in the initial marking, data is considered as correct (black token marking). So the inequation 1 for the observation δ_O does not hold.

$$\mathrm{M}_0+C\times\overrightarrow{\delta_O}\preceq\hat{M} \tag{1}$$

By extending the characteristic vector with the fault occurrences, which consist the diagnosis. The covering relation in inequation 2 holds:

$$\mathrm{M}_0+C\times\overrightarrow{\delta}\preceq\hat{M} \text{ such that } \forall t \in T_{obs},\ \overrightarrow{\delta}(t)=\overrightarrow{\delta_o}(t); \tag{2}$$

Let $\mathcal{D}=\langle N,\delta_o,\hat{M}\rangle$ be a diagnosis problem and let n_i be variables ranging over $\{0,1\}$, we construct the characteristic vector δ as follows:

- $\forall t \in T_{obs}$, $\overrightarrow{\delta}(t)=\overrightarrow{\delta_o}(t)$;
- $\forall t_{f_i} \in T_F \wedge \overrightarrow{\delta_o}(Type(t_{f_i})) \neq 0$, $\overrightarrow{\delta}(t_{f_i})=n_i$;
- $\forall t_f \in T_F \wedge \overrightarrow{\delta_o}(Type(t_f))=0$, $\overrightarrow{\delta}(t_f)=0$;

We can then construct an inequations system (one inequation for each place) for the diagnosis problem as follows:

$$Q_{\hat{M}}=\begin{cases} Eq_{p_1} : \hat{M}(p_1) \succeq M_0(p_1) + C(p_1,.)\overrightarrow{\delta} \\ \cdots \\ Eq_{p_i} : \hat{M}(p_i) \succeq M_0(p_i) + C(p_i,.)\overrightarrow{\delta} \\ \cdots \end{cases}$$

To each place p, we associate an inequation Eq_p where the left part is $l(Eq_p)=\hat{M}(p)$ and the right part is $r(Eq_p)=M_0(p) + C(p,.)\overrightarrow{\delta}$. We divide the set of inequations $Q_{\hat{M}}$ into three subsets:

- $Q^r_{\hat{M}}=\{Eq_p|l(Eq_p)=r\}$
- $Q^b_{\hat{M}}=\{Eq_p|l(Eq_p)=b\}$
- $Q^*_{\hat{M}}=\{Eq_p|l(Eq_p)=* \vee l(Eq_p)=0\}$

The diagnosis algorithm executes backward reasoning recursively (algorithm 2) for each inequation $Eq_p \in Q^r_{\hat{M}}$ within $Q_{\hat{M}}$ and then combines all the diagnosis results (algorithm 3). In the following, we give first the solution of one inequation and then that of an inequations system.

One inequation $Q^r_{\hat{M}}$ solving

The part on the right side of an inequation is a multi set composed by color functions, constants, and the corresponding place variables which may have positive or negative coefficients. Solving the inequation consists in canceling the negative terms in the right part, keeping the positive color functions, and evaluating the positive coefficient n_i of red tokens (r) to 1 to explain the red token on the left side of the inequation (algorithm 1). Algorithm 1 looks for the possible minimal diagnosis N^r_p corresponding to one symptom place p in a symptom marking. And at the same time, it looks for the candidate inequations C^r_p which can explain the symptom place but should be solved further. So to completely solve C^r_p, we to recursively back reason by reconstructing $Q^r_{\hat{M}}$, $Q^*_{\hat{M}}$ until getting all possible causes of the symptom place p (algorithm 2).

Algorithm 1 Partially solving a $Q^r_{\hat{M}}$ inequation: $solvAnEqu(Eq_p)$

Input: Eq_p: a $Q^r_{\hat{M}}$ inequation concerns a place p;
Output: $< C^r_p, N^r_p >$ {C^r_p:a set of color functions which generate red tokens; N^r_p: a set of faulty transitions;}

1: $C^r_p=\emptyset$; $N^r_p=\emptyset$;
2: **ForEach** $n_i \times c_i \in r(Eq_p)^+=\sum_{i\in I} n_i \times c_i$ **do**
3: **if** n_i is not a constant and $c_i = r$ **then**
4: $N^r_p=N^r_p \cup \{t_{f_i}\}$; {records the faulty transition t_{f_i} in N^r_p}
5: **else if** c_i is a color function concerning place p' **then**
6: $C^r_p=C^r_p \cup \{c_{p'}\}$;{records the place $c_{p'}$ if its color c_i is unknown for further solving}
7: **else if** c_i is a color propagation function d^c_i **then**
8: $C^r_p=\{C^r_p\} \cup \{c_{p_i} \in Var(c_i)\}$;{records all the input places of c_i for further solving}
9: **end if**
10: **end for**
11: **return** $< C^r_p, N^r_p >$;

An inequations system $Q_{\hat{M}}$ solving

By solving each inequation in $Q^r_{\hat{M}}$ with algorithm 2, we get the diagnosis for

Algorithm 2 Completely solving a $Q^r_{\hat{M}}$ inequation: $CSD(Q_{\hat{M}}, Eq_p)$

Input: $Q_{\hat{M}}$=$Q^r_{\hat{M}} \cup Q^b_{\hat{M}} \cup Q^*_{\hat{M}}$: the inequations system ;
$Eq_p \in Q^r_{\hat{M}}$: an inequation to solve;
Output: Sol_p: a diagnosis solution concerning a symptom place p;

1: Sol_p=$\emptyset$;
2: $\langle C^r_p, N^r_p \rangle$=$solvAnEqu(Eq_p)$;{get the first back reasoning result, C^r_p need to be resolve further}
3: Sol_p=$Sol_p \cup N^r_p$;{record the current diagnosis}
4: **if** $C^r_p \neq \emptyset$ **then**
5: **ForEach** $c_{p'} \in C^r_p$ **do**
6: **if** $\exists Eq_{p'} \in Q^*_{\hat{M}}$ **then**
7: **if** $l(Eq_{p'})$=$*$ **then**
8: $Sol_p = Sol_p \cup CSD(Q^r_{\hat{M}} \cup \{r \succeq r(Eq_{p'})\} \cup (Q^b_{\hat{M}} \cup Q^*_{\hat{M}}) \setminus \{Eq_p, Eq_{p'}\}, r \succeq r(Eq_{p'}))$; {evaluates the $l(Eq_{p'})$ as r, reconstructs the inequations system and recursively back reasoning until solved all the related places}
9: **else if** $l(Eq_{p'})$=0 **then**
10: Sol_p=$Sol_p \cup CSD(Q^r_{\hat{M}} \cup \{r \succeq r(Eq_{p'}) + c_{p'}\} \cup (Q^b_{\hat{M}} \cup Q^*_{\hat{M}}) \setminus \{Eq_p, Eq_{p'}\}, r \succeq r(Eq_{p'}) + c_{p'})$;{evaluates the $l(Eq_{p'})$ as r and add a red token on the right side of the inequation to balance $Eq_{p'}$, reconstructs the inequations system, and recursively back reasoning until solved all the related places}
11: **end if**
12: **end if**
13: **end for**
14: **end if**
15: **return** Sol_p;

a inequations system $Q_{\hat{M}}$ (algorithm 3). The union set of all the Sol_p is the diagnosis solution for $Q_{\hat{M}}$ which can contain multiple symptoms (faults).

Algorithm 3 Diagnosis solution for an inequations system $Q_{\hat{M}}$

Input: $Q_{\hat{M}}$=$Q^r_{\hat{M}} \cup Q^b_{\hat{M}} \cup Q^*_{\hat{M}}$: the inequations system ;
Sol_p=$\emptyset$: a diagnosis solution concerning a symptom place p;
Output: D: a diagnosis solution of $Q_{\hat{M}}$;
1: D=$\emptyset$;
2: **ForEach** $Eq_p \in Q^r_{\hat{M}}$ **do**
3: Sol_p=$CSD(Q_{\hat{M}}, Eq_p)$; {resolve each inequation in $Q^r_{\hat{M}}$ by back reasoning}
4: D=$D \overset{\cup}{\times} Sol_p$;[4]
5: **end for**
6: **return** D;

4.3 Example (cont'): diagnosis problem and solution of *FlightAgent*

In our diagnosis scenario, each BPEL process is associated with a monitoring platform, which records the status of the activities and variables of each execution instance, and a diagnosis is implemented as a WS, which contains the initialed (described in subsection 3.4) CPN model of the BPEL. The diagnosis WS it triggered by the BPEL executer (BPEL execution engine) or invoker (WS, application, etc) once a symptom is thrown by the executer or invoker, the (activation or data) places which correspond to the symptom is marked as r while the other data places are marked as $*$, and activation places are marked as 0. Now suppose we get a series of observed activities σ_0: C, W, S, O, W, S, O, $\overline{W}$, and P, which means the while iteration is processed twice. Then we construct a characteristic vector $\overrightarrow{\delta}^T$: ($t_{f_a}$ t_{f_0} t_{f_1} t_C t_W $t_{\overline{W}}$ t_S t_{f_3} t_O t_{f_2} t_P)=(n_0 n_1 n_2 1 2 1 2 n_4 2 n_3 1). Given an initial marking M_0 =(a^{in} a_2 a_3 a_4 a_5 a^{out} d_0 d_1 d_2 d_3 d_4 d_5)=(b 0 0 0 0 0 b b b b b b), we suppose that, in two diagnosis scenarios, we got two symptom markings M_{n_1} = (a^{in} a_2 a_3 a_4 a_5 a^{out} d_0 d_1 d_2 d_3 d_4 d_5)=(0 0 0 0 0 $*$ $*$ $*$ $*$ $*$ $*$ r), and M_{n_2} = (a^{in} a_2 a_3 a_4 a_5 a^{out} d_0 d_1 d_2 d_3 d_4 d_5)=(0 0 0 0 0 r $*$ $*$ $*$ $*$ $*$ r). For symptom marking M_{n_1}, we construct an inequations system as in equation 3.

[4] $\overset{\cup}{\times}$ is an operator that applies the union operator on couples resulting from the Cartesian product.

$$\begin{cases} Eq_{a^{in}} : 0 \succeq (r - C_{a^{in}}) \times n_0 - C_{a^{in}} + b \\ Eq_{a_2} : 0 \succeq FW^c(C_{a^{in}}) - C_{a_2} \times 2 - C_{a_2} + FW^c(C_{a_4}) \times 2 + 0 \\ Eq_{a_3} : 0 \succeq EL^c(C_{a_2}, C_{d_2}) \times 2 - C_{a_3} \times 2 + 0 \\ Eq_{a_4} : 0 \succeq FW^c(C_{a_3}) \times 2 - C_{a_4} \times 2 + 0 \\ Eq_{a_5} : 0 \succeq EL^c(C_{a_2}, C_{d_2}) - C_{a_5} + 0 \\ Eq_{a^{out}} : * \succeq FW^c(C_{a_5}) + 0 \\ Eq_{d_0} : * \succeq (r - C_{d_0}) \times n_1 + b \\ Eq_{d_1} : * \succeq (r - C_{d_1}) \times n_2 + b \\ Eq_{d_2} : * \succeq FW^c(C_{d_0}) - C_{d_2} + (r - C_{d_2}) \times n_3 + b \\ Eq_{d_3} : * \succeq FW^c(C_{d_1}) + (EL^c(C_{d_4}) - C_{d_3}) \times 2 - C_{d_3} + b \\ Eq_{d_4} : * \succeq (EL^c(C_{d_2}) - C_{d_4}) \times 2 + (r - C_{d_3}) \times n_4 + b \\ Eq_{d_5} : r \succeq EL^c(C_{d_3}) - C_{d_5} + b \end{cases} \tag{3}$$

Note that for final marking M_{n_2}, we can construct a similar inequations system except $Eq_{a^{out}}$ is different ($r \succeq FW^c(C_{a_5}) + 0$) from the one in equation system (3). By applying the diagnosis algorithms, the diagnosis that concerns the symptom marking M_{n_1} is illustrated in figure 4(a) while the diagnosis that concerns M_{n_2} is the $\overset{\cup}{\times}$ product of the diagnosis illustrated in figures 4(a+b) as the inequations system for symptom marking M_{n_2} contains one more red token in the activation output place a^{out}. In figure 4, we illustrate the diagnosis solving process in structured trees. The nodes represent the inequations needed to be solved and each leaf represents a diagnosis and the union of all leaves is a diagnosis solution.

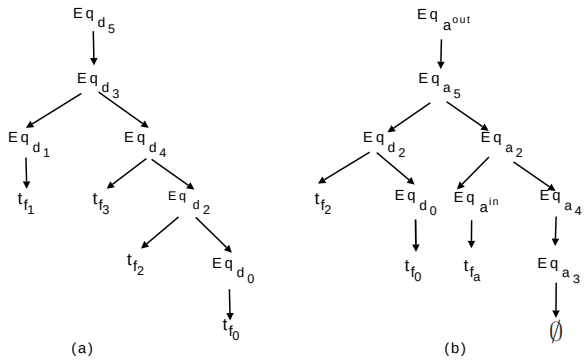


Fig. 4. Diag: M_{n_1} and Diag: $Eq_{a^{out}}$ in M_{n_2}

As a result, for symptom marking M_{n_1}, we have the diagnosis: D_1=$\{\{t_{f_0}\}, \{t_{f_1}\}, \{t_{f_2}\}, \{t_{f_3}\}\}$ represents 4 single faults. Either the input data fault d_0, or the input data fault d_1, or the transition fault on invoke activity S, or the transition

fault on invoke activity O. Concerning the symptom marking M_{n_2}, the diagnosis is extended as: $D{=}D_1 \overset{\cup}{\times} D_2$, where D_2 concerns the red token in activation place a^{out}. As illustrated in figure 4(b), $D_2{=}\{\{t_{f_a}\},\{t_{f_0}\},\{t_{f_2}\}\}$. So, we get diagnosis $D{=}\{\{t_{f_0}\}, \{t_{f_2}\}, \{t_{f_a}, t_{f_1}\}, \{t_{f_a}, t_{f_3}\}\}$, i.e., the fault is on input data place d_0, or on transition O, or on input activation place f_a and invoke activity C, or on input activation place f_a and invoke activity S.

5 Related work

Automata, process algebra, and Petri nets are the most popular DES models. We refer the reader to [14] for the surveys of formal methods of Web services modeling. The major method for diagnosing a DES is trajectory unfolding. Unfolding method is used on the observable trajectory of system evolution to find the faulty states as the diagnosis. For example, [16] proposes a decentralized model-based diagnosis algorithm based on the PNs model ([10]) by inversely unfolding the trajectory. But in [16], local diagnoser does not support iteration in BPEL processes.

We can also adapt the *FlightAgent* example according to the modeling methods of [2] by modeling the states of the BPEL service as places and activities as transitions. As this modeling approach loses the data dependency which cannot ensure the diagnosis is as minimal as ours. [12] models a modular interacting system as a set of place-bordered Petri nets and proposes a distributed online diagnosis which applies algebra calculations from the local models and the communicating messages between them. But when applying [12] on the *FlightAgent* example gets the explosion of the state space because the partition of the variables and messages into subtle parts, and its simple Petri nets definition are too limited to deal with the data aspects.

There are some works that model the WS system with other types of models. In [5], a system is modeled with process algebra containing faulty behavior models. The diagnosis is done by comparing all possible action traces with the observations. All the faulty actions of the matched traces are the diagnosed faults. But [5] models and diagnosis the general WS applications but not a concrete WS specification language. [15] models BPEL services as enriched synchronized automata pieces and diagnose by trajectory reconstruction from observation while the algorithm is incapable for diagnosing the control fault in the process.

A similar diagnosis approach has been proposed in [1], of which we use the same data dependency relation. But [1] does not support loops in WS process while we represent loops as the occurrence in a characteristic vector. In such way, we solve the loops without extra cost. The consistency-base diagnosis approach proposed in [1] is more abstract but loses the precision on modeling level. But we believe that the two methods can get the same minimal diagnosis.

We use the places to represent the data instead of states of the DES in other works. When simulating the CPN model, , the states of the data is more intuitive. And the markings of CPN represent the different states which contain plenteous information and can be formally analyzed.

6 Conclusion

Our CPN modeling approach addresses diagnosis of data fault(s) of orchestrated Web services. The paper constructs a model for the faulty data and faulty activities in a BPEL process. We construct an inequations system for the diagnosis of a BPEL service. And a concrete inequations solving algorithm is proposed. The diagnosis takes advantage of the matrix calculation, which helps to improve the effectiveness of the diagnosis. The interpretation of happened (1) or not happened (0) status of the fault transitions avoids the unfolding of Petri nets or trajectory reconstruction. So the iterative structure in BPEL services does not increase the calculation complexity of the diagnosis.

Our diagnosis approach can be easily extended into the distributed environments according to the approach proposed in [12] by defining a proper composition protocol of the CPNs. And we believe that the diagnosability analysis can also be done using algebra analysis based on the incidence matrix, which is another ongoing work.

References

1. L. Ardissono, L. Console, A. Goy, G. Petrone, C. Picardi, M. Segnan, and D. T. Dupré. Enhancing web services with diagnostic capabilities. In *European Conference on Web Services*, pages 182–191, 2005.
2. A. Benveniste, E. Fabre, C. Jard, and S. Haar. Diagnosis of asynchronous discrete event systems, a net unfolding approach. *IEEE Trans. on Automatic Control*, 48:714–727, 2003.
3. K. Boukadi, C. Ghedira, Z. Maamar, and H. Boucheneb. Specification and verification of views over composite web services using high level petri-nets. Technical Report RR-LIRIS-2006-014, LIRIS UMR 5205 CNRS/INSA de Lyon/Universit Claude Bernard Lyon 1/Universit Lumire Lyon 2/Ecole Centrale de Lyon, 2006.
4. T. Chatain and C. Jard. Models for the supervision of web services orchestration with dynamic changes. In *Advanced Industrial Conference on Telecommunications / Service Assurance with Partial and Intermittent Resources Conference / E-Learning on Telecommunications Workshop*, pages 446–451. IEEE CS, 2005.
5. L. Console, C. Picardi, and M. Ribaudo. Process algebras for systems diagnosis. *Artificial Intelligence*, 142(1):19–51, November 2002.
6. D. CPN Group, University of Aarhus. Cpn tools. `http://wiki.daimi.au.dk/cpntools/cpntools.wiki`, 2007.
7. M. Diaz. *Les réseaux de Petri de haut niveau.* Hermes Science Publications, Paris, France, 2001.
8. W. Hamscher, L. Console, and J. de Kleer, editors. *Readings in model-based diagnosis.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.
9. K. Jensen. *Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use.* Springer, USA, 1997.
10. Y. Li, T. Melliti, and P. Dague. Modeling bpel ws for diagnosis: towards self-healing ws. In *International Conference on Web Information Systems and Technologies*, pages 795–803. IEEE C.S., 2007.
11. OASIS. Bpel 2.0 specification. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html`, August 2006.

12. S.Genc and S.Lafortune. Distributed diagnosis of place-bordered petri nets. *Automated Software Engineering*, 4(2):206–219, April 2005.
13. W. Tan, Y. Fan, and M. Zhou. A petri net-based method for compatibility analysis and composition of web services in business process execution language. *Automated Systems Engineering*, 6:94–106, 2009.
14. Y. Yan. *Description Language and Formal Methods for Web Service Process Modeling*. M.E Sharpe Inc., Armonk USA, 2008.
15. Y. Yan and P. Dague. Modeling and diagnosing orchestrated web service process web services. In *International Conference on Web Services*, pages 9–13. IEEE C.S., 2007.
16. L. Ye and P. Dague. Decentralized diagnosis for bpel web services (poster). In *International Conference on Web Information Systems and Technologies*, pages 283–287. INSTICC, 2008.
17. X. Yi and K. Kochut. Process composition of web services with complex conversation protocols: A colored petri nets based approach. In *Design, Analysis, and Simulation of Distributed Systems*, 2004.
18. Z. Zhang, F. Hong, and H. Xiao. A colored petri net-based model for web service composition. *Journal of Shanghai University (English Edition)*, 105(4):323–329, 2008.

Synthesis of PTL-nets with Partially Localised Conflicts

Maciej Koutny and Marta Pietkiewicz-Koutny

School of Computing Science
Newcastle University
Newcastle upon Tyne, NE1 7RU
United Kingdom
{maciej.koutny,marta.koutny}@newcastle.ac.uk

Abstract. We discuss the problem of constructing PT-nets with localities (PTL-nets) from transition systems with arcs labelled by multisets of transitions (steps). We first outline how this can be done within the existing general solution based on the regions of step transition systems and fixed co-location relations. We then drop the latter assumption and show that this does not really matter when one aims at synthesising PTL-nets where all conflicts involve conflicts between co-located transitions.
Keywords: Petri nets, localities, regions of transition systems, conflict.

1 Introduction

An ever growing number of computing systems behave in the 'globally asynchronous locally (maximally) synchronous (or GALS)' manner. Examples can be found in hardware design, where a VLSI chip may contain multiple clocks responsible for synchronising different subsets of gates [4], and in biologically inspired membrane systems representing cells within which biochemical reactions happen in synchronised pulses [9]. To capture such systems in a formal manner, [6] introduced *Place/Transition-nets with localities* (PTL-nets), where each locality identifies a distinct set of events which must be executed synchronously, i.e., in a maximally concurrent manner (akin to *local maximal concurrency*).

To capture such systems in a formal manner, [6] introduced *Place/Transition-nets with localities* (PTL-nets), where each locality identifies a distinct set of transitions which must be executed synchronously, i.e., in a maximally concurrent manner (akin to *local maximal concurrency*).

In this paper, we aim at constructing PTL-nets from their behavioural specifications given in terms of transition systems with arcs labelled by multisets of transitions (steps). We first outline how this can be done within the existing general solution provided in [3] based on the regions of step transition systems and known co-location relations. We then investigate what happens if no co-location relation for transitions is given in advance, and show that this does not really matter when one aims at synthesising PTL-nets where all conflicts involve conflicts between co-located transitions.

2 Preliminaries

Let T be a fixed finite non-empty set of *(net) transitions*. A *co-location relation* is any equivalence relation $\hat{=}$ on T. For a transition t and a multiset of transitions α (i.e., a mapping $\alpha : T \to \mathbb{N}$, called throughout the paper a *step*), we will denote $t \hat{=} \alpha$ whenever there is at least one transition $u \in \alpha$ satisfying $t \hat{=} u$. Moreover, $\alpha|_t$ is α after deleting all the transitions which are not co-located with t.

Mappings like $f : T \to \mathbb{N}$ or $g : X \times T \to \mathbb{N}$, where X is a set, can accept steps α instead of single transitions, in the following way:

$$f(\alpha) = \sum_{t \in T} \alpha(t) \cdot f(t) \quad \text{and} \quad g(x, \alpha) = \sum_{t \in T} \alpha(t) \cdot g(x, t) \ .$$

A *step transition system* is a triple $\mathcal{T} = (Q, A, q_0)$ where Q is a non-empty finite set of *states*, $A \subseteq Q \times \mathbb{N}^T \times Q$ is a finite set of *transitions (arcs)*, and $q_0 \in Q$ is the *initial state*. We assume that each transition in T occurs in at least one of the steps labelling the arcs of $\mathcal{T}$. Below $\mathcal{T}$ is fixed.

We will write $q \xrightarrow{\alpha} q'$ if (q, α, q') is a transition. Moreover, for every state q:

- $\mathit{allSteps}_q$ is the set of all steps labelling arcs outgoing from q.
- $\mathit{minSteps}_q$ is the set of all non-empty steps $\alpha \in \mathit{allSteps}_q$ for which there is no non-empty $\beta \in \mathit{allSteps}_q$ strictly included in α.
- T_q is the set of all transitions occurring in the steps of $\mathit{allSteps}_q$.
- $\hat{=}_q$ is the restriction of a co-location relation $\hat{=}$ to $T_q \times T_q$.
- Two co-location relations, $\hat{=}$ and $\hat{=}'$, are *state consistent* if the restrictions $\hat{=}_q$ and $\hat{=}'_q$ are the same, for every state q.

A *PT-net with localities* (or PTL-net) is $N = (P, T, W^+, W^-, \hat{=}, M_0)$, where P is a set of *places* disjoint from transitions, $W^+, W^- : P \times T \to \mathbb{N}$ are directed arcs with non-negative integer weights, $\hat{=}$ is a co-location relation, and $M_0 : P \to \mathbb{N}$ is an *initial marking* (in general, any multiset of places is a marking).

