UNIVERSITÄT
KOBLENZ · LANDAU
Institut für Softwaretechnik

FB 4
Informatik

Proceedings

14. Workshop

Algorithmen und Werkzeuge für Petrinetze

20.-21. September 2007

Stephan Philippi
Alexander Pinl
(Hrsg.)

Nr. 25/2007

Arbeitsberichte aus dem Fachbereich Informatik

Arbeitsberichte des Fachbereichs Informatik

ISSN (Print): 1864-0346
ISSN (Online): 1864-0850

Herausgeber / Edited by:

Der Dekan:
Prof. Dr. Paulus

Kontaktdaten der Verantwortlichen

Stephan Philippi, Alexander Pinl
Institut für Softwaretechnik
Fachbereich Informatik
Universität Koblenz-Landau
Universitätsstraße 1
D-56070 Koblenz
EMail: philippi@uni-koblenz.de; apinl@uni-koblenz.de

Vorwort

Die Workshop-Reihe 'Algorithmen und Werkzeuge für Petrinetze' wurde 1994 mit dem Ziel initiiert, in der deutschsprachigen Petrinetz-Community den fachlichen Austausch und die inhaltliche Zusammenarbeit zwischen den mit der Entwicklung und Analyse von Algorithmen beschäftigten Arbeitsgruppen und den im Bereich der Implementierung von Werkzeugen tätigen Arbeitsgruppen zu fördern.

Nach

- Berlin 1994,
- Oldenburg 1995,
- Karlsruhe 1996,
- Berlin 1997,
- Dortmund 1998,
- Frankfurt 1999,
- Koblenz 2000
- Eichstätt 2001,
- Potsdam 2002,
- Eichstätt 2003,
- Paderborn 2004,
- Berlin 2005 und
- Hamburg 2006

findet 2007 der Workshop 'Algorithmen und Werkzeuge für Petrinetze' zum 14. Mal statt. Veranstalter ist wie immer die Fachgruppe 0.0.1 'Petrinetze und verwandte Systemmodelle' der Gesellschaft für Informatik. Veranstaltungsort ist in diesem Jahr die Universität in Koblenz.

Der vorliegende Sammelband enthält die Vorträge, die auf dem Workshop präsentiert worden sind. Um auch die Vorstellung von noch unfertigen Ideen oder von in Entwicklung befindlichen Werkzeugen zu ermöglichen, fand wie in den vergangenen Jahren kein formaler Begutachtungsprozess statt. Die eingereichten Beiträge wurden lediglich auf ihre Relevanz für das Workshop-Thema hin geprüft.

Eichstätt, Lyngby, Dortmund, Koblenz, Rostock und Hamburg im September 2007

J. Desel, G. Juhás, E. Kindler, P. Kemper, K. Lautenbach, K. Wolf, R. Valk

Inhaltsverzeichnis

Analysis of workflow graphs through SESE decomposition

Jussi Vanhatalo[1,2], Hagen Völzer[1], and Frank Leymann[2]

[1] IBM Zurich Research Laboratory, Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland
{juv,hvo}@zurich.ibm.com
[2] Institute of Architecture of Application Systems, University of Stuttgart
Universitätsstrasse 38, D-70569 Stuttgart, Germany
frank.leymann@iaas.uni-stuttgart.de

We describe a technique that enhances control-flow analysis of business process models in two ways: It considerably speeds up the analysis and it improves the diagnostic information that is given to the user for fixing the control-flow errors.

The control-flow of a business process model can be captured as a *workflow graph*, which is essentially the same as a free-choice workflow net. The absence of control-flow errors, such as *deadlocks* and *lack of synchronization* is often called *soundness* of the workflow graph.

Our technique consists of two parts: Firstly, the workflow graph is uniquely decomposed into a hierarchy of single-entry-single-exit (SESE) fragments, which are usually substantially smaller than the original graph. Checking soundness of the workflow graph then amounts to checking soundness of each fragment in isolation. This decomposition into fragments can be done in linear time.

Secondly, we use a fast heuristic that can decide soundness of many of the fragments occurring in practice. Any remaining fragments that are not covered by the heuristic can then be analyzed using any known complete soundness checker.

We also report on a case study where we used our technique with more than 340 real business processes modeled with the IBM WebSphere Business Modeler.

The described work is published in [1]. It was partially supported by the SUPER project (http://www.ip-super.org/) under the EU 6th Framework Programme Information Society Technologies Objective (contract no. FP6-026850).

References

1. Jussi Vanhatalo, Hagen Völzer, and Frank Leymann. Faster and more focused control-flow analysis for business process models through SESE decomposition. In Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *ICSOC*, volume 4749 of *Lecture Notes in Computer Science*, pages 43–55. Springer, 2007.

A Visual Editor for Reconfigurable Object Nets based on the ECLIPSE Graphical Editor Framework

Enrico Biermann, Claudia Ermel, Frank Hermann and Tony Modica
Technische Universität Berlin, Germany

{enrico,lieske,frank,modica}@cs.tu-berlin.de

Abstract

The main idea behind *Reconfigurable Object Nets (RONs)* is the integration of transition firing and rule-based net structure transformation of place/transition nets during system simulation. RONs are high-level nets with two types of tokens: object nets (place/transition nets) and net transformation rules (a dedicated type of graph transformation rules). Firing of high-level transitions may involve firing of object net transitions, transporting object net tokens through the high-level net, and applying net transformation rules to object nets. Net transformations include net modifications such as merging or splitting of object nets, and net refinement. This approach increases the expressiveness of Petri nets and is especially suited to model mobile distributed processes. The paper presents a visual editor for RONs which has been developed in a student project at TU Berlin in summer 2007. The visual editor itself has been realized as a plug-in for ECLIPSE using the ECLIPSE Modeling Framework (EMF) and Graphical Editor Framework (GEF) plug-ins.

Keywords: Petri nets, net transformation, graph transformation, visual editor, reconfigurable object nets, Eclipse, GEF

1 Introduction

Modelling the adaption of a system to a changing environment has become a significant topic in recent years. Application areas cover e.g. computer supported cooperative work, multi agent systems, dynamic process mining or mobile ad-hoc networks (MANETs). Especially in the context of our project *Formal modeling and analysis of flexible processes in mobile ad-hoc networks* [PEH07, For06] we aim to develop a formal technique which on the one hand enables the modeling of flexible processes in MANETs and on the other hand supports changes of the network topology and the transformation of processes. This can be achieved by an appropriate integration of graph transformation, nets and processes in high-level net classes.

The main idea behind *Reconfigurable Object Nets (RONs)* is the integration of transition firing and rule-based net structure transformation of place/transition nets during system simulation. RONs are high-level nets with two types of tokens: object nets (place/transition nets) and net transformation rules (a dedicated type of graph transformation rules). Thus, on the one hand, RONs follow the paradigm "nets as tokens", introduced by Valk in [Val98], and, on the other hand, extend this paradigm to "nets and rules as tokens" in order to allow for modelling net structure changes (reconfigurations) of object nets. Firing of RON-transitions may involve firing of object net transitions, transporting object net tokens through the high-level net, and applying net transformation rules to object nets. Net transformation rules model net modifications such as merging or splitting of object nets, and net refinements.

A rule r consists of a left-hand side L related to a right-hand side R and describes the local replacement of L by R. Similar to the concept of graph transformations [EEPT06], each application of a rule $r = (L \rightarrow R)$ to a source net N_1 leads to a net transformation step $N_1 \stackrel{r}{\Longrightarrow} N_2$, where in the source net N_1 the subnet corresponding

to the left-hand side L is replaced by the subnet corresponding to the right-hand side R, yielding the target net N_2. Rule-based Petri net transformations have been treated in depth in e.g. [EP04, PU03].

The formal basis for RONs is given in [HME05], where high-level nets with nets and rules as tokens are defined algebraically, based on algebraic high-level nets [PER95]. Here we present a visual editor for RONs which has been developed in a student project at the TU Berlin in summer 2007. The visual editor itself has been realized as a plug-in for ECLIPSE using the ECLIPSE Modeling Framework (EMF) [EMF06] and Graphical Editor Framework (GEF) [GEF06] plug-ins. In RONs, as presented in this paper, the algebraic operations defined for rule applications and transition firing are modeled as special RON-transition types which have a fixed firing semantics. This facilitates the implementation of the RON editor and simulator since no parser for algebraic specifications is needed. It turned out that the four RON-transition types are adequate to model various interesting examples. More information on RONs, case studies and downloads of the RON tools are available on our RON homepage [RON07].

Section 2 introduces RONs along the running example of distributed producers/consumers. After a brief overview of the ECLIPSE Graphical Editor Framework, the milestones of the students' project and the different parts of the RON tool are explained in Section 3. Section 4 gives an outlook on future extensions of the tool.

2 Reconfigurable Object Nets: An Example

In our example RONs are applied to model a distributed system of producers and consumers where several producers and consumers may interact with each other. In the initial state of the sample RON in Fig. 1 potential producers and consumers are distributed on different Net places as independent object nets without interaction. Producer

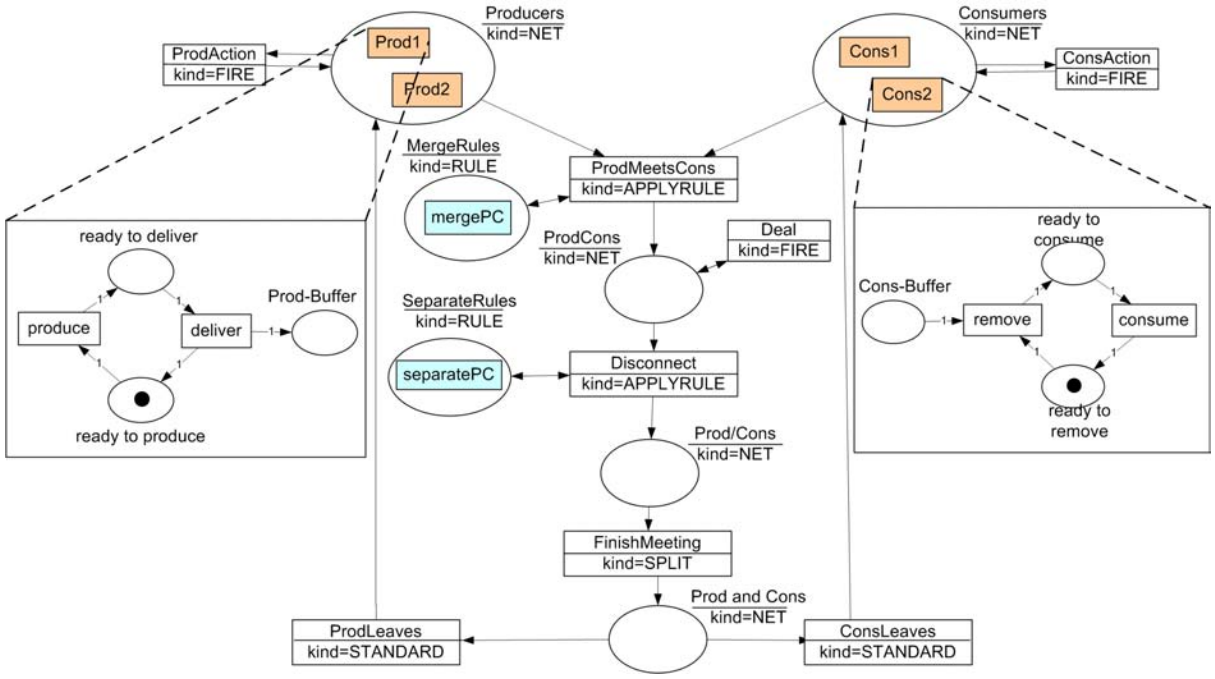

Figure 1: Distributed Producers/Consumers modelled as RON

nets may fire, e.g. they can produce items and place them on the buffer place. Firing in object nets is triggered by firing a RON transition of type FIRE, which takes one object net with marking M from the Net place in its pre-domain and puts the same object net, now marked by one of the possible successor markings of M, into all of

its post-domain places. For producer-consumer interaction, a producer net can be merged with a consumer net by firing the RON transition *ProdMeetsCons* of type APPLYRULE. A transition of this type takes an object net from each of the pre-domain Net places, a rule from the pre-domain Rule place, applies this rule to the disjoint union of all the taken object nets and puts the resulting net to all post-domain Net places. The rule *merge-PC*, depicted in Fig. 2, glues a producer object net and a consumer object net by inserting a *connect* transition between both buffers. A so-called negative application condition (NAC) forbids the application of the rule if there already exists a *connect* transition. Note that the transition *ProdMeetsCons* controls which producer interacts with which consumer.

By firing the FIRE transition *Deal*, in the glued net the consumer now can consume items produced by the producer as long as there are tokens on the place *Prod-Buffer*. Moreover, the producer may also produce more items and put them to the buffer. After the deal has been finished, the nets are separated again by firing the APPLYRULE transition *Disconnect*. This applies rule *separate-PC* in Fig. 2 to the glued net which deletes the *connect* transition from the net.

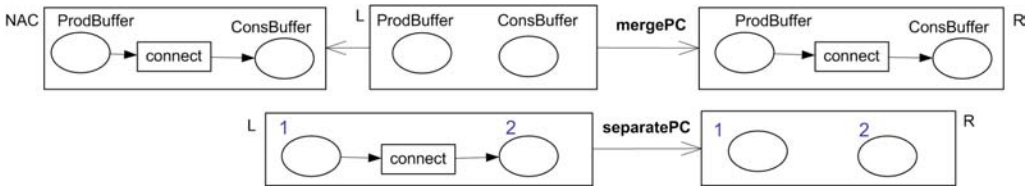

Figure 2: Rules for gluing and for separating a producer and a consumer net

Note that the resulting net, which is put on place *Prod/Cons*, is still one single object net which consists of two unconnected components. In order to split these components into two object nets, a transition of type SPLIT has to be fired. Firing RON transition FinishMeeting results in two separate object nets on place *Prod and Cons*. In a last step, the now separated producer and consumer nets are returned to their initial places by firing the STANDARD transitions *ProdLeaves* and *ConsLeaves*. STANDARD transitions simply remove a net token from each pre-domain place and add the disjoint union of all removed object nets to each of the post-domain Net places. In our example, this means that the STANDARD transitions may also put the object nets back to the "wrong" initial place. It is possible to avoid such behaviour by using APPLYRULE transitions instead where rules check the object nets for the occurrence of a *producer* or *consumer* place, respectively.

3 An Editor for RONs

Conceptually, the visual editor for RONs is divided into four main components which were scheduled to be implemented in four milestones, building on each other, each resulting in software which could be run and tested. Hence, the project consisted of short development cycles leading to fast results and error recognition.

Similar student projects dealing with the development of visual editors as ECLIPSE plug-ins have been carried out by our research group for some years now. It emerged that after a short period of getting used to GEF the students could concentrate on devising and implementing advanced domain specific editing features.

In addition to the usual advantages like Java's platform independence and ECLIPSE being a powerful IDE and available as open source we could make use of ECLIPSE's highly extensible plug-in architecture. We employ the ECLIPSE Modeling Framework (EMF) to automatically generate data model code for the editor from class diagrams and to handle persistence operations.

The Graphical Editor Framework (GEF) provides means of implementing a visual editor based on the Model-View-Controller pattern relying on ECLIPSE's Standard Widget Toolkit (SWT). It supports many operations and features common to most graphical editors like zooming, various layouting of figures, support for drag & drop etc.

To prevent the students from struggling with ECLIPSE's internal details especially at the beginning of the project we provided an abstract framework of an editor demonstrating several concepts and techniques ECLIPSE and GEF offer, e.g. graphical views showing multiple models for rules.

In the following, we present the results of the four milestones, i.e. the RON tree view based on the EMF model for RONs, and the visual editors for object nets, for transformation rules and for high-level nets with the four transition types FIRE, APPLYRULE, SPLIT and STANDARD. Fig. 3 shows a screenshot of the RON editor showing all views and editors.

RON Tree View. View 1 in Fig. 3 shows the main editor component, a tree view for the complete RON model from which the graphical views can be opened by double-click.

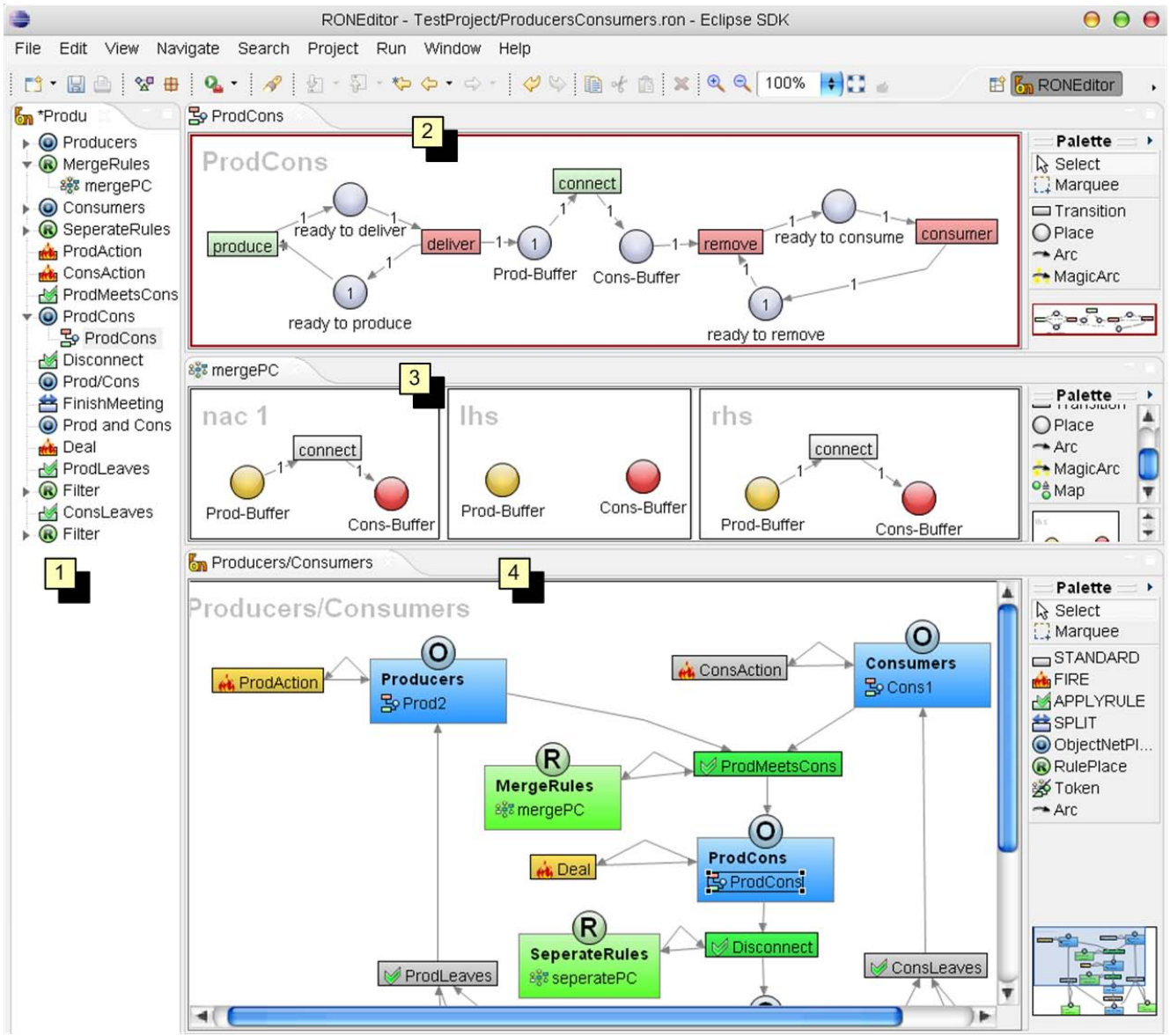

Figure 3: The RON Environment for Editing and Simulating Reconfigurable Object Nets

Object Net Editor. The first component to be implemented was a graphical editor for object nets, i.e. the net tokens on the RON's net-typed places. This component is actually a place/transition net editor, allowing the simulation of firing transitions. Its implementation could be reused for the rule editor and the RON editor as well. An object editor panel is shown in Fig. 3, View 2, holding the object net *ProdCons*, which models producer-consumer interaction.

Transformation Rule Editor. In the second milestone the RON framework was extended by an editor for transformation rules. It mainly consists of three editor panels, one for the left-hand side (LHS), one for the right-hand side(RHS) and one for a negative application condition(NAC) (see view 3 in Fig. 3). Each editor panel is basically an object net editor itself, but with the additional possibility to relate the object nets by defining mappings on places and transitions. Mappings are realized by the *mapping tool* of the rule editor that allows the matching of LHS objects to RHS or NAC objects to indicate which objects are preserved by the rule. In the editor, mappings are indicated by corresponding object colouring. In order to ensure that the mapping specified by the mapping tool is also a valid Petri net morphism, it is checked for each mapped transition that all places in its pre- (post-)domain in the LHS are mapped to the corresponding places in the pre- (post-)domain of in the RHS. Another restriction is that all rules are injective, so different LHS objects must be mapped to different RHS objects. Note that the object net *ProdCons* in Fig. 3, View 2, is the result from applying rule *mergePC* to two object nets *Prod1* and *Cons2* from the places *Producers* and *Consumers*, respectively. (For the situation before the rule is applied, see Fig. 1).

High-Level Net Editor. The third extension of the RON editor involved editing of high-level nets which control object net behaviour and rule applications to object nets. Such a high-level net is drawn in the RON editor panel, shown in Fig. 3, View 4. Here, NET places carrying object net tokens are blue containers marked by an "O" for *Object Nets*. RULE places carrying transformation rules are green containers marked by an "R" for *Rules*. Each transition type has a special graphical icon as visualization: for FIRE, for APPLYRULE, for SPLIT, and for STANDARD. Enabled RON-transitions are coloured, disabled ones are gray.

Simulation of RONs. A RON transition is fired when double-clicked. The simulation of firing transitions of kinds STANDARD, FIRE, and SPLIT has been implemented directly in the editor. In order to simulate firing of APPLYRULE transitions, internally the RON editor was extended by a converter to AGG, an engine to perform and analyze algebraic graph transformations [AGG]. If the user gives the command to fire an APPLYRULE transition he has to select the rule and the object net token(s) in the pre-domain the rule should be applied to. This is realized in the user interface shown in Fig. 3, View 4, by ordering the tokens in the corresponding NET and RULE containers in a way that the uppermost tokens are the ones considered by the rule application. Furthermore, the user is asked for a match defining the occurrence of the rule's left-hand side in the selected object net. Optionally, AGG can find or complete partial matches and propose them to the user in the RON editor. With the selected rule, match, and object net AGG computes the result of the transformation which is put on the post-domain places according to the firing semantics explained in Section 2.

4 Conclusion and Ongoing Work

Modelling mobile and distributed systems requires a modelling language which covers both, change and coordination. RONs meet these conditions and we have shown its power on a reduced but instructive example modelling interaction between producers and consumers. The abstract high-level net controls the flow, selection and use of object tokens being object nets but also rules for reconfiguration. Thus the details on the object level are hidden in these tokens such that they can be modelled separately keeping the view on the high-level net concise.