A step α of transitions is *resource enabled* at a marking M if, for every place $p \in P$, $M(p) \geq W^-(p, \alpha)$. Such a step is then *control enabled* if there is no transition t such that $t \hat{=} \alpha$ and the step $t + \alpha$ is resource enabled at M. Control enabled α can be fired leading to the marking M', for every $p \in P$ defined by:

$$M'(p) = M(p) - W^-(p, \alpha) + W^+(p, \alpha) \ .$$

We denote this by $M[\alpha\rangle M'$ or $M[\rangle M'$. We assume that for each transition t there is a place p such that $W^-(p, t) > 0$ (otherwise t would never occur in a control enabled step). As a consequence,

> *for every step α which is resource enabled at marking M there is a step containing α which is control enabled at M.* (‡)

The *concurrent reachability graph* $\mathit{CRG}(N)$ of N is the step transition system $\mathit{CRG}(N) = ([M_0\rangle, A, M_0)$ where $[M_0\rangle$ is the set of *reachable* markings (the least set containing M_0 and such that $M \in [M_0\rangle$ and $M[\rangle M'$ implies $M' \in [M_0\rangle$), and $(M, \alpha, M') \in A$ iff $M \in [M_0\rangle$ and $M[\alpha\rangle M'$.

3 From transition systems to PTL-nets

Let us consider the following net synthesis problem:

> *Given a finite $\mathcal{T}$ and a co-location relation $\doteq$, whenever possible construct a finite PTL-net N such that $\mathcal{T}$ is isomorphic to $CRG(N)$.* (†)

It was shown in [3] that synthesis problems like (†) can be solved using techniques coming from the theory of regions of transition systems (see, e.g., [1, 5, 8]).

In this instance, a *region* of the transition system $\mathcal{T}$ is a triple of mappings

$$(\sigma : Q \to \mathbb{N}, \eta^+ : T \to \mathbb{N}, \eta^- : T \to \mathbb{N})$$

such that, for every transition $q \xrightarrow{\alpha} q'$ of $\mathcal{T}$, we have

$$\sigma(q) \geq \eta^-(\alpha) \quad \text{and} \quad \sigma(q') = \sigma(q) - \eta^-(\alpha) + \eta^+(\alpha) \; .$$

Regions are used both to check the feasibility of (†) and to construct a target PTL-net. At the centre of the synthesis procedure outlined below is checking of two required properties of $\mathcal{T}$, called *state separation* and *forward closure*.

Assume that $Q = \{q_0, \ldots, q_m\}$ and $T = \{t_1, \ldots, t_n\}$. We use three vectors of non-negative variables: $\mathbf{x} = x_0 \ldots x_m$, $\mathbf{y} = y_1 \ldots y_n$ and $\mathbf{z} = z_1 \ldots z_n$. We also denote $\mathbf{p} = \mathbf{xyz}$ and define a homogeneous linear system, where $\alpha \cdot \mathbf{z}$ denotes $\alpha(t_1) \cdot z_1 + \cdots + \alpha(t_n) \cdot z_n$, etc.:

$$\mathcal{P} \; : \; \begin{cases} x_i \geq \alpha \cdot \mathbf{z} \\ x_j = x_i + \alpha \cdot (\mathbf{y} - \mathbf{z}) \end{cases} \qquad \text{for all } q_i \xrightarrow{\alpha} q_j \text{ in } \mathcal{T}$$

The regions of $\mathcal{T}$ are determined by the integer solutions $\mathbf{p}$ of $\mathcal{P}$ assuming that $\sigma(q_i) = x_i$ (for $0 \leq i \leq m$) as well as $\eta^+(t_j) = y_j$ and $\eta^-(t_j) = z_j$ (for $1 \leq j \leq n$).

Remark 1. Let α_i be the sum of the sequence of steps labelling arcs along the path from q_0 to q_i in a fixed spanning tree Tree of $\mathcal{T}$. One can eliminate each x_i with $i \geq 1$ through a substitution $x_i = x_0 + \alpha_i \cdot (\mathbf{y} - \mathbf{z})$, resulting in a system equivalent to $\mathcal{P}$ (note that as $\varnothing \in allSteps_{q_i}$ we do not need the inequality $x_0 + \alpha_i \cdot (\mathbf{y} - \mathbf{z}) \geq 0$):

$$\mathcal{P}' \; : \; \begin{cases} x_0 + \alpha_i \cdot (\mathbf{y} - \mathbf{z}) \geq \alpha \cdot \mathbf{z} & \text{for all } q_i \text{ and } \alpha \in allSteps_{q_i} \\ (\alpha_j - \alpha_i - \alpha) \cdot (\mathbf{y} - \mathbf{z}) = \mathbf{0} & \text{for all } q_i \xrightarrow{\alpha} q_j \text{ in } \mathcal{T} \text{ but not in Tree} \end{cases}$$

The set of rational solutions of $\mathcal{P}$ forms a polyhedral cone in $\mathbb{Q}^{m+2n+1}$ (while that of $\mathcal{P}'$ forms a polyhedral cone in $\mathbb{Q}^{2n+1}$). Following [2], one can compute finitely many integer generating rays of this cone $\mathbf{p}^1, \ldots, \mathbf{p}^k$ such that each rational solution $\mathbf{p}$ of $\mathcal{P}$ can be expressed as a linear combination with non-negative rational coefficients,

$$\mathbf{p} = \sum_{l=1}^{k} r_l \cdot \mathbf{p}^l \; .$$

Such rays are fixed and turned into net places if (†) is feasible.

Checking state separation is carried out for each pair of distinct states, q_i and q_j, and amounts to deciding whether there is an integer solution $\mathbf{p}$ of $\mathcal{P}$ with coefficients $r_1, \ldots, r_k$ such that $x_i \neq x_j$. Since the latter is equivalent to

$$\sum_{l=1}^{k} r_l \cdot x_i^l \neq \sum_{l=1}^{k} r_l \cdot x_j^l \ ,$$

one simply checks whether there is l such that $x_i^l \neq x_j^l$.

Checking forward closure is carried out for each state q_i, and starts by calculating the *region enabled* steps $\mathit{regSteps}_{q_i}$. One only needs to consider steps α with $|\alpha| \leq \mathit{max}$ where max is maximum size of steps labelling arcs in $\mathcal{T}$ (note that $\mathbf{p} = \mathit{max} \ldots \mathit{max}\, 1 \ldots 1\, 1 \ldots 1$ is an integer solution of $\mathcal{P}$). Such a step does not belong to $\mathit{regSteps}_{q_i}$ *iff* for some integer solution $\mathbf{p}$ of $\mathcal{P}$ with coefficients $r_1, \ldots, r_k$ we have $x_i < \alpha \cdot \mathbf{z}$. Since the latter is equivalent to

$$\sum_{l=1}^{k} r_l \cdot (x_i^l - \alpha \cdot \mathbf{z}^l) < 0 \ ,$$

one simply checks whether there is l such that $x_i^l - \alpha \cdot \mathbf{z}^l < 0$. Finally, one checks whether $\mathit{allSteps}_{q_i}$ is the set of all $\alpha \in \mathit{regSteps}_{q_i}$ for which there is no $t \in T$ such that $\alpha + t \in \mathit{regSteps}_{q_i}$ and $t \doteq \alpha$.

4 PTL-nets with partially localised conflicts

The synthesis procedure outlined above only works for a given co-location relation which may be unrealistic in practice. As the number of co-location relations for n transitions is finite, one might in principle try them all. This, however, would be impractical since this number is the n-th number in the fast-growing sequence of *Bell numbers*. An important observation helping to reduce this vast range of possibilities is that the synthesis procedure succeeds for $\doteq$ *iff* it succeeds for any co-location relation with which it is state consistent (the proof of a similar property in the context of the synthesis of ENL-systems can be found in [7]). In a special, yet still practically important, case discussed next all one needs to consider is a *single* co-location relation.

A PTL-net has *partially localised conflicts* (or is PTL/LC-net) if the following holds, for all reachable markings M and steps α which are resource enabled at M:

> *If t is a transition resource enabled at M but the step $\alpha+t$ is not resource enabled at M, then $\alpha|_t + t$ is also not resource enabled at M.* (††)

The idea behind a PTL/LC-net is that all the actual (dynamic) conflicts for resources in reachable markings involve only local conflicts. All conflicts between transitions that are not co-located are only static. To justify this, consider (††) and observe that $(\alpha - \alpha|_t) + t$ is resource enabled at M for any PTL/LC-net.

Indeed, otherwise we could take (††) with the same t and $(\alpha - \alpha|_t)$ instead of α reaching the conclusion that $(\alpha - \alpha|_t)|_t + t$ is not resource enabled. But this would mean that t is not resource enabled (as $(\alpha - \alpha|_t)|_t + t = \varnothing + t = t$), contrary to what has been assumed.

Figure 1 shows an example of a PTL/LC-net, where co-located transitions are depicted using the same shading. Note that it exhibits a dynamic conflict between (co-located) transitions u and v, but the conflict between (not co-located) transitions t and u is static.

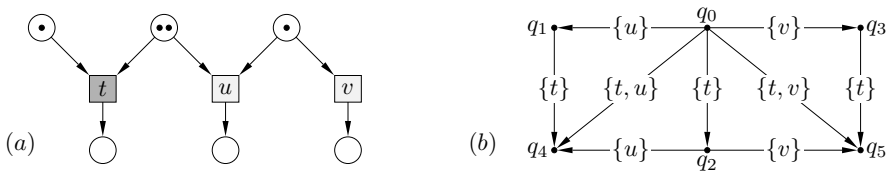


Fig. 1. A PTL/LC-net (a); and its concurrent reachability graph (b).

Let $\mathcal{T}$ be the concurrent reachability graph of a PTL/LC-net N with the co-location relation $\backsimeq$. Moreover, let q be one of its states and

$$max_t^q = \max\{\alpha(t) \mid \alpha \in allSteps_q\} ,$$

for every transition t in T_q. Below we treat q as a marking of N.

Proposition 1. *If $t \in \alpha \in allSteps_q$ then $\alpha|_t \in minSteps_q$.*

Proof. Suppose that $\alpha|_t \notin allSteps_q$. Then (since $\alpha|_t$ is resource enabled at q) there is a transition u such that $u \backsimeq \alpha|_t$ and $\alpha|_t + u$ is resource enabled at q. By (††), $\alpha + u$ is resource enabled at q, contradicting $\alpha \in allSteps_q$. As a result, $\alpha|_t \in allSteps_q$ and so, since all the transitions in $\alpha|_t$ are co-located, $\alpha|_t \in minSteps_q$. □

Corollary 1. *If $\alpha \in minSteps_q$, then all the transitions in α are co-located.* □

The next result shows that, for a concurrent reachability graph of a PTL/LC-net, the local information at a state q about the steps enabled there will determine the co-location relation of any two transitions that are involved in all these steps. More precisely, two transitions will be co-located if either there is no step enabled at q where they both have the maximal number of occurrences, or there is a minimal step at q to which they both belong.

Theorem 1. *Two distinct transitions $t, u \in T_q$ are co-located* iff *either there is no step $\alpha \in allSteps_q$ such that $max_t^q + max_u^q = \alpha(t) + \alpha(u)$, or there is a step in $minSteps_q$ to which the two transitions belong.*

Proof. ($\Longrightarrow$) Suppose that $\alpha \in \mathit{allSteps}_q$ is such that

$$\mathit{max}_t^q + \mathit{max}_u^q = \alpha(t) + \alpha(u) \ ,$$

and so $\alpha(t) \geq 1$ and $\alpha(u) \geq 1$. Then $t, u \in \alpha|_t$ and by Proposition 1, we obtain that $\alpha|_t \in \mathit{minSteps}_q$.

($\Longleftarrow$) Suppose that $t \neq u$ and there is no step $\alpha \in \mathit{allSteps}_q$ such that

$$\mathit{max}_t^q + \mathit{max}_u^q = \alpha(t) + \alpha(u) \ .$$

Let β be a step in $\mathit{allSteps}_q$ such that $\beta(t) + \beta(u)$ is maximal. Since

$$\beta(t) + \beta(u) < \mathit{max}_t^q + \mathit{max}_u^q \ ,$$

we assume, without loss of generality, that $\beta(t) < \mathit{max}_t^q$. Since $\beta \in \mathit{allSteps}_q$ we have that the step $\gamma = \beta(t) \cdot t + \beta(u) \cdot u$ is resource enabled at q. On the other hand, $\gamma + t$ is not resource enabled as otherwise there would have been a step in $\mathit{allSteps}_q$ containing it (see (‡)), contradicting the choice of β. Hence, by $t \neq u$ and $t \in T_q$ and (††),

$$\gamma|_t + t = \beta(t) \cdot t + t$$

is not resource enabled at q. But this contradicts the definition of max_t^q and $\beta(t) + 1 \leq \mathit{max}_t^q$.

If $t, u \in \alpha \in \mathit{minSteps}_q$ then we apply Corollary 1. □

It follows from Theorem 1 that if we can synthesise a PTL/LC-net then the projections $\doteq_q$ of all suitable co-location relations are unique and can be computed locally for each state q from the steps in $\mathit{allSteps}_q$ and $\mathit{minSteps}_q$.

For the transition system in Figure 1(b), we can show that the choice of a co-location relation as in Figure 1(a) was actually the only choice to make this transition system synthesisable to a PTL-net. To see this, let us apply Theorem 1 to states of the transition system in Figure 1(b). For the initial state, we have $T_{q_0} = \{u, t, v\}$ and

$$\begin{aligned} \mathit{allSteps}_{q_0} &= \{\varnothing, \{u\}, \{t\}, \{t, u\}, \{v\}, \{t, v\}\} \\ \mathit{minSteps}_{q_0} &= \{\{u\}, \{t\}, \{v\}\} \ . \end{aligned}$$

Consequently, $t \neq_{q_0} u$ as there is no step in $\mathit{minSteps}_{q_0}$ which contains both t and u, and there is a step $\alpha = \{t, u\} \in \mathit{allSteps}_{q_0}$ such that

$$\alpha(t) + \alpha(u) = 2 = \mathit{max}_t^{q_0} + \mathit{max}_u^{q_0} \ .$$

Similarly, one can show that $t \neq_{q_0} v$. For the last pair of transitions, u and v, we obtain that $u \doteq_{q_0} v$ as there is no step $\alpha \in \mathit{allSteps}_{q_0}$ such that

$$\alpha(u) + \alpha(v) = 2 = \mathit{max}_u^{q_0} + \mathit{max}_v^{q_0} \ .$$

For the remaining states we have

$$T_{q_1} = \{t\} \qquad T_{q_2} = \{u, v\} \qquad T_{q_3} = \{t\} \qquad T_{q_4} = T_{q_5} = \varnothing \ ,$$

and so we only need to check whether $u \doteq_{q_2} v$. The answer is positive since

$$\begin{aligned} \mathit{allSteps}_{q_2} &= \{\varnothing, \{u\}, \{v\}\} \\ \mathit{minSteps}_{q_2} &= \{\{u\}, \{v\}\} \end{aligned}$$

and so there is no step $\alpha \in \mathit{allSteps}_{q_2}$ such that

$$\alpha(u) + \alpha(v) = 2 = max_u^{q_2} + max_v^{q_2} \ .$$

After computing the projections $\doteq_q$, for all $q \in Q$, we form the transitive closure $\doteq_{\mathcal{T}}$ of their union and proceed as follows (note that in the case of our example, $\doteq_{\mathcal{T}} = \{(u,v),(v,u),(u,u),(v,v),(t,t)\}$).

First we check whether $\doteq_q$ is equal to $\doteq_{\mathcal{T}} |_{T_q \times T_q}$, for every state q. If this is not the case, we know that the synthesis problem to PTL/LC-nets fails. Otherwise, we proceed with the procedure outlined in the previous section with a given co-location relation $\doteq_{\mathcal{T}}$, and its outcome determines the outcome of the whole synthesis process. Moreover, if this succeeds, then any other good co-location relation can be obtained as follows.

Let $G^{\mathcal{T}}$ be an undirected graph whose vertices are the equivalence classes of the co-location relation $\doteq_{\mathcal{T}}$, and there is an edge between vertices V and W if there is a state q of $\mathcal{T}$ and two transitions, $t \in V$ and $u \in W$, such that $t, u \in T_q$ and $t \not\doteq_{\mathcal{T}} u$. Then all feasible co-location relations are given through different solutions of the vertex colouring problem for $G^{\mathcal{T}}$. Moreover, since the net structure can be the same as for $\doteq_{\mathcal{T}}$, we do need to re-run the synthesis algorithm again.

5 Final remarks

The idea of considering PTL-nets with partially localised conflicts was inspired by our recent work on the synthesis of elementary net systems with localities [7], where a class of systems without dynamic non-local conflicts has been shown to exhibit properties similar to those captured by Theorem 1.

In our future work we plan to investigate PTL-net synthesis from specifications other than step transition systems.

Acknowledgement We would like to thank the referees for their constructive comments.

References

1. Badouel, E., Darondeau, Ph.: Theory of Regions. LNCS 1491 (1998) 529–586
2. Chernikova, N.: Algorithm for Finding a General Formula for the Non-negative Solutions of a System of Linear Inequalities. USSR Computational Mathematics and Mathematical Physics 5 (1965) 228–233
3. Darondeau, P., Koutny, M., Pietkiewicz-Koutny, M., Yakovlev, A.: Synthesis of Nets with Step Firing Policies. LNCS 5062 (2008) 112–131

4. Dasgupta, S., Potop-Butucaru, D., Caillaud, B., Yakovlev, A.: Moving from Weakly Endochronous Systems to Delay-Insensitive Circuits. Electronic Notes in Theoretical Computer Science **146** (2006) 81–103
5. Ehrenfeucht, A., Rozenberg, G.: Partial 2-structures; Part I: Basic Notions and the Representation Problem, and Part II: State Spaces of Concurrent Systems. Acta Informatica 27 (1990) 315–368
6. Kleijn, H.C.M., Koutny, M., Rozenberg, G.: Towards a Petri Net Semantics for Membrane Systems. LNCS 3850 (2006) 292–309
7. Koutny, M., Pietkiewicz-Koutny, M.: Towards Efficient Synthesis of ENL-systems. Technical Report CS-TR-1141, School of Computing Science, Newcastle University (2009)
8. Mukund, M.: Petri Nets and Step Transition Systems. International Journal of Foundations of Computer Science **3** (1992) 443–478
9. Păun, G.: Membrane Computing, An Introduction. Springer-Verlag, Berlin Heidelberg New York (2002)

Managing Complexity in Model Checking with Decision Diagrams for Algebraic Petri Net*

Didier Buchs, Steve Hostettler

Software Modeling and Verification laboratory
University of Geneva,
route de Drize 7, CH-1227 Carouge Switzerland,
didier.buchs@unige.ch,steve.hostettler@unige.ch,
http://smv.unige.ch

Abstract. *Algebraic Petri Nets* (APN: Petri Nets + Abstract Algebraic Data Types) are powerful tools to model concurrent systems. Because of their high expressive power, allowing end-users to model complex systems, *State Space Explosion* is a big issue in APN. *Symbolic Model Checking* (SMC) and particularly *Decision Diagrams* (DD) based symbolic model checking is a proven technique to handle the state space explosion for simpler formalisms such as P/T Petri nets. This paper discusses how to use Binary Decision Diagrams' (BDD) evolutions (Data Decision Diagrams, Set Decision Diagrams, ΣDD, ...) to tackle the aforementioned problem in the APN world. The main contribution of this work is the notion of the *Algebraic Cluster* that tackles the concurrency induced by token multiplicity. The discussed algorithms have been implemented in a tool that is freely accessible on http://alpina.unige.ch.

Keywords: System design and verification, Higher-level Nets Models, Algebraic Petri Nets, State Space Generation, Decisions Diagrams

1 Introduction

Modeling complex systems needs high-level formalisms, algebraic net is one of the interesting approach that can be used. The Petri net part of the model is employed to express aspects of the system related to causality, non-determinism and concurrency while algebraic abstract data types are used to describe the data evolving in the model as well as the modification of these values.

This paper explores how to use model checking based on decision diagrams for algebraic Petri Nets. In particular, how to manage the inherent combinatorial complexity imposed by data types. We focus on principles that can be used to speed up the reachability analysis. Since we perform symbolic model checking

* This project was partially funded by the COMEDIA project of the Hasler foundation, ManCom initiative project number 2107. We also want to thank Levi Lùcio and Alexis Marechal for their useful comments.

and not bounded model checking, we expect the state space to be finite. These concepts can also be applied to more elaborated model checkers such as CTL model checkers.

We do not explain here the internal machinery (for more details see [1]) of the model checker but we rather focus on its usage. In particular, checking algebraic nets using symbolic representation based on decision diagrams requires an additional level of symbolism with respect to P /T Petri nets model checking. This additional layer, called algebraic clusters, is the main contribution of this paper. Algebraic clusters are clusters [2,3] that are indexed by an algebraic domain. The advantage of such an approach is that it enables discovery of localities of computation and the ability to apply saturation principles. Localities are no longer only defined based on net structure but also on the algebraic values.

Besides, partial or total unfolding greatly speeds up the exploration of the state space and saves memory. By unfolding we mean finding all possible values of a user-defined algebra (if finite) and build specialised operations to handle them instead of heavy general operations that can handle any value. This last aspect will not be detailed in this article however interested readers can find more information in [1]. Throughout this paper we mainly explore how the correct use of algebraic clusters can dramatically improve the construction of the state space both in terms of memory consumption and processing time. Comparison to existing approach can be made on two levels:

- Comparable high-level Petri nets analyzers such as Maria[4] and Helena[5] are completely outperformed if clusters can be mapped on significantly independent part of computation (see section 8).
- Decision diagrams on P/T nets with unfolding [6,7] and our approach are linear in term of performance. However with a significant advantage on our side for the modeling aspect as well as a clear separation of the heuristics (clustering and unfolding) necessary for doing efficient model checking.

These principles are illustrated in our tool called AlPiNA (Algebraic Petri Net Analyzer). [1]

This paper is organized as follows: first we introduce algebraic abstract data types and algebraic Petri nets, then we give an example of an algebraic Petri net that is sufficiently complex to introduce our approach. The fourth section shows, using said example, the kind of properties we can check. Section 5 gives an overview of the operational aspects of model checking while the sixth section presents an abstract definition of the decision diagrams as well as the encoding of basic elements and operational principles. The seventh section introduces the idea behind clustering and how algebraic clustering works. Finally, section 8 describes how clusters are applied to the example and some significant benchmarks demonstrates the approach.

[1] The engine is already available, a GUI will be released within a few months.

2 Short Introduction to Algebraic Nets

Algebraic nets [8] are an evolution of P/T Petri nets where tokens belong to domains defined by algebraic specifications. Although not very different from other extensions of Petri nets such as Coloured Petri Nets [9], APN have several significant advantages:

- The possibility to define any data structures that have first order axiomatisations (it is the case for usual structures that we find in classical modeling or programming languages: integer, lists, sets,... but not reals for example);
- An abstract level of axiomatization combined with concrete operational techniques based on rewriting;
- A formal notation allowing reasoning about data types and their usage. It can be particularly useful to automatically perform proofs.

An APN definition is split into two parts: a Petri Net with places holding typed tokens; and a set of ADT (Abstract Algebraic Data Types) representing data.

2.1 Algebraic Abstract Data Type

Algebraic Abstract Data Types (AADT or ADT for simplicity) [10] provide a mathematical way to define properties of data types. ADT modules define data types by means of algebraic specifications. Each module describes one or more sorts, along with generators and operations on these sorts. The properties of the operations are given in the axiom section of the modules, by means of positive conditional equational axioms.

An algebraic abstract data type is then composed of a signature $\Sigma = \langle S, OP \rangle$ where S is the set of sorts and OP the set of operations with their arity. In general, there are two types of ADTs:

- Primitive data ADTs: boolean, natural, and integer are simple data types.
- Container ADTs: set, list, pair, bag, etc. They represent structured data and allow to construct complex data types and operations. The contained types can be data ADTs or container ADTs.

Moreover, through equations (Ax) based on terms, defined by the signature (T_Σ) and variables (X) to form the term with variables ($T_{\Sigma,X}$), the properties are defined. A specification is given by $Spec = \langle \Sigma, Ax, X \rangle$. There are no predefined data types in ADT and all used types should be defined by ADT modules. Nevertheless, we provide a library of ADTs as part of COOPNBuilder[11]/AlPiNa[12] framework which includes the most commonly used data types: boolean, natural, string, pair, list, etc. These types are fully axiomatized; this allows inferring properties of models for verification purposes.

A calculus is defined through rewriting techniques [13,14], $Rew : T_\Sigma \rightarrow T_\Sigma$, which provides a normal form for any terms. This procedure can be used for deciding equalities between terms (the eval function evaluates terms into their semantic domains) i.e $\forall t, t' \in T_\Sigma, Rew(t) = Rew(t') \Rightarrow eval(t) = eval(t')$. In the rest of the paper, we only use term rewriting as semantics of ADT.

It must be noted that while the notation where kept as minimal as possible we support a rather powerful and useful algebraic extension of ADT, the order sorted algebraic specification [15]. A Complete description of the support of order sorting in our approach can be found in [16].

2.2 APN

Components are described by modular Algebraic Petri Nets which are defined by so-called *behavioural axioms*, similar to the axioms of an ADT.

An APN spec is noted $Apn\text{-}spec = \langle Spec, P, T, Beh, X, m_0 \rangle$ where $Spec$ is the algebraic specification, P the set of places (with $\tau : P \rightarrow S$ a typing function), T the transitions, Beh the set of behavioural axioms (detailed below) and X some variables and , m_0 the initial marking. The (behavioural) axioms have the following structure:

$$Cond \Rightarrow event :: pre \rightarrow post$$

In which each one of the terms has the following meaning: [2]

- $Cond$ is a set of equational conditions, similar to a guard. The $\vdash$ relation decides based on rewriting if a closed condition is satisfied;
- $event \in T$ is the name of a transition;
- pre and $post$ are typical Petri net flow relations (indexed by P) determining what is consumed and what is produced in the net's places.
- pre and $post$ are built on multiset of terms. Multisets are built with + (union), - (difference) and singleton $[a]$ operators.

For instance, the multiset with two numbers 1 and 2 using the natural number ADT described Fig. 2 is written $[s(0)] + [s(s(0))]$ or simply $[s(0), s(s(0))]$.

2.3 Semantics of APN

Knowing the semantics of an ADT, we can build the semantics of an APN. The semantics of an APN is given by a transition system where labels represent what is visible from outside i.e. the events. The states (also called the markings) are represented by a set of multisets of tokens, indexed by the places, expressed by algebraic terms of the ADTs.

Let M be the set of markings such that M is a P indexed family of $T_{[\Sigma],[\tau(p)]}$. The $[s]$ extension mean the addition of sort multiset and associated operations for each sort of the ADT[8]. This is made with empty set ϵ and operations such as marking union (+), marking difference (-) and comparisons $\subseteq$. Markings are vector extensions of the similar operation on multisets for describing the markings.

[2] In our tool only input arcs that are labeled by closed term or isolated variables are allowed in *pre*. Putting a variable in the input arc plus a condition in the guard simulates composed terms.

Given an axiom $Cond \Rightarrow event :: pre \rightarrow post$ and $m, m' \in M$ the markings, $event \in T$, the transitions induced by that behavioural property is:
$\forall \sigma$[3], if $\vdash Cond\sigma$, $Rew(pre\sigma) \subseteq m, m \xrightarrow{event} m - Rew(pre\sigma) + Rew(post\sigma)$ [4]

The resulting construction is the transition system built from the initial marking by applying this firing rule until reaching a fix point $TS_{apn}(m_0)$. From the transition system, we can also consider the reachability set $Reach_{apn}(m_0)$.

3 Modeling with apn

The model (Fig. 1) used in the rest of the paper is the distributed database manager used by many authors, originally presented by Genrich and Lautenbach in [17], working on Petri net and associated models. This model raises interesting problems concerning domains. For instance, the number of database is finite (type Databases) while the number of acknowledgments is modeled with natural numbers. Limited domains will be enumerated (we say here unfolded) before the state space computations while the domain of naturals will be explored during the state space exploration. We call this combination "partial unfolding". Alternatively, the graphical syntax can be replaced by the previous textual syntax. For instance, the "Receive Msg" transition is modeled by the following axiom:
$db1 = db2 \Rightarrow$ Receive Msg $:: \langle([db1])_{Pending}, ([db2])_{Inactive}\rangle \rightarrow \langle([db1])_{Syncing}\rangle$

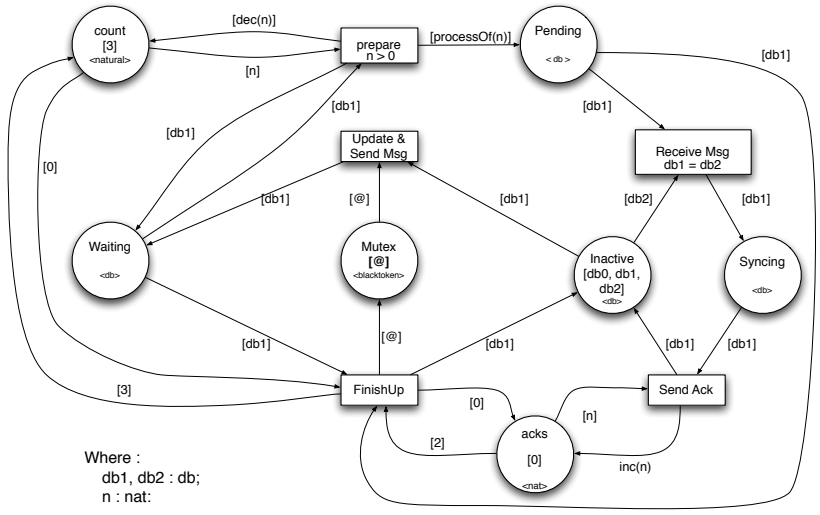


Fig. 1. Distributed database Algebraic Petri Net model

[3] σ is a term substitution. $t\sigma$ stands for applying the substition σ to the term $t \in T_{\Sigma,X}$
[4] *Rew* is the evaluation morphism extended to APN structures such as multiset and vector of places from the algebraic morphism *Rew*.

In Fig. 2, one can find a simplified version of the ADTs that describe the data part of the model. Only the operations and axioms that are required for the problem are written. The algebra of the booleans is composed of true and false without any operation. The natural numbers are described using Peano arithmetic and finally the databases are represented by a structure that is isomorphic to the natural numbers modulo 3.

```
ADT Booleans
    Sort bool
    Generators
        true : bool;
        false : bool;
End Booleans

ADT Databases
    Sort db
    Use nat
    Generators
        db0 : db;
        d : db -> db;
    Operations
        processOf : nat -> db;
    Axioms
        d(d(d(db0))) = db0;
        processOf(s(x)) = d(processOf(x));
     Where
        x:nat;
End Databases
```

```
ADT Naturals
    Sort nat
    Use bool
    Generators
        0 : nat;
        s : nat -> nat;
    Operations
        dec : nat -> nat;
        inc : nat -> nat;
        > : nat ,nat -> bool;
        2 : nat;
        3 : nat;
    Axioms
        inc(x) = s(x);
        dec(s(x)) = x;
        dec(0) = 0;
        >(s(x), s(y)) = >(x, y);
        >(s(x), 0) = true;
        >(0, s(x)) = false;
        >(0,0) = false;
        2 = s(s(0));
        3 = s(s(s(0)));
      Where
        x,y:nat;
End Naturals
```

Fig. 2. ADT of the Distributed Databases Protocol

4 The purpose of symbolic model checking

In order to check properties, a dedicated language is offered which check mainly in CTL temporal logic setting the formula of the shape: $AG(atm_{\Sigma,P})$ i.e. determines for a given property $at \in atm_{\Sigma,P}$ if $Reach_{apn}(m_0) \models AG(at)$, where $atm_{\Sigma,P}$ is an atomic formulae based on signature and place definitions, operation on sets and algebraic conditions. We currently do not support other CTL operators such as (F, U, ...) as they require the predecessor relation (pre) as defined in [18]. This restricts the checks to those can be verified by reachability analysis. In our model checker, we gave some possibilities to check more detailed properties at the atomic proposition level, such as inclusion or exclusion of particular tokens or sets of tokens. The following properties can be checked on the model of Fig. 1:

- A process is either idle, waiting for other processes to sync or syncing: $AG(inactive \cap syncing = \emptyset \wedge syncing \cap waiting = \emptyset \wedge waiting \cap inactive = \emptyset)$
- At most one database process is the master process: $AG(card(wating) \leq 1)$

These properties will be satisfied for a given number of databases, in the benchmarks we will show how many databases we can handle with our operational method. Please note the whole does not work for an arbitrary number of databases since we have to compute the state space entirely.

5 Basic Operational Techniques

In this section we define the model checking techniques that are used to build the set of reachable states. The usual process for computing the state space is to successfully apply transitions on the marking until reaching a fix point. For an account of the reachability algorithms, see [7]. Algorithm. 1 describes the brute force approach to build the state space.

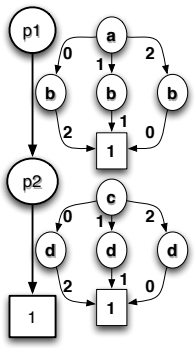
Algorithm 1: Compute the set of states reachable from m_0 using Beh

```
Input: m_0 the initial state.
Input: Beh the set of axioms to apply.
Result: the set of states reachable from the states m_0 using Beh
begin
    s, s' are set of states
    s := {m_0} ;
    repeat
        s' := s ;                          // Save the old set of states
        foreach beh ∈ Beh do
            foreach m ∈ s do
                foreach σ,⊢ Cond_beh σ and Rew(pre_beh σ) ⊆ m do
                    m' := m − Rew(pre_beh σ) + Rew(post_beh σ);
                    s := s ∪ {m'};
                end
            end
        end
    until s = s' ;                         // Until we reach a fixpoint
    return s ;
end
```

Decision diagrams (DD) are very efficient techniques to operationally implement the previous algorithm. Such approaches are both very efficient in term of memory consumption and processing time. In the DD framework, the repeat loop, both for loops as well as the application of the axioms are encoded using so called homomorphisms. The next section gives an overview of the DD framework as well as more details on the operational implementation of said homomorphisms. The above procedure is far from being optimized for complex systems. We will show how to improve it by splitting up the state spaces through clustering. The pieces are then composed afterwards to build the complete state.

6 Decision Diagrams

Data Decision Diagrams (DDD [19]) and Set Decision Diagrams (SDD [3]) are both evolutions of the well-known Binary Decision Diagrams (BDD)[20]. While BDD is often seen as representing a Boolean function, it can also be seen as a set of sequences of assignments of Boolean values to variables. DDD (resp. SDD) are similar for assignments of any kind of values (resp. sets) of the form $(var_1 := val_1).(var_2 := val_2)\ldots(var_n := val_n)$. Val will designate the possible values and Var the variable names.



The SDD on the left side represents the Cartesian product of $p1$ and $p2$ that is 9 paths or states. SDD ($e_{sdd} = \{p1, p2\}$) embed DDD ($e_{ddd} = \{a, b, c, d\}$):

$$p1 \xrightarrow{a\xrightarrow{1}b\xrightarrow{1}1} p2 \xrightarrow{c\xrightarrow{1}d\xrightarrow{1}1} 1 + p1 \xrightarrow{a\xrightarrow{0}b\xrightarrow{2}1} p2 \xrightarrow{c\xrightarrow{1}d\xrightarrow{1}1} 1 +$$
$$p1 \xrightarrow{a\xrightarrow{2}b\xrightarrow{0}1} p2 \xrightarrow{c\xrightarrow{1}d\xrightarrow{1}1} 1 + p1 \xrightarrow{a\xrightarrow{1}b\xrightarrow{1}1} p2 \xrightarrow{c\xrightarrow{0}d\xrightarrow{2}1} 1 +$$
$$p1 \xrightarrow{a\xrightarrow{0}b\xrightarrow{2}1} p2 \xrightarrow{c\xrightarrow{0}d\xrightarrow{2}1} 1 + p1 \xrightarrow{a\xrightarrow{2}b\xrightarrow{0}1} p2 \xrightarrow{c\xrightarrow{0}d\xrightarrow{2}1} 1 +$$
$$p1 \xrightarrow{a\xrightarrow{1}b\xrightarrow{1}1} p2 \xrightarrow{c\xrightarrow{2}d\xrightarrow{0}1} 1 + p1 \xrightarrow{a\xrightarrow{0}b\xrightarrow{2}1} p2 \xrightarrow{c\xrightarrow{2}d\xrightarrow{0}1} 1 +$$
$$p1 \xrightarrow{a\xrightarrow{2}b\xrightarrow{0}1} p2 \xrightarrow{c\xrightarrow{2}d\xrightarrow{0}1} 1$$

Again the power of the SDD lies in the Cartesian product symbolic encoding. Using SDD, thanks to the sets, we end up with a two-dimensional symbolic encoding.

Fig. 3. SDD

6.1 Abstract definition of Decision Diagrams

A decision diagram (DD) is a structure $\mathbb{DD}$ with properties of set and the following operations:

- $SeqA_{Var,Val}$ is the set of sequences of assignments of value Val to variables Var. This object will be efficiently represented by DD.
- internal operations ($\mathbb{DD} \times \mathbb{DD} \rightarrow \mathbb{DD}$) such as the above mentioned set operations $\cup_{DD}$, $\cap_{DD}$, $\setminus_{DD}$, one constant $\emptyset_{DD}$,
- the encoding and decoding operations: $encode_{DD} : \bigcup SeqA_{Var,Val} \rightarrow \mathbb{DD}$, $decode_{DD} : \mathbb{DD} \rightarrow \bigcup SeqA_{Var,Val}$,
- specific internal operations ($\mathbb{DD} \rightarrow \mathbb{DD}$) such as inductive homomorphisms hom_{DD}.

Moreover, these operations must have the following properties:

- all operations are homomorphic i.e. $op(d \cup_{DD} d') = op(d) \cup_{DD} op(d')$. This fundamental property ensures that computations can be made globally on a set avoiding costly value-by-value calculations.

- the domain $SeqA$ is the sequence of assignments specific to the encoded information of the domain Val. Sequences of assignments are composable with concatenation $'.'$ or $\bigotimes$ for an indexed sequence of concatenation. For instance 'v1:=1.v2:=2.v3:=3' is a sequence of assignments.
- an efficient comparison is provided $= : (\mathbb{DD} \times \mathbb{DD} \rightarrow \mathbb{B})$. It works in constant time due to the canonicity of the representations. This point is very important in DD and the formal representation given to DD hide this point for simplicity but the concrete realization must ensure this fundamental property.
- encoding and decoding are reverse operations: $encode_{DD} \circ decode_{DD} = decode_{DD} \circ encode_{DD} = Id$.

In the encoding necessary for this paper, we use various structures based on different kind of assignment (defined in the domain $SeqA_{Var,Val} = \{(v_1 := val_1).(v_2 := val_2)...(v_n := val_n)\}$):

- DDD (Data Decision Diagrams) where $SeqA$ is based on $v := val$, $v \in Var$, $val \in ValueDomain$[5]
- SDD (Set Decision Diagrams) where $SeqA_{Var,Val}$ is based on $v := val$, $v \in Var$, $val \in \mathcal{P}(ValueDomain)$
- MSDD (Multi Set Decision Diagram) where $SeqA_{Var,Val}$ is based on $v := val$, $v \in Var$, $val \in \mathcal{PMS}(ValueDomain)$[6]
- ΣDD (Signature based Decision Diagrams) k $SeqA_{Var,Val}$ is based on $v := val$, $v \in S$, $val \in \mathcal{P}(T_{\Sigma,X})$ with an homomorphism implementing rewriting $Rew_{\Sigma DD}$ compatible with rewriting on terms Rew. The ΣDD [16] structure implements rewriting on set of terms very efficiently (comparisons with reference implementations such as Maude [21] show real advantages of ΣDD for large set of terms). This means that we can perform proof of universally quantified formula on finite domains in a reasonable time.

6.2 Computing reachable states

Computing the state space requires a basic schema of how states are encoded, and the homomorphisms that compute new states from existing states. Without entering into a detailed description, we give a sketch of states are encoded.

Encoding The encoding is given for transition systems following its inductive definition:

- $encode_{SDD}(m) = \bigotimes_{p \in P}(p := encode_{MSDD}(m_p))$
- $encode_{MSDD}(\epsilon_s) = s := \emptyset_{MSDD}$, $s \in S$
- $encode_{MSDD}([t]_s) = (s := encode_{\Sigma DD}(\{t\}))$[7] , $s \in S$ and $t \in T_\Sigma$
- $encode_{MSDD}(ms + ms') = encode_{MSDD}(ms) \cup_{MSDD} encode_{MSDD}(ms')$
- $encode_{MSDD}(ms - ms') = encode_{MSDD}(ms) \setminus_{MSDD} encode_{MSDD}(ms')$
- $encode_{\Sigma DD} : \mathcal{P}(T_{\Sigma,X}) \rightarrow \mathbb{SIGDD}_\Sigma$ which encodes a term as a ΣDD [16].

[5] Where $ValueDomain$ is the domain of the variables.
[6] The $\mathcal{PMS}$ notation corresponds to the power multi-set.
[7] The basic encoding element of a sequence.

Homomorphisms Homomorphisms are used to encode the transition relation. We define in [1] homomorphisms for algebraic nets that compute successor states of given states.

Let $Beh_t = \langle event, pre, Cond, post \rangle$ be a transition behaviour. [8] We define $H^-_{Beh_t}$, $Check_{Beh_t}$ and $H^+_{Beh_t}$, based on elementary homomorphisms H^+ and H^- working on each individual places, [1] by:

- $H^-_{Beh_t} = \bigcirc_{p \in P} H^-(p, pre_p, event)$,
- $H^+_{Beh_t} = \bigcirc_{p \in P} H^+(p, post_p, event)$,
- $Check_{Beh_t} = \bigcirc_{\langle l,r \rangle \in Cond} check(\langle l, r \rangle)$.

The homomorphism Hom_{Beh} applies the behaviour of all transitions of T by combining the previous operators: $Hom_{Beh} = \bigcup_{Beh_t \in Beh} H^-_{Beh_t} \circ Check_{Beh_t} \circ H^+_{Beh_t}$ and finally we compute the transitive closure: $Hom^*_{Beh} = (Hom_{Beh} \cup Id)^*$.

At the end of this process, we obtain a SDD structure building the whole transition system $Reach_{apn}(m_0)$ from an initial marking :

$$\bigcup_{m \in Reach_{apn}(m_0)} encode_{SDD}(m) = Hom^*_{Beh}(encode_{SDD}(m_0))$$

which corresponds to the program schemata given at the beginning. When implementing such a procedure we can see that the performance, while interesting, is not as good as those of [7]. Reasons are that the processes are not well captured and non-necessary interleaving of computations are explicitly represented while they should have been symbolically represented. In the next section we will present algebraic clustering to overcome this problem.

7 When axioms can help discovering processes

In this section, we explain the principles that can be used for capturing processes and modules in a model (subparts of independent behaviour) and how they impact on the computation of the state space.

Let's divide the model M of a system into n components such that $M = C_1 \times \ldots \times C_n$ and thus the complexity of $|M| = |C_1| \cdot \ldots \cdot |C_n|$. Hence, in the best case, that is when each and every component is independent, we end up with the full Cartesian product of the states of the different components. This is exactly what we get when concatenating decision diagrams representing set of states. Therefore, to efficiently use decision diagrams, we need to split up the system in components, consider them locally and then to compose their state space. It is clear that in the general case, components are not completely independent and adjustments must be made.

If well chosen, these clusters help to compute a considerably smaller number of symbolic values whilst generating the state space. Moreover, the representation of the state space itself is dramatically reduced [2]. In the following we show

[8] $\bigcirc$ represent an indexed sequence of compositions.

how to define these subparts with the concept of clusters. This approach extends the one of [2,3] and is called algebraic clustering [1].

Notice that previous approaches tackled the problem by splitting up the model based on structural information [2,3]. Since our work is based on algebraic nets, we do require managing algebraic values or colors by generalizing the approach to algebraic values.

Our clustering function will define in an algebraic setting a mapping between place (the structural part of the model) and algebraic values present in the places to a dedicated finite algebraic domain of cluster identifiers. The P indexed Cl function dispatch the tokens (the algebraic values in place's sort) in the clusters for each place $p \in P$. Please note that operationally only the syntactical representation ($t \in T_{\Sigma,\tau(p)}$) of the algebraic values are manipulated.

$$Cl_P : T_{\Sigma,\tau(p)} \rightarrow Clusters$$

For homogeneity reasons, this function is specified in the same formal framework as the algebraic abstract data type. Nevertheless, we will add 'syntactic sugar' to the usual syntax in order to simplify the task of the user of the model checker tool. In order to easily manage cluster domains, order sorting is used for specifying that each inductive cluster forms a disjoint subpart of the clusters. This is also true for structural clusters. We will first define the domain $Clusters$ as union of structural ($structClusters$) and algebraic clusters ($algClusters$) and the show how to axiomatize the Cl function.

7.1 Structural clusters

Structural clusters (a.k.a static clusters) are clusters that can be finitely enumerated. Typically, structural clusters represent modules (and thus the structure) of the model. Such clusters are static with regards to the algebraic values that they contain. In other words, tokens are dispatched among the structural clusters according to structural information such as the place they belong to. Structural clusters in place $p \in P$ have the property that $\exists c \in Clusters, \forall t \in T_{\Sigma,\tau(p)}, Cl_p(t) = c$.

7.2 Algebraic clusters

Algebraic clusters (a.k.a inductive clusters) are finite domains that can be obtained by inductive definition limited by a bound formula. Typically, algebraic clusters represent objects or groups of objects that behave together. An inductive cluster is defined by a triple:

$$algCluster_i = \langle clBase, clInductive, clBound \rangle$$

Where $clBase$ is a constant, $clInductive$ is a unary operation inductively defining cluster names (isomorphic to natural number) and $clBound$ is an algebraic condition selecting only a finite subpart of these inductively defined set.

The defined set $UnboundCl_i$ is then the least set satisfying:

$$clBase \in UnboundCl_i$$
$$\forall c \in UnboundCl_i,\ clInductive(c) \in UnboundCl_i$$

Moreover, the bound is used to only keep values that satisfy the $clBound$ constraint: $Clusters_i = \{c \in UnboundCl_i \mid c \models clBound\}$. In this paper, we assume that such selection can be done easily in order to produce the finite set of clusters that can be exploited in the concrete computations. Decidable logics, that fits with this requirement, have been studied in [22,23].

7.3 Axiomatisation of clustering

Axioms must be given to semantically define the cluster mapping. As mentioned previously, for homogeneity reasons, clustering is defined by using the AADT framework. However, for the sake of simplicity, we simplified a bit the axiomatization and we provide syntactic sugar to easily define clustering properties on set of places. The following examples show some clustering properties:

- cluster of 0 in $\{p1, p2, p3\}$ is $clbase$
- cluster of $succ(x)$ in $\{p1, p2, p3\}$ is cluster of x in any

The first property says that the value '0' is dispatched in a cluster called $clbase$ if it occurs in the places $\{p1, p2, p3\}$. The second property is inductive and defines that every token of the form "succ(x)" is dispatched according to "x". In this case, the clustering dispatched every natural numbers in the same cluster ($clbase$) if it occurs in the places $\{p1, p2, p3\}$. A different set of axioms would then dispatch values in distinct clusters:

- cluster of 0 in $\{p4\}$ is $clbase$
- cluster of $succ(x)$ in $\{p4\}$ is cl(cluster of x in $\{p4\}$)

7.4 Properties of the clustering

The properties of the cluster functions can be linked to the axioms of the *apn* under study, in particular they have to express a notion of independence similar to what we have in reduction methods based on symmetry. Locality is one important aspect. A local transition is a transition that only affects one cluster. So given one axiom $Cond \Rightarrow event :: pre \rightarrow post$, it is local if it is involved in one cluster i.e.:

$$\forall \sigma, \vdash Cond\sigma \Rightarrow \exists cl \in Clusters, \forall p \in P, Cl_p^m(pre_p\sigma) = Cl_p^m(post_p\sigma) = \{cl\}.$$ [9]

Transitions are locals if all their axioms are local. In the example, we can show that transition *Receive Msg* is local because we give the cluster function:

[9] cluster function Cl is naturally extended to multisets with i.e $Cl_p^m(\epsilon) = \emptyset$, $Cl_p^m([t]) = \{Cl_p(t)\}$, $Cl_p^m(t + t') = Cl_p^m(t) \cup Cl_p^m(t')$

- cluster of db0 in any is cl_0
- cluster of $db(x)$ in any is cl(cluster of x in any)

It is then necessary to prove that $\forall\sigma$ s.t $db1 = db2$ we have $Cl^m_{pending}([db1]) = Cl^m_{inactive}([db2]) = Cl^m_{syncing}([db1])$. Using the definition of clusters and ΣDD this proof is performed by exhaustively instantiating the domain. This will split the set of axioms into local and non-local axioms before computing local and global fixpoints. Exhaustive proof can be very costly when domains are large. In some contexts, it would be interesting to have pattern of axiom that are local. For instance, we can prove by induction on the database that we have a local axiom:

- base case : $Cl^m_{pending}([db0]) = Cl^m_{inactive}([db0]) = Cl^m_{syncing}([db0]) = \{cl_0\}$ is obviously verified.
- induction $Cl^m_{pending}([db(x)]) = Cl^m_{inactive}([db(x)]]) = Cl^m_{syncing}([db(x)]]) = \{cl(Cl_{pending}(x))\} = \{cl(Cl_{inactive}(x))\} = \{cl(Cl_{syncing}(x))\}$: which is verified using the induction hypothesis and the substitutivity property.