Currently, RONs use specialized transitions in the high-level view, but we intend to extend the presented editor to cover full algebraic high-level nets, such that a high-level transition is controlled by firing conditions and arc inscriptions consisting of terms and attributes. Furthermore, rule specification shall be extended to cope also with P/T net morphisms which are *not* injective. This would allow net transformations like gluing and cloning while RONs are restricted to connecting object nets via new P/T net transitions. Additionally, we plan to allow tokens in rules, such that a rule application will depend on the presence of tokens, i.e. on the state of an object net. Finally, the editor for RONs shall be extended by analysis features like a check of conflicts between object net transition firing and rule application according to [EHP+07].

References

[AGG] AGG Homepage. http://tfs.cs.tu-berlin.de/agg.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer Verlag, 2006.

[EHP+07] H. Ehrig, K. Hoffmann, J. Padberg, U. Prange, and C. Ermel. Independence of net transformations and token firing in reconfigurable place/transition systems. In *Proc. Conf. on Application and Theory of Petri Nets and Other Models of Concurrency*, volume 4546 of *LNCS*, pages 104–123. Springer, 2007.

[EMF06] Eclipse Consortium. *Eclipse Modeling Framework (EMF) – Version 2.2.0*, 2006. http://www.eclipse.org/emf.

[EP04] H. Ehrig and J. Padberg. Graph grammars and Petri net transformations. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 496–536. Springer, 2004.

[For06] ForMA$_l$NET. DFG Project, Technical University of Berlin. *Formal Modeling and Analysis of Flexible Processes in Mobile Ad-hoc Networks*, 2006. http://www.tfs.cs.tu-berlin.de/formalnet.

[GEF06] Eclipse Consortium. *Eclipse Graphical Editing Framework (GEF) – Version 3.2*, 2006. http://www.eclipse.org/gef.

[HME05] K. Hoffmann, T. Mossakowski, and H. Ehrig. High-Level Nets with Nets and Rules as Tokens. In *Proc. Conf. on Application and Theory of Petri Nets and other Models of Concurrency*, volume 3536 of *LNCS*, pages 268–288. Springer, 2005.

[PEH07] J. Padberg, H. Ehrig, and K. Hoffmann. Formal modeling and analysis of flexible processes in mobile ad-hoc networks. *EATCS Bulletin*, 91:128–132, 2007.

[PER95] J. Padberg, H. Ehrig, and L. Ribeiro. Algebraic high-level net transformation systems. *Mathematical Structures in Computer Science*, 5:217–256, 1995.

[PU03] J. Padberg and M. Urbášek. Rule-Based Refinement of Petri Nets: A Survey. In Ehrig et al. *Advances in Petri Nets: Petri Net Technology for Communication Based Systems*, volume 2472 of *LNCS*, pages 161–196. Springer, 2003.

[RON07] Student's Project. TFS, Technical University of Berlin. *Reconfigurable Object Nets*, 2007. http://www.tfs.cs.tu-berlin.de/roneditor.

[Val98] R. Valk. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In *Proc. Conf. on Application and Theory of Petri Nets*, volume 2987 of *LNCS*, pages 1–25. Springer, 1998.

Snoopy - A Tool to Design and Animate/Simulate Graph-Based Formalisms

Monika Heiner Ronny Richter Martin Schwarick

Brandenburg University of Technology at Cottbus
Dep. of CS, Data Structures and Software Dependability
Postbox 10 13 44, 03013 Cottbus, Germany
snoopy @ informatik.tu-cottbus.de
http://www-dssz.informatik.tu-cottbus.de/software/snoopy.html

Abstract

We sketch the fundamental properties and features of Snoopy, a tool to model and execute (animate, simulate) hierarchical graph-based system descriptions. The tool comes along with several pre-fabricated graph classes (in particular some kind of Petri nets and other related graphs), and facilitates a comfortable integration of further graph classes due to its generic design. To support an aspect-oriented model engineering, different graph classes may be used simultaneously. Our tool runs on Windows and Linux operating systems, and is available free of charge for non-commercial use.

1 Preliminaries

Snoopy [Webd] is a generic and adaptive tool for modelling and animating/simulating of hierarchical graph-based formalisms. While concentrating our development as far on several kinds of Petri nets and related graph classes, the generic design of Snoopy facilitates also a comfortable extension by new graph classes. The simultaneous use of several graph classes is supported by the dynamic adaptation of the graphical user interface to the active one. So it is possible to treat qualitative and quantitative models of the system under investigation side by side. For example you can start with a qualitative Petri net model and check some essential properties with external analysis tools. To get a basic understanding of the causal dependencies it is possible to play the token game. Later you can easily move on to related quantitative models, deterministic, stochastic or continuous ones, for a deeper understanding of the time dependencies governing your system. These quantitative models can be simulated with internal or external tools. This integrating approach was employed in [GH06], [GHL07]. To support this style of model engineering it is possible to convert between different graph classes, obviously with loss of information in some directions.

Snoopy runs on Windows and Linux operating systems; it is available free of charge for non-commercial use, and can be obtained from our website [Webd]. The source code is available on request.

2 Graph Independent Features

Snoopy provides for all graph classes some consistently available generic features, e.g. edit (copy, paste, cut), and layout (mirror, flip, rotate, and automatic layout by Graphviz [Weba]), as well as some graphical file export for documentation purposes (eps, Xfig, FrameMaker).

Graph constraints permit only the creation of syntactically correct models of the implemented graph classes.

The construction of large graphs is supported by a general hierarchy concept of subgraphs (represented as macro nodes) and by logical (fusion) nodes, which serve as connectors of distributed net parts. Additionally, colours or different shapes of individual graph elements may be used to highlight special functional aspects (compare figure 1).

A generic interaction mode allows a communication between different graphs. Some events in one graph can trigger commands (colouring, creating or deleting of graph elements) in another graph, even if they are of different graph classes [Dub07].

Furthermore, a dynamic colouring of graph elements for the visualization of paths or node sets (e.g. P/T-invariants, structural deadlocks, traps) is available [Win06].

A digital signature by md5 hash ensures the structure and the layout of the graph separately, which increases the confidence in former analysis results during model development [Dub05].

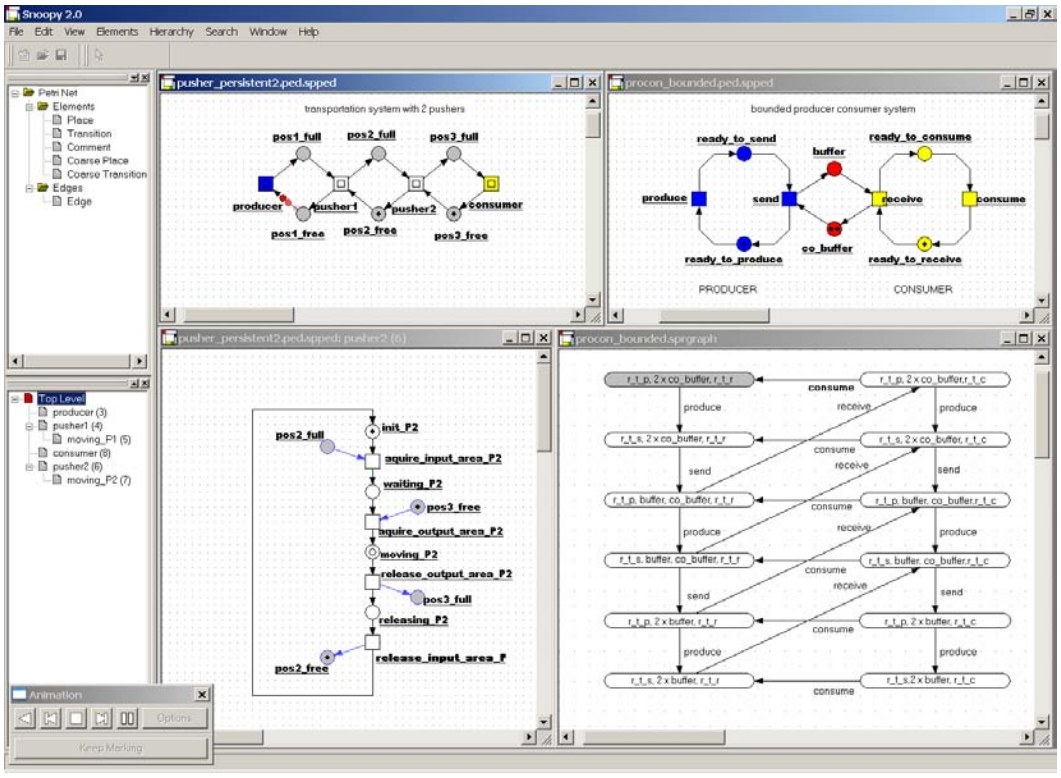

Figure 1: Snoopy Screenshot.

3 Realized Graph Classes

3.1 Reachability Graph

This simple graph class supports just one node and one arc type, besides comment nodes. The graph nodes can carry a name, a description and a Petri net state, which consists of a list of places and their markings. The arcs may be labelled by arbitrary character strings.

3.2 Petri Net

This bipartite graph class allows qualitative modelling by the standard notion of place/transition Petri nets. An animation by the token game gives first insights into the dynamic behavior and the causality of the model. The token game may be played step-wise or fully automated in forward or backward direction with different firing rules (maximal, minimal or intermediate steps). For modelling of large systems the hierarchy concept and fusion nodes have been proven to be useful.

The generic interaction manager [Dub07] allows to construct the reachability graph driven by the token flow animation of the Petri net. Furthermore, the export to a wide range of external analysis tools is available, among them INA, Lola, Maria, MC-Kit, Pep, Prod, Tina (see [Webc] for tool descriptions) as well as to our own toolbox Charlie [Sch06b]. Additionally, an import of the APNN file format supports advanced model sharing with other Petri nets tools.

3.3 Extended Petri Net

This graph class enhances place/transition Petri nets by some special arc types: read arcs, reset arcs, equal arcs and inhibitor arcs. A token flow animation and exports to external analysis tools are available, as in standard Petri nets. The special arc types are accepted only in the export to the APNN file format, which supports these graph elements.

3.4 Time Petri Net

This class introduces place/transition Petri nets with time. Up to now, time durations or time intervals can be assigned to transitions. The net analysis is supported by an export to INA.

3.5 Continuous Petri Net

Continuous Petri nets [Sch06a] may be considered as a structured approach to write systems of ordinary differential equations (ODEs), which are commonly used for a quantitative description of biochemical reaction networks (cf. section 4). For simulation several numerical algorithms (12 stiff/unstiff solvers, among others Runge-Kutta and Rosenbrock) are implemented, as well as an export to SBML [Webe] to use external analysis tools, popular in the systems biology community. Moreover, the ODEs defined by a continuous Petri net can be generated in LaTeX style.

3.6 MTBDD

For teaching purposes and documentation of small case studies we implemented multi-terminal binary decision diagrams.

3.7 Fault Tree and Extended Fault Tree

Fault trees describe the dependencies of component-based systems in failure conditions and are commonly used in risk management of systems with high dependability demands. Snoopy allows the qualitative and quantitative evaluation of fault trees. Several dependability measures may be computed, for example reliability, probability of system failure, and mean time to failure [Kur07].

3.8 Miscellaneous

The generic design of our graph tool allows an uncomplicated extension by new graph classes. For example EDL signatures (a formalism to describe patterns of computer network attacks) have been realized in [Roh07]. Snoopy is also involved in the tool chain of the approach for embedded system design presented in [GBC07]. You might want to find your own favorite graph class in a future version of this paper - we are open for suggestions and cooperations.

4 Case Studies

Snoopy has been used for a wide range of case studies, technical as well as biochemical ones, such as the control software of a production cell [HDS99], the metabolism of the potato tuber [KJH05], or various signal transduction networks [HKW04], [Neu04], [GH06], [GHL07] – just to mention a few due to space limitations.

5 Implementation

Snoopy was started in 1997 as a student's project [Men97], [Fie04] and is still under development and maintenance. It is based on the experience gathered by its predecessor PED, which it replaces. The tool is written in the programming language C++ with use of the Standard Template Library. A crucial point of the development is its platform-independent realisation, so Snoopy is now available for Windows and Linux operating systems. For this purpose, the graphical user interface employs the framework wxWidgets [Webf]. The object-oriented design (compare Figure 2) uses several design patterns, thus special requirements may be added easily. Due to a strict separation of internal data structures and graphical representation it is straightforward to extend Snoopy by a new graph class.

6 Future Work

In [GHL07], an integrative approach for biochemical network modelling has been proposed, using qualitative as well as stochastic and continuous Petri nets. This technology will be supported by the extension developed in [Leh07]. The import

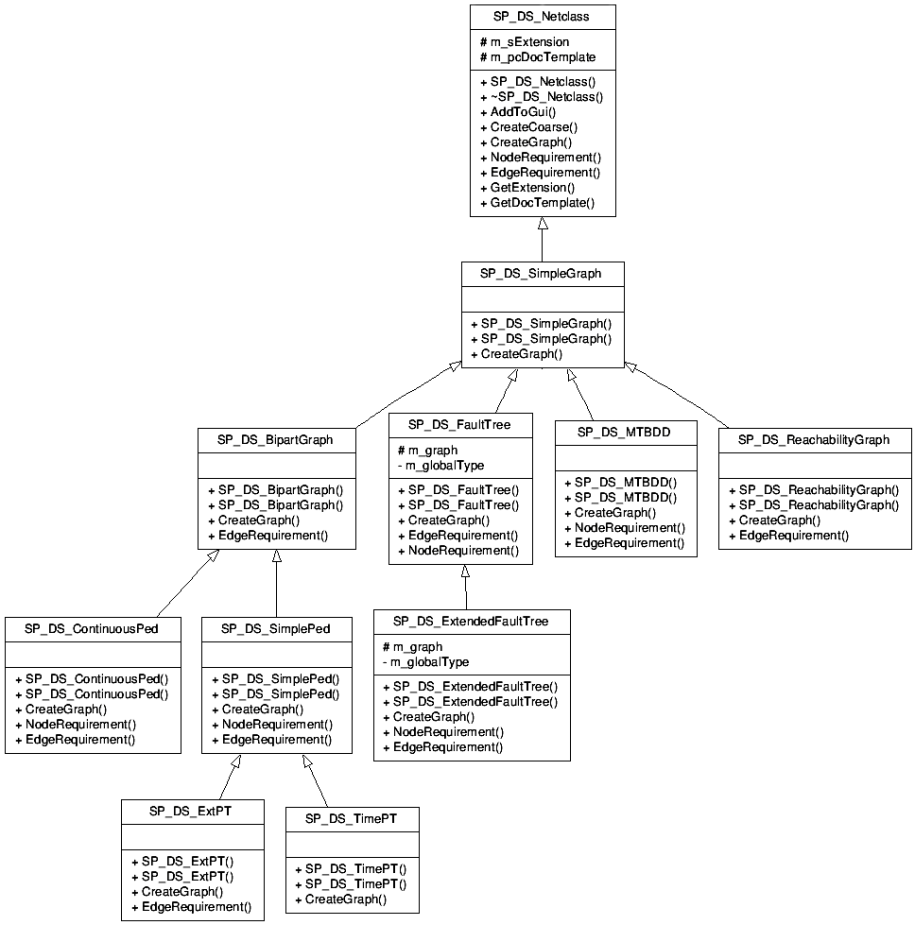

Figure 2: Extract of Snoopy's class hierarchy as generated by Doxygen.

of biochemical network models in the KEGG and SBML data formats is about to be released [Sch07], which will allow the direct re-use and re-engineering of models from the systems biology community. An ongoing student's project works on managing and executing animation sequences, especially counter examples produced by external analysis tools.

Finally, a PNML [Webb] import and export will be implemented, as soon as a (preliminary) final standard will be available.

References

[Dub05] Matthias Dube. Signing and Verifying SNOOPY files. Technical Report, Universit degli Studi di Milano, Dipartimento di Informatica e Comunicazione, 2005.

[Dub07] Matthias Dube. Design and Implementation of a Generic Concept for an Interaction of Graphs in Snoopy (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 2007.

[Fie04] Markus Fieber. Design and Implementation of a Generic and Adaptive Tool for Graph Manipulation (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 2004.

[GBC07] Luis Gomes, Joao Barros, and Aniko Costa. Petri Nets Tools and Embedded Systems Design. In *Proc. Workshop on Petri Nets and Software Engineering (PNSE07) at Int. Conf. on Application and Theory of Petri Nets (ICATPN '07 Siedlce)*, pages 214–219, 2007.

[GH06] David Gilbert and Monika Heiner. From Petri Nets to Differential Equations - an Integrative Approach for Biochemical Network Analysis;. In *Proc. 27th ICATPN 2006, LNCS 4024*, pages 181–200. Springer, 2006.

[GHL07] David Gilbert, Monika Heiner, and Sebastian Lehrack. A Unifying Framework for Modelling and Analysing Biochemical Pathways Using Petri Nets. In *Proc. CMSB 2007, LNCS/LNBI 4695*, pages 200–216. Springer, 2007.

[HDS99] Monika Heiner, Peter Deussen, and Jochen Spranger. A Case Study in Design and Verification of Manufacturing Systems with Hierarchical Petri Nets. *Journal of Advanced Manufacturing Technology*, 15:139–152, 1999.

[HKW04] Monika Heiner, Ina Koch, and Jürgen Will. Model Validation of Biological Pathways Using Petri Nets - Demonstrated for Apoptosis. *BioSystems*, 75:15–28, 2004.

[KJH05] Ina Koch, Björn H. Junker, and Monika Heiner. Application of Petri Net Theory for Modeling and Validation of the Sucrose Breakdown Pathway in the Potato Tuber. *Bioinformatics*, 21(7):1219–1226, 2005.

[Kur07] Anja Kurth. Fault Trees in Snoopy (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 2007.

[Leh07] Sebastian Lehrack. Stochastic Petri Nets in Snoopy (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., in preparation, 2007.

[Men97] Thomas Menzel. Design and Implementation of a Framework for Petri Net Oriented Modelling (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 1997.

[Neu04] Gerry Neumann. Modeling of Biochemical Processes with Petri Nets; Hemostasis vs. Fibrinolysis vs. Inhibitors (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 2004.

[Roh07] Christian Rohr. Design and Implementation of an Editor for Visualizing and Debugging of EDL Signatures (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 2007.

[Sch06a] Daniel Scheibler. A Tool to Design and Simulate Continuous Petri Nets (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 2006.

[Sch06b] Martin Schwarick. A Tool to Analyze Petri Nets (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 2006.

[Sch07] Daniel Schrödter. Re-engineering of Biochemical Networks (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., submitted, 2007.

[Weba] Website Graphviz. Graph Visualization Software. `http://www.graphviz.org` last visit: 08/2007.

[Webb] Website Petri Net Markup Language. PNML Framework Release 1.2.0. `http://www-src.lip6.fr/logiciels/mars/PNML/`, last visit: 08/2007.

[Webc] Website Petri Nets World. The Home Site. `http://www.informatik.uni-hamburg.de/TGI/PetriNets/`, last visit: 08/2007.

[Webd] Website Snoopy. A Tool to Design and Animate/Simulate Graphs. `http://www-dssz.informatik.tu-cottbus.de/software/snoopy.html`.

[Webe] Website Systems Biology Markup Language. The Home Site. `http://sbml.org/index.psp`, last visit: 08/2007.

[Webf] Website wxWidgets. A Toolkit for cross-platform GUI Application. `http://www.wxwidgets.org`, last visit: 08/2007.

[Win06] Katja Winder. Invariant-based Structural Characterization of Petri Nets (in German). Master's thesis, Brandenburg University of Technology Cottbus, Computer Science Dept., 2006.

Aspekte der Modularität und Flexibilität im Analyse-Framework des Tools NeMo (ToMASEn)

Alexander Pinl*, Markus Pinl, Jan Poganski, Stephan Philippi

Universität Koblenz-Landau, Institut für Informatik,
Postfach 201 602, 56016 Koblenz
{apinl, mpinl, janp, philippi}@uni-koblenz.de

Zusammenfassung

Für die Modellierung komplexer technischer Systeme ist eine adäquate Tool-Unterstützung essenziell. Stetig wachsende Anforderungen an die modellbasierte Analyse technischer Systeme sowie neue Algorithmik und Auswertungen führen zu einer steigenden Komplexität der Software-Komponenten für die Analyse, welche daher modular, leicht erweiterbar und flexibel ausgelegt sein müssen. Gleichzeitig werden hohe Anforderungen an die Effizienz und Performance der Algorithmen gestellt. Dieser Beitrag gibt Einblicke in die Konzeption des Analyseframeworks des Tools *NeMo*, welches unter anderem für die Modellierung und Analyse sicherheitskritischer technischer Systeme im Eisenbahnverkehr eingesetzt wird.

1 Analyseframework

Als wesentlicher Bestandteil von NeMo[1] hat das Analyseframework seit seiner Erwähnung in [PhPiRa05] und [Phil*06] einen grundlegenden Ausbau erfahren. Dabei lag der Schwerpunkt weniger auf der Integration weiterer Analyseverfahren, sondern vielmehr darauf

- einen einfachen, aber flexiblen Zugriff auf die Analysemöglichkeiten zu schaffen,
- analysebezogene Daten geschickt zu verwalten und
- eine performante Durchführung von Analysevorgängen zu ermöglichen.

Doch auch der Erweiterung der Analysemöglichkeiten wurde Rechnung getragen, was im Abschnitt 2 für den Bereich der Erreichbarkeitsanalyse näher ausgeführt wird. Es folgt nun die aktuelle Konzeption des Analyseframeworks.

1.1 Gestaltung der Benutzungsoberfläche

Einen groben Eindruck, wie dem Benutzer die Analysefunktionalität von NeMo zugänglich gemacht wird, zeigt Abb. 1. Zu einem aktuell geöffneten Graphen zeigt die *Analysis-Control*-Sicht dem Benutzer, welche Analyseverfahren grundsätzlich auf das gerade betrachtete Modell anwendbar sind. Dabei sind diese hierarchisch angeordnet und mit Hilfe von Checkboxen einzeln oder gruppiert auswählbar. Um einen Analysedurchlauf zu starten, zu unterbrechen oder abzubrechen, stellt die Sicht außerdem drei Schaltflächen zur Verfügung. Zu jeder Analyse wird angezeigt, ob schon ein aktuelles oder ein veraltetes Ergebnis vorliegt.

Die *Analysis-Result*-Sicht hat den Hauptzweck, die vorliegenden Analyseergebnisse anzuzeigen. Nach Auswahl eines Ergebnisses werden nützliche Informationen, wie z.B. die Berechnungsdauer, ersichtlich. Möchte man verschiedene Analyseergebnisse gleichzeitig anzeigen, bietet es sich an zu diesem Zweck weitere Instanzen der *Analysis-Result*-Sicht zu nutzen. Diese Flexibilität ist auch gerade dann von Vorteil, wenn man auf verschiedenen Graphen die gleiche Untersuchung macht und die Ergebnisse anschließend miteinander vergleichen will. Sollte es Ergebnisse geben, welche nicht übersichtlich in der beschriebenen Sicht dargestellt werden können, sollen diese zumindest von dort aus aufrufbar sein.

*A. Pinl wird von der DFG unterstützt (GZ: LA 1042/7-2).