This pattern of axioms can be generalized to any axiom with input and output places of the same sort and same clustering definition, where conditions must be neutral. In this case, the axiom is local. This is used to quickly discover localities.

7.5 The computation of the reachability set revisited

Symbolically, instead of computing the reachability set for the whole system, $Reach_{apn}(system)$, we will compute the same state space using the computation of the state space of the clusters.

Algorithm 2 starts by saturating (computing the transitive closure) of each subpart of the model then it saturates the model itself. Please note that this is a simplified version of the saturation algorithm. The actual one does first saturate variables that have been impacted by a previous saturation.

8 A good use of clusters to exploit processes

In our example (the distributed database), it seems natural to dispatch the different database process among different clusters. By doing that, behaviours that are local to a process that is without side effects on other database processes can be juxtaposed in order to get the Cartesian product of such behaviours.

- cluster of any in {counter, mutex, acks} is *semaphore*
- cluster of db0 in any is cl_0
- cluster of $db(x)$ in any is cl(cluster of x in any)

Algorithm 2: Compute the clustered set of states reachable from the states m_0 by applying $Beh = \cup_{c \in Clusters} Beh_c \cup Beh_{global}$ with Beh_c the set of local behaviours of the cluster $c \in Clusters$ and Beh_{global} the set of global transitions. It first saturates local transitions before global ones.

Input: m_0 the initial set of states.
Input: $Hom_{Beh_c} = (\cup_{b \in Beh_c} encode_{Hom}(b) \cup Id)^*$
Input: $Hom_{Beh_{global}} = (\cup_{b \in Beh_{global}} encode_{Hom}(b) \cup Id)^*$
Result: the clustered set of states reachable from the states m_0 using Beh
begin
  s, s' are clustered set of states
  $s := clusteredEncode_{SDD}(m_0)$; ; `// extends` $encode_{SDD}$ `to clusters`
  repeat
    $s' := s$; `// Save the old set of states`
    foreach $c \in Clusters$ **do**
      $s := Hom_{Beh_c}(s)$; `// Apply the local axioms to` s_c
    end
    $s := Hom_{Beh_{global}}(s)$; `// Apply the global axioms to` s
  until $s = s'$; `// Until we reach a fixpoint`
  return s ;
end

The first axiom tells us that any algebraic values present in one of the following places {counter, mutex, acks} must be put in the cluster c_0. When the clustering is independent from the algebraic values and only dependant of the places, it is typically an example of a structural cluster. In this case, all the values shared by all processes (semaphores) are put in the same cluster. Since those values are shared, they can be part of any local behaviour. The second axiom gives the base case of the algebraic clustering and the third one gives the inductive step. Intuitively, it means that the local behaviour can be computed by only working on a subpart of the model.

		AlPiNA						Maria		Helena	
		Partial Unfold.			Total Unfold.						
Model Size	States #	DD #	Mem (MB)	Time (s)	DD #	Mem (MB)	Time (s)	Mem (MB)	Time (s)	Mem (MB)	Time (s)
10	196821	15336	10	0.8	13402	12.4	1.3	47	44.3	24	9
15	7.17E7	51E3	32.6	2.6	49E3	41	5.8	-	-	1.4E3	7.5E3
35	5.84E17	993E3	544	69.4	117E4	789	278	-	-	-	-

Table 1. State space generation

We compared our implementation to Maria[10][4] and Helena [5], both well known in the domain of High Level Petri Net Model Checking. Thanks to clustering and unfolding enabled (partial or total) AlPiNA outperforms the other tools. Results are not as good as those for P/T nets from [7] because of the unfolding cost and because some symmetries cannot be exploited due to token heterogeneity (can be improved using a symbolic-symbolic approach). However, user gains in expressivity and simplicity since APN models are more tractable than their P/T equivalents. We used partial unfolding by ignoring naturals and total unfolding with a bound that limits naturals numbers to the number of databases. The cost of static analysis (included in total time in the table) starts to be prohibitive for 35 databases and thus the partially unfolded version performed better.

When performing explicit model checking, properties can be checked whilst generating the state space. This is not the case with the presented techniques. In the later case, properties are checked once the state space generation is complete. Table 2 shows the time required to check a property. Times for state space generation and property checking have been cumulated. Again, AlPiNA outperforms the competition both in terms of speed and in terms of the size of the model.

	AlPiNA		Maria		Helena	
Model Size	Prop 1 (s)	Prop 2 (s)	Prop 1 (s)	Prop 2 (s)	Prop 1 (s)	Prop 2 (s)
10	1	1.4	49	46	10	9
15	2.9	3.4	-	-	7.5E3	7.5E3
35	76	78	-	-	-	-

Table 2. Property Check

The models we used, our implementation, as well as the programs for Maria and Helena can be found under http://alpina.unige.ch.

9 Conclusion & Future Work

During this article, we provided the readers with a description on the techniques and principles that we use for the model checking of complex system specifications based on algebraic Petri Nets. We proposed methods that not only use symbolic representation on the place structure but also on subcomponents (structural clusters) and on the multi-flow of computation determined by algebraic values (algebraic clusters). These principles combined with an adequate encoding of the states and consequently the state space by means of decision

[10] with parameters –compile tmp -Y -R -Z

diagrams of various natures leads to very efficient model checking algorithms. Convincing benchmarks have been shown.

In this paper, several topics have been briefly mentioned for the coherence of the explanation but never deeply explained. The interested reader is encouraged to consult other papers about decision diagrams [19,3], our recent work on ΣDD for term representation and computation [16], the detailed description of the homomorphisms used for encoding and computing state space of algebraic nets [1] and our prototype tool AlPiNA [12].

Future work are planned in various directions: generalizing the use of ΣDD, model checking modular systems such as COOPN, extension of clusters to hierarchy of clusters, saturation control, and prototype tool implementation.

References

1. Didier Buchs and Steve Hostettler. Toward efficient state space generation of algebraic petri net. Technical report, Centre Universitaire D'Informatique, Université de Genève, January 2009. TR206 - Available as http://smv.unige.ch/technical-reports/TR206-APNClustering.pdf.
2. Ming-Ying Chung and Gianfranco Ciardo. Saturation now. In *QEST '04: Proceedings of the The Quantitative Evaluation of Systems, First International Conference*, pages 272–281, Washington, DC, USA, 2004. IEEE Computer Society.
3. J.M. Couvreur and Y. Thierry-Mieg. Hierarchical decision diagrams to exploit model structure. In *FORTE*, pages 443–457, 2005.
4. Modular reachability analyzer. http://www.tcs.hut.fi/Software/maria/.
5. S. Evangelista C. Pajault. High level net analyzer. http://helena.cnam.fr/.
6. LIP6. Libddd. http://move.lip6.fr/software/libddd.
7. Alexandre Hamez, Yann Thierry-Mieg, and Fabrice Kordon. Hierarchical set decision diagrams and automatic saturation. In *Petri Nets*, pages 211–230, 2008.
8. Wolfgang Reisig. Petri nets and algebraic specifications. In *Theoretical Computer Science*, volume 80, pages 1–34. Elsevier, 1991.
9. K. Jensen. *Coloured Petri Nets*. Springer, Berlin, 1996.
10. Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1 : Equations and Initial Semantics*, volume 6 of *EATC Monographs*. Springer-Verlag, 1985.
11. Ali Al-Shabibi, Didier Buchs, Mathieu Buffo, Stanislav Chachkov, Ang Chen, and David Hurzeler. Prototyping object oriented specifications. In *(ICATPN 2003), LNCS*, volume 2679, pages 473–482. Springer-Verlag, June 2003.
12. SMV. Algebraic petri nets analyzer. http://alpina.unige.ch.
13. A. J. J. Dick and P. Watson. Order-sorted term rewriting. *Comput. J.*, 34(1):16–19, 1991.
14. Enno Ohlebusch. *Advanced topics in term rewriting*. Springer-Verlag, London, UK, 2002.
15. Joseph A. Goguen and José Meseguer. Order-sorted algebra 1: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.
16. Didier Buchs and Steve Hostettler. Sigma decision diagrams : Toward efficient rewriting of sets of terms. In Andrea Corradini, editor, *TERMGRAPH 2009 : Premiliminary proceedings of the 5th International Workshop on Computing with Terms and Graphs*, number TR-09-05, pages 18–32. Universita di Pisa, 2009.

17. H. J. Genrich and K. Lautenbach. The analysis of distributed systems by means of predicate/ transition-nets. *Lecture Notes in Computer Science: Semantics of Concurrent Computation*, 70:123–146, 1979.
18. E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, pages 995–1072. Elsevier, 1990.
19. J.-M. Couvreur, E.Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P. Wacrenier. Data decision diagram for petri nets analysis. In *23rd international conference on application and theory of Petri Nets (ATPN 2002), jun 2002, Australia.*, volume LNCS vol 2360, 2002.
20. R. Bryant. Graph-based algorithms for boolean function manipulation. In *Transactions on Computers, C-35*, pages 677–691. IEEE, 1986.
21. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.
22. M. Presburger. Ueber die vollstaendigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In *Comptes Rendus du I congrès des Mathématiciens des Pays Slaves*, pages 92–101, 1929.
23. Agnès Arnould, Pascale Le Gall, and Bruno Marre. Dynamic testing from bounded data type specifications. In *EDCC-2: Proceedings of the Second European Dependable Computing Conference on Dependable Computing*, pages 285–302, London, UK, 1996. Springer-Verlag.

Model Analysis via a Translation Schema to Coloured Petri Nets

Visar Januzaj[1] and Stefan Kugele[21]

[1] Technische Universität Darmstadt, Fachbereich Informatik,
FG Formal Methods in Systems Engineering,
Hochschulstr. 10, 64289 Darmstadt, Germany
{januzaj, kugele}@forsyte.de
[2] Technische Universität München, Institut für Informatik,
Boltzmannstr. 3, 85748 Garching bei München, Germany
kugele@in.tum.de

Abstract. Model-driven development (MDD) has become a success story and a de facto standard in the development of safety-critical embedded systems. The daily work in the development of such systems cannot be imagined without industry standard CASE tools like e.g. MATLAB/Simulink. Often however, the analysis capabilities of such tools are limited. Therefore, we propose to combine them with the powerful analysis tools developed for Coloured Petri Nets (CPNs).
In this paper, we present a translation schema from COLA—a synchronous data-flow language—to CPNs. We believe this approach to be also feasible for other data-flow languages as long as they have a well-defined syntax and semantics. The combination of both modelling languages allows us to verify properties of COLA models using algorithms and tools designed for CPNs. An example demonstrates the viability of this approach.

Key words: Coloured Petri Nets, Model-driven development (MDD), embedded systems, synchronous data-flow languages

1 Introduction

Embedded systems development is seeing a surge of interest both in academia and industry, this is due to the rapid growth of the market share of embedded systems. About 98% [1] of all processors are nowadays used in embedded systems. Their presence becomes ubiquitous, ranging from portable music players and mobile phones to airbag controllers and flight control systems (FCS). For the first mentioned consumer electronics products properties like reliability, robustness, and correctness are circumstantial. Whereas in the field of automotive or avionics control software, failures of any kind may be fatal or at least result in large warranty costs.

A multitude of different development tools, frequently based on model-driven development (MDD) approaches, have been invented to tackle the complexity

of embedded systems design. Nowadays, modelling large system designs without industry standard CASE tools like e.g. MATLAB/Simulink [2] or SCADE by Esterel Technologies (A380, FCS) [3] cannot be imagined. Here, most different aspects play a major role: due to abstraction and different model views, the complexity apparent to the system's developer is reduced. This helps to reduce design errors by supporting the engineer at the daily work. In the desirable case, that the chosen modelling technique (language) has a well-defined basis, the use of formal methods is facilitated. This improves the quality of the system de novo. In the automotive domain for instance, OEMs are working in a highly competitive mass market, where the time to market is essential for the success of a product and for the company as a last consequence. There, a reduction or—in the best case—the absence of errors detected late in the overall development process considerably saves money and shortens the development process, which in return reduce development costs.

Since the widely used MATLAB/Simulink lacks a formally-defined semantics [4], in a co-operation project together with an industry partner from the automotive domain, the synchronous data-flow language COLA (Component Language) [5] has been developed. During its development, aims like usability, soundness, and reusability were in mind. Around this language, a fully integrated tool [6–14] was created supporting the complete development process ranging from the early requirements engineering, the system modelling, to the system deployment phase.

Coloured Petri Nets (CPNs) [15–17]—similar to COLA—are a graphical modelling language emerged from the combination of Petri Nets [18] and the functional programming language Standard ML (SML) [19, 20]. CPNs and their corresponding computer tools (*CPN Tools* [21]) have been successfully applied in various application areas and industry projects [17], ranging from VLSI chip design, communication protocols [22] to military systems [23–25].

In this paper, we describe how to benefit from both modelling techniques by first translating COLA designs into the CPN formalism and second using analysis techniques and tools applicable to Coloured Petri Nets. This combination allows to augment the well-defined COLA syntax and semantics with the comprehensive CPN Tools.

1.1 Related work

Coloured Petri Nets have been extensively used to model and verify business processes. Gottschalk et al. [26] translated Protos models, i. e. a popular tool for business process modelling, into Coloured Petri Nets for simulation, testing, and configuration reasons. Moreover, in the field of Web services, CPNs are used. There, questions concerning correctness and reliability arise, when composing single Web services to more complex ones. Kang et at. [27] and Yang et al. [28] have studied the translation of WS-BPEL (*Web Services Business Process Execution Language*) or BPEL specifications into CPNs. This allows for analysis and verification of the composed Web services using for example CPN Tools [21]. However their translations are rather informal than a formal defined translation

schema. Hinz et al. [29] translated BPEL specifications into Petri Nets in order to use the model checking tool LoLA to verify relevant properties.

Akin to the presented approach, the authors of [30] bring together the two modelling languages UML and CPN. They translate Use Cases and UML 2.0 Sequence Diagrams into CPN models for formal analysis. In the automotive domain or in the field of embedded systems design in general, *Live Sequence Charts (LSC)* are widely used as specification language. The authors of [31] claim, that LSC do not provide the possibility for analysis and verification and thus a translation into CPN is appropriate and which is given in a well-defined formal way.

1.2 Organisation

The remainder of this work is structured as follows: First, we will give a brief introduction into Coloured Petri Nets in Sect. 2 followed by a more detailed description of the Component Language COLA in Sect. 3. Sect. 4 presents the basic contribution of this work, namely a translation schema from COLA to CPN. An Example demonstrating the feasibility of this approach is given in Sect. 5. Finally, we conclude in Sect. 6.

2 Coloured Petri Nets

Coloured Petri Nets (CPNs) [15–17] belong to the family of high-level Petri nets. Their modelling power strongly relies on the composition between Petri nets and the high-level functional programming language Standard ML (SML) [19,20]. On the one hand, the usage of Petri nets offers an effective framework for *modelling* concurrency, communication, and synchronisation. On the other hand, the application of SML facilitates the *definition* and *manipulation* of the data. In order to be able to cope with the normally large size of real life systems and to introduce a better system overview, CPNs offer the possibility of hierarchically modelling, i. e. parts of the model are combined into submodules. In addition to the possibility of modular modelling, CPNs include a time concept. The integration of the notion of time allows the investigation of especially for distributed, real-time and embedded systems important timing requirements, such as deadline and delay constraints. Furthermore, a set of graphical computer tools is developed to support the modelling, editing, simulation, and analysis of various important properties of systems modelled as CPNs. This set of tools is integrated into the *CPN Tools* [21] framework.

In the following we briefly introduce the definitions of hierarchical and non-hierarchical CPNs. These definitions should serve to easier understand the terminology used for the translation of COLA models. For more detailed and complete definitions see [15–17]. We assume, however, that the reader is familiar with Petri nets and CPNs as well as with the notions such as *multi-sets* (multiple appearances of the same element, denoted *MS*), *marking* (a function mapping each place into a multi-set of tokens of the same colour), *reachability*, *firing*, etc.

Definition 1 (Non-hierarchical CPN (cf. [16])). *A non-hierarchical CPN is a 9-tuple* $\Omega = (P, T, A, \Sigma, V, C, G, E, I)$ *with:*

- *A finite set of* ***places*** P *and* ***transitions*** T *such that* $P \cap T = \varnothing$.
- *A set of directed* ***arcs*** $A \subseteq P \times T \cup T \times P$.
- *A finite set of* ***colour sets*** Σ.
- *A finite set of* ***variables*** V, $Type(v) \in \Sigma, \forall\, v \in V$.
- *A* ***colour set function*** $C : P \to \Sigma$, $C(p) \in \Sigma, \forall\, p \in P$.
- *A* ***guard function*** $G : T \to Expr$, $Type(G(t)) = \texttt{BOOL}, \forall\, t \in T$.
- *An* ***arc expression function*** $E : A \to Expr$, $Type(E(a)) = C(p)_{MS}, \forall\, a \in A$ *and* a *is connected to* $p \in P$.
- *An* ***initialisation function*** $I : P \to Expr$, $Type(I(p)) = C(p)_{MS}, \forall\, p \in P$.

In order to easier understand the following definition, we need to introduce some basic notions: *substitution transitions* are transitions that represent an abstraction of a more detailed submodule of a CPN system. The set of places belonging to the preset and the postset of a transition t is denoted $X(t) = {}^\bullet t \cup t^\bullet$. *Socket nodes* are called the places p surrounding a substitution transition t, i. e. $p \in X(t)$. The *socket type* function ST is defined as follows (cf. [16]):

$$ST(p,t) = \begin{cases} in & \text{if } p \in ({}^\bullet t \text{ - } t^\bullet) \\ out & \text{if } p \in (t^\bullet - {}^\bullet t) \\ i/o & \text{if } p \in ({}^\bullet t \cap t^\bullet) \end{cases}$$

Definition 2 (Hierarchical CPN (cf. [16])). *A hierarchical CPN is a 9-tuple* $\Omega_H = (S, SN, SA, PN, PT, PA, FS, FT, PP)$ *with:*

- *A finite set of* ***pages*** S. *Each page is a non-hierarchical CPN:* $\forall\, s \in S, s = (P_s, T_s, A_s, \Sigma_s, V_s, C_s, G_s, E_s, I_s)$, *the set of net elements of each page pair are disjoint.*
- *A set of* ***substitution nodes*** $SN \subseteq T$ - *the set of all transitions in* Ω_H.
- *A* ***page assignment function*** $SA : SN \to S$, *such that no page is a subpage of itself.*
- *A set of* ***port nodes*** $PN \subseteq P$ - *the set of all places in the entire* Ω_H.
- *A* ***port type function*** $PT : PN \to \{in, out, i/o, general\}$.
- *A* ***port assignment function*** $PA : SN \to Pot(X(SN) \times PN)$ *such that:*
  - *The relation between socket nodes and port nodes is defined as follow:* $\forall\, t \in SN$: $PA(t) \subseteq X(t) \times PN_{SA(t)}$.
  - *Correct types for socket nodes are required:* $\forall\, t \in SN, \forall (p_1, p_2) \in PA(t) : [PT(p_2) \neq general \Rightarrow ST(p_1, t) = PT(p_2)]$.
  - *Related nodes have the same colour set and initialisation:* $\forall\, t \in SN, \forall (p_1, p_2) \in PA(t) : [C(p_1) = C(p_2) \wedge I(p_1) <> = I(p_2) <>]$.
- *A finite set of* ***fusion sets*** $FS \subseteq P_s$, *such that all elements have the same colour set and initialisation.*
- *A* ***fusion type function*** $FT : FS \to \{global, page, instance\}$, *such that page and instance fusion sets belong to a single page.*
- *A multi-set of* ***prime pages*** $PP \in S_{MS}$.

3 COLA—The Component Language

The key concept of COLA is that of *units*. Consequently, all COLA models are built up by simple units—so-called *basic blocks*. They are at the lowest level in a hierarchy of composed units. Those composed units are called *networks* and are used to build up more complex data-flow networks. The basic blocks are atomic, i. e. they cannot be further decomposed. They define the basic (arithmetic and boolean) functions of a system. Environmental interaction is given via typed *ports*. Port to port communication is established via *channels*. According to the synchronous paradigm, which COLA is based on, computation and communication takes no time. Thus, cycles realised by channels that connect an output port to an input port of the same network have to contain at least a *delay* block. It has an initial value and defers value propagation by one *clock tick*. This construct is well-suited to realise memory and feedback loops often used in control systems.

In addition to basic blocks and networks, units can be decomposed into *automata*, i. e. finite state machines similar to Statecharts [32]. The behaviour in each state is again determined by units corresponding to each of the states. This capability is well-suited to express disjoint system modes, also called *operating modes* (cf. [33, 34]).

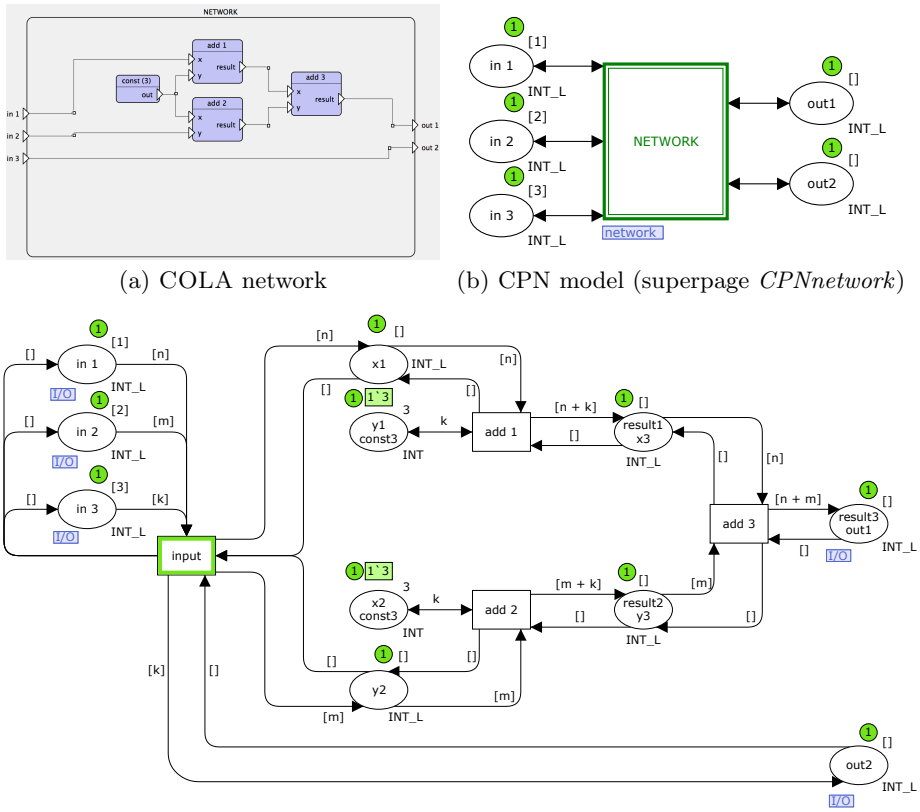
Figure 1(a) shows a COLA network consisting of a couple of add blocks and a const block with value 3. An impression of a similarly behaving CPN is given in Fig. 1(c). The top level of the latter is depicted in Fig. 1(b). In the following, we give a formal definition of each COLA language construct in order to develop a formal translation schema into Coloured Petri Nets.

3.1 COLA in Detail

COLA models are build up by very few, but powerful primitives. The definition of COLA language elements is rather short and builds upon other industry standard data-flow language elements found in MATLAB/Simulink, SCADE, or Lucid Synchrone. COLA's advantage, however, is the reduction to only the bare necessities. Moreover, this slender syntactical core is well-defined and provides a rigourous semantics. The authors of COLA [5] defined the semantics by providing an interpreter for COLA that can be seen as a reference implementation. Moreover, a graphical as well as a textual syntax definition is given in the mentioned article. This formal framework is required, e. g. to simulate the model in a well-defined way, or to perform static analysis like type checking and behavioural verification.

The following sections provide a definition of each syntactical language element.

Units define a relation between its typed input and output *ports*. Ports are used for environment interaction. The combination of all input ports P_{in} =



(a) COLA network

(b) CPN model (superpage *CPNnetwork*)

(c) CPN model (subpage *network*)

Fig. 1. (a) A COLA network consisting of a couple of basic blocks. (b) The top level of the corresponding CPN with the same behaviour (c).