[1] Net Modeling Toolkit. Das Tool wurde auf der AWPN 2005 in Berlin und der AWPN 2006 in Hamburg im jeweils aktuellen Entwicklungsstand vorgestellt. [PhPiRa05, Phil*06]

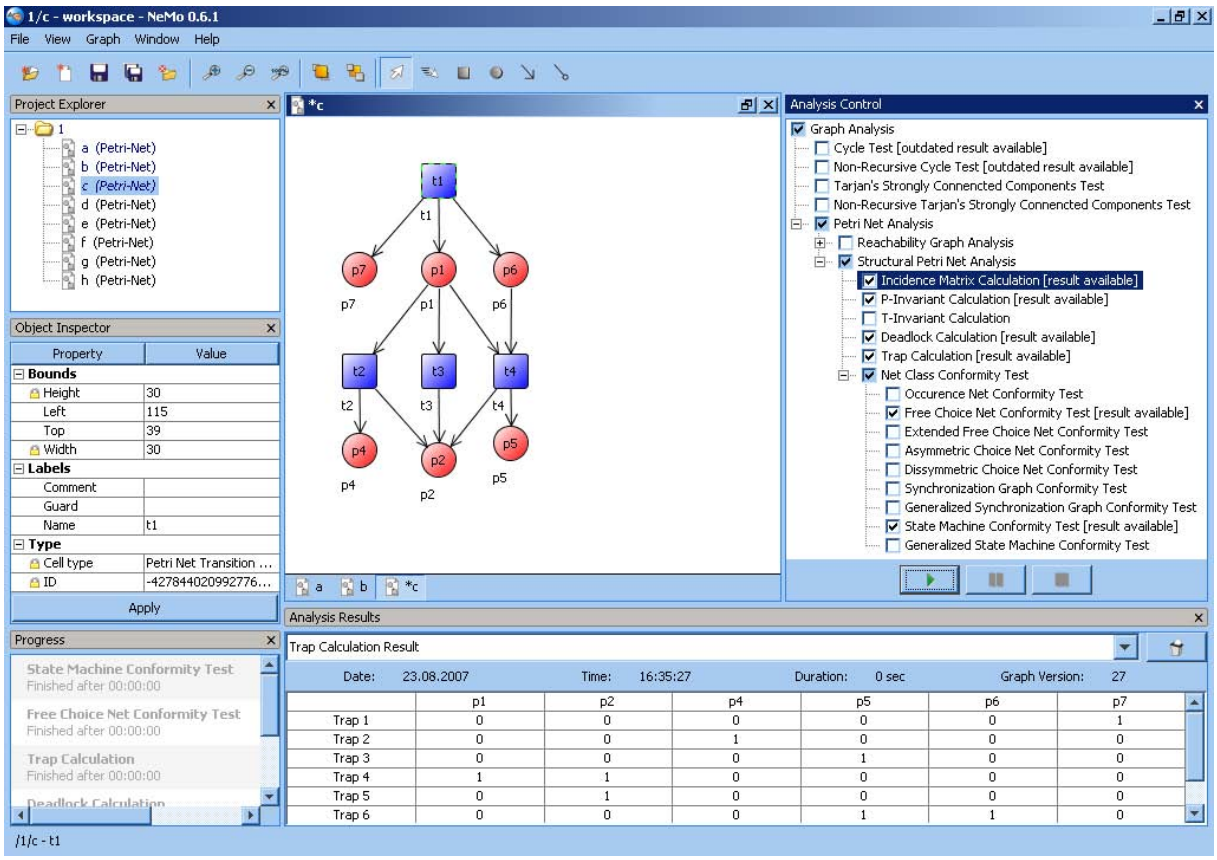

Abbildung 1: Benutzungsoberfläche

1.2 Organisation analysebezogener Daten

Die in Abb. 1 ersichtliche Baumstruktur, in der Analyseverfahren in Kategorien zusammengefasst sind, ermöglicht nicht nur die einfache Auswahl der gewünschten Verfahren für einen Analysevorgang, sondern verwaltet auch alle zugehörigen Ergebnisse. Das zugrunde liegende hierarchische Datenmodell wird passend zum jeweiligen Graphtyp zusammengestellt. So wird beispielsweise für ein Petrinetz das Analysemodell für Graphen übernommen und um alle Petrinetz-spezifischen Analyseverfahren erweitert. Neben einfachen Baumknoten, welche eine Analysekategorie repräsentieren, gibt es in einem solchen Analysemodell ausführbare Knoten, welche sich dadurch auszeichnen, dass sie mit einem Analyseverfahren verknüpft sind und ein Ergebnis aufnehmen können.

Ausgangspunkt aller Analysen sind immer Graphen und sämtliche Analyseergebnisse gehören letztlich zu den Graphen, aus denen sie ursprünglich hervorgehen. Auch für den Fall einer mehrstufigen Analyse, wenn also Analyseergebnisse selbst wieder einer Analyse unterzogen werden, bleibt der Bezug zum Graphen erhalten. Um diesen eindeutigen Bezug sicherzustellen, muss das Datenmodell einer analysierbaren Graphklasse als analysierbar gekennzeichnet werden. In der Hierarchie der Analyseklassen, von der Abb. 2 einen Ausschnitt darstellt, schlägt sich dies auch darin nieder, dass jede Analyse ein analysierbares Graphmodell erwartet.

Die Klassen der Analyseergebnisse müssen die in Abb. 3 dargestellte Klasse `AnalysisResult` erweitern. Nur solche Ergebnisse darf eine Analyse liefern. Bei Erzeugung eines solchen wird die eindeutige Graphnummer des analysierten Graphen und die sog. semantische Version des Graphmodells vermerkt. Somit ist der Bezug zwischen Graph und Ergebnis hergestellt. Das Konzept der semantischen Versionsnummer eines Graphen eröffnet die Möglichkeit, bei Änderungen am Graphen zu erkennen, ob eine Änderung der Semantik vorlag und somit ein bereits berechnetes Ergebnis veraltet ist. Darüber hinaus ist es auch möglich zu erkennen, wenn ein Ergebnis wieder aktuell wird, was durch Rückgängigmachen von Änderungen am Graphen verursacht werden kann. Um die Berechnungsdauer eines Ergebnisses bestimmen zu können, wird auch noch Start- und Endzeit der Berechnung gespeichert.

1.3 Ablauf von Analysevorgängen

Um die performante Ausführung der Analyseverfahren zu begünstigen sieht das Analyseframework vor, dass Analyseobjekte erst bei Bedarf generiert werden. Zu diesem Zweck besitzen die ausführbaren Knoten

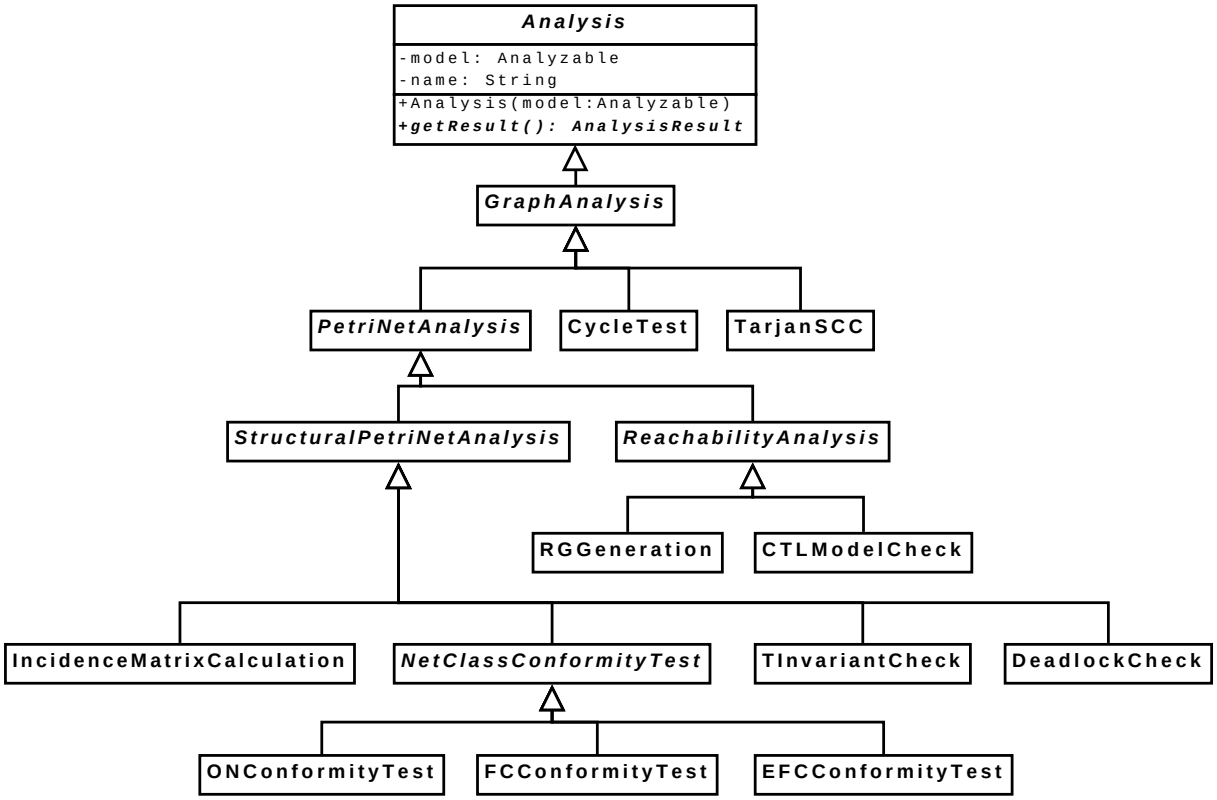

Abbildung 2: Analyseklassen (Ausschnitt)

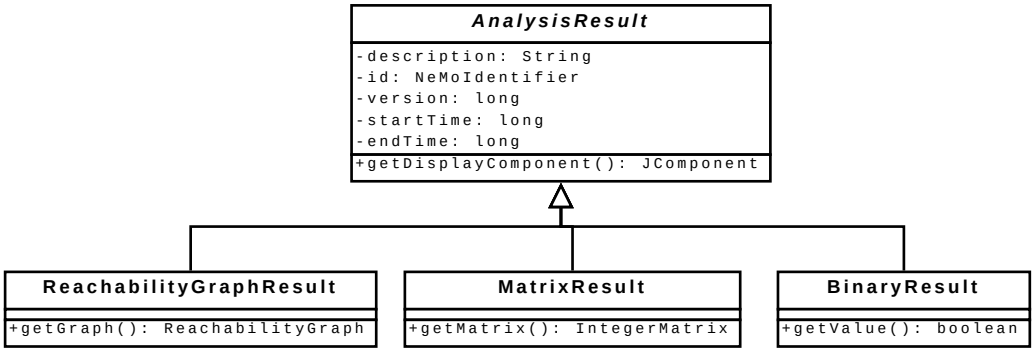

Abbildung 3: Analyseergebnisklassen

im Analysebaum ein sog. *Invokerobjekt*, das auf das entsprechende Analyseverfahren zugeschnitten ist. Wird ein solcher Invoker angewiesen die zugeordnete Analyse in Gang zu setzen, instanziiert er die entsprechende Analyseklasse und führt sie in einem eigenen Thread aus. Dass sich NeMo hier Multithreading zunutze macht, trägt gerade bei der Ausführung mehrerer berechnungsintensiver Analyseverfahren dazu bei die Leistungsfähigkeit von Mehrkernprozessoren auszuschöpfen.

Schließlich spielt bei der Optimierung eines Analysevorgangs noch die Wiederverwendung bereits vorhandener Ergebnisse eine wichtige Rolle. Zu diesem Zweck wird bei der Anforderung eines Ergebnisses (insbesondere auch durch ein darauf aufbauendes Analyseverfahren) geprüft, ob bereits ein aktuelles Ergebnis vorliegt. Nur wenn das nicht der Fall ist, wird es neu berechnet. Somit können Analyseverfahren wechselseitig aufeinander zugreifen und es werden keine überflüssigen Berechnungen angestoßen.

Besonders im Hinblick auf komplexe Algorithmen, z.B. im Bereich des Model-Checkings, spielen Effizienz und Performance bereits auf sehr niedrigen (maschinennahen) Ebenen eine große Rolle. Die in NeMo getroffenen Design-Entscheidungen bezüglich einer effizienten IO-Architektur werden im folgenden Abschnitt skizziert.

2 IO-Architektur für CTL-Model-Checking

Bei der Entwicklung von Model-Checkern für große entfaltete Zustandsräume entsteht immer der Bedarf an entsprechend skalierende Datenstrukturen. Übliche Markierungsalgorithmen zum CTL-Model-Checking [Emer95] speichern als temporäre Daten zu jedem Zustand einige Informationen, z.B. ob ein bestimmter Teilausdruck in einem Zustand bereits ausgewertet wurde [Helj97].

Selbst mehrere Gigabyte Arbeitsspeicher reichen bei entsprechend großen Strukturen nicht mehr aus, um die Menge an Daten zu bewältigen. Als Beispiel verwenden wir einen lokalen CTL-Model-Checking-Algorithmus, der in [VerLew93] erstmals vorgestellt und in [Helj97] verbessert wurde. Dieser speichert zu jedem Zustand und zu jedem Teilausdruck der CTL-Formel zwei Wahrheitswerte sowie zu jedem Zustand eine Tiefensuch-Nummer. Der Speicherbedarf bewegt sich also in $O(|S| \cdot |f|)$, wobei S die Menge der Zustände und $|f|$ die Länge der CTL-Formel bezeichnet. Einige Millionen Zustände zu verwalten ist bei Arbeitsspeichergrößen von zum Beispiel zwei Gigabyte denkbar, steigt die Anzahl der Zustände in die Milliarden, muss eine alternative Speicherverwaltung genutzt werden.

Betriebssysteme nutzen seit langem Festplatten, um den physischen Arbeitsspeicher für Anwendungen transparent zu erweitern. Diese Möglichkeit bietet sich auch hier an. Bei der Implementation kann nicht angenommen werden, dass die Speicherverwaltung des Betriebssystems so effizient arbeitet, dass von einer Anwendung einfach beliebig viel Speicher alloziert werden kann. Dies scheitert bereits daran, dass die maximale Größe des virtuellen Arbeitsspeichers üblicherweise fest ist und nicht einfach erweitert werden kann. Verlangt eine Anwendung nach mehr Speicher, als vom Betriebssystem an virtuellem Speicher zur Verfügung gestellt wird, schlägt diese Anforderung fehl. Weiterhin würde dieses Vorgehen alle Anwendungen, die auf dem Rechner laufen, negativ beeinflussen. Deshalb sind spezielle Implementationen und ein eigenes Speichermanagement notwendig.

Ein zweiter Problemschwerpunkt sind Arbeitsdaten, die bei der konkreten Implementation ebenfalls anfallen: in erster Linie für Tiefen-/Breitensuche benötigte Stacks, Queues oder Dequeues. Auch hier reicht es nicht aus, auf die üblichen Datenstrukturen zurückzugreifen, die von Java geboten werden, denn eine Tiefensuche kann theoretisch jeden Zustand des Systems besuchen. Folglich müssen auch hier spezielle Implementationen benutzt werden. Die nötigen Datenstrukturen für den momentan implementierten Model Checker sind Stacks und Maps.

2.1 Evaluation von Objektdatenbanken

Datenbanken (relational, objekt-relational oder objektorientiert) könnten als Datenspeicher verwendet werden. Bei Verwendung solcher Datenbanksysteme entsteht allerdings ein Zusatzaufwand durch die Datenbankverwaltung, Sperrung von Datensätzen, Transaktionsverwaltung und Objektallokationen (und damit verbunden auch Objektfreigaben). Da all dies für die Implementation des Model-Checking-Algorithmus nicht notwendig ist, wurde auf den Einsatz einer Datenbank verzichtet

Bei der Evaluation von schlankeren Bibliotheken stach vor allem JDBM[2] hervor, welches ein allgemeiner Speicher für Java-Objekte ist und bei dem bereits Implementationen für B+-Bäume und H-Bäume (JDBM-Nomenklatur für als Baum organisierte Hash-Maps) für beliebige Daten beiliegen. JDBM wird für einige Unterbereiche der Speicherung (momentan ist das die zu untersuchende Struktur sowie Teile der temporären Daten des Model-Checkers) eingesetzt, soll aber langfristig auch in diesen Bereichen abgelöst werden. Durch die Möglichkeit, Objekte beliebiger Struktur zu speichern und auch zu aktualisieren, wobei die persistierten Objekte durchaus anwachsen können, bietet es eine enorme Flexibilität, erreicht jedoch nicht die Performance, die mit speziellen Implementationen möglich ist. Dies liegt unter anderem daran, dass JDBM immer auf der Ebene von Objektinstanzen arbeitet und damit die Verarbeitung von primitiven Datentypen, die beim implementierten Algorithmus vornehmlich notwendig sind, zusätzlichen Aufwand zur Instanziierung und Freigabe erfordert. Bei nur sehr kurzer Verwendung von Objektinstanzen sind beides in aktuellen Implementierungen der Java Virtual Machine zwar keine teuren Operationen, bei mehreren Millionen oder Milliarden von Zugriffen wird der zusätzliche Aufwand jedoch signifikant. Diese Einschränkung betrifft jede allgemeine Implementation einer Datenstruktur, unabhängig davon, ob es ein Stack, eine Queue oder ein Baum ist[3].

Ein weiteres Problem ist das intransparente Caching von JDBM. Es ist nicht kontrollierbar, wieviel Arbeitsspeicher zum Caching von deserialisierten Objekten verwendet wird, da generell mit einer festen Anzahl an zwischengespeicherten Objekten gearbeitet wird. Ob jede Insanz 10 Byte, 10 Kilobyte oder 100 Kilobyte groß ist, ist davon unabhängig. Bei Applikationen, die die Arbeitsspeicher moderner Rechner komplett ausreizen, ist es jedoch von enormer Bedeutung, Kontrolle über den Speicherverbrauch einzelner

[2] `http://sourceforge.net/projects/jdbm/`

[3] Entwicklungen werden häufig in die Richtung allgemeiner Wiederverwendbarkeit getrieben. Daraus resultiert gerade in der Welt der Objektorientierung automatisch zusätzlicher Aufwand. Die durch Objektorientierung erreichten Vorteile sind im konkreten Kontext jedoch nicht relevant und wiegen die Nachteile zur Laufzeit nicht auf.

Systemkomponenten zu haben[4]. Deswegen wurde eine eigene low-level Datenzugriffsschicht entwickelt, die einerseits genau auf die speziellen Anforderungen beim Model-Checking und bei der Verwaltung großer unveränderlicher Datenbestände angepasst und andererseits um Fähigkeiten zur Speicherung beliebiger Objekte ähnlich JDBM erweiterbar ist.

2.2 Notwendige Datenstrukturen

Wie bereits erwähnt werden in erster Linie Stacks zur Tiefensuche sowie Maps benötigt. Während die Arbeit bei ersteren bereits abgeschlossen sind, steht die Implementation effizienter Map-Strukturen noch aus. Angedacht sind hier optimierte B+-Bäume.

Durch die Systemgröße entstehen auch besondere Anforderungen an die Implementation der eigentlichen Algorithmen: So elegant rekursive Algorithmen oft wirken, ist eine Rekursion über den gesamten Zustandsraum unmöglich, weshalb alle Algorithmen iterativ implementiert werden müssen. Je nach Art der Umwandlung von rekursiven in iterative Algorithmen müssen verschiedene Datentypen auf den Stack gelegt werden können: Mit dem selbstimplementierten Stack wird der Stack der virtuellen Maschine simuliert, auf den Methodenvariable gelegt werden, bevor die Rekursion stattfindet, um sie nach Rückkehr des rekursiven Aufrufs wieder vom Stack zu nehmen. Auch hier wird deutlich, dass noch hardwarenah gedacht werden muss.

2.3 Verwendete Konzepte

Die Implementation setzt vollständig auf Javas NIO-Architektur[5] auf. NIO steht für *new input output* und ermöglicht Java-Programmen erstmals wirklich effiziente Ein-/Ausgabeoperationen auf Dateien und Netzwerksockets, denn es stellt besonders betriebssystemnahe Konzepte zur Verfügung. Dies sind unter anderem die für diesen Anwendungszweck relevanten Puffer im Kernelspeicher sowie effiziente Schreibzugriffe.

Die Speicherorganisation der Datenzugriffsschicht basiert auf Byte-Puffern, die im Kernelspeicher liegen. Ist der verfügbare Speicher ausgeschöpft, können Puffer zum Beispiel auf die Festplatte ausgelagert werden, was für höhere Applikationsschichten vollständig transparent abläuft. Weil sich die Daten bereits im Kernelspeicher befinden, können sie vom Betriebsysstem direkt an die Treiberschicht für Plattenzugriffe weitergeleitet werden. Dies beschleunigt Schreib- und Leseoperationen. Beim Schreiben können die Daten weiterhin aus mehreren Puffern in einem physischen Schreibvorgang auf die Festplatte geschrieben werden (*gathering writes*). Dies erhöht die Effizienz, da weniger physische Schreibvorgänge durchgeführt werden müssen und die Daten dadurch in größeren Blöcken geschrieben werden. Dadurch wird auch die Fragmentierung der Daten auf den physischen Speichern verringert. Weiterhin ist es denkbar, vom Betriebssystem Bereiche der geschriebenen Dateien direkt in den virtuellen Speicher einzublenden (*file mapping*). Einer Anwendung, die auf einem mapped file arbeitet, scheinen die Inhalte der Datei in einem ganz normalen Puffer zu liegen. Die Daten werden vom Betriebssystemcache bei Bedarf gelesen und wieder zurückgeschrieben, so dass das eigentliche Java-Programm überhaupt keine Dateiein-/ausgabe mehr durchführen muss. Durch die Nähe zum Betriebssystem kann die Effizienz weiter gesteigert werden [RoLoAn00, Teva*87].

Da die Allokation und Freigabe von Kernelspeicher und im Zusammenhang damit die Instanziierung von Puffer-Objekten eine teure Operation ist, wurde die Datenzugriffschicht so implementiert, dass nicht mehr benötigte Puffer-Objekte nicht freigegeben, sondern zur Wiederverwendung vorbereitet werden (*Pooling*). So wird als Nebeneffekt auch der Aufwand für den Garbage Collector der virtuellen Maschine deutlich reduziert, was sich in einer verbesserten Gesamtperformance niederschlägt. Weiterhin ist eine globale Caching-Architektur der Puffer-Objekte implementiert. Damit wacht eine zentrale Instanz über den durch alle Puffer belegten Speicher und kann bei Bedarf die EA-Schicht anweisen, Puffer freizugeben. Da bei dem implementierten Algorithmus gerade für die verwendeten Stacks das Lokalitätsprinzip zutrifft, ist hier eine MRU-Strategie (most recently used) denkbar, die zuerst Puffer, die schon länger nicht mehr für Ein-/Ausgabeoperationen genutzt wurden, auslagert. Die gerade verwendeten „heißen“ Stacks sind von Auslagerungen dann am wenigsten betroffen. Zusammen mit dem Pooling ist durch diese zentrale Verwaltung eine ständige komplette Kontrolle über den verbrauchten Speicher möglich.

3 Ausblick

Durch den modularen und flexiblen Aufbau des Analyseframeworks ist eine Erweiterung für höhere Petrinetz-Klassen und andere Graph-Klassen mit vergleichsweise geringem Aufwand möglich. Konkret

[4] Auch die Speicherverwaltung der virtuellen Maschine ist für diese Zwecke gänzlich ungeeignet.
[5] Eingeführt mit Java Version 1.4.

ist geplant, die Analyse um Algorithmen für stochastische Petrinetze zu ergänzen.

Der weitere Ausbau des Analyseframeworks selbst betrifft die Handhabung von parameterabhängigen Analysen und die Persistenz von Analyseergebnissen. Da manche Analyseverfahren Parameter, wie beispielsweise eine Petrinetz-Markierung, erfordern, ist die ergonomische Bearbeitung von Analyseparametern ein wichtiges Ziel. Durch die Parametrisierung kann es mehrere aktuelle Ergebnisse einer Art zu einem Graphen geben, so dass Analyseergebnissen die zugehörigen Parameter beigefügt werden müssen. Dies wirkt sich auch auf die Speicherung von Analyseergebnissen aus, bei der insbesondere die Konsistenz zwischen Graphen und zugehörigen Ergebnissen eine entscheidende Rolle spielt.

Längerfristig soll die Modellierung und Analyse von Bayes-Netzen sowie von daraus generierten Wahrscheinlichkeits-Ausbreitungs-Netzen (siehe [LaPhPi06, LauPin07]) implementiert werden.

Literatur

[Emer95] **E. Allen Emerson**. Automated Temporal Reasoning about Reactive Systems. *Banff Higher Order Workshop*, pp. 41–101, 1995.

[Helj97] **Keijo Heljanko**. Model Checking the Branching Time Temporal Logic CTL. Research Report A45, Helsinki University of Technology, 1997.

[LaPhPi06] **K. Lautenbach, S. Philippi und A. Pinl**. Bayesian Networks and Petri Nets. *Proceedings of the workshop "Entwurf komplexer Automatisierungssysteme" (EKA) 2006*, Braunschweig, 2006.

[LauPin07] **Kurt Lautenbach und Alexander Pinl**. Probability Propagation Nets. Arbeitsberichte aus dem Fachbereich Informatik 20–2007, Universität Koblenz-Landau, Institut für Informatik, Universitätsstr. 1, D-56070 Koblenz, 2007.

[Phil*06] **S. Philippi, A. Pinl, J.R. Müller und R. Slovák**. Towards tool support for the formally based analysis of safety-critical systems with Petri-Nets. Daniel Moldt, *Bericht 276, 13. Workshop "Algorithmen und Werkzeuge für Petrinetze" (AWPN 2006)*, pp. 69–74, Hamburg, 2006.

[PhPiRa05] **S. Philippi, A. Pinl und G. Rausch**. A first View on a Generalised Modelling Toolkit for Graph-based Languages. K. Schmidt und Chr. Stahl, *12. Workshop "Algorithmen und Werkzeuge für Petrinetze" (AWPN 2005)*, pp. 25–30, Berlin, 2005.

[RoLoAn00] **Drew Roselli, Jacob R. Lorch und Thomas E. Anderson**. A Comparison of File System Workloads. *Proceedings of 2000 USENIX Annual Technical Conference*, pp. 41–54, June 2000.

[Teva*87] **Avadis Tevanian, Richard F. Rashid, Michael Young, David B. Golub, Mary R. Thompson, William J. Bolosky und Richard Sanzi**. A UNIX Interface for Shared Memory and Memory Mapped Files Under Mach. *USENIX Summer*, pp. 53–68, 1987.

[VerLew93] **Bart Vergauven und Johan Lewi**. A linear local model checking algorithm for CTL. *CONCUR'93*, pp. 447–461. Springer, 1993.

Modular PNML revisited: Some ideas for strict typing

Ekkart Kindler
Informatics and Mathematical Modelling
Technical University of Denmark
DK-2600 Lyngby
Denmark
eki@imm.dtu.dk

***Abstract*— The Petri Net Markup Language (PNML) is currently standardised by ISO/IEC JTC1/SC7 WG 19 as Part 2 of ISO/IEC 15909. But, there is not yet a mechanism for structuring large Petri nets and for constructing Petri nets from modules. To this end, modular PNML has been proposed some time ago. But, modular PNML has some problems. These problems along with ideas for their solution will be discussed in this paper.**

As a first step toward standardising a module concept for PNML in Part 3 of ISO/IEC 15909, this paper proposes a refined concept of modular PNML, which is independent of a particular kind of Petri net, but still has a strict type system. This paper focuses on the ideas and concepts; the technical details still need to be worked out. To this end, this paper also raises some issues and questions that need to be discussed before standardising modular PNML.

I. Introduction

The *Petri Net Markup Language* (*PNML*) is an interchange format for all kinds of Petri nets [5], [8], [2]. It is currently standardised as Part 2 of the International Standard ISO/IEC 15909 [4]. The *PNML Framework* helps tool makers implementing this standard [3].

However, Part 2 of ISO/IEC 15909 covers *Place/Transition nets*, *Symmetric Nets*, and *High-level Petri Nets* only. It does not cover other versions of Petri nets, such as timed or stochastic Petri nets or features such as inhibitor or reset arcs. This will be covered by Part 3 of ISO/IEC 15909. In addition, Part 3 of ISO/IEC 15909 will cover concepts for structuring large nets and for constructing nets from components.

In this paper, we discuss some ideas and concepts for defining *Petri net modules* and for constructing nets from different *instances* of such modules. This should serve as a staring point for the structuring and modularisation concepts of Petri nets that will be defined in Part 3 of ISO/IEC 15909. Actually, there was a proposal for defining and using Petri net modules quite early in the development of PNML, which was called *modular PNML* [7], [6], [8]. Since PNML is independent of a specific kind of Petri net, modular PNML was also designed to work with any kind of Petri net. The semantics of modules and the Petri nets that are constructed from instances of these modules is defined completely independent of the semantics of a particular version of Petri nets. It is defined purely syntactically by making copies of the module definitions, by connecting them via reference nodes, and by flattening the resulting net (see [7] for details) — without knowing anything about the concrete Petri net type.

The universality of modular PNML came for a price, however: The syntactical correctness of a system could only be checked after the complete system was built from the modules. Modules could be easily used in a way that was syntactically incorrect, but this would be known only after the system was complete. Therefore, modules had a very loose concept of typing. Basically, there were two reasons for this loose typing concepts: First, the focus of interfaces of modules was on places and transitions (i. e. nodes of the Petri net). Additional information attached to places or transitions, such as types, markings, or transitions, was not taken into account. Second, there was only a very simple concept for importing or exporting other information such as operators or sorts from or to a module. All this information was represented by *symbols*; but — since the module concept should be independent of a particular version of Petri nets — symbols did not have any specific structure. Therefore, the first version of modular PNML could not guarantee that symbols were combined in a syntactically correct way. Moreover, the concepts for symbols have never been worked out in full detail, since the concepts of modular PNML were not included to Part 2 of ISO/IEC 15909.

In this paper, we will discuss some ideas how the problems with the original version of modular PNML can be solved. To this end, a refined concept of symbols is introduced so that symbols can be used in a syntactically correct way—though not knowing their meaning. Then we will show how this idea carries over to labels of places and transitions. Altogether, this will result in a *strict typing* of modules: the overall syntactical correctness of a system built from modules can be checked locally in every module definition and where it is used. Still, these concepts retain one of the most important principles of PNML: it works for any kind of Petri net. For each new Petri net type, one needs to define only its *symbols* and how they can be combined. The rest of the concepts of modular PNML can be defined once and for all Petri net types.

Though these concepts are not yet worked out in full detail, they should be detailed enough for discussing the pros and cons of that approach and to discuss and decide on the future direction of the structuring and modularisation concepts that should be included to Part 3 of ISO/IEC 15909.

II. Example

Though the concepts proposed in this paper are independent of a particular kind of Petri net, we start with an example of a module using high-level Petri nets.

A. Module definition

Figure 1 shows an example of a module channel that transmits some information from a place *in* to a place *out*. In fact, this example is a slightly modified version of the example from [7], [8], where we represent the transmitted data explicitly now. Figure 1 shows the *definition* of the module.

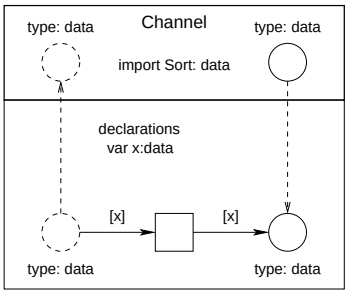

Fig. 1. The module Channel

This definition consists of two parts. The upper part in the bold-faced box defines the interface of the module and its name Channel. This interface consists of different parts: it imports a place (the dashed circle on the left-hand side) and it exports a place (the solid circle on the right-hand side). The difference between import and export nodes will become clear later in this paper. For now, it should be sufficient to know that these are the places seen and useable from outside the module: The import place will be provided by the environment of the module when it is used; the export place is a place that can be used by the environment of the module. In addition to the import and export nodes, the module definition also imports a symbol: a symbol representing a sort. As indicated by its name, this sort represents the type of data that should be transmitted over the channel. In order to make the symbol an import symbol, we use the keyword import here[1], the additional text Sort indicates that this symbol is a sort.

The lower part of the module definition in the thinly outlined box is the implementation of the module. It consists of a normal place on the right-hand side, which is the place that is exported – expressed by the dashed arrow. And there is a reference place which actually represents the imported place on the left-hand side – expressed by the dashed arrow pointing to the import place. Moreover, there is a transition between these two places; the arc annotations of the two arcs is a variable x of sort data. This variable is also defined in the implementation; in this declaration, we make use of the imported symbol for sort data.

[1]Actually, the keywords import and export and the graphical notation for import and export nodes, are not the point of this paper. In the end, the concrete syntax will be some XML and, possibly, some recommendation for the graphical representation. Since the focus of this paper is on concepts, we use some abstract syntax here.

Note that both places in the implementation of the module also have the *type* data, which exactly corresponds to the type of the import and export places in the interface. Note that we can check this net and, in particular, the labels of the places for syntactical correctness – without knowing which sort will be imported for symbol data, and which place will be imported for the import place. But, the interface requires that the imported symbol data is a sort, and the imported place has this type.

B. Module instances

Next, we will build a simple system from the module Channel. Figure 2 shows the use of three *instances* of module *Channel*, which are named ch1, ch2, and ch3, respectively. To indicate the instantiation, we use the name of the instance followed by the name of the module definition – a notation that is well-known from UML object diagrams. Moreover, the instances graphically resemble the interface definition of the module.

Here, we can actually understand the meaning of import places and import symbols more clearly. For each instance of the module, the import place needs to refer to some place outside the module, which will be the one imported for that instance. This is indicated by dashed arrows again. Note that export nodes of module instances are also seen from outside a module. So, we can use them for referring to them from import nodes. This way, we get a sequence of three channels. Once the data from the leftmost place are transmitted to the rightmost channel, the additional transition increments the value of that token and sends it back to the start place.

This is where the import symbol data representing a Sort comes in again. For each instance of the module Channel, we must provide a sort for the symbol data. In this example, we use the sort int, which is a built-in sort of high-level Petri nets. This way, the chain of channels transmits integer values. But, we could have used any other built-in or user-defined sort for that.

Again, we can check the syntactical correctness of this Petri net built from the module without having a look into the implementation of the modules. We need to make sure only that, for every import place, the attached type is the same as the type of the place it refers to. Since data is now bound to the sort int everywhere, this condition is obviously met.

From the model in Fig. 2 and the definition of the module Channel as shown in Fig. 1, the actual Petri net defined is the one shown in Fig. 3. It, basically, is obtained by making a copy of the module implementation for each module instance (and by prefixing all names inside the module implementation with the name of that module instance) and then merging every reference node with the node it refers to. For the nodes, this process was defined in more detail in [7], [8]. Here, we apply this idea also to symbols: Every occurrence of data in the module implementation is now replaced by the sort it is assigned in this instance of the module.

Actually, this idea was already mentioned in [7], [8]. The new idea here is that we know more about a symbol, e. g.

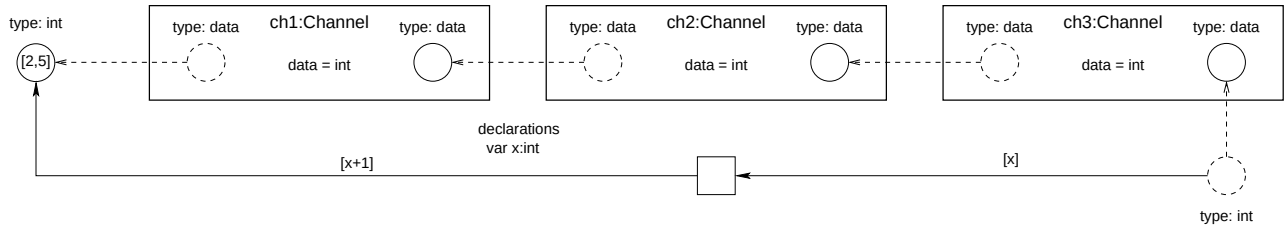

Fig. 2. Instances of module Channel

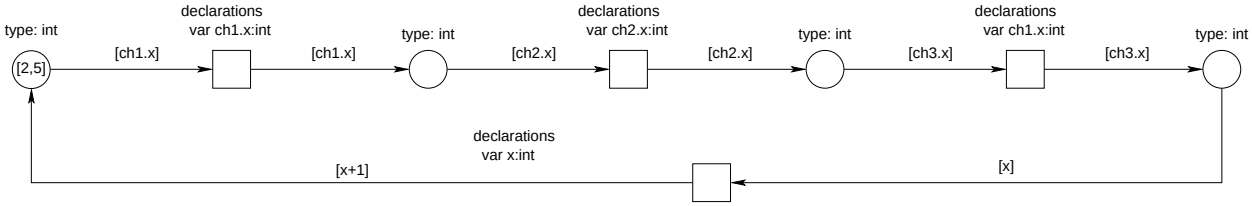

Fig. 3. The resulting model

we know that symbol data represents a sort. This way, we can make sure that module definitions and their use are syntactically correct.

In the rest of this paper, we will discuss some of the details necessary to make this idea work – independently of a specific Petri net type.

III. Concepts

In the previous section, we have seen the main concepts of modular PNML. Here we briefly rephrase the concepts again as defined in [7], [8].

A. Basic concepts

We distinguish between a *module definition* and a *module instance*. The module definition defines the *module interface* as well as the *module implementation*. The *module interface* defines *import* and *export* nodes[2]. Once a module is defined, we can use its *module instances* in other nets. We can even use module instances in the definition of other modules. The only condition on this *uses relation* among modules is that it is acyclic. This way, modules form a hierarchy.

For every import node of a *module instance*, there must be a reference to some node of the net or module which uses this module. The export nodes of the module instances can be used as if they were nodes defined in that net or module itself. Note that the users of a module do not see – or at least it is not necessary that they see – the implementation of the module. For properly using a module (syntactically), they need to know the interface definition only.

Of course, different modules might use the same identifier for naming places, transitions, or symbols. By using namespaces, the same name in different instances can be distinguished. In our example, this is indicated by prefixing all names inside a particular module instance by the name of that instance. In the original proposal of modular PNML, this concept was realised by the above mentioned prefixing mechanism, which is a bit ad hoc. Today, we could use XML namespaces for that purpose [1], but this technical issue is beyond the scope of this paper.

[2]In our example, we had import and export places only. But, in general, we can also have import and export transitions. Concerning the module concept, there is no difference between places and transitions.

B. The problem

In our example, we have used the very same idea for importing *symbols*, and we can also export symbols. The example, however, was quite simple. The imported symbol was a sort, which has no further structure. It could be used, basically, at every place where a sort was required. This is no longer true for other kinds of symbols. When we import a symbol for an operator, say f for example, it is not enough to know that it is an operator. In order to construct syntactically correct terms from this operator f, we need to know the number of parameters it takes and their types. This is but one example of a symbol with more structure.

Now, there are two question: How does modular PNML know this structure of the symbols? And, how does it know how to use a symbol in a syntactically correct way? Of course, we could built in the structure of sorts and operations to modular PNML. But, this would violate one of the principles of PNML: its independence of a specific kind of Petri nets. For example, for timed or stochastic Petri nets a module might import some delays or some firing rates for some transitions. And other types of Petri nets could have something completely different. Therefore, we cannot make specific types of symbols an integral part of modular PNML. Rather, it is necessary, to have a general concept of symbols and along with a new Petri net type, we need to define the structure of these symbols, which will be closely related to the concepts occurring in a type anyway.

In the rest, of this section, we illustrate how this can be done. We start again with a concrete example of a high-level Petri net module, where the symbols of interest are *sorts* and *operators*. Then, we show how this structure can

be generalised and how the structure of the symbols can be expressed along with a Petri net type definition.

C. Structured symbols

Figure 4 shows one of my favourite Petri net examples. For some operation $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ and some value y of type B, which is put to the import place, it calculates a pair (x, y) such that $\mathsf{f}(x) = y$ and puts this pair to the output place – if such a pair exists. So, it magically computes the inverse of f for some given value y. Note that the module is independent of a particular operator f; it works for any operator, which will be provided when the module is instantiated.

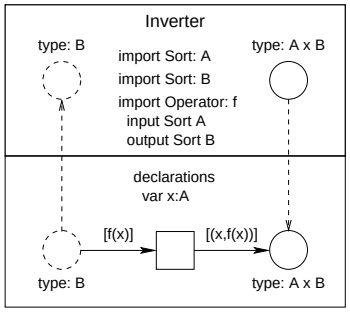

Fig. 4. The module Inverter

Let us have a closer look at the symbols used in this example. The module definition of Inverter imports three symbols: two sorts A and B, and one operator f. Though, there is no specific order in which the symbols are imported, there is a dependency here. The operator f has some structure, which is made explicit in the interface. It has one input sort and one output sort. In principle, we could use any sort here. But, in this specific example, we refer to the imported sorts A and B as the input sort and output sort for that operator. Therefore, the symbol f depends on the other two symbols. The import sorts are also used for defining the type of the import place and the export place of the module. Which are sort B and the product $\mathsf{A} \times \mathsf{B}$, respectively.

The module implementation is pretty standard, except for the fact that instead of true sorts and operations, we use imported symbols. The interesting part of this example is, that we know the structure of the imported operator now; therefore, we can check the syntactical correctness of the arc inscriptions. And we can check that the sort of the arc inscriptions fit the type of the corresponding place. In order to do that, the Petri net type, here high-level Petri nets, just needs to know the input and the output sort of an operator; this is exactly the information that is provided in the module definition for the imported symbol f. From the syntactical point of view, the symbol f of high-level Petri nets is now as good as any built-in or user defined operator.

The syntax of the definition of the structure of the operator symbol f is quite verbose and appears a bit unusual. We might rather expect something like $\mathsf{import\ f} : \mathsf{A} \to \mathsf{B}$. Again, this is an issue of concrete syntax, which is not the issue of this paper. Actually, an adequate concrete syntax for such definitions depends on the Petri net type and even on the particular kind of symbol. The syntax $\mathsf{import\ f} : \mathsf{A} \to \mathsf{B}$ is specific to operators. The syntax used here is closer to the underlying concepts, which will become clearer in the next section, when we discuss the definition of the symbols of a particular kind of Petri net.

D. Defining symbol types

One important question is still open, however: How does modular PNML know that sorts do not have an internal structure, whereas operators have an internal structure. Even more, how does modular PNML know that an operator needs to have an input sort and an output sort; actually, we will see in a minute that there can be any number of input sorts, but there must be exactly one output sort for an operator.

In order to answer that question, we briefly revisit the way in which Petri net types can be defined in PNML[3]. A Petri net type is defined by the labels that can be attached to the different objects of the net or the net itself. The structure of these labels is defined by a UML meta model[4]. Such meta models define which labels are there in a particular version of Petri net and defines the relation between these concepts. For lack of space, we do not present the full meta model for high-level Petri nets – this alone would take over six pages. We rather have a closer look to a small fragment of it, which is shown in Fig. 5. It shows some classes of the package *Terms*, in which all concepts related to terms are defined, but some details are omitted.

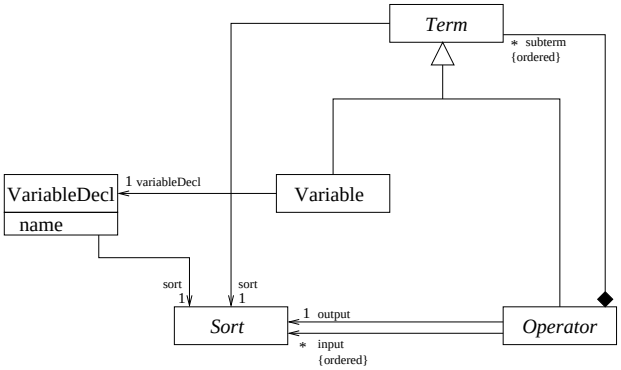

Fig. 5. Fragment from the *Terms* package

It says that a *term* can be built from a *variable* or from an *operator* and some *subterms*, where the conditions for correct typing are also omitted here. The relevant concepts, resp. classes, for our purpose are the Sort and the *Operator*, since these are the symbols we would like to import and export for high-level Petri nets. In this diagram, we can see the structure of the operator symbol immediately: the directed association

[3]Note that this Petri net type concept is used for defining three different versions of Petri nets in Part 2 of ISO/IEC 15909, but the concept for defining Petri net types itself is not standardised in Part 2 of ISO/IEC 15909. The basic concepts have been outlined in earlier papers [8] and it will be included as Part 3 of ISO/IEC 15909.

[4]Originally, the meta model was defined in terms of a RELAX/NG grammar, but now PNML uses UML meta models.

output to class Sort says that there must be exactly one output sort for an operator; and the directed association input says that there can be an arbitrary number of input sorts. Since there are no such associations for class sort, symbols for sorts do not have further structure (as far as modules are concerned). This exactly corresponds to the structure of the information that was provided for the imported operator symbol f in the example of Fig. 4.

Altogether, the information on the structure of the symbols can be derived from the meta model of the Petri net type. But, not completely: In Fig. 5, we have some other classes which we do not consider to be symbols, and, in the complete package *Terms* of the standard, there are even more. So, we need a mechanism to make explicit which concepts from the meta model should or could be used as symbols in this particular Petri net type. The simplest way is by marking the relevant classes and relevant associations by a *stereotype* ⟨⟨symbol⟩⟩ as shown in Fig. 6.

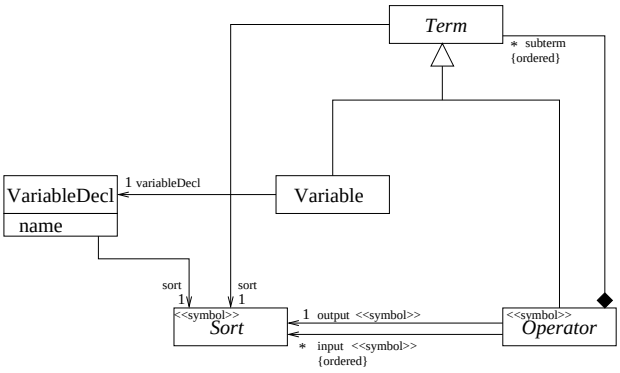

Fig. 6. Definition of symbols

Of course, there are other ways of making the symbols of a Petri net type explicit for the use with modular PNML. For example there could be two classes that inherit from Sort or Operator, respectively, and in addition inherit from some class Symbol. The latter is a more implementation oriented realisation, whereas the first one is more on the conceptual level. These details, however, are beyond the scope of this paper.

In whichever way, this concept is realised, it guarantees that symbols can be used in exactly the same way as their conceptual counterparts. This is why, syntactical correctness can be checked–without knowing its details.

IV. Issues

In the previous, section we have discussed the main idea and concepts of a module concept for PNML, which supports strict typing, i. e. syntactical correctness can be guaranteed locally without computing the full flattened model. The technical details, however, still need to be worked out. Moreover, there are some conceptual issues that need to be discussed before finalising the concepts and before implementing the technical details. In this section, we will briefly enumerate some of these issues.