$\langle i_1 : t_1, \ldots, i_m : t_m \rangle$ where t_j, $1 \leq j \leq m$, is the type of port i_j, and all output ports $P_{out} = \langle o_1 : t_{m+1}, \ldots, o_n : t_{m+n} \rangle$, with $m, n \in \mathbb{N}$, defines the unit's *signature* $\sigma = (P_{in} \rightarrowtail P_{out})$.

A unit u is defined as a 3-tuple $\langle n, \sigma, I \rangle$ where n is its name, σ defines the signature, and I specifies the actual implementation. A unit can be considered as the superclass for special types of units, namely *functional block*, *timing block*, *network*, and *automaton*. Depending on the used type, the implementation I is chosen adequately. A detailed description of the different unit types is presented next.

Functional blocks can realise a multitude of different operators: first, fundamental arithmetic operations ($+$, $-$, $/$, $*$) can be used. Second, COLA provides the basic comparison operators ($=$, $\neq$, $<$, $\leq$, $>$, and $\geq$). Third, Boolean connectivities are supported ($\wedge$, $\vee$, and $\neg$). A functional block u is defined as follows:

$u = \langle n, (\langle lop : t, rop : t\rangle \rightarrowtail \langle result : b\rangle), I\rangle$. All operations are binary with input ports *lop* (*left operand*) and *rop* (*right operand*) and provide a result port *result*. The type b of the result depends on the operation. For arithmetic operations holds that b is equal to t, e. g. `Int` or `Real`. All other operators return a result value of type `Boolean`. Their implementation I is defined by the used operator, i. e. the functional block `add`, cf. Fig. 2(a), (operator $+$) for example is mathematically defined and implemented as $result := lop + rop$.

Delays (timing blocks) retaining a value for a single time unit (tick) and thereby provide a low-level realisation of variables as found in high-level programming languages. This is indispensable in the context of feedback control system. There, computed values have to be stored and fed-back as input for the next clock tick. Initially, delays are initialised with a constant as default value. Each cycle in a network has to contain at least one delay. Otherwise, the modelled system cannot be interpreted.

A delay is a unit, defined in the following way: $d = \langle n, \sigma, I\rangle$, with n being an identifier, the signature $\sigma = (\langle in : t\rangle \rightarrowtail \langle out : t\rangle)$ and an implementation I defined as its valuations over the infinite sequence of discrete time steps $(s_j)_{j\in\mathbb{N}_0}$

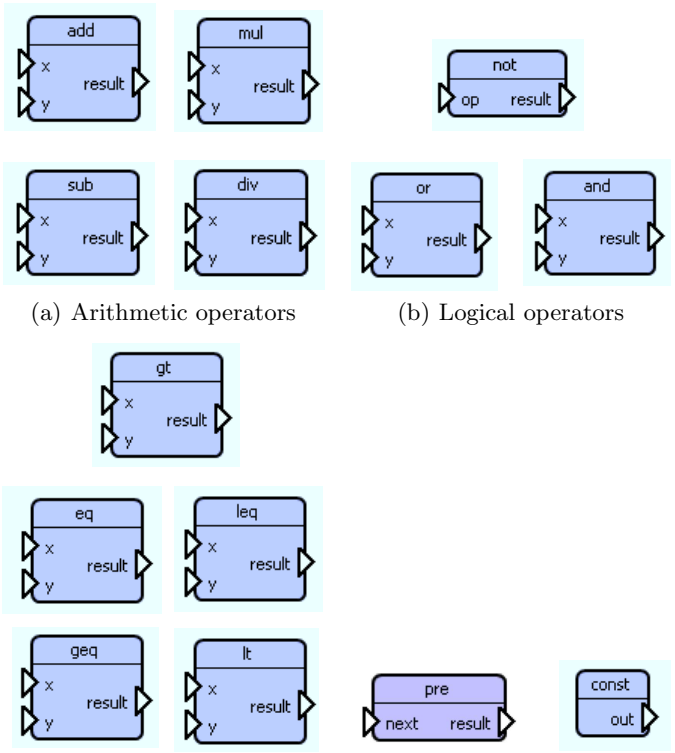
$$out[s_j] := \begin{cases} default & \text{if } j = 0 \\ in[s_{j-1}] & \text{if } j > 0 \end{cases}$$

where $in[s_j]$ indicates the value of the input port at time step s_j and $out[s_j]$ that of the output port, respectively.

For basic blocks, i. e. functional blocks and timing blocks, the concrete graphical syntax is depicted throughout the Fig(s) 2(a), 2(b), 2(c), and 2(d).

Networks are used to structure the overall system. They are used to provide a high-level system view in order to abstract from implementation details and thus reduce the complexity apparent to the developers. By descending, or decomposing a network, the initial hidden implementation becomes visible. They are realised using so-called *channels* to interconnect a set of units and build up larger data-flow networks. A channel c is a triple $c = \langle n, s, \{d_1, \ldots, d_k\}\rangle$ with n being the identifier and s is the source port which is connected to a set of destination ports $d_i \in \{d_1, \ldots, d_k\}$, with $1 \leq i \leq k$. Together with the included sub-units, networks are defined as: $\langle n, \sigma, \langle U, C\rangle\rangle$ where n is the identifier, σ is the signature as defined for units, and the implementation consists of a set of units U contained in the network together with the set of interconnecting channels C.

The graphical syntax of networks can be learned from the example given in Fig. 1(a).



(a) Arithmetic operators (b) Logical operators

(c) Comparison operators (d) Timing and constant block

Fig. 2. Basic blocks provided by COLA: (a) arithmetical operators, (b) logical operators, (c) comparison operators, and (d) timing block (delay or pre) and the constant block.

Automata are special units, whose implementation is a finite automaton with states and transitions guarded by predicates. Both, states and guards are itself implemented by units: a state's behaviour is defined by a network, the guards are stateless networks, i. e., networks without occurrences of automata and delays since these units are statefull. They have to store their current state in the case of an automaton and their last value in the case of a delay for at least one execution cycle.

Formally, an automaton is a unit $\langle n, \sigma, I \rangle$ with identifier n and signature σ. The implementation of an automaton is given by: $I = \langle Q, q_o, \Delta \rangle$, where Q is a finite set of state labels, that refer to the names of units, which implement the state's behaviour. Their signature is equal to that of the automaton. q_0 is the name of the initial state and $\Delta \subseteq Q \times \mathsf{dom}(\mathsf{in}(\sigma)) \times Q$ is the transition relation. $\mathsf{dom}(\mathsf{in}(\sigma))$ is defined as

$$\mathsf{dom}(\mathsf{in}(\sigma)) \stackrel{def}{=} \mathsf{dom}(\mathsf{type}(a_1)) \times \ldots \times \mathsf{dom}(\mathsf{type}(a_k))$$

where $\mathsf{dom}(\mathsf{type}(a_i))$ denotes the domain of the typed ports $a_i \in \mathsf{in}(\sigma), 1 \leq i \leq k$. $\mathsf{in}(\sigma)$ defines the projection onto the input ports of the signature and respectively $\mathsf{out}(\sigma)$ defines the projection onto the output ports.

Starting from the initial state, the semantics is defined as follows: let q be the current state, if there is an outgoing transition whose guard evaluates to *true*, take it and execute the unit referenced by the target state. If there is no such transition predicate evaluating to *true*, execute the unit referenced by the current state.

An example for the graphical syntax of a COLA automaton is depicted in Fig. 3.

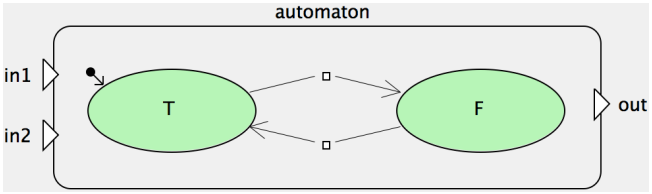
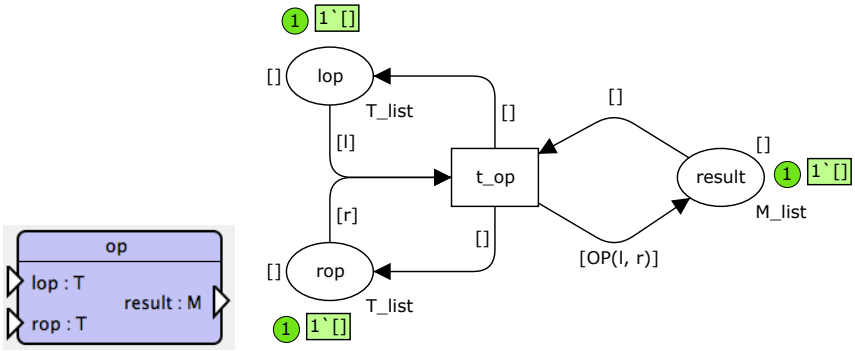


Fig. 3. A COLA automaton with two states T (initial state) and F and from each state a transition to the other one.

4 Translation Schema

In the following, a translation schema from COLA to CPNs is proposed. Therefore, each language construct is translated one after another. Beginning with basic units, namely functional blocks, we give stepwise more and more complex translation schemas for units like networks and automata. We will, however, not introduce translation schemas neither for constant blocks nor for delay units as their translation is straightforward: *constant blocks* – are translated into a single CPN place initialised with the corresponding value, *delays* – for each input and output we generate a separate CPN place as well as a transition to connect them. The input place holds the initialisation value. A translation of a delay, e.g. `pre_1`, can be found in Fig. 8(a). Note that the delay is modelled as a substitution transition. The translation described above is modelled in its subpage (not visible in Fig. 8(a)). Keeping the same structure as the original COLA model should serve for a better understanding of the translation process.

For the translation we use both hierarchical and non-hierarchical CPNs. Units that can be decomposed are translated into hierarchical CPNs, those that cannot into non-hierarchical CPNs. In order to make sure that no value is written into a non-empty place, we define input and output places (and where necessary) as lists of a given data type. Thus, apart from other constraints, each transition connected to such places fires only if its postset is empty. This reflects the behaviour defined in COLA.



(a) COLA (b) CPN

Fig. 4. (a) COLA basic block with two input ports of type T and an output port of type M. (b) Corresponding CPN with three places and one transition.

Before we start with the definition of the translation schema we need first to define a function $\pi : io(\sigma) \rightarrow P$ which maps the set of COLA input and output ports into the set of CPN places, with $\mathsf{io}(\sigma) = in(\sigma) \cup out(\sigma)$.

4.1 Functional Block

In Fig. 4 a COLA functional block and its translation is depicted. The translation schema for a functional block is defined as shown in Fig. 5. Since functional blocks cannot further be decomposed their translation is straightforward. Input and output ports are transformed into CPN places (P), including their corresponding data types (C). A transition (T) is generated to reflect the operation OP and is accordingly connected to places by arcs (A). Arc inscriptions (E) matching the empty list [ ] play a key role for the generated CPN model. On the one hand, they force the transition to fire only if its postset is empty, cf. $a = (result, t_{op})$ in Fig. 4(b). In this way the behaviour defined in COLA is reflected, i. e. no new value is added to an output port unless old values are consumed. On the other hand, they notify other modules connected to them, that the data residing in the input ports has been consumed (cf. the outgoing arcs from t_{op}), i. e. new values can proceed. To achieve this we define lists of used data types (Σ). Variables (V) corresponding to a data type are used to read the input and process the data according to the operation OP, cf. the arc inscription of $a = (t_{op}, result)$ in Fig. 4(b). The guard (G) of the transition is always `true`. All places are initialised (I) with the empty list.

4.2 Network

In COLA networks can consist of a large number of subunits. An example of a network is depicted in Fig. 1. We will describe only the translation of the highest level of a network.The translation schema for COLA data-flow networks

FunctionalBlock

A COLA functional block

$$\mathsf{FB} \;=\; \langle n, \sigma = (\langle lop : tt, rop : tt\rangle \rightarrowtail \langle result : m\rangle), \mathsf{OP}\rangle$$

is translated into a CPN

$$cpn = (P, T, A, \Sigma, V, C, G, E, I)$$

using the following schema:

Schema

$P = \pi(\mathsf{in}(\sigma) \cup \mathsf{out}(\sigma))$, i.e. $\{lop, rop\} \cup \{result\}$

$T = \{t_{op}\}$

$A = \{(lop, t_{op}), (rop, t_{op}), (t_{op}, result), (t_{op}, lop), (t_{op}, rop), (result, t_{op})\}$

$\Sigma = \{tt\_l, m\_l\}$, tt_l and m_l are lists of type tt and m, resp.

$V = \{l : tt, r : tt\}$

$$C(p) = \begin{cases} tt\_l & \text{if } p \in \{lop, rop\} \\ m\_l & \text{if } p \in \{result\} \end{cases}$$

$G(t) = \mathtt{TRUE}, \forall\ t\ \in\ T$

$$E(a) = \begin{cases} [l] & \text{if } a = (lop, t_{op}) \\ [r] & \text{if } a = (rop, t_{op}) \\ [\mathsf{OP}(l, r)] & \text{if } a = (t_{op}, result) \\ [\,] & \text{if } a \in \{(t_{op}, lop), (t_{op}, rop), (result, t_{op})\} \end{cases}$$

$I(p) = [\,], \forall\, p \in P$

Network

A COLA network

$$\mathsf{NET} \;=\; \langle n, \sigma, \langle U, C\rangle\rangle$$

is translated into a hierarchical CPN

$$Hcpn = (S, SN, SA, PN, PT, FS, FT, PP)$$

using the following schema:

Schema

$S = \{CPNnetwork\} \cup S_U$

$SN = \{n_{\mathsf{NET}}\} \cup SN_U$, n_{NET} is the identifier of NET

$SA(SN) = \bigcup_{s \in SN} SA(s)$

$PN = \pi(\mathsf{io}(\sigma)) \cup PN_U$

$$PT(p) = \begin{cases} i/o & \text{if } p \in \pi(\mathsf{io}(\sigma)) \\ PT(PN_U) & \text{if } p \in PN_U \end{cases}$$

$$PA(t) = \begin{cases} \{(in_1@CPNnetwork, in_1@network), \\ (in_2@CPNnetwork, in_2@network), \ldots, \\ (out_1@CPNnetwork, out_1@network), \ldots\} & \text{if } t = n_{\mathsf{NET}} \\ PA(SN_U) & \text{if } t \in SN_U \end{cases}$$

$PP = 1\text{`}CPNnetwork$

Note: FS and FT are not considered during the translation.

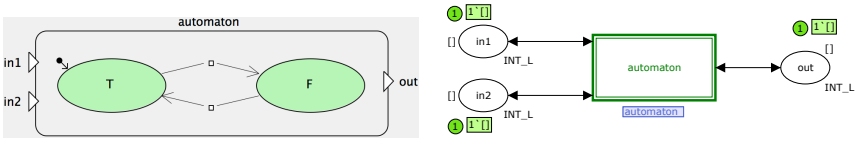
Fig. 5. Functional Block and Network Schema.

is defined as shown in Fig. 5. Each network is translated into a corresponding hierarchical CPN. For the top level of each COLA network a page, the super page, is generated, e. g. *CPNnetwork* in Fig. 1(b). In the following we will refer to the COLA and CPN models and their components in Fig. 1, when necessary to achieve a better understanding of the translation process. The set of other pages, representing the implementation $I = \langle U, C \rangle$ of the network, are included in S_U, where U is the set of subunits participating in the network. Each of these units is separately translated corresponding to its schema type. The translation of the set of channels C is not explicitly given. However, they are important for establishing the connectivity between translated components, e. g. if there is a connection/channel from a COLA unit A to a unit B, in the corresponding CPN model the output places of A are correspondingly glued together with the input places of B (unless some other criteria apply). Each subunit is represented by the set of substitution transitions SN, which consists of a transition, e. g. $n_{\mathsf{NET}} = \mathsf{NETWORK}$, and the set of those ($SN_U$) appearing in the subunits in U. SA maps each substitution transition to their implementations in the subpages, e. g. transition NETWORK to the subpage *network*. The set of input and output nodes of *CPNnetwork* (in_1, in_2, ...) are unified with those of the subpages PN_U building the set PN. Most of port nodes are of type (PT) *i/o* as described in the schema. Now we just need to define the assignment (PA) of port nodes to socket nodes, e. g. in_1 in *network*, denoted in_1@*network*, is assigned to in_1 in *CPNnetwork* (in_1@*CPNnetwork*). Since the nodes in both pages share commonly the same name, the tuple (out_1@*CPNnetwork*, $result_3 out_1$@*network*) illustrates best such an assignment.

4.3 Automaton

Automata are the most complex units of COLA. Figure 6 shows a COLA automaton and its CPN representation. For the translation of an automaton we introduce a two-step schema (cf. Fig. 6). In the first step we describe the highest abstraction level as a hierarchical CPN. In the second step the functionality of the automaton, i.e. guard evaluation and state switching, is described as a non-hierarchical CPN. For each state of the automaton, e.g. `T` and `F`, there exists a separate transition, which serves as a substitution transition for the implementation of the underlying network unit (cf. Fig. 8(b)). The same figure would represent also the functionality of the automaton in Fig. 6, by only replacing `do_nothing` and `working` with `T` and `F`, respectively. The first translation step is similar to the translation of a network, thus we give no further description. We have, however, to stress that the index Q represents the implementation of the underlying network for each automaton state in Q. The set of their corresponding substitution transitions is denoted Q_T.

The second schema describes by means of non-hierarchical CPNs the next lower level page (*automaton)*. Besides the port nodes, determined by $\pi()$, which are needed to be assigned to sockets of the parent or super page (*CPNautomaton*), there are two additional places *State* and *Activated* added to the set of places P. Place *State* is of type *State* and holds the identifiers of each state



(a) COLA automaton (b) CPN page *CPNautomaton*

Automaton

A COLA automaton
$\mathsf{AUT} = \langle n, \sigma, I\rangle$, $I = \langle Q, q_0, \Delta\rangle$, $\Delta \subseteq Q \times \mathsf{dom}(\mathsf{in}(\sigma)) \times Q$
is translated into a hierarchical CPN

$$Hcpn = (S, SN, SA, PN, PT, FS, FT, PP)$$

using the following schema:

*Schema*1

$S = \{CPNautomaton\} \cup S_Q$
$SN = \{n_{\mathsf{AUT}}\} \cup SN_Q$, n_{AUT} is the identifier of AUT
$SA(SN) = \bigcup_{s \in SN} SA(s)$
$PN = \pi(\mathsf{io}(\sigma)) \cup PN_Q$

$$PT(p) = \begin{cases} i/o & \text{if } p \in \pi(\mathsf{io}(\sigma)) \\ PT(PN_Q) & \text{if } p \in (PN_Q) \end{cases}$$

$$PA(t) = \begin{cases} \{(in_1@CPNautomaton, in_1@automaton), \\ (in_2@CPNautomaton, in_2@automaton), \ldots, \\ (out_1@CPNautomaton, out_1@automaton), \ldots\} & \text{if } t = n_{\mathsf{AUT}} \\ PA(SN_Q) & \text{if } t \in (SN_Q) \end{cases}$$

$PP = 1`CPNautomaton$

automaton represents the subpage of n_{AUT}.
Note: FS and FT are not considered during the translation.

*Schema*2 − *page* automaton

$P = \{State, Activated\} \cup \pi(\mathsf{in}(\sigma) \cup \mathsf{out}(\sigma))$
$T = \{activate\ State\} \cup Q_T$, $A = \{(State, activate\ State), (activate\ State, State), (activate\ State, Activated), (Activated, activate\ State), \ldots\}$
$\Sigma = \{State, State\_L, \} \cup \mathsf{D}$, with $\mathsf{D} = \mathsf{dom}(\mathsf{in}(\sigma))$
$V = \{s : State\} \cup \{v_1 : t_1, \ldots, v_n : t_n\}, t_i \in \mathsf{D}, 1 \leq i \leq n, n =\mid \mathsf{in}(\sigma) \mid$

$$C(p) = \begin{cases} State & \text{if } p = State \\ State\_L & \text{if } p = Activated \\ \mathsf{D} & \text{if } p \in \pi(\mathsf{io}(\sigma)) \end{cases}$$

$G(t) = \texttt{TRUE}, \forall\ t\ \in\ T$

$$E(a) = \begin{cases} s & \text{if } a = (State, activate\ State) \\ state(s, \{v_1, v_2, \ldots\}) & \text{if } a = (activate\ State, State) \\ [state(s, \{v_1, v_2, \ldots\})] & \text{if } a = (activate\ State, Activated) \\ \ldots & \end{cases}$$

$I(State) = q_0$

Fig. 6. Exemplary (a) COLA automaton with two states and (b) its translation, and the translation schema.

in Q. *Activated* holds the currently active state. The transition *activateState* is responsible for the initialisation of state switching, by feeding the function *state()* with input data and the actual active state. The purpose of function *state()* is to check and control the switching between states, according to the defined guards of the automaton. How this works has already been described in Sect. 3.1. Let $G = \{g_1, g_2, \ldots, g_n\}$ be the set of the guards of an automaton, $V = \{v_1, v_2, \ldots, v_m\}$ the set of variables used in the guards and $S = \{s_1, s_2, \ldots, s_i\}$ the set of states of the automaton, with $s \in S$. We define the *state*() function and the colour sets *State* and *State_L* as follows:

```
fun state(s, v_1, ..., v_m) = if s = s_1 andalso g_1 then s_2
                              else
                              if s = s_2 andalso g_2 then s_3
                              ...
                              else s;
colset State   = with s_1 | s_2 | ... | s_i;
colset State_L = list State;
```

The rest of the translation schema is straightforward.

4.4 Translation Algorithm

The idea of the outlined Algorithm 1 is to translate COLA models into CPNs in a DFS manner. For each visited unit, the corresponding translation schema is applied. Once all units are translated there will be loose components and a superfluous number of places (representing each input and output port of each component). To reduce the number of places and establish the corresponding connectivity between components, we glue together input and output places (cf. line 19) regarding the defined channels in the original COLA model, i. e. the corresponding source and destination ports. Finally, to accommodate the structure of the generated hierarchical CPN, the connection between subpages and their parent pages is established by assigning ports to sockets (cf. line 20).

There are two special cases that need to be considered during the translation of a network: first, if multiple ports read from one and the same port (cf. port *out* of the constant block in Fig. 1(a)). In this case, we translate the connection in that way that the source of the channel is translated to as many places as there are destinations (cf. places $y_1\,out_3$ and $x_2\,out_3$ in Fig. 1(c)). Second, the input and output of a unit are not connected (cf. Fig. 7 the implementation of the `do_nothing` state). Therefore, we create a *new* place and connect it with the *input* transition and other transitions accordingly (cf. Fig. 8(c)). This is done to make sure that the data flow in the network is not broken, i. e. we want to establish a correct consumption of the input data in order to proceed to the output, as required in COLA.

Note that one can merge the transitions *input* and *out*, thus not needing to add the *new* place at all. The transition *input* can often get merged with other transitions and reduce the size of the net, e. g. one could merge *input* and

Algorithm 1: COLA2CPN

```
Data: COLA model
Result: CPN model
while (not all units u ∈ U have been visited) do
    perform a DFS traversal on the COLA model;
    switch (u instanceof) do
        case (functional block)
            if (u isA constant) then
                create a single place p;
                initialise p accordingly;
            else
                FunctionalBlock(u);
        case (network)
            Network(u);
            create a transition input to collect the incoming data;
            connect input according to the connections in u (channels);
        case (automaton)
            Automaton(u);
        case (delay)
            Delay(u);
            initialise the translation
glue input and output places together, according to their connectivity in the
    COLA model;
assign ports to sockets;
```

add (cf. Fig. 8(d)) and deleting the places *in_1* and *in_2*, without changing the behaviour of the net.

5 Example

In Fig. 7, a screenshot of the COLA simulation tool is depicted. It shows a high level COLA system consisting basically of two automata, two input constants and two delay operators (pre). Each automaton has two states, namely do_nothing and working. In both cases, do_nothing always provides the value 0 as output, concerning the behaviour of working, however, both automata show a different implementation. automaton_1 performs the subtraction of the values present at the input ports in_1 and in_2 (out := in_1 - in_2). The state working of automaton_2 increases the input value at port in_1 by 3 (out := in_1 + 3). The modelled transition relations are omitted for the sake of clarity.