Up to now, PNML ignored all labels of reference nodes and, in modular PNML, import nodes where considered to be reference nodes. Therefore, import and export nodes did not have labels or the labels did not have any meaning. Our examples, however, show that it is necessary to consider these labels and to check whether the labels of an import node fits the label of the place it refers to. The types should, basically, be the same. Therefore, it might be worthwhile to drop the idea of ignoring all labels of reference nodes and import nodes, and rather check that the labels fit to each other. The issue to be discussed here, is which labels need to be required for guaranteeing syntactical correctness and which are not. For example, names can always be ignored and they do not need to fit to each other. Types, however, should coincide. A marking might be missing; but, if present, it should be the same as in the place it refers to – sometimes we might even want to add the markings up.

In our examples, we briefly mentioned that, for one instance of a module, an import node can refer to an export node of another or even the same module instance. This, however, needs some care in order to avoid cyclic references. This can be achieved in different ways: One way would be not to allow any direct or indirect references from export to import nodes inside an implementation of a module. But, there are examples where such references make sense. If we want to have references from export nodes to import nodes inside an implementation of a module, we could make these dependencies explicit in the interface of the module. When using these modules, we could check that the references do not contain cyclic references. It needs to be discussed whether the extra effort of maintaining the dependencies between export and import nodes in the interfaces of a module is worth the effort. For deciding on this issue, we needed some convincing examples, with references from export to import nodes.

Another question is how strictly the typing should be enforced resp. mandated. On the one hand, the concepts presented in this paper could be used in such a way that, we always guarantee the syntactical correctness and strictly enforce it. But, it is not clear, whether we always want that. For example, think of a module with two imported sorts, say A and B and a place in the implementation of the module has type A, but the inscription of an outgoing arc has type B. Clearly, this is syntactically incorrect at this point. But, if someone uses this model and instantiates both import sorts A and B with the same sort, this use results in a correct overall system. Whether such modules should be allowed or not needs to be discussed.

For high-level nets and some other Petri net types with time or some similar extensions, the proposed concepts seem to have enough expressive power to construct systems in the way the are usually built. Still, it is not clear whether this is true for other kinds of Petri nets and, possibly, with completely different constructs for building systems from components. Since, modular PNML should eventually work for all Petri net types, we need to investigate some more example Petri net types in different application areas, and compare the

concepts from modular PNML with other existing structuring mechanisms in Petri nets and Petri net tools.

In analogy to import and export nodes, we proposed import and export symbols for modular PNML. This could, for instance, be useful for defining some data types in some data type module that can be used in other nets without revealing the details of the data type. Still, this is more a data type issue than a Petri net issue. Therefore, this is not a too convincing example for the use of export symbols. All the other examples, we could think of were quite artificial or could be easily rephrased in terms of modules with import symbols only. So, we are not sure whether we really need export symbols at all. On the other hand, they come almost without any extra cost; therefore, there is no real argument for excluding export symbols.

At last, there are some more technical issues, which were mentioned earlier already. One is the concept of namespacing for the instances of modules. The other is the way in which concepts of a Petri net typed are distinguished or marked to be symbols in the meta model of that Petri net type. For answering these questions we need some more experience with existing XML tools and PNML implementations.

V. Conclusion

In this paper, we have revisited modular PNML. We have presented some ideas that guarantee strict typing of Petri net modules and still works for any version of Petri nets, at almost no extra cost. The only extra effort in addition to defining a new Petri net type itself is to make the concepts that can be symbols explicit in the meta model of that Petri net type.

The focus of this paper is on the ideas of modular PNML and the concepts necessary to achieve strict typing. Some details of the realisation and, in particular, the exact XML syntax for modules need to be discussed and defined.

Moreover, the issues raised in Sect. IV need a more detailed discussion and investigation, in which the Petri net community as well as the WG 19 of ISO/IEC JTC1/SC 7 should be involved. Any kind of response, suggestion, or comments are welcome.

References

[1] Namespaces in XML 1.0 (second edition). W3C recommendation, The Object Management Group, Inc., August 2006.

[2] Jonathan Billington, Søren Christensen, Kees van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The Petri Net Markup Language: Concepts, technology, and tools. In W. van der Aalst and E. Best, editors, *Application and Theory of Petri Nets 2003, 24th International Conference*, volume 2679 of *LNCS*, pages 483–505. Springer, June 2003.

[3] L. Hillah, F. Kordon, L. Petrucci, and N. Trèves. Model engineering on Petri nets for ISO/IEC 15909-2: API framework for Petri net types metamodels. *Petri Net Newsletter*, 69:22–40, October 2005.

[4] ISO/JTC1/SC7/WG19. Software and Systems Engineering - High-level Petri Nets Part 2: Transfer Format. Technical Report FCD 15909-2, v. 1.2.0, ISO/IEC, June 2007.

[5] Matthias Jüngel, Ekkart Kindler, and Michael Weber. The Petri Net Markup Language. *Petri Net Newsletter*, 59:24–29, October 2000.

[6] Ekkart Kindler and Michael Weber. Modules in pictures. *Petri Net Newsletter*, 61:5–8, October 2001.

[7] Ekkart Kindler and Michael Weber. A universal module concept for Petri nets – an implementation-oriented approach. Informatik-Bericht 150, Humboldt-Universität zu Berlin, Institut für Informatik, April 2001.

[8] Michael Weber and Ekkart Kindler. The Petri Net Markup Language. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Petri Net Technologies for Modeling Communication Based Systems*, volume 2472 of *LNCS*, pages 124–144. Springer, 2003.

Ein Vorschlag zur Modellierung von Ultra Large Scale Systems

Daniel Moldt, Matthias Wester–Ebbinghaus

Universität Hamburg, Fakultät für Mathematik, Informatik und Naturwissenschaften,
Department Informatik, Vogt-Kölln-Str. 30, D-22527 Hamburg,
http://www.informatik.uni-hamburg.de/TGI/

Zusammenfassung Die aktuelle Diskussion um *Ultra Large Scale* (ULS) - Systeme wirft die Frage nach einer geeigneten Modellierung der grundsätzlichen Systemarchitektur komplexer Systeme auf. In diesem Beitrag wird auf Basis von höheren Petrinetzen (Referenznetze) und insbesondere mittels des Konzeptes der Netze-in-Netzen ein Referenzmodell für ULS Systemarchitekturen vorgestellt. Eine Besonderheit der Modellierung ist die operationale Semantik und die Möglichkeit der echten und virtuellen Schachtelung von Systemen, wie sie für ULS Systeme besonders geeignet erscheint. Dabei werden in Anlehnung an soziale Systeme vier Ebenen der Modellierung vorgeschlagen: Gesellschaft, Feld, Organisation und Multiagentensystem.

1 Einleitung

In Arbeiten zu sogenannten *(Software-)Anwendungslandschaften* ([8]) und *Ultra Large Scale (ULS) Systemen* ([9]) werden die Probleme der gezielten Konstruktion großer Systeme beschrieben. Dort werden Charakteristika von Systemen angegeben, die eine hohe Flexibilität bei gleichzeitiger Zusicherung von Systemeigenschaften erfordern. Es gilt als offene Forschungsfrage wie in diesem Bereich eine Modellierung erfolgen soll. Die UML (Unified Modeling Language) gilt in dieser Hinsicht als völlig unzureichend, da gerade die übergreifenden Architekturfragen durch die kleinschrittige Modellierung nicht gelöst werden. Im Bereich der Mulitagentensysteme (MAS) (oft auch als Middleware betrachtet) werden diese Fragen zwar adressiert, aber auch nicht organisationsübergreifend beantwortet. Genau in diesem Themengebiet zeigen sich aber die besonderen Charakteristika von komplexen Systemen (die auch innerhalb von traditionellen MAS auftreten können). Durch die Idee, Systemarchitekturen an die Anwendung anzupassen, liegt dann ein organisationsübergreifendes Modellieren nahe. Aufgrund der Komplexität bietet sich jedoch, wie in [15] beschrieben, der Wechsel der grundlegenden Modellierungseinheit vom Agenten hin zur Organisation an.

In diesem Beitrag wird daher ein Referenzmodell vorgeschlagen, das ausgehend von der Organisationsmetapher im Wesentlichen vier Ebenen umfasst: Gesellschaft, Feld, Organisation und MAS. Mit Hilfe dieser vier Abstraktionsebenen, die in ihrer abstrakten Notation und in konkreten Verwendungssituationen auch rekursiv und reflexiv verwendet werden könnnen, lassen sich die aus der Sicht der Autoren wichtigsten Ebenen bei der Modellierung von ULS (ähnlichen) Systemen vollständig charakterisieren. Die in [15] skizzierten Ebenen werden in diesem Beitrag um wesentliche Elemente ergänzt und in ihrer abstrakten Darstellung konkretisiert. Dabei wird auf existierende Organisationstheorien zurückgegriffen, insb. ([12]).

2 Modellierung von Softwaresystemen

Ein inhärentes Merkmal von ULS-Systemen liegt in der Schachtelung von Systemen, die in Systemen eingebettet sind. Hier ist wesentlich, die jeweiligen Systeme modellierungstechnisch zu identifizieren. Für die folgenden Diskussionen ist daher die Beschreibung eines Systems notwendig.

In [13] wurde ein petrinetzbasiertes Modell vorgestellt, das das Konzept der Einheit als abstrakte Beschreibung grundlegender Systemeinheiten konzeptualisiert. In Kombination mit der Modellierung dynamischer Architekturen mittels des Netze-in-Netzen Konzeptes in Form von Plugins (siehe [1]) ergibt sich hier ein sehr einfaches erstes Systemmodell. Die zentralen Operationen einer Systemeinheit sind Einfügen (add), Ändern/Nutzen (modify) und Löschen (remove), die jeweils auf die internen Einheiten der Einheit einwirken. Damit ergibt sich unmittelbar die Abbildung 1 des grundlegenden Systemverständnisses. Diese stark vereinfachende Modellierung liefert die Grundlage für weitere Differenzierungen.

Diese weiteren Differenzierungen ergeben sich aus der jeweiligen Perspektive des Modellierers. Die Modellierung komplexer Systeme erfordert unterschiedliche Ebenen der Abstraktion. Dabei sollen auf den jeweiligen Ebenen Inhalte so dargestellt werden, dass sie auf dieser (Abstraktions-)Ebene eine weitgehend vollständige und homogene Darstellung der gewählten Perspektive der Modellierung umfassen. Je detaillierter eine Betrachtung gewählt wird, desto weniger können übergeordnete Belange in die Modellierung einbezogen werden. So erfolgt bei der Programmierung einer konkreten Eingabebehandlung keine

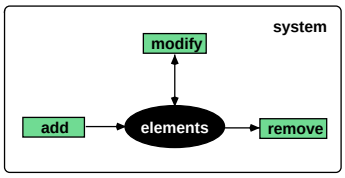

Abbildung 1. Grundlegendes Systemverständnis: Einbettung von Einheiten in ihre Umgebung

Überprüfung der Firmenziele, wenn dies nicht explizit Gegenstand der Eingabebehandlung ist. In Hinblick auf Softwarearchitekturen ist daher eine gezielte Unterscheidung verschiedener Ebenen vorzunehmen. Ein Vorschlag für eine solche Unterscheidung für die Architektur von Multiagentensystemen wurde beispielsweise in [4] und [11] vorgeschlagen. Von dieser Architektur inspiriert schlagen wir in diesem Beitrag ein Referenzmodell für ULS Systeme vor. Im Zentrum steht dabei die Erhöhung der Abstraktion im Vergleich zu den MAS. So ist nicht mehr der Agent die Kerneinheit der Strukturgebung, sondern die Organisation.

3 Ein Referenzmodell für ULS Systeme

Nachdem die prinzipiellen Hintergründe der Systemmodellierung kurz umrissen wurden, wird in diesem Abschnitt ein konkretes Referenzmodell für ULS-Systeme skizziert.

3.1 Überblick

Der Fokus auf die Organisation als zentrale Einheit der Architektur führt unmittelbar zu drei notwendigen Ebenen der Betrachtung: die Organisation selber, ihre Interna (hier MAS) und ihre Umgebung (hier das organisationale Feld). Da diese Einheiten geschachtelt werden können, wird ein Abschluss in Form einer vierten Ebene benötigt. Er folgt aus den Voraussetzungen, dass die Architektur erstens mehrere Organisationen umfassen soll und zweitens jede dieser Organisationen mehrere unmittelbare Umgebungen hat. Interessanterweise bestätigt die Organisationstheorie nach [12] diese 4 Ebenen (social psychological level, organizational structure level, ecological levels (organizational fields, society)). Das Referenzmodell umfasst daher vier Ebenen, die über jeweils drei Beziehungen (bzw. sechs, da für jede Richtung eine eigene vorzusehen ist) angeordnet sind. Abbildung 2 zeigt, wie die jeweiligen Einheiten in Form von Referenznetzen ([7]) als ineinander geschachtelt gesehen werden können.

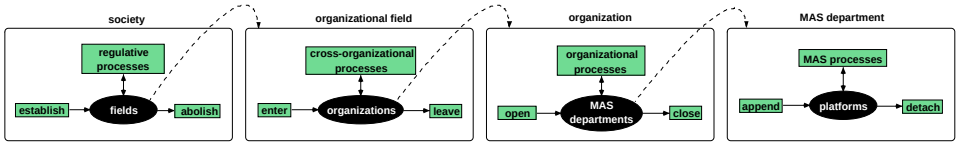

Abbildung 2. Mehrschichtige Referenzarchitektur organisationsorientierter Softwaresysteme

Aufgrund des geringen Platzes werden hier nur einige Charakteristika der gewählten Modellierungstechnik der Referenznetze erwähnt.[1] Referenznetze zeichnen sich dadurch aus, dass sie auf andere Netze (hier dann als Einheiten/Systeme interpretiert) verweisen können. Jede Netzinstanz kann dann mittels ihrer enthaltenen Transitionen über synchrone Kanäle mit anderen (Transitionen in evtl. anderen) Netzinstanzen kommunizieren. Die Referenzen sind gerichtet, bilden aber wie in [5] beschrieben die Möglichkeit, unterschiedliche Orte der Steuerung (in der übergeordneten Einheit, der untergeordneten Einheit oder gleichberechtigt zwischen beiden Einheiten) zu etablieren. Fragen zur Semantik der Art der Einbettung werden insbesondere in [2] und [14] diskutiert. Für einen ersten Einstieg reicht die wertmäßige Betrachtung, die eine eindeutige Einbettung der jeweiligen Systeme vorsieht, aus. Aufgrund der Überlagerung von verschiedenen Systemen bietet sich jedoch die (zusätzliche) Verwendung der Referenzsemantik für (einzelne ausgewählte, nicht-disjunkte) Zerlegungen von Systemen an.

[1] Eine detailliertere Darstellung findet sich demnächst auf den Webseiten zu unserem Forschungsbereich der organisationsorientierten Softwareentwicklung, siehe http://www.informatik.uni-hamburg.de/TGI/organ.

Für eine weitere Diskussion sind die einzelnen Einheiten in Abbildung 2 zu uniform. Im Folgenden wird daher eine detailliertere Differenzierung der Ebenen vorgenommen. Dabei sollen die Interaktionen der Einheiten einer Systemebene untereinander und die Interaktionen mit den übergeordneten und untergeordneten Systemeinheiten deutlich werden. Die Art der unterschiedlichen Einbettungen, die jeweilis vorhandenen Strukturen und die Menge der tatsächlich vorhandenen Einheiten bestimmen dann das Verhalten und die Struktur des gesamten Systems und seiner enthaltenen Einheiten.

Im Folgenden werden daher die Ebenen Gesellschaft, Organisatorisches Feld, Organisation und MAS beschrieben. Dabei erfolgt hier nur eine Skizze der Ebenen, da eine vollständige Beschreibung aus Platzgründen nicht erfolgen kann. Die zentralen Beziehungen lassen sich aber verdeutlichen. Durch die Verwendung der Objektenetze bzw. Referenznetze wird eine operationale, nebenläufige Semantik geliefert und da diese Petrinetzformalismen auf dem Konzept der Netze-in-Netzen beruhen, können die geschachtelten und verwobenen Subsysteme der ULS Systeme adäquat und unmittelbar mittels first-order concepts modelliert werden.

3.2 Ebenen im Detail: Gesellschaft

Da diese Ebene die höchste Ebene ist und sie den Abschluss des Referenzmodells bilden soll, gibt es lediglich untergeordnete Ebenen. Die untergeordnete Ebene ist die der organisationalen Felder. Daher werden auf dieser Ebene lediglich die Beziehungen zu dieser Ebene dargestellt. Insgesamt werden zwei zentrale Richtungen unterschieden: die zur Einbettung der Felder durch die Gesellschaft und die zur Umgebung der Felder. Zahlreiche verschiedene Facetten können hier unterschieden werden. Im Folgenden, dies gilt auch für die anderen Ebenen, werden lediglich beispielhaft einige zentrale Beziehungen beschrieben. Die Beschreibung der Beziehungen erfolgt durch die Beschreibung der jeweiligen Elemente der gerade modellierten Ebene. Das Gegenstück auf der unteren Ebene lässt sich dann spiegelbildlich beschreiben. Die einzelnen Elemente sind wie alle Systeme durch Strukturen und Prozesse zu beschreiben. Die Modellierung durch Petrinetze erlaubt dabei die unveränderlichen Strukturen durch Stellen, Transitionen und deren Relation zu beschreiben. Die Prozesse ergeben sich dann durch die auf diesen Strukturen potenziell möglichen Petrinetzprozesse. Bei dem Netze-in-Netzen Konzept ergeben sich somit dynamische Strukturen, da durch die Petrinetzprozesse wiederum Netze und damit Strukturen erzeugt werden.

Die Gesellschaft hat als Abschluss gebende Ebene unveränderliche Elemente (hier die Verfassung).[2] Hier ergeben sich gesetzliche Rahmenbedingungen, die für *alle* Einheiten des gesamten Systems gelten. Auf der Ebene der Gesellschaft werden die gegebenen Einheiten (hier die Verfassung) genutzt, um aktuelle Gesetze zu erzeugen, die bei der Behandlung der Felder durch die Gesellschaft eingesetzt werden. Dies gilt für die Erzeugung, die Änderung oder Unterstützung und das Entfernen von Feldern gleichermaßen. Abbildung 3 zeigt dies auf. Die Testkanten deuten an, dass mit der jeweiligen Aktion (z.B. die Transition

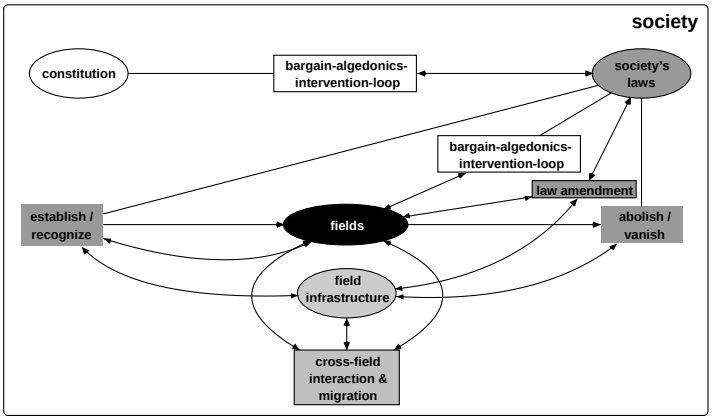

Abbildung 3. Architekturebene: Gesellschaft

establish / recognize) die Gesetze nicht verändert werden.

Die Änderung der Gesetze kann auf zwei grundsätzliche Arten erfolgen. Zum einen können die in der Gesellschaft ablaufenden Prozesse eine Änderung herbeiführen und zum anderen können aus den Feldern

[2] Bei der Betrachtung als offene oder als reflexive (d.h. durch eigene Elemente veränderbar) Systeme könnten auch diese Elemente wiederum Veränderungen unterworfen werden. Dann bildet die Ebene der Gesellschaft jedoch keinen absoluten, eindeutigen Abschluss.

heraus Änderungen durch dort durchgeführte Aktionen bewirkt werden. Diese Wirkungen müssen, damit sie auch tatsächlich auf der Ebene der Gesellschaft wirksam werden können, mittels einer hinreichend ausgerichteten Struktur so an die Gesetze angekoppelt sein, dass diese auch eintreten.

Durch diese Modellierung lassen sich zum einen steuernde Prozesse oder beherrschende Strukturen in der oberen Ebene verorten und zum anderen auch erzeugende Strukturen mit evtl. emergenten Ergebnissen auf den unteren Ebenen darstellen.

Die Referenznetze erlauben es, die Felder als unterschiedliche Einheiten auf der Ebene der Gesellschaft zu behandeln, aber da es sich gleichzeitig um Referenzen handeln kann, müssen die Felder *nicht* disjunkt sein. Die Überlagerung erfolgt auf den unteren Ebenen. Die Problematik für eine *saubere* Systemarchitektur wird damit jedoch offensichtlich, da gegen das *information hiding* Prinzip verstoßen wird. Die Überlagerung von Feldern (und natürlich auch anderen Einheiten in den folgenden Ebenen) ergibt sich aus der zu modellierenden Domäne. Gibt es dort eine entsprechende Mehrfachzuordnung, so gibt es sie auch im korrespondierenden Modell bzw. der Architektur.

3.3 Ebenen im Detail: Organisatorisches Feld

Das organisatorische Feld ist in die Gesellschaft eingebettet und enthält wiederum Organisationen als untergeordnete Einheiten. Die oben beschriebene und für die Gesellschaft schon verwendete allgemeine Einheitenstruktur kommt auch auf dieser Ebene zur Anwendung. Es gibt Organisationen, die zu dem Feld hinzugefügt werden, die interagieren, d.h. mit der Gesellschaft, anderen Feldern, anderen Organisationen auf der selben oder anderen Feldern kommunizieren, und die aus dem Feld entfernt werden. Dabei werden die jeweils erzeugten Rahmenbedingungen verwendet. Diese können von der Gesellschaft vorgegeben werden und von dem Feld beeinflusst werden oder auch nicht. Letzteres hängt von der jeweiligen Einbettung ab. Weiterhin können die Rahmenbedingungen auf dem Feld etabliert werden indem *administrative* Vorgänge innerhalb des Feldes diese erzeugen, bearbeiten oder entfernen, oder aber die einzelnen Organisationen können über durch das Feld vorgegebene Aktionen Einfluss nehmen. Abbildung 4 zeigt die prinzipiellen Strukturen. Felder stellen die Umgebung für auf sie ausgerichtete Organisationen dar

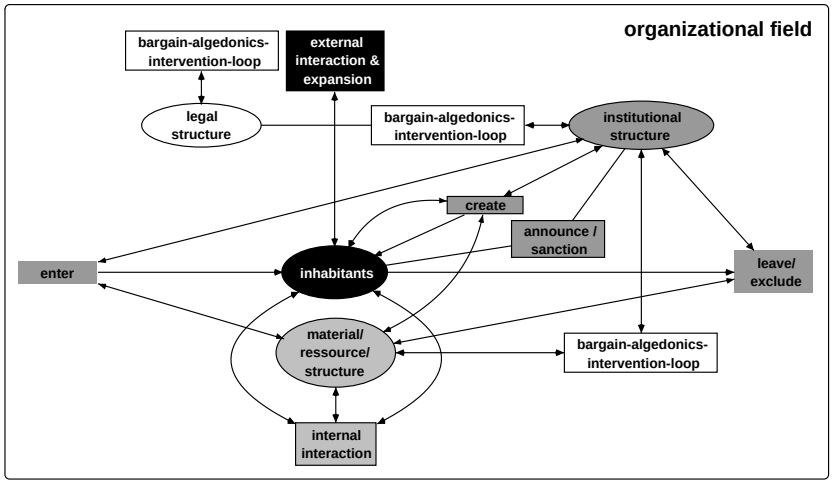

Abbildung 4. Architekturebene: Organisatorisches Feld

und sichern die Einbettung in die Gesellschaft ab.