In COLA a deadlock in a classical sense is not possible. This is due to the fact that the COLA semantics dictates that at each tick of the system execution a new value is assigned to each output port. A deadlock from a Petri net point of view is compared best with a COLA system, that is stuck in an automaton

state, which cannot be changed anymore. This might, however, be a system design decision. But in many cases, as in the given example, it is a modelling error. Regarding the example, the values present at the output ports result of both delays have a special behaviour: at the first tick both ports emit the value 1. To simplify matters, we write these values as a result vector $\mathbf{r} = \left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)$ where the upper value corresponds to the port value of the upper delay, and the lower value to the lower delay, respectively. Both values have been set by the developer as default values for the delays. When considering the behaviour over time we use a matrix-like notation, i. e., the ith column of the matrix represents the output values after the ith tick: $\mathbf{M}^{\infty} = \left(\begin{smallmatrix}1 & 2 & -3 & -4 & 0 & 0 \ldots\\ 1 & 6 & 7 & 0 & 0 & 0 \ldots\end{smallmatrix}\right)$. For this simple example, the following infinite sequence

$$\mathbf{M}^{\infty} = \begin{pmatrix}1 & 2 & -3 & -4\\ 1 & 6 & 7 & 0\end{pmatrix} \circ \begin{pmatrix}0\\0\end{pmatrix}^{\omega}$$

of port valuations is obtained, i. e., after a finite number of steps (four in this case) the system reaches a deadlock-like state and from then on only emits $\mathbf{r} = \left(\begin{smallmatrix}0\\0\end{smallmatrix}\right)$ as result. However, for more complex examples, similar behaviour cannot be detected by the developer by solely using the COLA simulator. Here, the power of the CPN Tools becomes important.

After translating the COLA model into a CPN, using the outlined translation algorithm, the CPNs depicted in Fig. 8 are obtained. The idea is to automatically construct the state space of the CPN models and finally create the *state space report* which contains information about standard behaviour properties: dead markings, dead and live transitions, etc. These information collected in the report support the analysis of a system in an early stage of its development and help to decreases the number of design errors. Furthermore, one can check other specific behavioural properties by using predefined *query functions* provided by CPN Tools to write user-defined analysis algorithms. For our example, the CPN Tools reported a set of live transitions shown as an excerpt of the report below.

```
Live Transition Instances
--------------------------------
    automaton1'activate State 1
    automaton2'activate State 1
    doNothing1'input 1
    doNothing1'out 1
    doNothing2'input 1
    doNothing2'out 1
    pre1'delay 1
    pre1'init 1
    pre2'delay 1
    pre2'init 1
```

The expected behaviour is reflected in this result to the effect that the transitions representing both working states are not contained. That means for the CPN, that there is a marking from which there exists no path containing these

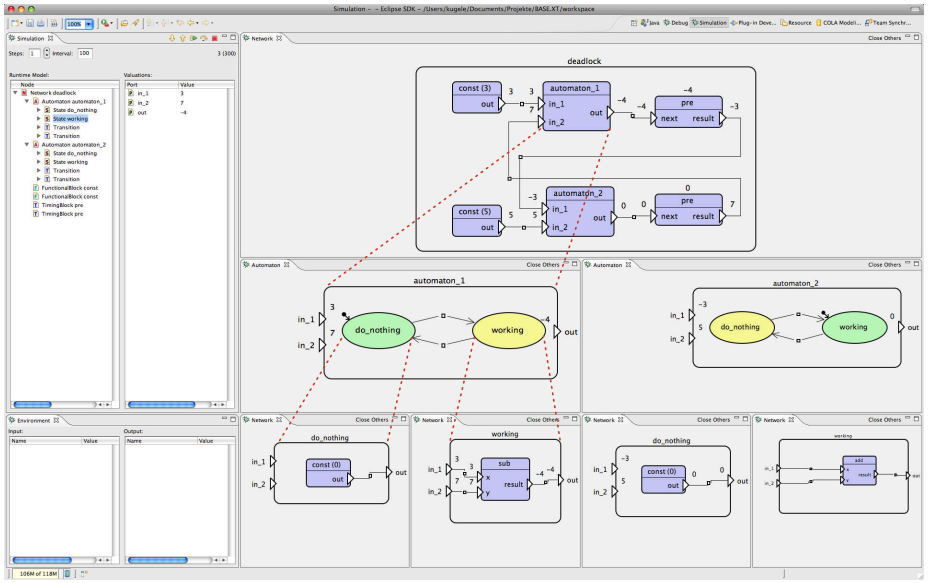
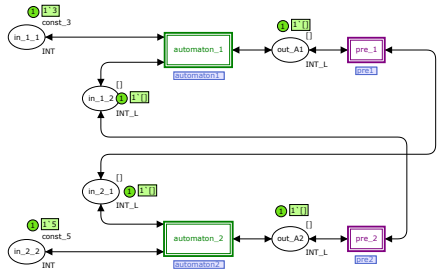
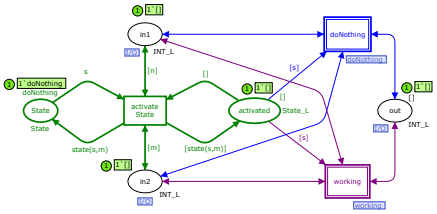


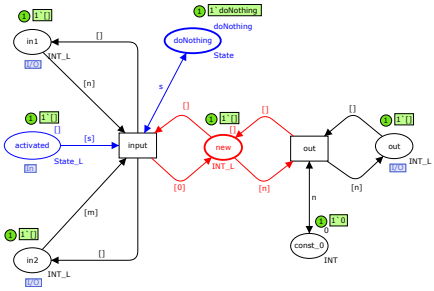
Fig. 7. COLA Simulator: Dashed lines are added manually to clarify the hierarchical decomposition.



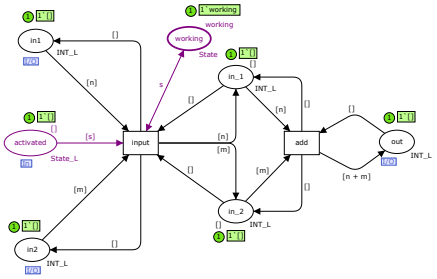
(a) CPN example



(b) Automaton



(c) State *do nothing*



(d) State *working*

Fig. 8. CPN example: (a) The highest abstraction level of the CPN example. (b) Realisation of an automaton. (c) Realisation of the state *do nothing*. (d) Realisation of the state *working* (*automaton_2*).

transitions. In other words—from a COLA point of view—it is possible to reach a system state that prohibits a change to a distinguished system state (working in our case). Based on this information, the developer has to check whether the modelled system behaviour is what was desired. If this it not the case, a modelling error has been detected. This is only one of the many observation one can receive, i. e. by far not all what an analysis process can yield. The intention of the analysis example was however to show how helpful can be such analysis result. Being beyond the scope of this paper, the analysis of COLA models will not be discussed.

An issue, however, remains the state space explosion problem. To alleviate it a number of state space reduction methods (symmetry, equivalence, sweep line) have been developed and integrated into CPN Tools. Furthermore, Khomenko et al. [35] presented an improvement of the *unfolding* technique which can be applied to all classes of high-level Petri nets. Based on this work, in [36] a prototype has been proposed and developed for unfolding a subclass of n-safe CPNs. In the ASCoVeCO project [37] a platform (ASAP) is being developed aiming for the integration of various analysis methods into one environment, as well as giving the possibility to extend the existing tool collection, thus increasing the analysis possibilities for CPNs.

Our translation concept can only profit from such an analysis environment, facilitating broader analysis aspects for COLA models in return.

6 Conclusions

In this paper we introduced a mathematically sound schema for the translation of the synchronous data-flow language COLA into Coloured Petri Nets. This translation schema allows the combining of the strengths of both modelling techniques to have a powerful model analysis methodology at hand. The toy example presented here showed the applicability of the presented approach by providing hints for possible design errors. A possible future extension could be the use of Timed Coloured Petri Nets to fit better the synchronous paradigm that COLA follows. Furthermore, we want to facilitate the automatically translation of COLA models into CPNs. This will allow us to deal with and analyse more interesting, larger and real-life systems modelled in COLA.

We believe that this approach is feasible to be also applied to other synchronous data-flow languages, like Lustre for instance.

Acknowledgments

We would like to thank Andreas Holzer and the anonymous reviewers for their fruitful comments in finalising this paper.

References

1. Broy, M.: Automotive software and systems engineering (panel). In: MEMOCODE. (2005) 143–149

2. The MathWorks Inc.: Using Simulink. (2000)
3. Berry, G., Gonthier, G.: The esterel synchronous programming language: design, semantics, implementation. Sci. Comput. Program. **19**(2) (1992) 87–152
4. Tripakis, S., Sofronis, C., Caspi, P., Curic, A.: Translating discrete-time simulink to lustre. Trans. on Embedded Computing Sys. **4**(4) (2005) 779–818
5. Kugele, S., Tautschnig, M., Bauer, A., Schallhart, C., Merenda, S., Haberl, W., Kühnel, C., Müller, F., Wang, Z., Wild, D., Rittmann, S., Wechs, M.: COLA – The component language. Technical Report TUM-I0714, Institut für Informatik, Technische Universität München (September 2007)
6. Kugele, S., Haberl, W.: Mapping Data-Flow Dependencies onto Distributed Embedded Systems. In: Proceedings of the 2008 International Conference on Software Engineering Research & Practice, SERP 2008, Las Vegas, Nevada, USA (July 2008)
7. Wang, Z., Haberl, W., Kugele, S., Tautschnig, M.: Automatic Generation of SystemC Models from Component-based Designs for Early Design Validation and Performance Analysis. In: Proceedings of the 7th International Workshop on Software and Performance, WOSP 2008, Princeton, NJ, USA, ACM (June 2008) 23–26
8. Kugele, S., Haberl, W., Tautschnig, M., Wechs, M.: Optimizing automatic deployment using non-functional requirement annotations. In Margaria, T., Steffen, B., eds.: Leveraging Applications of Formal Methods, Verification and Validation. Volume 17 of CCIS., Springer (2008) 400–414
9. Haberl, W., Tautschnig, M., Baumgarten, U.: Running COLA on Embedded Systems. In: Proceedings of The International MultiConference of Engineers and Computer Scientists 2008. (March 2008)
10. Herrmannsdoerfer, M., Haberl, W., Baumgarten, U.: Model-level Simulation for COLA. In: International Workshop on Modeling in Software Engineering (MISE'09: ICSE Workshop 2009). (2009)
11. Haberl, W., Birke, J., Baumgarten, U.: A Middleware for Model-Based Embedded Systems. In: Proceedings of the 2008 International Conference on Embedded Systems and Applications, ESA 2008, Las Vegas, Nevada, USA (July 2008)
12. Haberl, W., Tautschnig, M., Baumgarten, U.: From COLA Models to Distributed Embedded Systems Code. IAENG International Journal of Computer Science **35**(3) (September 2008) 427–437
13. Kühnel, C., Bauer, A., Tautschnig, M.: Compatibility and reuse in component-based systems via type and unit inference. In: Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), IEEE Computer Society Press (2007)
14. Wang, Z., Sanchez, A., Herkersdorf, A.: Scisim: a software performance estimation framework using source code instrumentation. In: Proceedings of the 7th international workshop on Software and performance (WOSP '08), New York, NY, USA, ACM (2008) 33–42
15. Jensen, K.: Coloured Petri nets (2nd ed.): basic concepts, analysis methods and practical use, volume 1. Springer-Verlag, London, UK (1996)
16. Jensen, K.: Coloured Petri nets: basic concepts, analysis methods and practical use, volume 2. Springer-Verlag, London, UK (1997)
17. Jensen, K.: Coloured Petri nets: basic concepts, analysis methods and practical use, volume 3. Springer-Verlag New York, Inc., New York, NY, USA (1997)
18. Reisig, W.: Petri nets: an introduction. Springer-Verlag New York, Inc., New York, NY, USA (1985)
19. Paulson, L.C.: ML for the working programmer (2nd ed.). Cambridge University Press, New York, NY, USA (1996)

20. Standard ML. http://www.standardml.org/
21. CPN Tools. http://www.daimi.au.dk/CPNTools/
22. Kristensen, L.M., Jensen, K.: Specification and validation of an edge router discovery protocol for mobile ad hoc networks. In: SoftSpez Final Report. (2004) 248–269
23. Kristensen, L.M., Billington, J., Qureshi, Z.: Modelling military airborne mission systems for functional analysis. (2001)
24. Petrucci, L., Billington, J., Kristensen, L.M., Qureshi, Z.H.: Developing a formal specification for the mission system of a maritime surveillance aircraft. In: ACSD '03: Proceedings of the Third International Conference on Application of Concurrency to System Design, Washington, DC, USA, IEEE Computer Society (June 2003) 92–101
25. Qureshi, Z.H.: Formal modelling and analysis of mission-critical software in military avionics systems. In: SCS '06: Proceedings of the eleventh Australian workshop on Safety critical systems and software, Darlinghurst, Australia, Australia, Australian Computer Society, Inc. (2006) 67–77
26. Gottschalk, F., van der Aalst, W.M.P., Jansen-Vullers, M.H., Verbeek, H.M.W.: Protos2cpn: using colored petri nets for configuring and testing business processes. STTT **10**(1) (2008) 95–110
27. Kang, H., Yang, X., Yuan, S.: Modeling and verification of web services composition based on cpn. In: NPC '07: Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing Workshops, Washington, DC, USA, IEEE Computer Society (2007) 613–617
28. Yang, Y., Tan, Q., Xiao, Y., Liu, F., Yu, J.: Transform bpel workflow into hierarchical cp-nets to make tool support for verification. In: APWeb. (2006) 275–284
29. Hinz, S., Schmidt, K., Stahl, C.: Transforming bpel to petri nets. In: Business Process Management. (2005) 220–235
30. Fernandes, J.M., Tjell, S., Jorgensen, J.B., Ribeiro, O.: Designing tool support for translating use cases and uml 2.0 sequence diagrams into a coloured petri net. In: SCESM '07: Proceedings of the Sixth International Workshop on Scenarios and State Machines, Washington, DC, USA, IEEE Computer Society (2007) 2
31. Amorim, L., Maciel, P.R.M., Jr., M.N.N., Barreto, R.S., Tavares, E.: Mapping live sequence chart to coloured petri nets for analysis and verification of embedded systems. ACM SIGSOFT Software Engineering Notes **31**(3) (2006) 1–25
32. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley (1998)
33. Bauer, A., Broy, M., Romberg, J., Schätz, B., Braun, P., Freund, U., Mata, N., Sandner, R., Ziegenbein, D.: AutoMoDe — Notations, Methods, and Tools for Model-Based Development of Automotive Software. In: Proceedings of the SAE 2005 World Congress, Detroit, MI, Society of Automotive Engineers (April 2005)
34. Maraninchi, F., Rémond, Y.: Mode-automata: a new domain-specific construct for the development of safe critical systems. Science of Computer Programming **46**(3) (2003) 219–254
35. Khomenko, V., Koutny, M.: Branching processes of high-level petri nets. In Garavel, H., Hatcliff, J., eds.: TACAS. Volume 2619 of Lecture Notes in Computer Science., Springer (2003) 458–472
36. Januzaj, V.: CPNunf: A tool for McMillan's Unfolding of Coloured Petri Nets. In: Proceedings of 8th Workshop on Practical use of Coloured Petri Nets and the CPN Tools. (2007) 147–166
37. The ASCoVeCO project. http://www.daimi.au.dk/~ascoveco/

Transforming WS-CDL specifications into Coloured Petri Nets *

Valentín Valero, Hermenegilda Macià, and Enrique Martínez

Escuela Superior de Ingeniería Informática.
Universidad de Castilla-La Mancha, 02071 Albacete, SPAIN
{Valentin.Valero, Hermenegilda.Macia, Enrique.Martínez}@uclm.es

Abstract. The Service Oriented Architecture (SOA) allows us to provide more powerful services by composing several Web Services. The W3C has specified WS-CDL (Web Services Choreography Description Language) as a candidate language to describe interactions between different parties in a Web Services composition. Nevertheless, WS-CDL lacks of rigorous formalization of its semantics. In this paper we provide an operational semantics and a Petri net semantics for a relevant subset of WS-CDL, focusing on collaborations and elements related to concurrency. The operational semantics is based on barred terms, which allow us to capture the state of the choreography during its execution, while the Petri net semantics is obtained as a translation of the proposed process algebra into coloured Petri nets.

Keywords: Web Services, Web Services Composition, Choreography, Operational Semantics, Petri Nets.

1 Introduction

A Web Service [5] can be defined as a self-describing, self-contained modular application that can be published, located and invoked over a network, usually the Internet. Web Service composition is an interesting way to provide new services as a combination of two or more established Web Services. In this manner, services provided by different suppliers can collaborate to provide a new service, regardless of the implementation languages or the hosting platforms. Web technologies are thus a new way of doing business cheaply and efficiently, but the composition of Web services still requires a solid technology and new techniques to support their developments [11].

The current Web technology is based on the Service Oriented Architecture (SOA) stack [3], where the top layers are orchestration and choreography. Orchestration layer describes the execution logic of a Web Service by defining its control flows. Execution languages like WS-BPEL [6] are used for that purpose. Choreography layer describes interactions between different parties in a

* Supported by the Spanish government (cofinanced by FEDER founds) with the project TIN2006-15578-C02-02, and the JCCLM regional project PEII09-0232-7745

Web Services composition, from a global viewpoint. For that purpose, the W3C has proposed WS-CDL description language [2]. Furthermore, the use of formal methods may help in the development of composite Web Services, as these techniques provide us a solid foundation for the languages and tools used in this task.

However, the semantics of WS-CDL given by the W3C lacks of formalization. One of the main goals of this paper is the formalization of the WS-CDL semantics, that is, the definition of an operational semantics and a Petri net semantics for a relevant subset of WS-CDL. The operational semantics is a barred semantics, similar to that used to define the operational semantics of the Petri Box Calculus PBC [7]. This semantics covers the main aspects of WS-CDL, being based on choreographies, activities and exception blocks. It also takes into account variables, with their values ranging over an integer domain, and the exceptions that could arise when using unassigned variables. We also provide the translation of the proposed algebraic language into a specific model of coloured Petri nets [12]. The benefits of this encoding are that Petri nets offer us a graphical representation of the system, and they are supported by many tools (e.g. CPN Tools [1]), while allow us to simulate and analyze the system behavior.

WS-CDL developers claim that this choreography language is based on the formal language π-calculus [13]. In [8, 9] the authors study the relationship between a formalized version of WS-CDL, called global calculus, and the π-calculus. In this context, there is also a work of W.S.P. van der Aalst [4] that poses a discussion about the use of Petri nets or π-calculus in the context of Web Services Composition Languages. In that paper the author presents some arguments in favor of using Petri nets instead of π-calculus for the design and development of composite Web Services. In this paper, however, we consider both visions, the algebraic, by providing a syntax and a formal operational semantics for a subset of WS-CDL, and a Petri net semantics by using coloured petri nets. The syntax of the model we present here is closer to WS-CDL syntax than the global calculus syntax, in the sense that our operators are directly taken from the main WS-CDL structural elements. Furthermore, we are not splitting some operators into two or more separate operators. For instance, with our barred semantics we can maintain the *workunit* construction as a single operator, instead of splitting it in several different constructions. In [16, 14] the authors also translate WS-CDL into a formal model. They use a small language called CDL, which is a formal model of simplified WS-CDL. The formal operational semantics of CDL is also given by the authors. However, data aspects are not considered in these works. In comparison, we consider WS-CDL integer variables, which can be used as guards in workunits, in assignments, or in interactions. Finally, in [10] the authors investigate formal representations of service interaction patterns in π-calculus and Petri nets. However, they do not use any choreography description language as starting point, modeling the composition directly into the formal representation.

The outline of the paper is as follows. Section 2 contents a brief description of the main elements of WS-CDL. In Section 3 the operational semantics of a

relevant subset of WS-CDL is defined. In Section 4 coloured Petri nets are introduced and the Petri net semantics for WS-CDL is given. Finally, the conclusions are presented in section 5.

2 WS-CDL

The Web Services Choreography specification offers a precise description of collaborations between the parties involved in a choreography. WS-CDL specifications are contracts containing "global" definitions of the common ordering conditions and constraints under which messages are exchanged. The contract describes, from a global viewpoint, the common and complementary observable behavior of all the parties involved. Each party can then use the global definition to build and test solutions that conform to it. The global specification is in turn realized by a combination of the resulting local systems, on the basis of appropriate infrastructure support.

The WS-CDL model [2] describes the participants of a composite Web Service, their role types and the relationships between the parties. It also contains a description of the information exchanged, the channels used for communication and the visible information of the different roles.

The main elements of a WS-CDL description are choreographies, which are defined in a hierarchical way. There is a root choreography, and any choreography can perform other inner choreographies. Choreographies prescribe the common rules that govern the ordering of exchanged messages and the collaborative behavior of the different parties. They consist of three parts:

- **Choreography Life-line**: This describes the progression of a collaboration. Initially, the collaboration is established between the parties; then, some work is performed within it, and finally it completes either normally or abnormally.
- **Choreography Exception Block**: This specifies the additional interactions that should occur when a Choreography behaves in an abnormal way.
- **Choreography Finalizer Block**: This describes how to specify additional interactions that should occur to modify the effect of an earlier successfully completed Choreography (for example to confirm or undo the effect).

Each of these parts basically contains one activity, which describes the work to be done. There are basic activities (which perform the lowest level actions) and ordering structures. Basic activities are used to assign the variable values, or to indicate that a role type is performing some internal (non-visible) actions. Other basic activities supported by WS-CDL are the *interaction activities*, which describe the exchange of information between parties and the possible synchronizations of their observable information changes and the actual values of the exchanged information.

The ordering structures combine activities with other Ordering Structures in a nested structure to express the ordering conditions in which the information within the Choreography is exchanged. The ordering structures are the sequence,

choice and parallel. There is also another class of activity supported by WS-CDL, the so-called workunits, which allow the execution of some activities when a certain condition holds. Workunits also permit the iteration of activities.

3 Operational Semantics

We first introduce the algebraic language that serves us as a metamodel of WS-CDL, and that allows us to define an operational semantics for it. For that purpose, we consider a WS-CDL document with only one choreography (the so called *root choreography*), i.e., we do not allow a hierarchy of choreographies in our starting document. We call *Var* the set of variable names used in the choreography. We assume that each role type uses its own variable names, i.e., a variable name can only be used by a single role type[1]. For simplicity we only consider integer variables, although it would not be problematic to extend this assumption to any number of data types. Furthermore, we also consider that each interaction only contains one exchange element, which is used to communicate the value of a variable from one role type to the other.

The specific algebraic language, then, that we use for the activities is defined by the following BNF-notation:

$$A ::= \mathit{fail} \mid \mathit{assign}(r, v, n) \mid \mathit{noaction}(r) \mid \mathit{inter}(r_1, r_2, v_1, v_2) \mid A\,;A \mid A \,\square\, A \mid A \parallel A \mid \mathit{workunit}(g, \mathit{block}, g', A)$$

where r, r_1, r_2 range over the roletypes of the choreography, v, v_1, v_2 range over *Var*, $n \in \mathbb{Z}$, g, g' are predicates that use the variable names in *Var*, and *block* is a boolean. Given a predicate g, we will call $\mathit{Vars}(g)$ the set of variables used in g, which may have been initialized or not when they are used.

The correspondence between the syntax of WS-CDL and our metamodel is shown in Table 1. The basic activities are *fail*, *assign*, *noaction* and *inter* operations; *fail* is used to raise an exception, the control flow is transferred to the exception block, and after that the choreography terminates. The *assign* operation is used to assign the variable v at role r to n, the *noaction* captures either a silent or internal operation at role r. The *inter* operation is used to capture an interaction between roles r_1 and r_2, where the value of variable v_2 in r_2 is assigned to the value of variable v_1 of r_1. An interaction fails when the variable v_1 in r_1 is unassigned, then the exception block of the choreography is executed, after which the choreography terminates.

The ordering structures are the sequence, choice, parallel and workunit operations. The workunit operator has the following interpretation: first, if some of the variables used in g are not available, or if g evaluates to false, then, depending on the *block* attribute the workunit is skipped or blocked until g is evaluated to true. When the guard evaluates to true, the activity inside the workunit is executed, and when it terminates, the repetition condition g' is evaluated. If some variable used in g' is not available or if g' is false, then, the workunit

[1] Actually, WS-CDL does not allow the use of shared variables.

WS-CDL Syntax	Metamodel
<assign roleType="r"> <copy name="CName"> <source expression="n"/> <target variable="cdl:getVariable('v',',')" /> </copy> </assign>	assign(r,v,n)
<noAction roleType="r"/> or <silentAction roleType="r"/>	noaction(r)
<interaction name="name" ... <participate relationshipType="rname" fromRoleTypeRef="r1" toRoleTypeRef="r2" /> <exchange name="Cname" ... action="request"> <send variable="cdl:getVariable('v1',',')" ... > <receive variable="cdl:getVariable('v2',',')" ... > </exchange> ... </interaction>	inter(r1,r2,v1,v2)
<sequence> activity$_1$ activity$_2$ </sequence> | <choice> activity$_1$ activity$_2$ </choice> | <parallel> activity$_1$ activity$_2$ </parallel>	$\frac{activity_1 \ ; \ activity_2}{activity_1 \ \square \ activity_2}$ $activity_1 \parallel activity_2$
<workunit name="Name" guard="g" repeat="g' " block="true | false"> Activity </workunit>	workunit(g,block,g',Activity) — — — — — — — — block = true | false
Choreography exception handling	fail
<choreography name="Name" ... Activity1 <exceptionBlock name="EName"> Activity2 </exceptionBlock> ... </choreography>	(Activity1, Activity2) — — — — — — — —

Table 1. Conversion table

terminates, otherwise the activity inside it is executed again. The sequence and parallel operators have the usual interpretation.