3.4 Ebenen im Detail: Organisation

Die Organisation ist der zentrale Teil des Referenzmodells, da hier der Mittelpunkt der Abstraktion liegt, wie dies bei den Multiagentensystemen die Agent ist. Hier liegt ein Großteil der Modellierung. Charakterisierend sind die Einbettungen in die jeweiligen Felder und die eingebetteten Multiagentensysteme. Die Einbettung nach außen erfolgt über flow in und flow out, die den Einfluss und die Einbettung in die Felder repräsentieren. Details zu dieser Ebene finden sich in Abbildung 5. Die internen Strukturen zur Behandlung dieser Einbettungen bestimmen, in welcher Weise die Organisation tatsächlich den Einflüssen der Felder und der Gesellschaft folgt.[3]

[3] In wie weit die Organisationen direkt in die Gesellschaft eingebettet werden (können) soll hier nicht diskutiert werden. Die Autoren gehen davon aus, dass durch die Referenzsemantik und entsprechende Modellierungskon-

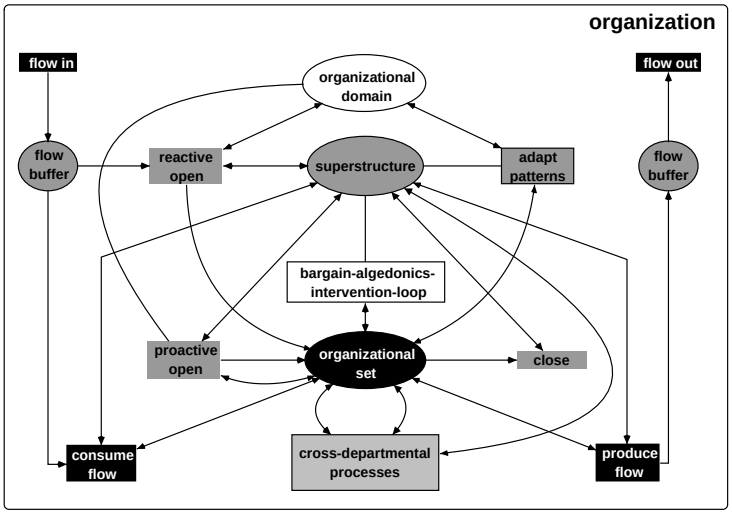

Abbildung 5. Architekturebene: Organisation

Die Einbettung der MAS erfolgt über die jeweiligen Organisationsaktionen, die unmittelbar auf die MAS abgestellt sind. Auch hier gibt es die Möglichkeit der losen unverbindlichen Einbettung, die keinen Einfluss von der oberen zur unteren Ebene oder umgekehrt zulässt, genau so, wie es strikte Durchgriffsaktionen der Organisation auf die MAS gibt. Ein wesentlicher Unterschied der Beziehung *Organisation/MAS* zu der Beziehung *Feld/Organisation* ist, dass die Annahme besteht, dass die MAS wesentlich abhängiger von der Organisation sind als dies bei der Abhängigkeit einer Organisation von einem Feld der Fall ist. Die Organisation *kontrolliert* die MAS. Charakteristisch für MAS ist jedoch, dass sie sich durch einen hohen Grad an Autonomie auszeichnen. Genau die Kontrolle und Steuerung durch die Organisation durchbricht diese Autonomie der MAS in gewisser Weise. Von Bedeutung ist hier der Begriff der Autarkie, der in [15] angeführt wurde. Die Organisation ist somit die Umgebung der MAS, beschreibt deren Einbettung und bündelt sie so, dass der Zweck der Organisation erreicht wird, wobei die Autonomie der MAS in den zur Steuerung / Verwaltung notwendigen Aktionen (Strukturen und Verhalten) der Organisation explizit berücksichtigt werden.

3.5 Ebenen im Detail: MAS

Die Multiagentensysteme (MAS) bilden den Abschluss des Referenzmodells. Hier wird „lediglich“ die Einbettung in Organisationen modelliert. Dass, wie dies in [5] beschrieben wurde, die MAS selbst wiederum komplexe Systeme sind, bleibt an dieser Stelle unberücksichtigt, da der gewählte Abstraktionsgrad zur Modellierung von komplexen Systemarchitekturen auf dieser Betrachtungsebene enden soll. Auf dieser Modellierungsebene sind die Einbettung in die Organisation und die eigentliche Leistung (die Menge der zur Verfügung gestellten Prozesse) von Bedeutung. Abbildung 6 liefert den entsprechenden Abschnitt des Referenzmodells. Das MAS ist in die Organisationen eingebettet und enthält (dann nicht näher beschriebene) Plattformen mit ihren Agenten, die so in die Organisation eingebettet werden. Wichtig ist hier, dass sich die einzelnen Plattformen (oder die Agenten auf ihnen) mehreren Organisationen oder Feldern innerhalb der Gesellschaft zuordnen können. Durch die Verwendung von Referenzen wird dies möglich. Soll eine eindeutige Zuordnung eingehalten werden, kann modellierungstechnisch auf Wertsemantik zurückgegriffen werden. Dann können (versehentliche) Mehrfachzuordnungen nicht mehr vorkommen. Mehrfachzuordnungen wären dann explizit zu modellieren und damit kein first order Konzept mehr.

Die Einbettung der MAS ist so zu gestalten, dass die Möglichkeiten realer Agenten auf den MAS in der Gestalt möglich sind, wie dies in der Anwendungsdomäne der Fall ist. Die organisations- oder feldübergreifende Kommunikation von einzelnen MAS (oder deren interne Einheiten) sollte gewahrt bleiben. Dies wird in der Abbildung 6 durch die Unterstützung der Kommunikation (siehe z.B. cross-platform interaction & migration oder external interaction) deutlich. Die Bereitstellung MAS-interner Strukturen und Prozesse der internen Verwaltung, wie z.B. Teamprozesse, werden hier nur schematisch durch individual position... dargestellt und nicht weiter differenziert.

---

strukte sowohl eine unmittelbare Einbettung möglich ist, als auch eine die vollständig durch eine übergeordnete Ebene *kontrolliert* wird.

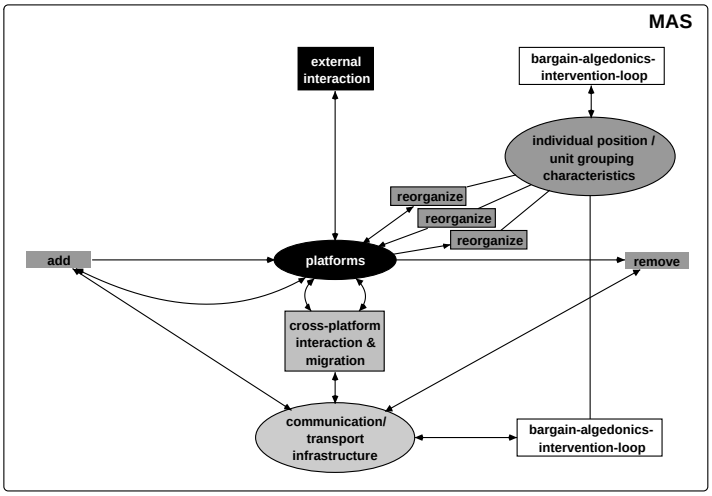

Abbildung 6. Architekturebene: MAS-Abteilung

3.6 Übergreifende Diskussion der Ebenen

Der Idee folgend, dass sich Software möglichst nahe an der von ihr unterstützten Umgebung orientieren sollte, werden für komplexe Softwaresysteme Ebenen der Strukturierung vorgeschlagen, die es erlauben aus dem Bereich der sozialen Systeme Konzepte unmittelbar auf Softwarearchitekturen zu übertragen. Hier wurden keine Beispiele angeben, aber die jeweiligen Ebenen der Gesellschaft, Felder (Märkte), Organisationen (Firmen, Vereine etc.) und MAS (Abteilungen, Gruppen, Personen, Programme etc.) mit den alltäglichen Begriffen der Wirtschaft in Verbindung zu setzen, sollte leicht fallen. Das Referenzmodell ist jedoch *nicht* auf wirtschaftsinformatische Modelle beschränkt. Es wird hier der Anspruch erhoben, auf beliebige komplexe Software anwendbar zu sein, da die Grundprinzipien der komplexen Systeme sich prinzipiell ähneln, wie dies für ULS Systeme gerade argumentiert wird und da informatische Systeme in sehr vielen realweltlichen Anwendungsbereichen anzutreffen sind.

Ausgehend von der abstrakten Modellierung zum grundlegenden Systemverständnis aus Abbildung 1 wurde für die Modellierungen der einzelnen Ebenen sowohl eine Verfeinerung der modify-Transition als auch eine Färbung der internen Systemeinheiten vorgenommen. Generell wurde dabei jeweils eine Unterscheidung zwischen den „operationalen“ und den „administrativen“ Einheiten einer Systemebene vorgenommen. Die operationalen Einheiten sind verantwortlich für die *primären Akvititäten* des Systems während die administrativen Einheiten für die *Systemeinbettung* der operationalen Einheiten und damit erst für die Existenz eines Gesamtsystems sorgen. Die Beschaffenheit der administrativen Einheiten ergibt sich dabei ganz im Sinne der *Zwillingseigenschaft* offener Systeme auf zweifache Weise. Einige Eigenschaften werden autoritär von der oberen Systemebene vorgeschrieben. Andere wiederum ergeben sich emergent aus den (interaktiven) Prozessen der internen operationalen Systemeinheiten. Auf diese Weise ergibt sich ein Zusammenspiel aus externer Kontrolle oder Regulierung und Selbstorganisation.

Dabei ist wesentlich, dass sich *alle* Strukturen und Prozesse des gesamten Systems auf die Multiagentensysteme zurückführen lassen da sie die elementaren Architekturbausteine darstellen. So hat beispielsweise auch eine Gesetzesvorgabe der Gesellschafsebene für eine Teilmenge ihrer organisatorischen Felder ihren Ursprung (über mehrere Einbettungsschritte in unterschiedliche Richtungen) auf der Multiagentensystemebene.

Die Wahl der Multiagentensysteme als die kleinsten Einheiten der Modellierung ergibt sich dabei aus der Notwendigkeit einer hohen Abstraktionsebene auf der einen und der konzeptionellen und technischen Machbarkeit auf der anderen Seite. Die Agentenforschung hat sich in den letzten Jahren vermehrt auf die „Vergegenständlichung“ eines Multiagentensystems fokussiert, auf die Auffassung eines Multiagentensystems als eine (organisatorische) Entität mit eigenen strukturellen, funktionalen und normativen Eigenschaften ([15]). Damit wurde der Weg bereitet, mit der in diesem Beitrag vorgestellten Architektur nahtlos an die Ergebnisse der Agentenforschung anzusetzen und dabei das Abstraktionsniveau auf die Ebene von Organisationen mit Multiagentensystemen als internen Einheiten zu heben.

Damit wird auch offenbar, wie mit der Problematik, Multiagentensysteme trotz ihrer inhärenten Komplexität als unteren Systemabschluss betrachten, umzugehen ist. Die nähere Betrachtung von Plattforminterna eines Multiagentensystems fällt nicht unter den hier vorgestellten Ansatz. Stattdessen ist ein Wechsel der Methodik vorzunehmen. Genau für die dann benötigte Methodik liefert die Agentenorientierung einen reichen Fundus.

4 Zusammenfassung und Ausblick

Mit der vorgestellten Referenzarchitektur organisationsorientierter Softwaresysteme wurde in diesem Artikel eine erste Konzeptualisierung eines Denkmodells für die Gestaltung und Instandhaltung großer Softwareysteme, bestehend aus einer Fülle von vernetzten und heterogenen Teilsystemen, geliefert. Die Architektur integriert system- und organisationstheoretische Modelle und setzt auf den praktischen Ergebnissen der Multiagentensystemforschung auf. Sie ist damit nach Auffassung der Autoren ein idealer Kandidat für die nächste Generation von Architekturen für Softwaresysteme der Kategorie ULS System.

Neben der vertieften Spezifikation der einzelnen Ebenen ist die konkrete Verwendung des Referenzmodells ein wichtiger Aspekt der zukünftigen Arbeit. Hierbei bieten sich insbesondere interorganisationale Prozesse und Anwendungen an. Diese wurden z.B. in [10] betrachtet und im Kontext von MAS auch umgesetzt. Das Ziel ist die Gestaltung von organisationenübergreifenden Anwendungen, die trotz der oben beschriebenen Problematik von ULS Systemen bestimmte gewünschte Eigenschaften haben. Die Beschränkung auf Prozesse ermöglicht eine gezieltere Gestaltung der Systeme, deren Eigenschaften sich dadurch auch im Wesentlichen auf Prozesse beziehen. Formale Nachweise liegen hier wiederum durch Workflownetze auf Basis der Petrinetze nahe. Hier liegt ein großes Potenzial für den hier vorgestellten Ansatz. Zu bedenken ist aber, dass die traditionelle Gestaltung durch Workflowmanagementsysteme Strukturen in den Systemen erzwingt, die recht restriktiv sind und den hier für ULS Systemen formulierten Anforderungen nicht gerecht werden. In [3] und [6] wurden hingegen Team-orientierte Ansätze verfolgt, die ein ähnliches Ziel haben, jedoch stärker die KI-Aspekte betonen. Beide Richtungen unterstützen das vorgestellte Denkmodell in direkter Weise und bieten einen guten Ausgangspunkt für die praktische Ausweitung auf organisationsübergreifende komplexe Anwendungen. Dabei bleiben die prinzipiellen Grenzen der konkreten Gestaltung von ULS Systemen zwar bestehen, werden aber ein bischen weiter verschoben.

Literatur

1. Lawrence Cabac, Michael Duvigneau, Daniel Moldt, and Heiko Rölke. Modeling dynamic architectures using nets-within-nets. In *Applications and Theory of Petri Nets 2005. 26th International Conference, ICATPN 2005, Miami, USA, June 2005. Proceedings*, pages 148–167, 2005.
2. Michael Köhler. *Objektnetze: Definition und Eigenschaften*, volume 1 of *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2004.
3. Michael Köhler. A formal model of multi-agent organisations. *Fundamenta Informaticae*, 2007. To appear.
4. Michael Köhler, Daniel Moldt, and Heiko Rölke. Einheitliche Modellierung von Agenten und Agentensystemen mit Referenznetzen. In S. Jablonski, S. Kirn, M. Plaha, E. Sinz, A. Ulbrich-vom Ende, and G. Weiß, editors, *Tagungsunterlagen: Verteilte Informationssysteme auf der Grundlage von Objekten, Komponenten und Agenten (vertIS 2001), Universität Bamberg, 4–5. Oktober 2001*, pages 3–20, October 2001.
5. 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.
6. Michael Köhler and Matthias Wester-Ebbinghaus. Petri net-based specification and deployment of organizational models. 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 67–81, Siedlce, Poland, June 2007. Akademia Podlaska.
7. Olaf Kummer. *Referenznetze*. Logos Verlag, Berlin, 2002.
8. Josef Lankes, Florian Matthes, and Andre Wittenburg. Softwarekartographie: Systematische Darstellung von Anwendungslandschaften. *Wirtschaftsinformatik 2005*, 2005.
9. Linda Northrop. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon, 2006.
10. Christine Reese, Matthias Wester-Ebbinghaus, Till Dörges, Lawrence Cabac, and Daniel Moldt. An agent-based process infrastructure for multi-agent systems. Accepted paper for the international workshop on languages, methodologies and development tools for multi-agent systems (LADS'07), 2007.
11. Heiko Rölke. *Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen*, volume 2 of *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2004.
12. W. Richard Scott. *Organizations: Rational, Natural and Open Systems*. Prentice Hall, 2003.
13. Volker Tell and Daniel Moldt. Ein Petrinetzsystem zur Modellierung selbstmodifizierender Petrinetze. pages 36–41. Humboldt Universität zu Berlin, Fachbereich Informatik, 2005.
14. Rüdiger Valk. Object Petri Nets – Using the Nets-within-Nets Paradigm. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets: Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 819–848. Springer-Verlag, 2004.
15. Matthias Wester-Ebbinghaus, Daniel Moldt, Christine Reese, and Kolja Markwardt. Towards Organization–Oriented Software Engineering. In Heinz Züllighoven, editor, *Software Engineering Konferenz 2007 in Hamburg: SE'07 Proceedings*, volume 105 of *LNI*, pages 205–217. GI, 2007.

Automaten- und Petrinetzmodelle für die Darstellung und Analyse von Itembanken für computerbasiertes Testen

Heiko Rölke

Deutsches Institut für Internationale Pädagogische Forschung
Schloßstraße 29, 60486 Frankfurt
roelke@dipf.de

Zusammenfassung. Dieses Papier beschreibt die Erstellung von Zustands- und Erreichbarkeitsdiagrammen für mögliche Testabläufe im Rahmen von adaptiven Tests. Bei adaptiven Tests wird die Reihenfolge von zu lösenden Aufgaben erst unmittelbar bei der Testbearbeitung und in Abhängigkeit vom bisherigen Testverlauf berechnet. Deshalb ist auch bei gegebenem Aufgabenbestand a priori nicht klar, welche Aufgaben überhaupt und in welcher Reihenfolge diese für einen Probanden gestellt werden.
Zur Lösung damit verbundener Probleme, wie nicht oder zu häufig benutzte Aufgaben, werden Automaten- und Petrinetz-Modelle vorgestellt, die direkt oder indirekt mittels Zustandsraumanalyse die Verwendung von Items in allen möglichen Testabläufen widerspiegeln.

Keywords: Automaten, Petrinetze, Technologie-basiertes Assessment, TBA, adaptives Testen, CAT

1 Motivation: Analyse von Itembanken

Beim Testen und insbesondere beim computer- oder technologiebasierten Testen (TBA) wird auf Datenbanken mit bereits vorhandenen Aufgaben, sogenannten Items, zurückgegriffen. Aus diesen werden manuell oder automatisch Tests zusammengestellt. Der Spezialfall des computer-adaptiven Testens (CAT) betrifft das automatisierte Zusammenstellen von Tests während der Testdurchführung und einzeln für jeden Probanden - angepasst an dessen Leistungsfähigkeit.

Für die Qualität der erstellten Tests ist die Zusammensetzung der Item-Datenbanken maßgeblich. In der Psychometrie eingebürgert sind deshalb Simulationsläufe, gerade für computerbasierte und insbesondere für adaptive Tests. Unbekannt oder zumindest in der Literatur nicht vertreten ist dagegen die Möglichkeit, eine erschöpfende Analyse aller möglichen Testabläufe unabhängig von einer konkreten Testdurchführung vorzunehmen. Auf diesem Weg können Qualitätskriterien für Itembanken ermittelt werden.

Der Rest des Papiers ist wie folgt aufgebaut: Im folgenden Kapitel 2 wird ein knapper Überblick über das Gebiet des computer-basierten Testens gegeben

und auf weiterführende Literatur verwiesen. Danach werden in 3 unterschiedliche Möglichkeiten vorgestellt, wie die Analyse von Testabläufen über einer gegebenen Itembank durchgeführt werden kann. Das Papier schließt in Kapitel 4 mit einem Ausblick auf weitere Arbeiten.

2 Computer-basiertes und adaptives Testen

Vorbemerkung: Im hier zur Verfügung stehenden Raum kann notwendigerweise nur eine sehr verkürzte und ausschnitthafte Darstellung der Testtheorie stattfinden. Zur Vertiefung sei der Leser auf allgemeine Literatur wie [2] hingewiesen.

Das Gebiet der Testtheorie befasst sich mit den Hintergründen und Zusammenhängen zwischen abstrakten Fähigkeiten und ihren konkreten Ausprägungen beziehungsweise ihrer Messbarkeit in Testszenarien. Im Gegensatz zur sogenannten „klassischen Testtheorie“ beschreibt die „Item-Response-Theory“ (IRT) einen Zusammenhang zwischen latenter Fähigkeit und Antwortverhalten eines Probanden. Dieser Zusammenhang ist in der IRT wahrscheinlichkeitsbasiert.

2.1 Item-Response-Theory

In Abhängigkeit von seiner tatsächlichen (unbekannten und nicht direkt ermittelbaren) Fähigkeit θ löst ein Proband eine Aufgabe der Schwierigkeit σ mit der Wahrscheinlichkeit:

$$p(X = 1) = \frac{e^{\theta-\sigma}}{1 + e^{\theta-\sigma}}$$

Dies ist das sogenannte „Rasch-Modell“. Es beschreibt den Sachverhalt dass, je höher die latente Fähigkeit des Probanden, desto höher auch die Wahrscheinlichkeit, dass eine Aufgabe richtig gelöst wird. Fähigkeit und Schwierigkeit werden auf derselben Skala gemessen. Stimmen beide überein, so beträgt die Lösungswahrscheinlichkeit $\frac{1}{2}$.

Im praktischen Testablauf werden Fähigkeit und Schwierigkeit im nachhinein gemeinsam geschätzt. Dazu wird eine genügend große und repräsentative Stichprobe an gelösten Aufgaben benötigt. Liegen Daten über die Schwierigkeit von Aufgaben vor, können diese beispielsweise für den Entwurf neuer Tests eingesetzt werden. Auch die Bewertung bearbeiteter Tests vereinfacht sich. Eine weitere Möglichkeit ist der adaptive Einsatz von Aufgaben (Items) anhand des Verhaltens des jeweiligen Probanden: Löst ein Proband eine Aufgabe richtig, so bekommt er nachgehend schwierigere Aufgaben gestellt und anders herum. Dies ist die Grundidee des adaptiven Testens.

2.2 Adaptives Testen

Adaptives Testen baut darauf auf, dass durch *gezieltes* Stellen von Aufgaben der Testablauf insgesamt kürzer und durch Vermeidung von Unter- oder Überforderung weniger frustrierend für den Probanden sein kann. Trotzdem soll dessen

Fähigkeit mit derselben Genauigkeit geschätzt werden wie bei einem (längeren) nicht-adaptiven Test. Dazu wird mit Items gearbeitet, deren Schwierigkeit bereits bekannt ist (anhand einer ausreichenden Stichprobe geschätzt wurde). Nun kann für einen Probanden, der zu einem solche Item eine Lösung angibt eine Schätzung seiner Fähigkeiten vorgenommen werden. Daraufhin wird als nächstes Item dasjenige ausgewählt, das für den Probanden den höchsten Informationsgehalt hat – das also am Besten für eine Fortführung des Schätzvorganges geeignet ist. Während des Testverlaufes wird dieser Schätzvorgang iteriert sobald der Proband eine Aufgabe bearbeitet hat. Der Test wird nach einer festen Anzahl Items abgebrochen oder wenn die Schätzgenauigkeit ein vorher definiertes Maß überschritten hat.

Das oben vorgestellte Rasch-Modell ist das einfachste Modell, das für adaptives Testen eingesetzt wird. Elaboriertere Modelle verwenden beispielsweise zwei oder drei Parameter, neben der Schwierigkeit eines Items noch dessen Diskriminationsfähigkeit und einen Rate-Parameter.