Concerning the choice operator semantics, we can read its textual semantics in [2], which states that "only one activity of those involved in the choice can be executed, but when the choice has workunits with guard conditions, the first workunit whose guard condition is true must be executed". Thus, there is a prioritization by means of lexical ordering in this case. However, in the case of a choice having both guarded workunits and other activities as alternatives, it states that "the selection criteria for those activities are non-observable". This textual description is rather ambiguous, since in some cases lexical ordering is

used, whereas in others any activity can be selected. We consider that lexical ordering is not the best way to prioritize the activities. This could be actually done by introducing specific priorities in the activities, which is the subject of research that we are currently undertaking (see [15]). In this paper, then, we consider the following approach, which is in our opinion the most natural and best matches the goals of a choreography: any activity of those enabled[2] in the choice can be executed. We also impose the condition for the block attribute of the workunits which are alternatives of a choice that it be true, since in this case we need only consider those workunits whose guard evaluates to true, and abandoning the choice when a guard of a workunit is false would be pointless.

A choreography is now defined as a pair (A_1, A_2), where A_1 and A_2 are activities defined by the previous syntax. A_1 is the activity of the *life-line* of the choreography and A_2 is the activity of its exception block, which can be empty (denoted by $\emptyset$), because the exception block is optional. We do not consider a separate finalizer activity, because it can be part of A_1 (concatenated with it by a sequence operator).

We now introduce the operational semantics for this language, by using both *overbarred* and *underbarred* dynamic terms, which are used to capture the current state of the choreography throughout its execution in a similar way to that used to define the operational semantics of the Petri Box Calculus [7].

We will use letters $A, A_1, A_2, \ldots$ to denote activities, which are used to define the *dynamic terms*, these are defined by the following BNF-notation:

$$D ::= \overline{A} \mid \underline{A} \mid D\,;A \mid A\,;D \mid D \,\square\, B \mid A \,\square D \mid D \parallel D \mid \mathit{workunit}(g, \mathit{block}, g', D)$$

The set of dynamic terms will be called *Dterms*.

The *overbars* are used to indicate that the corresponding term can initiate its execution, whereas *underbarred* terms have already finished their execution. Thus, as the activity evolves along its execution the bars are moving throughout the term syntax.

Example 1. Consider the activity $A = \mathit{workunit}(g, \mathit{true}, g', \mathit{assign}(r, v, 1))$. Its execution starts with the dynamic term $\overline{A}$, from which the guard g is evaluated. If all the variables in g are available, and g becomes true, then, we reach the dynamic term $D_1 = \mathit{workunit}(g, \mathit{true}, g', \overline{\mathit{assign}(r, v, 1)})$, which means that the assignment of v can now start at role r. Otherwise, if some variable needed to evaluate g is not available, or if g is false, as the *block* condition is *true*, the activity blocks until g changes its value to true. Once the assignment of v is done, the following dynamic term is reached: $D_2 = \mathit{workunit}(g, \mathit{true}, g', \underline{\mathit{assign}(r, v, 1)})$, from which g' is evaluated. If some variable needed to evaluate g' is not available or g' is false, then, the workunit ends and the dynamic term $\underline{A}$ is reached. Otherwise, when g' is true, D_1 is reached again. □

[2] In the sense that it can execute some action at the current instant.

(Seq1)	$\overline{A_1; A_2} \equiv \overline{A_1}; A_2$	**(Seq2)** $\underline{A_1}; A_2 \equiv A_1; \overline{A_2}$
(Seq3)	$A_1; \underline{A_2} \equiv \underline{A_1; A_2}$	
(Cho1)	$\overline{A_1 \,\square\, A_2} \equiv \overline{A_1} \,\square\, A_2$	**(Cho2)** $\overline{A_1 \,\square\, A_2} \equiv A_1 \,\square\, \overline{A_2}$
(Cho3)	$\underline{A_1} \,\square\, A_2 \equiv \underline{A_1 \,\square\, A_2}$	**(Cho4)** $A_1 \,\square\, \underline{A_2} \equiv \underline{A_1 \,\square\, A_2}$
(Par1)	$\overline{A_1 \parallel A_2} \equiv \overline{A_1} \parallel \overline{A_2}$	**(Par2)** $\underline{A_1} \parallel \underline{A_2} \equiv \underline{A_1 \parallel A_2}$
(Cong1)	$\dfrac{\forall op \in \{;, \square\}, \quad D_1 \equiv D_2}{B \; op \; D_1 \equiv B \; op \; D_2\,, \quad D_1 \; op \; B \equiv D_2 \; op \; B}$	
(Cong2)	$\dfrac{D_1 \equiv D_2}{D \parallel D_1 \equiv D \parallel D_2\,, \quad D_1 \parallel D \equiv D_2 \parallel D}$	
(Cong3)	$\dfrac{D_1 \equiv D_2}{workunit(g, block, g', D_1) \equiv workunit(g, block, g', D_2)}$	

Table 2. Equivalence rules

In this example we have used dynamic terms to represent the current state of the system. However, dynamic terms like $\overline{B_1 \,\square\, B_2}$, $\overline{B_1} \,\square\, B_2$ and $B_1 \,\square\, \overline{B_2}$ correspond to the same state in the system, a state in which any alternative of the choice must be enabled. This means that in some cases the bars can be redistributed on a dynamic term yielding to an equivalent state. Thus, we now define the equivalence relation $\equiv$, as the **least equivalence relation** satisfying the rules of Table 2. By means of this equivalence relation we can identify those dynamic terms that can be obtained by moving backwards or forwards the bars on the terms without executing any action and which correspond to the same state in the system.

For any dynamic term D we will denote the class of dynamic terms equivalent to D by $[D]_{\equiv}$, and the set of classes of dynamic terms will be called *CDterms*.

The rules of Table 2 are very intuitive in general. *Seq1* is used to activate the first activity of a sequence when the sequence becomes activated, *Seq2* allows us to activate B_2 when B_1 terminates, and *Seq3* establishes that once B_2 ends, the sequence $B_1; B_2$ ends too. *Cho1* and *Cho2* allow us to activate either alternative of a choice, while *Cho3* and *Cho4* establish that once the selected alternative terminates the choice itself ends too. *Par1* is used to activate both arguments in a parallel activity, and *Par2* establishes that, when both argument activities terminate, the parallel activity terminates, too.

Lemma 1. The relation $\equiv$ defined as the least equivalence relation fulfilling the rules in Table 2 is a congruence.

Proof. $\equiv$ is defined as the least reflexive, symmetric, and transitive relation fulfilling the rules in Table 2. It is immediate that such a relation exists, and also that it is a congruence, due to rules *Cong1, Cong2* and *Cong3*. □

The following definition introduces the so-called *initial* and *final* dynamic terms, which are those dynamic terms that are equivalent to an overbarred (underbarred) activity.

Definition 1. (Initial and final dynamic terms)
Given a dynamic term D, we say that D is initial (resp. final), denoted by $init(D)$ (resp. $final(D)$), when there exists an activity A such that $\overline{A} \in [D]_{\equiv}$ (resp. $\underline{A} \in [D]_{\equiv}$). In such a case we will say that the class $[D]_{\equiv}$ is initial (resp. final) too. □

According to this definition, $(\overline{assign(r,v,n)})$, $(\overline{assign(r,v,n)} \,\square\, noaction(r))$ and $(\overline{assign(r,v,n)} \parallel \overline{noaction(r)})$ are initial, but not $(\underline{assign(r,v,n)})$ or $((assign(r,v,n); \overline{assign(r',v',n')}) \,\square\, noaction(r))$. Similar examples can be written for final dynamic terms.

A choreography is executed within the context of the variables defined in it. We now define the *context* of a choreography, which captures which variables are available and their current values.

Definition 2. (Context)
Given a choreography $C = (A_1, A_2)$, with roletypes $\mathcal{R}$ and variables Var, we define a *context* of C as a function $\mu : Var \to \mathbb{Z} \cup \{\epsilon\}$. Unavailable variables are assigned the ϵ value, otherwise this function provides us with the current value of the variable.

We denote the set of possible contexts of a choreography by *Contexts*. The *initial context*, denoted by μ_0, is that defined by assigning ϵ to all the variables in the choreography.

$$\mu_0(v) = \epsilon \quad \forall v \in Var$$

Given a context μ, a variable v and an integer value n, we denote by $\mu[v/n]$ the context obtained from μ by changing the value of v to n:

$$\mu[v/n](v') = \begin{cases} \mu(v') & \text{if } v' \neq v \\ n & \text{if } v' = v \end{cases}$$

We will also use this definition for n being an integer arithmetic expression that uses some variables of the choreography, with the natural interpretation, the value of v is replaced by the resulting value of n.

Now, given a predicate g and a context μ, we will write $sat(\mu, g)$ when $\forall v \in Vars(g)$, $\mu(v) \neq \epsilon$, and g evaluates to true under μ. □

Notice also that this definition covers all the possible cases for the syntax of *CDterms*, taking into account the $\equiv$-equivalence.

Definition 3. (Contextual activity terms)
A *contextual activity term* is a pair $([D]_{\equiv}, \mu)$, where D is a dynamic term and μ a context. □

Definition 4. We define a *dynamic choreography term* as a pair of one of the following forms: $([D]_{\equiv}, A_2)$ or $(A_1, [D]_{\equiv})$, where $[D]_{\equiv}$ corresponds to the activity in execution in the choreography (the life-line or its exception block), and A_2 can be empty.

We also define a *contextual dynamic choreography term*, as a pair $(\mathcal{C}, \mu)$, where $\mathcal{C}$ is a dynamic choreography term and μ is a context.

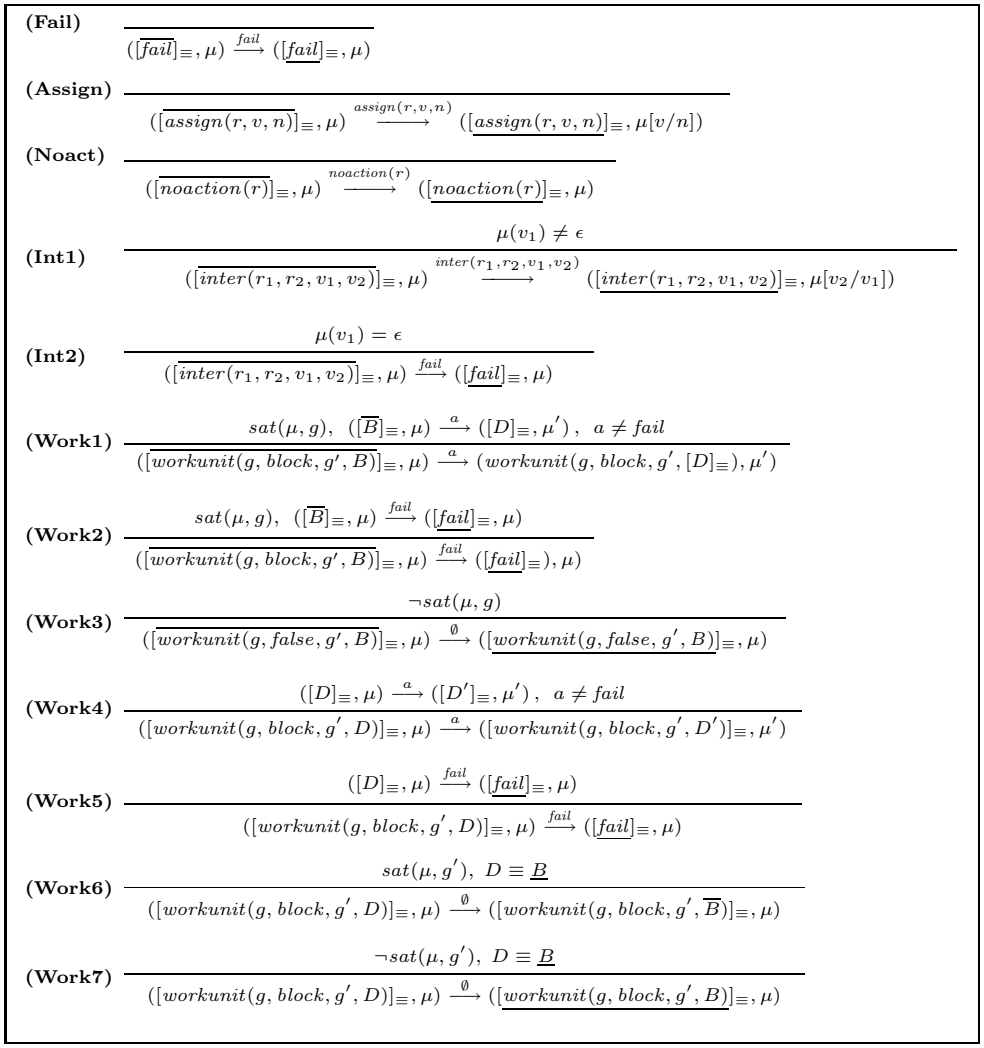


Table 3. Transition rules for contextual activity terms (I)

Given a choreography $C = (A_1, A_2)$, the *initial contextual dynamic term* of C is[3] $([\overline{A_1}]_{\equiv}, A_2, \mu_0)$. □

In Tables 3 and 4, we introduce the rules that define the transitions for the contextual activity terms, where

$$([D]_{\equiv},\mu) \xrightarrow{a} ([D']_{\equiv},\mu')$$

[3] We will write contextual dynamic choreography terms as triples, by omitting the parentheses for the dynamic choreography term.

(Seq1-2) $\dfrac{([D]_{\equiv},\mu) \xrightarrow{a} ([D']_{\equiv},\mu')\,,\ a \neq fail}{([D;B]_{\equiv},\mu) \xrightarrow{a} ([D';B]_{\equiv},\mu')} \qquad \dfrac{([D]_{\equiv},\mu) \xrightarrow{a} ([D']_{\equiv},\mu')\,,\ a \neq fail}{([B;D]_{\equiv},\mu) \xrightarrow{a} ([B;D']_{\equiv},\mu')}$

(Seq3-4) $\dfrac{([D]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)\,,}{([D;B]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)} \qquad \dfrac{([D]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)\,,}{([B;D]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)}$

(Choi1-2) $\dfrac{([\overline{B_1}]_{\equiv},\mu) \xrightarrow{a} ([D]_{\equiv},\mu')\,,\ a \neq fail}{([\overline{B_1 \,\square\, B_2}]_{\equiv},\mu) \xrightarrow{a} ([D \,\square\, B_2]_{\equiv},\mu')} \qquad \dfrac{([\overline{B_2}]_{\equiv},\mu) \xrightarrow{a} ([D]_{\equiv},\mu')\,,\ a \neq fail}{([\overline{B_1 \,\square\, B_2}]_{\equiv},\mu) \xrightarrow{a} ([B_1 \,\square\, D]_{\equiv},\mu')}$

(Choi3) $\dfrac{([\overline{B_1}]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)\,,\ ([\overline{B_2}]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)}{([\overline{B_1 \,\square\, B_2}]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)}$

(Choi4) $\dfrac{([D]_{\equiv},\mu) \xrightarrow{a} ([D']_{\equiv},\mu')\,,\ \neg init(D)\,,\ a \neq fail}{([D \,\square\, B]_{\equiv},\mu) \xrightarrow{a} ([D' \,\square\, B]_{\equiv},\mu')}$

(Choi5) $\dfrac{([D]_{\equiv},\mu) \xrightarrow{a} ([D']_{\equiv},\mu')\,,\ \neg init(D)\,,\ a \neq fail}{([B \,\square\, D]_{\equiv},\mu) \xrightarrow{a} ([B \,\square\, D']_{\equiv},\mu')}$

(Choi6-7) $\dfrac{([D]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)\,,\ \neg init(D)}{([B \,\square\, D]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)} \qquad \dfrac{([D]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)\,,\ \neg init(D)}{([D \,\square\, B]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)}$

(Par1) $\dfrac{([D_1]_{\equiv},\mu) \xrightarrow{a} ([D_1']_{\equiv},\mu')\,,\ a \neq fail\,,\ ([D_2]_{\equiv},\mu) \xrightarrow{\not{fail}}}{([D_1 \| D_2]_{\equiv},\mu) \xrightarrow{a} ([D_1' \| D_2]_{\equiv},\mu')}$

(Par2) $\dfrac{([D_2]_{\equiv},\mu) \xrightarrow{a} ([D_2']_{\equiv},\mu')\,,\ a \neq fail\,,\ ([D_1]_{\equiv},\mu) \xrightarrow{\not{fail}}}{([D_1 \| D_2]_{\equiv},\mu) \xrightarrow{a} ([D_1 \| D_2']_{\equiv},\mu')}$

(Par3-4) $\dfrac{([D_2]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)}{([D_1 \| D_2]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)} \qquad \dfrac{([D_1]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)}{([D_1 \| D_2]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)}$

Table 4. Transition rules for contextual activity terms (II)

which represents the execution of some basic activity a or an empty movement (denoted by $a = \emptyset$).

In rules *Par1* and *Par2* of Table 4 we use the notation $([D]_{\equiv},\mu) \xrightarrow{\not{fail}}$ to mean that no transition labelled with *fail* can be executed from $([D]_{\equiv},\mu)$.

Let us see the informal interpretation of these rules:

- Rules *Fail*, *Assign* and *Noact* are evident. *Int1* captures the execution of an activated interaction, when the source variable has a value assigned. Otherwise, rule *Int2* is used to raise an exception.
- Rules *Work1* to *Work7* establish the semantics of workunits. For a workunit whose guard condition evaluates to true, we can execute any initial movement of the activity inside it (rule *Work1*). Once the workunit is activated, if the activity inside the workunit can execute a *fail* movement, we immediately

raise an exception (rules *Work2* and *Work5*). When the *block* attribute is false, and the guard condition is not fulfilled[4], the workunit is skipped at once (rule *Work3*). The execution of the activities inside the workunit is captured by rule *Work4*. Rule *Work6* allows us to restart the activity inside the workunit when it has finished and the repetition condition holds, whereas rule *Work7* is used to abandon the workunit when that condition does not hold.

- Rules *Seq1* to *Seq4* capture the semantics of the sequence operator, while *Choi1* to *Choi7* define the semantics of the choice. The rules for the sequence are highly intuitive, so we omit an explanation about them. In the case of the choice operator, *Choi1* and *Choi2* are used to resolve the choice when one argument activity can execute a movement (different from *fail*). Once the choice has been decided by executing a movement of one of its argument activities, this activity continues executing until completion (*Choi4-5*). If the activity in execution can make a *fail* movement, an exception is raised (rules *Choi6-7*). In rule *Choi3* we can see that the choice can only execute a *fail* movement when both arguments are able to do that. Accordingly, when an alternative fails.
- Finally, rules *Par1-2* capture the (independent) parallel execution of the argument activities of a parallel operator, and *Par3-4* are used to raise an exception when one component is able to do so.

(Cor1)	$\dfrac{([D]_{\equiv},\mu) \xrightarrow{a} ([D']_{\equiv},\mu'),\ a \neq fail}{([D]_{\equiv},A_2,\mu) \xrightarrow{a} ([D']_{\equiv},A_2,\mu')}$	**(Cor2)**	$\dfrac{([D]_{\equiv},\mu) \xrightarrow{a} ([D']_{\equiv},\mu'),\ a \neq fail}{(A_1,[D]_{\equiv},\mu) \xrightarrow{a} (A_1,[D']_{\equiv},\mu')}$
(Cor3)	$\dfrac{([D]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu),\ A_2 \neq \emptyset}{([D]_{\equiv},A_2,\mu) \xrightarrow{fail} (A_1,[\overline{A_2}]_{\equiv},\mu)}$	**(Cor4)**	$\dfrac{([D]_{\equiv},\mu) \xrightarrow{fail} ([\underline{fail}]_{\equiv},\mu)}{(A_1,[D]_{\equiv},\mu) \xrightarrow{fail} (A_1,[\underline{fail}]_{\equiv},\mu)}$

Table 5. Transition rules for choreographies

The rules for choreographies are those introduced in Table 5, which capture the evolution of contextual dynamic choreography terms. *Cor1-2* allow the evolution of the activity in execution, except in the case of failure. In that case, rules *Cor3-4* are used, the first to activate the activity of the exception block, and the second to terminate the activity of the exception block when it fails. In rule *Cor3* the term A_1 is that obtained by removing the bars on D.

Definition 5. (Labelled transition system)
For any contextual activity term $([D]_{\equiv},\mu)$ we define its labelled transition system, denoted by $lts([D]_{\equiv},\mu)$, as that obtained by the application of the rules in Tables 3 and 4, starting from $([D]_{\equiv},\mu)$: $lts([D]_{\equiv},\mu) = (Q, q_0, \rightarrow)$, where Q is the set of contextual activity terms that are reachable by using the rules in

[4] This case cannot occur as alternative of a choice, due to the syntactical restriction introduced.

Tables 3 and 4, starting from $q_0 = ([D]_{\equiv}, \mu)$, and $\rightarrow = \{\xrightarrow{a} \mid$ for all basic activity a, or $a = \emptyset\}$.

Then, for any choreography $C = (A_1, A_2)$, we define the semantics of C as the labelled transition transition system obtained by the application of rules in Table 5 for the initial contextual dynamic choreography term of C, $c_0 = ([\overline{A_1}]_{\equiv}, A_2, \mu_0)$:

$$lts(C) = (\mathcal{Q}, c_0, \rightarrow)$$

where $\mathcal{Q}$ is the set of contextual dynamic choreography terms that are reachable by the rules in Table 5, starting from c_0, and $\rightarrow = \{\xrightarrow{a} \mid$ for all basic activity a, or $a = \emptyset\}$. □

Example 2. Let us consider the choreography $C = (A_1, A_2)$, where

$A_1 = assign(r_1, v_1, 1)\,;\, assign(r_3, v_3, 3)\,;\, (noaction(r_1) \,\square\, inter(r_1, r_2, v_1, v_2))$
$A_2 = assign(r_2, v_2, 0); assign(r_3, v_3, 0)$

Then, in Fig.1 we show the labelled transition system of C, where

$D_1 = assign(r_1, v_1, 1); \overline{assign(r_3, v_3, 3)}\,;\, \overline{(noaction(r_1) \,\square\, inter(r_1, r_2, v_1, v_2))}$
$D_2 = assign(r_1, v_1, 1); assign(r_3, v_3, 3)\,;\, \overline{(noaction(r_1) \,\square\, inter(r_1, r_2, v_1, v_2))}$

and

$$\mu_{ijk}(v_1) = i, \quad \mu_{ijk}(v_2) = j, \quad \mu_{ijk}(v_3) = k, \quad \mu_{ijk}(v_n) = \epsilon \quad \forall v_n \neq v_1, v_2, v_3$$

□

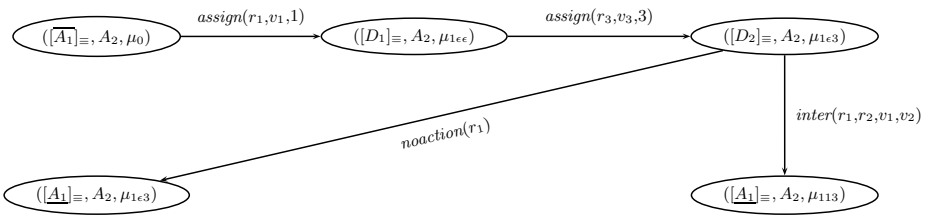


Fig. 1. $lts(C)$

4 Petri Net Semantics

4.1 Coloured Petri Nets

In this section we introduce the specific model of coloured Petri net [12] that we consider for the translation. The particular model that we use includes places with coloured tokens and normal places, in which no information is associated with tokens. We will have a coloured place for each variable, a place with a single token whose colour is used to capture the variable value (or ϵ). Transitions may have asssociated a guard condition, which must be true in order to allow the firing of the transitions. We will assume this guard to be true where this is not indicated.