Allen Verfahren gemein ist die Abhängigkeit von der Qualität und Zusammensetzung der Sammlung vorhandener Items, der Itembank. Nur mit einer optimalen Auswahl lassen sich die Ideen des adaptiven Testens auch umsetzen. Stehen nur Items mit beispielsweise stark abweichender Schwierigkeit zur Verfügung, so verringert sich die Genauigkeit des Schätzvorganges oder es lassen sich überhaupt keine neuen Informationen gewinnen. Eine andere Problematik ergibt sich, falls wenige, sehr gut diskriminierende Items mittlerer Schwierigkeit vorhanden sind: Ein optimaler Auswahlprozess wird immer wieder diese Items verwenden, so dass diese „überstrapaziert“ werden (*overexposure*) und damit als allgemein bekannte Items nicht mehr für Testzwecke eingesetzt werden können.

Aus diesem Grund kommt einer eingehenden Untersuchung der Itembank große Bedeutung zu. Dazu sind in der Psychometrie Simulationsläufe Standard, bei denen mit unterschiedlichen Start- und Auswahlparametern Testabläufe betrachtet werden. Zu weiteren Details und Literaturhinweisen siehe beispielsweise [1].

3 Modelle zur Itembank-Analyse

Bei Verwendung einer bekannten Itembank[1] sind die möglichen Testverläufe auch im Voraus berechenbar. Je nach Startbedingung – zufällige Auswahl eines Start-Items, Vorwissen oder Schätzung der Fähigkeit des Probanden – und beobachtetem Testverhalten ergeben sich unterschiedliche „Wege“ durch den Test. Die Gesamtheit aller möglichen Wege kann als kreisfreier gerichteter Graph[2] mit Items als Knoten aufgefasst werden. Dies ist jedoch nicht die einzige mögliche Struktur, im weiteren Verlauf des Kapitels werden noch Alternativen diskutiert.

[1] Dies ist der Standardfall. Abweichungen wie die automatische Generierung von Items während der Testzeit sind bisher eher als experimentell anzusehen.

[2] Abweichungen von einer Baumstruktur ergeben sich daraus, dass Items über unterschiedliche Wege erreichbar sind.

3.1 Item-Erreichbarkeitsbaum

Werden die erreichbaren Items als Knoten aufgefasst, so kann man von einem *Erreichbarkeitsgraphen* für Items sprechen. Sind mehrere Start-Items möglich, so enthält der Graph auch mehrere Startknoten oder diese werden von einem neuen virtuellen Startknoten aus erreicht.

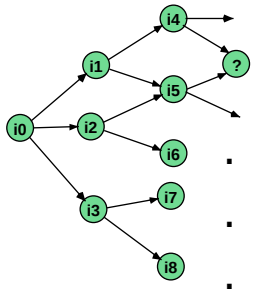

Abb. 1. Beispiel für einen Item-Erreichbarkeitsgraphen

Diese Situation ist in Abbildung 1 exemplarisch dargestellt: Item i0 ist virtueller Startpunkt, die Items i1, i2 und i3 sind die eigentlichen Startpunkte. Von diesen aus gibt es je zwei mögliche Nachfolger, für eine richtige und eine falsche Antwort. Zu den entsprechenden Nachfolgern gelangt man, indem die Fähigkeit des Probanden neu geschätzt und ein entsprechend schwieriges Item ausgewählt wird. Nach einer festen Anzahl Items (Tiefe des Graphen) oder anderen Abbruchkriterien wird die Konstruktion – genau wie der Test – beendet.

Diese Konstruktion kann unter Umständen zu Problemen führen, da die Historie des Testverlaufes eine Rolle spielt: Kein Proband soll ein Item zweimal präsentiert bekommen. Im Beispiel käme für das mit einem Fragezeichen beschriftete Item nur dann i2 in Frage, falls anfangs der Testverlauf über i1 gewählt wurde. Wird i5 aber über i2 erreicht, so darf dieses nicht nochmals verwendet werden. Aus diesem Grund wird statt der intuitiven Graphenstruktur eine (gerichtete) Baumstruktur verwendet, in der Knoten mehrfach vorkommen dürfen, falls sie nicht in einer Linien-Relation stehen (direkter Pfad entweder nur mit oder nur gegen die Pfeilrichtung).

Aus einem solchen Item-Erreichbarkeitsbaum lassen sich direkt erste interessante Aussagen ableiten. So sind Items, die mehrfach im Baum vorkommen, mögliche Kandidaten für eine Überstrapazierung. Noch interessanter sind Items, die gar nicht vorkommen: Sie spielen für den Testablauf keine Rolle. Letzteres Ergebnis lässt sich aus den üblicherweise verwendeten Simulationen nicht (mit Sicherheit) ablesen!

3.2 Fähigkeitsbaum

Eine ähnliche Konstruktion, die aber doch zu anderen Aussagen führt, stellt der *Fähigkeitsgraph* dar. Bei diesem werden die Fähigkeitswerte als Knoten aufgefasst, zwischen denen durch Items gewechselt werden kann. Es handelt sich hierbei um einen gerichteten, aber nicht notwendigerweise kreisfreien Graphen, da ein Fähigkeitswert prinzipiell während eines Testablaufes mehrfach vorkommen kann. Aus ähnlichen Überlegungen wie oben wird diese Graphenstruktur zu einer Baumstruktur, dem *Fähigkeitsbaum* erweitert: Kommt derselbe Fähigkeitswert in einem Test eines Probanden erneut vor, so kann **nicht** dasselbe Item ausgewählt werden wie vorher.

Aus dem Fähigkeitsbaum kann beispielsweise abgelesen werden, welches Intervall an Fähigkeiten ein Test abdecken kann. Eine weitergehende Analyse könnte die Fähigkeitswerte als Histogramm auftragen und mit dem erwarteten Histogramm – üblicherweise eine verschobene Glockenkurve – für die jeweilige Probandengruppe vergleichen. Interessant ist auch ein Vergleich der geschätzten Fähigkeiten mit den jeweils noch zur Verfügung stehenden Schwierigkeiten der Items. Bestehen hier starke Abweichungen, so ist der Testverlauf alles andere als optimal.

3.3 Testablaufmodell

Die in 3.1 und 3.2 vorgestellten Zustandsmodelle können mit einem geeigneten Algorithmus direkt generiert werden. Einfacher jedoch ist der Rückgriff auf bereits vorhandene Werkzeuge und Verfahren wie etwa die Modellierung des Tests selber als Petrinetz und eine anschließende (automatisierte) Analyse.

In Abbildung 2 ist ein (abstraktes) Petrinetzmodell angegeben, aus dem der Fähigkeitsbaum generiert werden kann. Die Schätzung der Fähigkeit startet mit dem Wert 0. Mit dem Fähigkeitswert f_alt wird aus der Itembank ein item ausgewählt. Dies wird vom Probanden gelöst (nicht expliziert) und ein neuer Fähigkeitswert geschätzt. Verändert die Item-Auswahl die Itembank, so ist auch die Erstellung einer Baumstruktur (statt eines Graphen) sichergestellt. Ist das (nicht näher spezifizierte) Abbruchkriterium abbruch erfüllt, kann kein weiteres Item mehr ausgewählt werden.

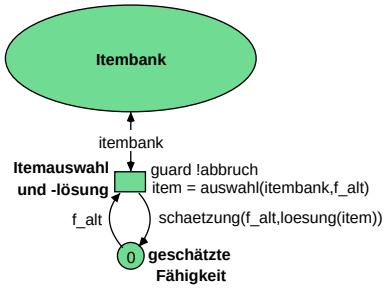

Abb. 2. Test als Petrinetzmodell

Aus der 1:1-Übertragung der Baumstruktur in ein generierendes Petrinetz ergibt sich noch keine Erhöhung der Modellierungsmächtigkeit. Bei komplexeren Testszenarien kann sich aber eine erweiterte Petrinetzdarstellung als sehr vorteilhaft erweisen: Beispielsweise sind (adaptive) Tests gebräuchlich, bei denen jeweils nicht nur ein einziges, sondern eine Menge von Items ausgewählt werden – oft mehrere Fragen zu einem gemeinsamen sogenannten Stimulus[3]. Zwischen diesen Items kann der Proband frei hin- und herspringen und sie in beliebiger Reihenfolge bearbeiten. Dies entspricht im Petrinetzmodell nebenläufigen Transitionen. Erst wenn alle Items bearbeitet sind, wird die Fähigkeit neu geschätzt und so weiter. Eine Modellierung als Automat müsste alle Möglichkeiten aufzählen. Als Petrinetz ist beispielsweise eine Verwendung von Netzen-in-Netzen denkbar: Der Testablauf von Items oder Itemgruppen wird unabhängig als Objektnetz(e) in einem übergreifenden Systemnetz modelliert. Letzteres stellt den Testablauf dar und ist beispielsweise für die Itemauswahl und den Testabbruch verantwortlich.

[3] Beispiele für Stimuli sind Texte, Bilder usw., zu denen dann Fragen gestellt werden.

4 Ausblick

Die Verwendung einer erschöpfenden Analyse statt ausschnitthafter Simulation stellt einen neuen Zugang zu Qualitätskriterien für Itembanken für Tests dar. Nach bestem Wissen des Autors ist eine solche Analyse bisher in der Literatur nicht vertreten.

In diesem Papier konnten nur die grundsätzlichen Möglichkeiten der Itembank-Analyse vorgestellt werden. Vor allem der Bereich der Psychometrie blieb offen: Was sind interessante und praktisch relevante Kriterien für die Qualität einer Sammlung von Items? In dieser Richtung forscht der Autor zur Zeit im Verbund mit Fachwissenschaftlern und Praktikern. Die daraus gewonnenen Erkenntnisse sollen mittelfristig in ein Werkzeug einfließen. Dafür sind auch die Modelle zu konkretisieren und eine Auswahl zu treffen, die sich an den bereits vorhandenen, frei verfügbaren und erweiterbaren Werkzeugen orientiert.

Leicht ausbauen lassen sich die hier vorgestellten Gedanken auch durch die Verwendung probabilistischer Automaten- und Petrinetzmodelle. Dann wäre die direkte Ableitung quantitativer Aussagen möglich. Des Weiteren lässt die Modellierungsmächtigkeit bisher noch viel zu wünschen übrig: Unterschiedliche Testszenarien und Anfangsbedingungen lassen sich bisher nur schwer integrieren. Hier muss die Analyse noch gegenüber der Simulation aufholen, um praktische Relevanz zu erlangen.

Literatur

1. Tim Davey and Mary J. Pitoniak. *Designing Computerized Adaptive Tests*, chapter 24. Lawrence Erlbaum Associates, Inc., Publishers, 2006.
2. R.K. Hambleton, H. Swaminathan, and H.J. Rogers. *Fundamentals of item response theory*. Newbury Park: Sage, 1991.

Cooperative Interaction with a Multi-Partner Service

Karsten Wolf, Universität Rostock, Institut für Informatik

Abstract

Consider the model of a service S that interacts with more than one partner. How can one of the partners interact with S without exactly knowing how the remaining ones behave? We propose a solution to this task which is significantly more general than our first solution proposed in [Sch05].

1 Introduction

There exist web services which interact with not just one partner (like a provider/requester pair), but with several services. Examples are online shops (which interact with customers, a financial institute for handling payment, and manufacturers) and travel agencies (which interact with customers, flight reservation services, hotel reservation services etc.). We call such a service a *multi-partner* service. Each partner of such a service S needs to construct or select its behaviour based just on the description of S itself, i.e. without knowledge of the operational behaviour of other communication partners of S. This task is not always simple, and in some cases impossible.

In this article we propose an approach for supporting the construction or selection of *one* partner S_i of a multi-partner service, without knowledge on the particular way in which the remaining partners are constructed or selected. The approach is based on a set of rules that, if obeyed by every partner of S, guarantee that the overall system is correct in the following sense: The system will never run into a deadlock, and for some globally fixed number k (for instance, provided by S), there will never be more than k messages pending in a communication channel between S_i and S. We see these properties as a minimum requirement for a correct interaction between services. Any reasonable specification of "proper interaction" will indeed include the properties listed ones, so an approach to any other notion of correctness would certainly be an extension of the approach proposed here.

The rules for cooperative behaviour are formulated such that it is straightforward to adapt existing approaches for deciding (autonomous) controllability and generating operating guidelines from the case of single-partner services to the (autonomous) multi-partner case.

2 Formalisms

Modeling services

We model a service as a *service automaton*. Service automata may be easily retrieved from any operational formal semantics [Fer04, AFFK04, FBS04, FGV04, FR05, HSS05, OVA+05] of the popular service description language WS-BPEL [AAA+07]. In [LMW07], a translation from WS-BPEL via a feature complete Petri net semantics into service automata is briefly explained.

Throughout the paper, we denote S the service automaton under consideration. With $S_1, \ldots, S_n$, we denote the partner services that interact with S.

Def. 1 (Service automaton, well formed) *A* service automaton S *consists of the following ingredients:*

- *a nonempty finite set* Q *of* states*, including an* initial state s_0 *and a set* F *of* final states*;*
- *a finite interface* $\mathcal{M} = \mathcal{M}_{in} \uplus \mathcal{M}_{out}$*;*
- *an* arity *(number of partners)* n *and a* partner assignment $\pi : \mathcal{M} \to \{1, \ldots n\}$*;*
- *a nondeterministic transition relation* $\delta \subseteq Q \times \mathcal{M} \uplus \{\tau\} \times Q$*;*

such that $q \in F$ *and* $[q, m, q'] \in \delta$ *implies* $m \in \mathcal{M}_{in}$*. A service automaton is* well-formed *if there is no infinite sequence* $q_1 q_2 \ldots$ *of states* $q_i \in Q$ *where, for all* i*,* $[q_{i-1}, \tau, q_i] \in \delta$*.*

Services are only composable if their communication interfaces given by π fit.

Def. 2 (Composable services) *Two services* S_1 *and* S_2 *are composable iff there are numbers* i $(1 \leq i \leq n_1)$ *and* j $(1 \leq j \leq n_2)$ *such that, for each* $m \in \mathcal{M}_1 \cup \mathcal{M}_2$*,*

- $\mathcal{M}_1 \cap \mathcal{M}_2 = \{m \mid \pi_1(m) = i\} = \{m \mid \pi_2(m) = j\}$*;*
- *for all* $m \in \mathcal{M}_1 \cap \mathcal{M}_2$*,* $m \in \mathcal{M}_{out,1}$ *iff* $m \in \mathcal{M}_{in,2}$ *and vice versa.*

Integrating the message channels into the state of a composed system may yield an infinite set of states as $Bags(\mathcal{M})$ may be infinite. It is indeed possible to compose services such that eventually there are arbitrarily many messages pending. On the other hand, all services that are correct in the sense stated in the introduction yield a finite state space as there are never more than k messages of a kind pending in a channel. We thus take a pragmatic approach and represent pending messages with elements of the finite set $Bags_b(\mathcal{M})$, for some b which is assumed to be larger than the number k apppearing in our correctness criterion.

Def. 3 (Composition of Services) *Let* S_1 *and* S_2 *be composable services. Then their composition* $S = S_1 \oplus_b S_2$ *has the following ingredients:*

- $Q_S = Q_{S_1} \times Bags_b(\mathcal{M}_1 \cap \mathcal{M}_2) \times Q_{S_2}$*,* $q_{0,S} = [q_{0,S_1}, \emptyset, q_{0,S_2}]$*,* $F_S = F_{S_1} \times \{\emptyset\} \times F_{S_2}$*;*
- $\mathcal{M}_S = (\mathcal{M}_{S_1} \setminus \mathcal{M}_{S_2}) \cup (\mathcal{M}_{S_2} \setminus \mathcal{M}_{S_1})$*;*
- $n_S = n_{S_1} + n_{S_2} - 2$*,* $\pi_S(m) =$ *if* $m \in \mathcal{M}_{S_1}$ *then* $\pi_{S_1}(m)$ *else* $\pi_{S_2}(m) + n_{S_1} - 1$*);*
- δ_S *contains the following transitions:*
  - $[[q_1, M, q_2], \tau, [q_1', M, q_2]]$ *for all* $[q_1, \tau, q_1'] \in \delta_{S_1}, M \in Bags_b(\mathcal{M}), q_2 \in Q_{S_2}$ *(internal action in* S_1*);*
  - $[[q_1, M, q_2], \tau, [q_1, M, q_2']]$ *for all* $[q_2, \tau, q_2'] \in \delta_{S_2}, M \in Bags_b(\mathcal{M}), q_1 \in Q_{S_1}$ *(internal action in* S_2*);*
  - $[[q_1, M, q_2], \tau, [q_1', M + [a], q_2]]$ *for all* $[q_1, a, q_1'] \in \delta_{S_1}, a \in \mathcal{M}_{out,S_1} \cap \mathcal{M}_{in,S_2}, M \in Bags_b(\mathcal{M}), q_2 \in Q_{S_2}$*, if* $M(a) < b$ *(send from* S_1 *to* S_2*);*
  - $[[q_1, M, q_2], \tau, [q_1, M + [a], q_2']]$ *for all* $[q_2, a, q_2'] \in \delta_{S_2}, a \in \mathcal{M}_{out,S_2} \cap \mathcal{M}_{in,S_1}, M \in Bags_b(\mathcal{M}), q_1 \in Q_{S_1}$*, if* $M(a) < b$ *(send from* S_2 *to* S_1*);*

- $[[q_1, M, q_2], \tau, [q_1', M-[a], q_2]]$ *for all* $[q_1, a, q_1'] \in \delta_{S_1}, a \in \mathcal{M}_{in,S_1} \cap \mathcal{M}_{out,S_2}, M \in Bags_b(\mathcal{M}), M(a) > 0, q_2 \in Q_{S_2}$ *(receive in* S_1 *from* S_2*);*
- $[[q_1, M, q_2], \tau, [q_1, M-[a], q_2']]$ *for all* $[q_2, a, q_2'] \in \delta_{S_2}, a \in \mathcal{M}_{in,S_2} \cap \mathcal{M}_{out,S_1}, M \in Bags_b(\mathcal{M}), M(a) > 0, q_1 \in Q_{S_1}$ *(receive in* S_2 *from* S_1*);*
- $[[q_1, M, q_2], a, [q_1', M, q_2]]$ *for all* $[q_1, a, q_1'] \in \delta_{S_1}, a \in \mathcal{M}_{S_1} \setminus \mathcal{M}_{S_2}, M \in Bags_b(\mathcal{M}), q_2 \in Q_{S_2}$ *(communication between* S_1 *and other partners);*
- $[[q_1, M, q_2], a, [q_1, M, q_2']]$ *for all* $[q_2, a, q_2'] \in \delta_{S_2}, a \in \mathcal{M}_{S_2} \setminus \mathcal{M}_{S_1}, M \in Bags_b(\mathcal{M}), q_1 \in Q_{S_1}$ *(communication between* S_2 *and other partners).*

In the sequel, we consider a multi-partner service S with partners which communicate only with S and not with each other. In this setting, composition of S with all its partners can be traced back to a sequence of of compositions that starts with S and adds its partners one at a time. The final result of this composition is essentially independent of the order in which partners are composed with S.

Correctness

According to Def. 3, the occurrence of a transition labeled with an $a \in \mathcal{M}_{in}$ (a receiving transition) depends on the presence of the message in the bag of pending messages. There may be two different kinds of situations where a service cannot continue its execution in a non-final state. First, it may run into a state that does not have any successor states. We call such a state a *deadlock*. Second, it may run into a *wait state*. This is a state which can be left only via input transitions. In a wait state, the service cannot continue execution unless a partner has sent an appropriate message. Wait states require particular care in the construction of partners.

Def. 4 (Deadlock, wait state, transient state) *In a service* S*, a state* $q \in Q \setminus F$ *such that there is no* a *and* q' *with* $[q, a, q'] \in \delta$ *is called* deadlock*. A state* $q \in O \setminus F$ *where* $[q, a, q'] \in \delta$ *implies* $a \in \mathcal{M}_{in}$*, and which is not a deadlock, is called* wait state*. A state* $q \in Q$ *such that there is a transition* $[q, a, q'] \in \delta$ *with* $a \notin \mathcal{M}_{in,S}$*, is called transient state of* S*.*

A wait state of a service S may or may not yield a deadlock in a composition of S with some other service. A state that is neither a deadlock nor a wait state will not yield a deadlock in any composition of S with another service. In a transient state, a service has the ability to continue execution without external influence.

As stated in the introduction, we consider only those compositions correct where the message bags representing the pending messages do not contain more than k messages of a kind, for some given k. In the following definition, remember that we used an artificial bound b for the size of message bags.

Def. 5 (k-limited communication) *Two composable services* S_1 *and* S_2 *have* k*-limited communication if, for any* $b > k$ *and every state* $[q_1, M, q_2] \in Q_{S_1 \oplus_b S_2}$*,* $M \in Bags_k(\mathcal{M}_{S_1} \cap \mathcal{M}_{S_2})$*.*

In other words, the services do never attempt to put more than k messages on a channel. With k-limited communication and deadlocks, we can now summarize our correctness criterion. We thereby consider a system consisting of a multi-partner service S with arity n and n single partner services S_i, each communication with S but not with any other partner of S.

Def. 6 (Correctness) *A system consisting of a service* S *having arity* n *and* n *Services* S_i $(1 \leq i \leq n)$ *where* S *and* S_i *is composable for all* i*, and* $\mathcal{M}_{S_i} \cap \mathcal{M}_{S_j} = \emptyset$ *for* $1 \leq i \neq j \leq n$ *is* k-correct *if all services* $S \oplus S_i$ *have* k*-limited communication and the composition* $(\ldots((S \oplus S_1) \oplus S_2) \oplus \ldots) \oplus S_n$ *does not have any (reachable) deadlocks.*

Reasoning about services

We are now ready to provide the basic formal tools used in the subsequent considerations and algorithms. First, we introduce the concept of *bad states*. Informally, a bad state of S is a state from which the occurrence of a deadlock can no longer be avoided by the partners of S.

Def. 7 (Bad state) *Let S be a service. Then the set of* bad states *of S is inductively defined as follows:*

- *If $q \in Q \setminus F$ is a deadlock of S then q is a bad state of S.*
- *If $q \in Q$, $[q, a, q'] \in \delta$ a transition where $a \notin \mathcal{M}_{in}$, and q' is a bad state, then q is a bad state of S as well.*
- *If $q \in Q$ is a wait state of S, and, for all transitions $[q, a, q'] \in \delta$, q' is a bad state, then q is a bad state of S as well.*

Avoiding a bad state is not always easy since a partner S' of S does not necessarily know exactly, in which state S is. This is due to the nondeterminism and the presence of internal transitions in S. Typically, we can, to each state q' of S', only assign a *set* of states where S and the pending messages could be in while S' is in q'. This information can be easily retrieved from the composition of S and S'.

Def. 8 (Knowledge) *Let S and S' be composable services. Let $q' \in S'$. Then the b-Knowledge of S' about S in state q', $K_{b,S,S'}(q') = \{[q, M] \mid [q, M, q'] \in Q_{S_1 \oplus_b S_2}$.*

Assuming a sufficiently large b, we shall drop the index b in the sequel. We shall also drop the remaining indices as they will always be clear from the context.

3 Cooperative partners

In this section, we define the rules according to which we expect all partners of a given service S to behave. We show that, if all partners obey the rules, the system consisting of S and all of them is correct. The rules can be informally summarized as follows.