Definition 6. (Coloured Petri Nets)
We define a Coloured Petri Net (CPN) as a tuple $N = (P, T, F, \lambda, \Sigma, G)$, where:

- P is a finite set of *places*, where $P = P_e \cup P_x \cup P_{er} \cup P_i \cup P_c$, which corresponds to:
  - *entry-places*, P_e, they are labelled with e. They will be marked with a token in the initial marking. Each CPN will have an only *entry place*.
  - *exit places*, P_x, they are labelled with x. They will be marked with a token if the activity finishes correctly.
  - *error places*, P_{er} they are labelled with x_{er}. If one of the *error* places is marked, this will indicate that the activity has not finished correctly.
  - *internal places*, P_i, they are labelled with i.
  - *coloured places*, P_c, they are labelled with $r_i v_i$, indicating that this place is associated to role r_i and variable v_i, and it will be marked with the associated value of the variable v_i in the role r_i. In the initial marking they will be marked with ϵ.
- T is a finite set of *transitions* $(P \cap T = \emptyset)$, such that $T = T_w \cup T_b$, with $T_w \cap T_b = \emptyset$. Transitions in T_w are called *white* and they will be associated with a basic activity *assign*, *noaction*, *inter*, $\emptyset$ or a guard; whereas transitions in T_b are called *black* and these will be associated with a *fail*. Black transitions will be fired first in case of conflict with white transitions, i.e., they are considered as having greater priority.
- $\lambda : T_b \cup T_w \rightarrow L \cup \{\emptyset\}$ is a labelling function, where L is the set of basic activities.
- F is the *flow relation* $(F \subseteq (P \times T) \cup (T \times P))$.
- Σ is the set of colours.

$$\Sigma : P_c \rightarrow \mathbb{Z} \cup \{\epsilon\}$$

- G is a guard function. It is defined from T into expressions such that:

$$\forall t \in T : [G(t) = Boolean]$$

We will omit this function in the grafical representation when it is *true*.

We use the classical notation on Petri nets to denote the precondition and postcondition of both places and transitions:

$$\forall x \in P \cup T : {}^\bullet x = \{y \,|\, (y, x) \in F\} \qquad x^\bullet = \{y \,|\, (x, y) \in F\}$$

Markings are defined as an annotation of tokens over uncoloured places (they will be represented with black dots), and with the value of the variable over the coloured places. Hence, a marking M is formally defined as a function $M : P \rightarrow \mathbb{Z} \cup \{\epsilon\}$, where $M(p_k)$ indicates the number of tokens on p_k, when it is a uncoloured place, or it indicates the value of the variable v_i (or ϵ for an unavailable variable) if it is a coloured place labelled with $r_i v_i$.

We consider as initial marking M_e, the marking in which the *entry* place is marked with one token, and the coloured places are marked wit ϵ. □

Semantics of CPNs is captured by the following definitions, which extend the firing rule of Petri nets by considering the priority information obtaining from two transition types that we have introduced.

Definition 7. (Enabling transitions)
Given a CPN, $N = (P, T, F, \lambda, \Sigma, G)$, a marking M of it and a transition $t \in T$, we say that t is *enabled at* M if each of its input uncoloured places contains at least one token, i.e., $\forall p \in {}^\bullet t,\ p \notin P_c\,,\ M(p) > 0$, and its associated guard function is evaluated to true. As usual, we denote this by $M[t\rangle\,.$ □

Now the firing rule can be precisely defined.

Definition 8. (Firing rule)
Given a CPN, $N = (P, T, F, \lambda, \Sigma, G)$, a marking of it M, and an enabled transition t, we say that t *can be fired* at that state if and only if there is no other enabled transition having a greater priority (black transitions are fired first in case of conflict).

The firing of t leads us to a new marking, M', which is defined as follows:

1. The marking M' for uncoloured places is obtained by applying the classical firing rule on Petri nets, i.e. $M'(p) = M(p) - W_F(p,t) + W_F(t,p)$, where $W_F(a) = 1$ for $a \in F$, and $W_F(a) = 0$ for $a \notin F$.
2. The marking M' for a coloured place p_c labelled with $r_i v_i$ ($p_c \in t^\bullet$) is obtained by considering the action labelling the transition, i.e., for $\lambda(t) = \mathit{asssign}(r_i, v_i, n)$, then $M'(p_c) = n$; for $\lambda(t) = \mathit{inter}(r_j, v_j, r_i, v_i)$, then $M'(p_c) = M(\widetilde{p_c})$, where $\widetilde{p_c}$ is the coloured place labelled with $r_j v_j$.

□

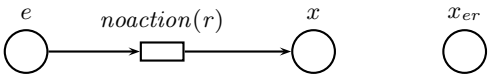
4.2 CPN Semantics for WS-CDL

Let us see the translation for the different constructions of WS-CDL :

– Basic activities. These are translated easily, as follows:

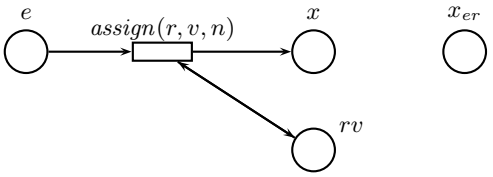
- No action activity.

$N_{noaction(r)}$ =



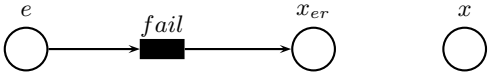
- Assign activity:

$N_{assign(r,v,n)}$ =



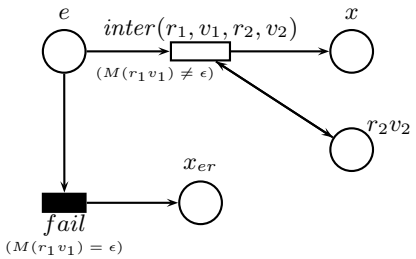
- Fail activity:

N_{fail} =



- Interaction activity:

  $N_{inter(r_1,v_1,r_2,v_2)}$ =

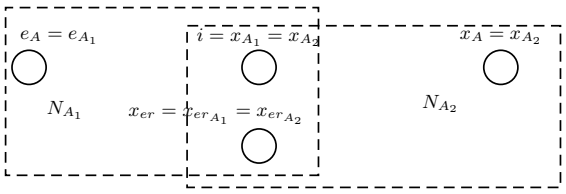


  Observe that *fail* transition are black, i.e., they have greater priority that the other ones.

– Ordering structures.
These are used to combine activities in a nested structure that uses the sequence, parallel and choice constructs. For all of these cases we provide the translation only considering two activities; nevertheless, the generalization to a greater number of activities is straightforward for all of them. We consider N_{A_i} $i = 1, 2$ the CPNs associated to the activity A_i.

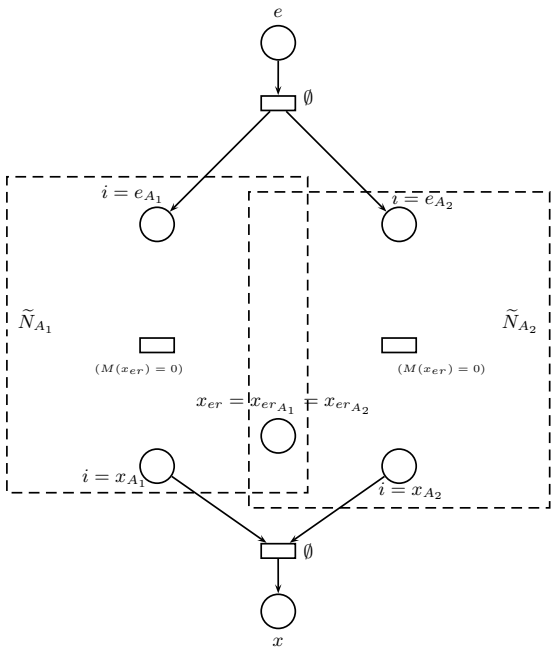
- Sequence:
  A sequence of two activities A_1 and A_2 is translated in an easy way, by just collapsing in a single place (this will be an internal place of the new Petri net) the *exit* place of the N_{A_1} and the *entry* place of N_{A_2}. The *entry* place of the new Petri net will be the *entry* place of N_{A_1}. The *exit* place of the new Petri net will be the *exit* place of N_{A_2}, and we also collapse the *error* places.

$N_{A_1;A_2}$ =



- Parallel.
  For the parallel execution of two activities A_1 and A_2 we have considered in the associated Petri net the above structure, where we have added two new transitions, one to fork both parallel activities and the other to join them upon their termination. Furthermore, we have joined the *error* places and a guard condition is added to every transition of both CPNs, which prevent their execution when the *error* place is marked.

$N_{A_1 \| A_2} =$



where $\widetilde{N}_{A_1}$ is obtaining from N_{A_1} adding for every transition the guard $M(x_{er}) = 0$ (respectively in $\widetilde{N}_{A_2}$).

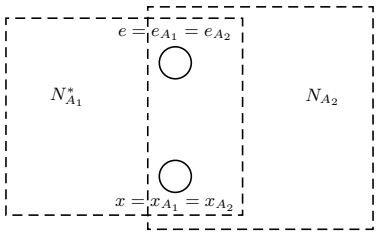
Observe that in the event of a failure in one of these separate parallel executions, the *error* place would be marked.

- Choice.
  Here, we introduce a restriction in the syntax that we consider for the translation. Specifically, we will require that no parallel or workunit operator appears at the first level of the arguments of the choice. In later versions we will introduce these operators, which require a certain distinction of subcases in order to define a proper translation.

  A choice between two activities A_1 and A_2 (with CPNs N_{A_1} and N_{A_2}, respectively) is translated by collapsing in a single place (this will be the *entry* place of the new Petri net) the *entry* place of N_{A_1} and the *entry* place of N_{A_2}. The *exit* place of the new Petri net will be that obtaining by collapsing the *exit* places of N_{A_1} and N_{A_2}. However, we need to distinguish whether A_1 and A_2 can initially fail or not, because in a choice a *fail* action can only be performed when both arguments are able to do this.

  * When at most one of the arguments (let us say A_1) has one initial *fail* transition ($t \in p_e^\bullet$, $p_e \in P_{N_{A_1}}$, $\lambda(t) = fail$), then, we must remove all the initial *fail* transitions of N_{A_1}.

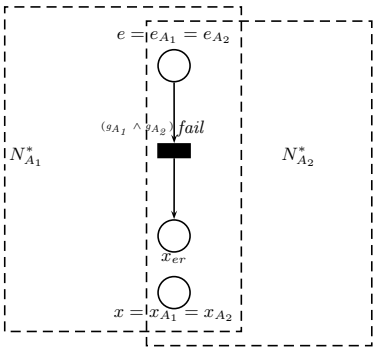
$N_{A_1 \square A_2} =$



where $N^*_{A_1}$ is obtaining by removing initial *fail* transitions of N_{A_1}.

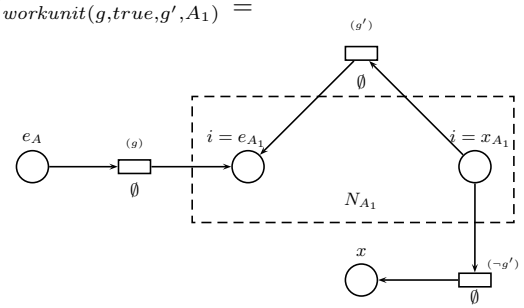
∗ In the case that both CPNs have initial *fail* transitions, these are joined in a single one labelled with *fail* and considering in it all the guards associated to these fail transitions.

$N_{A_1 \square A_2} =$

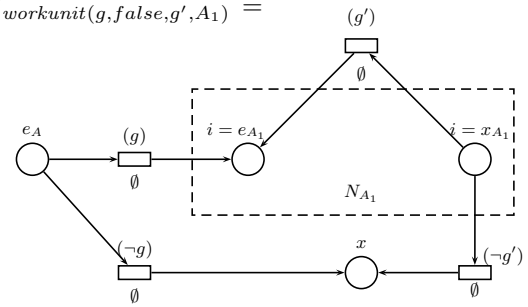


- Workunit.
  We distinguish two cases, depending on the block value of the workunit.

∗ $N_{workunit(g,true,g',A_1)} =$



∗ $N_{workunit(g,false,g',A_1)} =$

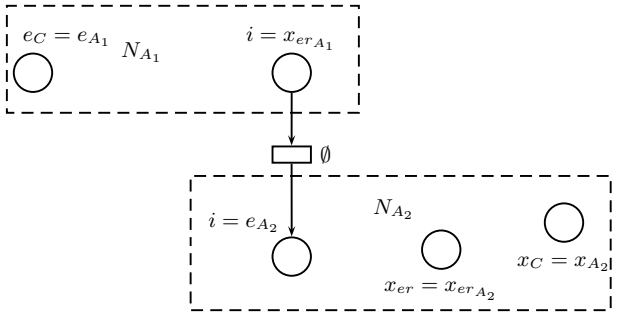


Next, we show that given an activity A (with the syntactic restriction introduced), the operational semantics for $\overline{A}$ and the semantics of the corresponding CPN are bisimilar.

Theorem 1. For any activity A (with the syntactic restriction introduced), $lts([\overline{A}]_{\equiv}, \mu_0)$ and the reachability graph of the marked CPN (N_A, M_e) are bisimilar.

□

Definition 9. For any choreography $C = (A_1, A_2)$, we define the CPN associated to C as follows:



□

Corollary 1. For any choreography $C = (A_1, A_2)$ with both activities fulfilling the syntactic restriction introduced, $lts(C)$ and the reachability graph of the marked CPN (N_C, M_e) are bisimilar.

□

Example 3. Let us now consider the choreography $C = (A, \emptyset)$ where $A = A_1 \| A_2 \| A_3$ and:
$A_1 = assign(r_1, v_1, 1); workunit(v_1 + v_3 = 4, true, v_1 + v_2 = 3, inter(r_1, r_2, v_1, v_2))$
$A_2 = assign(r_2, v_2, 2) \Box noaction(r_2) \Box fail$
$A_3 = assign(r_3, v_3, 3); assign(r_3, v_3, 4)$
Its corresponding N_C is that shown in Figure 3

This choreography may behave correctly or not, depending on the order in which the different actions are executed. For instance, the trace:

$$assign(r_1, v_1, 1).assign(r_3, v_3, 3).inter(r_1, r_2, v_1, v_2).$$
$$noaction(r_3).\emptyset.assign(r_2, v_2, 2).assign(r_3, v_3, 4)$$

corresponds to a correct execution, in which the contextual dynamic term $(\underline{[A_1 \| A_2 \| A_3]}_{\equiv}, \emptyset, \mu_{124})$ is reached. However, the trace:

$$assign(r_3, v_3, 3).assign(r_3, v_3, 4).assign(r_2, v_2, 2).assign(r_1, v_1, 1)$$

corresponds to a situation in which the system becomes deadlocked, because the guard of t_2 prevents their execution.

□

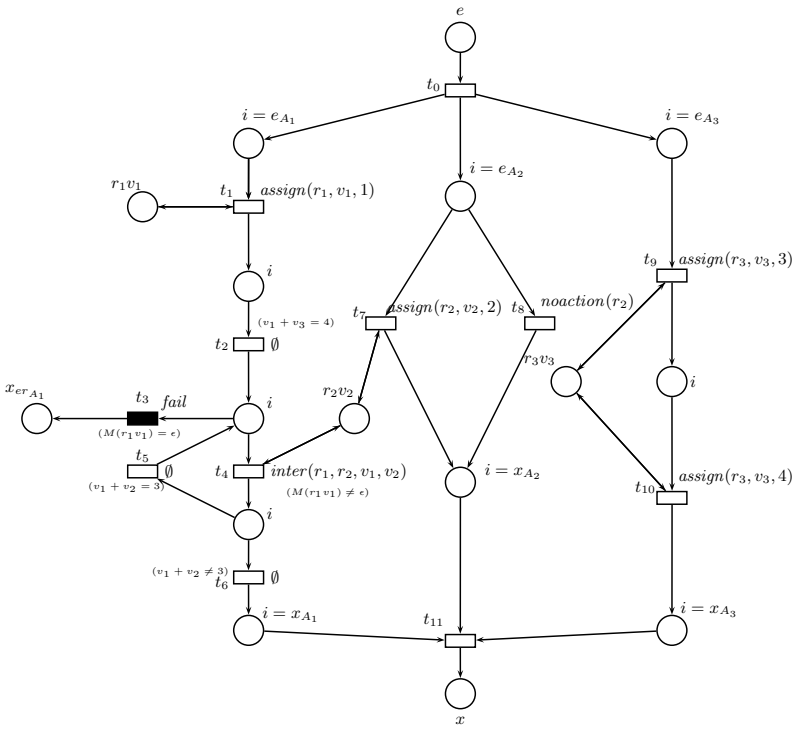


Fig. 2. CPN for the choreography C of Example 3

5 Conclusions

In this paper we have presented a barred operational semantics and a Petri nets semantics for a relevant subset of WS-CDL, taking into account the more relevant aspects of composite Web Services. The official semantics of WS-CDL [2] is defined in a textual manner. Thus, an important advantage of the provided semantics is that it can be used as alternative to the textual document with the purpose of obtaining the WS-CDL semantics in a more rigorous way.

We have considered the main activities of WS-CDL, including both the basic and the ordering structures, and we have defined a formal syntax for them, providing a set of operators that constitute the metamodel for which the barred operational semantics is defined. We have specifically covered the main structural elements of WS-CDL: choreographies and activities, but we have not considered a hierarchy of choreographies, neither the perform construction (as a consequence), nor the finalizer activities. These additional features of WS-CDL can be object of further research, in order to extend this translation. Another contribution of this paper is that this operational semantics is defined by using *barred* terms, which are syntactical terms that are either barred or underbarred in order to capture the current state of the described system. A relevant benefit of this

barred semantics is that we do not need to split the *workunit* construction into two or more separate operators, which would be required in order to define a classical operational semantics. We have also presented a denotational semantics considering a special kind of Coloured Petri Net. We also plan to implement the presented translation.

The semantics has been defined for a subset of WS-CDL, which, as future work, we plan to extend allowing it to support a richer subset of WS-CDL and considering more extensions like time issues or priorities.

References

1. *CPN Tools.* http://wiki.daimi.au.dk/cpntools/cpntools.wiki.
2. *Web Services Choreography Description Language Version 1.0 (WS-CDL).* http://www.w3.org/TR/ws-cdl-10/.
3. *World Wide Web Consortium (W3C).* http://www.w3.org/.
4. W.M.P. van der Aalst. *Pi calculus versus Petri nets: let us eat "humble pie" rather than further inflate the "Pi hype".* BPTrends, 3(5), pp. 1-11, 2005.
5. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services.* Springer-Verlag, 2002.
6. T. Andrews and et. al. *BPEL4WS – Business Process Execution Language for Web Services, Version 1.1, May 2003,* http://www.ibm.com/developerworks/library/specification/ws-bpel/.
7. E. Best, R. Devillers, and M. Koutny. *Petri Net Algebra.* EATC, Springer, 2001.
8. M. Carbone, K. Honda, and N. Yoshida. *Calculus of Global Interaction based on Session Types.* Electronic Notes in Theoretical Computer Science, vol. 171(3), pp. 127–151, 2007.
9. M. Carbone, K. Honda, and N. Yoshida. *A Theoretical Basis of Communication-Centred Concurrent Programming.* Electronic Notes in Theoretical Computer Science, vol. 209, pp. 125–133, 2008.
10. G. Decker and F. Puhlmann. *Formalizing Service Interactions.* S., Fiadeiro, J., Sheth, A., eds.: Proceedings of the 4th International Conference on Business Process Management (BPM 2006), volume 4102 of LNCS, Berlin, Springer Verlag 414û-419, 2006.
11. R. Hamadi and B. Benatallah. *A Petri Net-based Model for Web Service Composition.* In Proceedings of the 14th Australasian database conference, vol. 17, pp. 191–200, 2003.
12. K. Jensen. *Coloured Petri Nets: A high-level Language for System Design and Analysis.* LNCS vol. 483, Springer-Verlag, 1990.
13. J. Milner, R.and Parrow and D. Walker. *A Calculus of Mobile Processes, part I and II.* Information and Computation, vol. 100(1), pp. 1–40 and 41–77, 1992.
14. Z. Qiu, X. Zhao, C. Cai, and H. Yang. *Towards the Theoretical Foundation of Choreography.* In Proceedings of the 16th International Conference on World Wide Web, WWW 2007, pp. 973-982, 2007.
15. V. Valero, M.E Cambronero, G. Díaz, and H. Macià. *A Petri Net Approach for the Design and Analysis of Web Services Choreographies.* To appear in Journal of Logic and Algebraic Programming, 2009.
16. H. Yang, X. Zhao, Z. Qiu, G. Pu, and S. Wang. *A Formal Model for Web Service Choreography Description Language (WS-CDL).* International Conference on Web Services (ICWS'06), pp. 893-894, IEEE Computer Society Press, 2006.

Poster

Verification of Large-Scale Distributed Database Systems in the NEOPPOD Project*

Olivier Bertrand[1], Aurélien Calonne[2], Christine Choppy[1], Silien Hong[3], Kais Klai[1], Fabrice Kordon[3], Yoshinori Okuji[2] Emmanuel Paviot-Adet[3], Laure Petrucci[1], and Jean-Paul Smets[2]

[1] LIPN, CNRS UMR 7030, Université Paris XIII,
99 avenue Jean-Baptiste Clément, 93430 Villetaneuse, France
[2] NEXEDI, 270 bd Clémenceau, 59700 Marcq-en-Barœul, France
[3] LIP6 - CNRS UMR 7606, Université Pierre & Marie Curie,
4 PLace Jussieu, 75252 Paris Cedex 05, France

1 Introduction

Nowadays, large applications are developed that must access and maintain huge data bases. Examples of such software are e-government, internet based information systems, commerce registries, etc.). They are characterised not only by the huge amount of data they manipulate, but also by a mandatory high level of security and reliability.

Hardware has become rather cheap: 150€ for a 64-bit dual core server, and 100€ for a 1TB disk. A large application and data base server could be composed of hundreds of such disks and servers. It would thus be possible to handle as much as 1PB of data and thousands simultaneous transactions, for a moderate investment of 175,000€. However, this requires to elaborate reliable and safe distributed data base management software.

ZODB, the *Zope Object Database*, has become within a few years the most used object data base. This libre software, associated to the Zope application server is used for a Central Bank, to manage the monetary system of 80 million people in 8 countries [2]. It is also used for accounting, ERP, CRM, ECM and knowledge management. It is now a major libre software as PHP or MySQL is.

However, the current Zope architecture does not apply yet for data as huge as those mentioned earlier. In order to attain such performances, the architecture had to be revisited. It led to the design of an original peer-to-peer transaction protocol: NEO. This protocol must also ensure both safety and reliability, which is not easy to achieve for distributed systems using traditional testing techniques.

The aim of the NEOPPOD project is to formally verify safety and reliability properties for the new NEO protocol. The process thus involves the model design, expected properties verification and eventual revision of the protocol according to the results obtained.

* This work is supported by FEDER Île-de-France/System@tic—libre software.

2 Challenges

The new NEO protocol is expected to handle clusters of 100 to 10,000 server nodes. Therefore, safety and reliability are critical issues. The project aims at considering both qualitative (e.g. data consistency) and quantitative (e.g. performance aspects) characteristics of the system. This requires the use of different formal methods that should be operated from a common specification for consistency purposes.

Modelling issues: Hence, designing an appropriate specification is a first challenge. Starting from the protocol description, a reverse-engineering process allows for extracting step-by-step a corresponding symmetric Petri net mode [1]. Since the original program description is very large and well structured, it is mapped to a modular specification. However, in order to mimic different configurations of the cluster architecture, as well as the different roles of the servers involved, w.r.t. the protocol operation, the model must also be highly parameterised.

Verification issues: Since numerous instances of several actors are involved in the system, the combinatorial explosion of the state space is a major difficulty. Dedicated techniques exploiting characteristics of distributed systems must be elaborated. These techniques rely on both exploiting symmetries and use of compact data structures such as decision diagrams. Finally, the hierarchical architecture must be exploited to separate local actions in the system from those affecting several components.

Since the model is highly modular, compositional and/or modular verification approaches must be investigated. This should be particularly interesting to check the dimensioning of the system elements by means of place bounds.

3 Expected outcome

The expected outcome of this project are :

- a modular specification designed with several levels of abstraction ;
- verification of critical properties of the protocol, such as data consistency, correct fault recovery, detection of bottlenecks ;
- pushing further the limits of verification tools and techniques, enhancing the CPN-AMI platform [3].

References

1. G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. A symbolic reachability graph for coloured Petri nets. *Theoretical Computer Science*, 176(1–2):39–65, 1997.
2. ERP5. *Central Bank Implements Open Source ERP5 in Eight Countries after Proprietary System Failed.* `http://www.erp5.com/news-central.bank`.
3. MoVe-Team. The CPN-AMI Home page, url: `http://www.lip6.fr/cpn-ami`.

Organized by the MeFoSyLoMa Group

Cédric ibisc lacl LIP6 Laboratoire Spécification et Vérification TELECOM ParisTech

Printed by Université Paris 13