1. Do not allow S to enter a bad state;
2. Do not let one of your channels get filled beyond k messages;
3. If S could be in a final state and you are not, then have a transition ready to be executed.
4. Help S out of a wait state if that is possible.
5. Do not leave surplus messages with final states, unless you can assure that they can be handled.

The formal version of these rules reads as follows.

Def. 9 (Cooperative partner) *Let S be a service and S' a service of arity 1 that is composable with S. S' is called* k-cooperative *with respect to S if the following conditions hold:*

1. *$q_{0,S}$ is not a bad state of S and, for all $q' \in Q'$ and all $[q, M] \in K(q')$, if q is not a bad state of S, $[q, a, q_1] \in \delta_S$, and q_1 is a bad state of S, then $M(a) = 0$.*

2. $q' \in Q_{S'}$ *and* $[q, M] \in K(q')$ *implies* $M \in Bags_k(\mathcal{M})$.

3. *For all* $q' \in Q'$ *and all* $[q, M] \in K(q')$, *if* $q \in F_S$, q *is not bad, and* $q' \notin F_{S'}$ *then* $[q, M, q']$ *is transient in* $S \oplus S'$.

4. *For all* $q' \in Q'$ *and all* $[q, M] \in K(q')$, *if* q *is a wait state of* S *and not bad such that, for all* $[q, a, q_1] \in \delta_S$, $a \in \mathcal{M}'_S$ *implies* $M(a) = 0$, *and there exists a message* $b \in \mathcal{M}_{S'}$ *such that* $[q, b, q_2] \in \delta_S$ *and* q_2 *is not bad in* S, *then* $[q, M, q']$ *is transient in* $S \oplus S'$.

5. *For all* $q' \in Q'$ *and all* $[q, M] \in K(q')$, *if* $q \in F_S$ *and not bad then, for all* $a \in \mathcal{M}_S$, $M(a) > 0$ *implies that* $[q, M, q']$ *is transient in* $S \oplus S'$.

For some S, cooperative partners do not exist. We consider them ill-designed. As the following result shall show, the benefit of being cooperative is that, if all partners of S are cooperative then the overall system consisting of S and its partners is correct.

Thm. 1 *Let* S *be a service of arity* n, *composable with* n k*-cooperative partners* $S_1, \ldots, S_n$ *which have pairwise disjoint message sets. Then* $S^* = (\ldots((S \oplus S_1) \oplus S_2) \ldots) \oplus S_n$ *is* k*-correct.*

Proof. (Sketch) Assume first that a deadlock q^* is reachable in S^* and consider an execution sequence $q_0^* q_1^* \ldots q_j^* = q^*$ in S^*. Consider the projections $q_0, q_1, \ldots, q_j$ to S, $q_{i,0}, q_{i,1}, \ldots, q_{i,j}$ to S_i, and $M_{i,0}, M_{i,1}, \ldots, M_{i,j}$ to the message bags representing the messages pending between S and S_i. It is easy to show that, for all i, all states $[q_0, M_{i,0}, q_{i,0}], [q_1, M_{i,0}, q_{i,1}], \ldots, [q_j, M_{i,j}, q_{i,j}]$ are reachable in $S \oplus S_i$.

We start by showing that none of the states $q_0, \ldots, q_j$ is bad in S. Assume that there is a bad state in this sequence. Then there must be a first appearance of a bad state in this sequence, say q_h. Consider the transition that transformed q_{h-1} into h. As $q_{h-1} = q_h$ for all transitions in S^* which stem from transitions not in S, the considered transition corresponds to one in S. It must further be an input transition since otherwise badness of q_h would imply (by Def. 7) badness of q_{h-1}, that is, there is a $a \in \mathcal{M}in, S$ with $[q_{h-1}, a, q_h] \in \delta_S$. This means that In the considered configution there is a partner S_g where $a \in \mathcal{M}_{out,S_g}$ and $M_{g,h-1}(a) > 0$. This contradicts the first item of Def. 9 as $[q_{h-1}, M_{h-1}] \in K(q_{g,h-1})$.

q^* can only be a deadlock if q^* is not a final state of S^*. There can be several reasons for q^* being non-final. First, one of the $M_{i,j}$ can contain a surplus message, second, any of the $q_{i,j}$ could be non-final, or, third, q_j could be non-final.

Case 1: $q_j \notin F_S$. Since q_j is not transient in S (otherwise, q^* would not be a deadlock), q_j cannot be a deadlock of S (otherwise, q_j would be bad), q_j is a wait state, and, for all $[q, a, q'] \in \delta_S$, we have $M(a) = 0$.. As q_j is nod bad, there is at least one transition $[q, a, q'] \in \delta_S$. Let S_i be the partner of S where $a \in \mathcal{M}_{out,S_i}$. Obviously, $[q_j, M_{i,j}] \in K(q_{i,j}$, so S_i violates the fourth item of Def. 9.

Case 2: $q_j \in F_S$ and there is an i such that $q_{i,j} \notin F_{S_i}$. Then, by the third item of Def. 9, $[q_j, M_{i,j}, q_{i,j}]$ is transient in $S \oplus S_i$, so there must be a transition of S or S_i executable in q^*, in contradiction to the fact that q^* is a deadlock in S^*.

Case 3: $q_j \in F_S$, $q_{i,j} \in F_{S_i}$ $(1 \leq i \leq n)$, but there are an i and an a such that $M_{i,j}(a) > 0$. Now the fifth item of Def.9 requires that $[q_j, M_{i,j}, q_{i,j}]$ must be transient in $S \oplus S_i$, so there must be a transition executable in state q^* of service S^*, in contradiction to the assumption that q^* is a deadlock.

As none of the considered cases applies, q^* cannot be a deadlock, so S^* is deadlock-free. Further, S^* has k-limited communication, as a violation of this property would be reflected in some $M_{i,h} \notin Bags_k(\mathcal{M}$, in contradiction to the second item of Def. 9.

Assume now that S^* violates the property of limited communication, that is, at least one channel x gets filled beyond k messages. But then x gets filled beyond k messages in $S \oplus S_i$, too, where S_i is the partner of S that is connected to x. This is a contradiction to the second item of Def. 9. q.e.d.

4 Computing a cooperative partner (Sketch)

As in [Sch05, LMW07], we may construct a partner S_i of S by identifying every state q of S_i with its corresponding knowledge value $K(q)$. This way it is easy to evaluate the first two items of Def. 9. The remaining items express, for a given state q of S_i, requirements on the presence of edges leaving q. We remove edges only if we forced to do so by Def. 9. If this procedure removes the initial state, S_i cannot behave cooperatively. Otherwise, we obtain a most permissive cooperative behaviour B_i for S_i. The requirements of Def. 9 can then be translated into Boolean annotations to the states of B_i and yield an operating guideline for S_i, i.e an operational description of *all* cooperative behaviours for S_i.

5 Conclusion

The construction of a most permissive cooperative behaviour for any partner of a multi-partner service S gives us the opportunitiy to decide existence of cooperative partners. Shipping an operating guideline to every partner of S gives the partners of S the possibility to chose their desired operational behaviour (within the frame given by the operating guideline) independently from the remaining partners. For every choice of the partners, the overall system behaves correctly in the sense stated in the introduction.

References

[AAA+07] A. +Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guzar, N. Kartha, C.K. Liu, R. Khalaf, D. Knig, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web Services Business Process Execution Language Version 2.0. Committee Specification, OASIS, 2007.

[AFFK04] J. Arias-Fisteus, L. Sánchez Fernández, and C. Delgado Kloos. Formal Verification of BPEL4WS Business Collaborations. In *Proc. EC-Web 2004*, pages 76–85, 2004.

[FBS04] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proc. WWW 2004*, pages 621–630, 2004.

[Fer04] A. Ferrara. Web services: a process algebra approach. In *Proc. ICSOC*, pages 242–251, 2004.

[FGV04] R. Farahbod, U. Glässer, and M. Vajihollahi. Specification and Validation of the Business Process Execution Language for Web Services. In *Proc. ASM*, volume 3052 of *LNCS*, pages 78–94, 2004.

[FR05] D. Fahland and W. Reisig. ASM-based semantics for BPEL: The negative Control Flow. In *Proc. ASM*, pages 131–151, 2005.

[HSS05] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri nets. In *Proc. BPM*, volume 3649 of *LNCS*, pages 220–235, 2005.

[LMW07] N. Lohmann, P. Massuthe, and K. Wolf. Operating Guidelines for Finite-State Services. In *Proc. Petri Nets*, volume 4546 of *LNCS*, pages 321–341, 2007.

[OVA+05] C Ouyang, E. Verbeek, W.M.P. van der Aalst, S. Breutel, M. Dumas, and A.H.M. ter Hofstede. Formal Semantics and Analysis of Control Flow in WS-BPEL. Technical report (revised version), Queensland University of Technology, 2005.

[Sch05] K. Schmidt. Controllability of Open Workflow Nets. In *Proc. EMISA*, number P-75 in LNI, pages 236–249, 2005.

Towards Applicability of Language Based Synthesis for Process Mining

Robin Bergenthum, Robert Lorenz, Sebastian Mauser

Lehrstuhl für Angewandte Informatik
Katholische Universität Eichstätt-Ingolstadt, Eichstätt, Germany
e-mail: {robin.bergenthum, robert.lorenz, sebastian.mauser}@ku-eichstaett.de

Abstract

In this paper we give an overview of methods to adapt Petri net synthesis based on regions of languages for process mining. These methods are understood to supplement the basic process mining approach presented in [BDLM07].

1 Introduction

Many of today's information systems record information about performed activities of processes in so called event logs. Not only classical workflow management systems, but also web services, middleware systems, embedded systems in high-tech equipment such as medical systems and many other information systems produce event logs. Process mining techniques attempt to extract useful, structured information from the vast amount of information recorded in event logs. In this paper we focus on constructing a process model, which matches the actual workflow of the recorded information system, from an event log. This prevalent aspect of process mining is known as process or control-flow discovery. There are many process discovery techniques in literature (see e.g. [ADH + 03]), often implemented in the ProM framework [PE].

In [BDLM07] we proposed a process discovery approach based on the theory of regions of languages. It aims at constructing a Petri net from a typical event log recording the executed activities together with cases (i.e. process instances) the activities belong to. Assuming that the cases of a log are executed independently, each case defines a sequence of activities describing the ordering of the execution of the activities. Thus an event log can be interpreted as a set of traces defining a language. That means, in a natural way methods derived from the theory of regions of languages can be applied to synthesize a Petri net. In [BDLM07] we presented two basic process discovery algorithms constructing a Petri net from an event log. This process discovery approach is a precise one in the sense that it basically constructs a Petri net reproducing the traces of the event log and having *minimal* additional behaviour.

In contrast, many classical process discovery approaches are imprecise. In order to be efficient in time and memory consumption and to keep the resulting process models small, these process models allow for much more additional behaviour as necessary – they *overapproximate* the given event log. Our precise approach overcomes many of the limitations of such classical imprecise approaches. These have problems with complex control-flow structures such as non-free-choice constructs, unbalanced splits and joins, nested loops, etc., although in reality processes often exhibit such features. These problems are resolved by the precise algorithms in [BDLM07]. Of course any process discovery algorithm is in particular restricted to control-flow structures allowed by the target language for the process model. The approach in [BDLM07] allows to easily vary the target language between several classes of Petri nets. The second main problem of imprecise methods is their tendency to *underfit* the event log, i.e. the resulting model allows for much more behaviour without any indication to be reasonable real behaviour. This problem of underfitting is of course irrelevant for precise models.

On the other hand, precise methods are often criticized for their tendency to *overfit* [AG07]. Of course a log is usually incomplete, i.e. there may be possible traces not occurring in the log. On the first glance this argues against precise approaches. But without additional information it is unclear, which possible traces are missing in the log. It is hardly possible to extract information about such missing traces solely from the event log. The overapproximation performed by most existing imprecise approaches seems to be quite arbitrary, at least hardly controllable and predictable. When using the precise approach as a basis, it is possible to specifically introduce overapproximation. In this paper we develop several modular methods to adapt the basic precise approach in order to get a process model overapproximating the event log in a specific, targeted direction. Some additional information or expert knowledge may be used to determine the directions of overapproximation. This makes the overapproximation controllable and predictable, and (since the starting point is a precise algorithm) in particular leads to reliable results. The selection possibility of overapproximation methods makes the final process discovery algorithm configurable to address different needs of the user.

A problem with precise process mining methods is, that they may be inefficient in time and memory consumption. In our situation, one of the two algorithms presented in [BDLM07] has a polynomial runtime and memory consumption, while the other one may be exponential in the worst case. Since we have no experimental results so far, we cannot

say if the complexity of the two basic algorithms is sufficient for practical applications. In any case, we will propose modular implementable techniques to improve the runtime and the memory consumption of the algorithms in this paper. In particular some overapproximation techniques improve the runtime and the memory consumption.
Lastly, the key drawback of a precise process discovery method is, that it usually generates a quite large process model. The number of components of the model – in our situation mainly the number of places of the Petri net – often has to be large in order to potentiate an exact reproduction of the event log. This is the real problem arising from the overfitting of precise algorithms. The size of the model has to be in such a range, that the model can easily be interpreted by the user. Practitioners and process analysts in industry are usually interested in concise and controllable reference models. "Spaghetti"-models that can only be evaluated with computer support are mostly not satisfactory. Therefore we propose modular methods to reduce the number of places and arcs of the net, mined by our approach. Of course there is a trade-off between simplifying the mined net and maintaining a preferably precise model. Therefore many of the proposed simplification methods overlap or coincide with overapproximation methods. In order to get a compact model, also abstraction and visualization techniques for the mined model are useful. The danger of mining "spaghetti"-models is especially problematic in logs of unstructured processes in less restrictive environments [AG07]. However this is not a specific problem of our approach, but also of most classical process mining techniques.
We also propose some methods to improve the mining quality by integrating possible additional information going beyond the pure event log. That means, we are interested in exploiting specific additional information sources or expert knowledge to improve the matching of the constructed process model with the actual flow of work. Besides a better reproduction of reality, additional information is also used in the improvement methods concerning performance, compactness and overapproximation. If such additional information is existent, it is in our opinion important to offer possibilities to the user to exploit such information for the construction of the process model.
Altogether in this paper we make proposals for modularly implementable improvement methods to yield a practically useful and applicable process discovery tool from the precise algorithms in [BDLM07]. The methods primarily cover four aspects: specific targeted overapproximation, improvement of the performance, reduction and simplification of the resulting process model and better reproduction of the reality. The methods are either targeted heuristical techniques or techniques exploiting additional information sources. The methods can be modularly integrated to the process mining algorithms. The considerations of this paper will show that our precise algorithms are very well suited to integrate such methods. Thus we propose a huge number of methods all applicable to our approach. In a practical setting our process mining approach can then very well be configured to the needs of the user by offering the possibility to modularly choose an appropriate selection of these methods. The effects and biases of these methods are well predictable, such that the complete process mining approach leads to reliable, interpretable and reasonable results. The next section gives a comprehensive overview of possible improvement methods to our precise process mining algorithms. We sketch basic methods, their integration to our mining approach and their contribution to make our mining algorithm applicable and high-quality in practice. We do not explain the methods, especially their technical fundamentals, in detail in this paper.

2 Methods to Improve the Process Mining Approach Based on Regions of Languages

In [BDLM07] we described two basic process mining algorithms based on regions of languages. In this section we will restrict ourself to the more promising polynomial algorithm using separating regions and propose improvement methods towards a practical applicability. This algorithm computes a Petri net from an event log basically as follows: The transitions are given by the set of action names occurring in the given language. Each place (together with its connections to all transitions) is computed as a non-negative integer solution of some linear inequation system. This inequation system is chosen in such a way, that w.r.t. a place, which is a solution, all traces of the event log are still possible, while there is at least one trace not belonging to the log, which is prohibited by the place. Such places are computed for a finite set of such traces not belonging to the log, called *wrong continuations*. Wrong continuations extend traces from the log by one additional event.
Before we sketch the improvement methods for our process mining algorithm, we give a systematic classification of these methods. We identified four dimensions for the methods:

(A) *The aim of the improvement method*: Combination of one or more of the issues targeted overapproximation, better performance of the algorithm, more compactness of the resulting model and better reproduction of reality.

(B) *The procedure of the method*: Methods based on heuristics or additional information beyond the pure event log. Heuristics are general principles that can be applied for each log and bias the approach to a certain direction. They often try to extract probably reasonable assumptions for the resulting process model from the pure log. In contrast, additional information is typically very specific, e.g. expert knowledge about relationships of certain activities or some other external information sources. Also logged information in the event log beyond activities and cases is additional information in our context.

(C) *The phase of the process mining approach, to which the method is applied*: Pre-processing (filtering) phase of the event log, processing phase (the actual mining algorithm working on the pre-processed log), and phase

of post-processing of the resulting process model. We are not only concentrating on methods adapting the processing phase, but we also consider pre-processing and post-processing methods specifically tailored to the mining algorithm. Pre-processing and post-processing methods are especially important methods to realize the formulated aims, because these methods do in no way influence the actual precise mining algorithm. Therefore the changes made to the result of the process mining approach by pre- and post-processing methods are completely controllable.

(D) *The user interaction*: Different shapes depending on the improvement methods. Of course there may be no user interaction, but asking an experienced user for help in some problematic or unclear situations may significantly improve the process mining result. The degree of user interaction may be determined by the user. A possibility to avoid constant user interaction is to offer the user the possibility to parameterize the improvement methods. The parameter then defines to which degree a certain method is applied in the final process mining algorithm.

We organize the presentation of the methods according to their phase (C) (pre-processing, processing, post-processing). A classification of the methods to the other three dimensions is no problem: The aims covered by a method are usually obvious. It is also clear, if additional information is required for the method. The user interaction for a method can usually be varied from non to a-priori settings to a confirmation request for every step of the method.

2.1 Pre-processing phase

In this phase the given event-log is filtered to get a less complicated event log. Typically, some *activities are completely deleted* from the log. These activities are not considered in the resulting process model. Such activities are *infrequent activities* or activities classified as *unimportant activities* (e.g. detected from data fields or resources in the event log), *noise activities* or *uninteresting activities* (e.g. start events for activities, that are divided into a start and a complete event). These activities can be identified using some additional information, but also algorithms searching in the pure log for special characteristics to detect such activities are imaginable, e.g. based on frequencies or more advanced stochastic models. User interaction may be very helpful in this case.
Sometimes it is also important to *combine several activities to one activity* (e.g. because of similar names or relations to occurrences of other activities) or to *split one activity into several activities* (e.g. because of extremely different relations to occurrences of other activities). Combined activities will refer to the same transition in the process model, while events of a split activity refer to different transitions. These transformations of the log are of special importance in our approach, because our basic mining algorithm allows no label splitting. Also additional information can very well be exploited for these transformations, e.g. the coincidence of logged data fields, transactional information and originators of events or knowledge about similarity of certain activities. Even adding completely *new routing activities* to the log can be useful. That means one could try to detect patterns in the log, that provide an indication for hidden routing activities.
Furthermore *removing*, *adding* or *editing entire traces, sub-traces or parts of traces* of the log are of special interest. Since the algorithm is precise, a removed behaviour will not be included in the mined process model, while an added or edited behaviour will. Removing traces simplifies the log. As for activities, infrequent (several cases define the same trace and thus traces have frequencies w.r.t. the log), unimportant, noise, uninteresting, exceptional or even faulty traces may be identified and deleted. If it is possible to identify faulty traces, they seem to be a problem in the real system and deserve special attention. It is reasonable to not only delete them, but also store them as *undesirable behaviour*. There may also be faulty traces not occurring in the event log. These may also be detected, altogether yielding a set of explicitly unwanted traces. Such a set can very well be exploited by the actual mining algorithm as sketched later on. Adding traces supports a specific overapproximation. The aim is to add traces, that are (probably) possible in the real system, or do marginally influence the real system but simplify the mined model. Traces may be edited by aggregating sub-traces or parts of traces to one activity. Editing traces is also important to account for special dependencies of cases, e.g. by analyzing resources, originators and data of events of different cases. This is an important task of pre-processing, because the actual mining algorithm assumes, that the cases are executed independently. Editing traces also includes tasks like splitting a trace (e.g. to better reflect a recorded case) or combining traces (e.g. to collapse similar cases or to merge related cases), similarly as for single activities. The possibilities to detect candidate traces for methods described in this paragraph are similar as for respective methods concerning single activities.
For traces as well as for activities it makes sense to build *stochastic models*. That means instead of simply deleting less relevant traces or activities, one can associate each trace or activity with a certain relevance score. Such *relevance rating* differentiates in detail between less representative and highly significant traces or activities. Analogously, instead of simply combining similar traces or activities one can introduce *similarity ratings*. These rating procedures can also be refined by rating sub-traces or certain patterns of activities. Not only frequencies, but also typical patterns and interrelationships of activities or traces as well as additional information can be used to develop a stochastic model based on such ratings. As shown later on, evolved ratings can very well be used by our mining approach, possibly leading to better results than just deleting or combining traces or activities.
Pre-processing also plays a role in detecting probable specific relations of events, e.g. *causal independency/dependency* of events, *cycles* of events or events in *conflict*. Information on causally independent events can be used to transform

the log into a *partial language*, translating the traces to partially ordered scenarios. Another application is a reasonable *decomposition of the set of traces* of the log into behavioural (relatively) independent parts. Alternatively, a *decomposition of the set of activities* of the log into subsets of activities defining distributed components of the process is reasonable. There is a huge amount of possibilities to extract such relations of events from a log, e.g. frequencies and ratings (see above) of traces, activities and of certain patterns in the traces, similarities (given by ratings) of certain patterns, anomalies in the log, systematic deviations of certain patterns or dependencies of activities in the traces. Additional information in the log such as data fields of activities, originators of activities, time spans between certain events or transactional information as well as expert knowledge (e.g. given by some a-priori potentially incorrect model of the process) are particularly useful. Some information systems even directly record partially ordered cases or independencies of events (e.g. the ARIS tool). How information about such relations is integrated to the actual mining algorithm or post-processing methods, is discussed in the next two subsections.

2.2 Processing

Concerning the actual mining algorithm, first one can *vary the target language* of the process model according to the needs of the user. Our approach offers several Petri net classes, that vary in expressivity: General place-transition nets (p/t-nets), pure nets, workflow nets, nets with capacities, elementary nets, nets with unweighted arcs, etc. The structural requirements defining a target Petri net class are introduced into the algorithm by adapting the inequation systems, which are solved by our mining algorithm to define places. Many properties of the final net can easily be introduced by extending these systems with additional requirements for the places and their connections. In this way it is even possible to directly introduce some correctness properties for the final net into the algorithm. We do not refer here to correctness properties concerning the relation of the model and the log, but to the internal consistency of the model (e.g. concerning properties such as the popular soundness requirement for workflow nets).
Adaption of these inequation systems can also be used to introduce some *specific behavioural information* extracted in the pre-processing phase or from an external source (causal (in)dependencies of activities, cycles of activities, resource sharing of activities, etc.). There are three specially interesting examples of such behavioural information:

(i) *Translating the log into partially ordered behaviour*: Then arbitrary concurrency relations are reflected, also referred to as true concurrency. An adaption of our basic algorithm shown in [LBMD07], which regards flows of tokens, can be applied to yield analogous mining results in this more complex case.

(ii) *Decomposition of the set of traces into behaviourally (relatively) independent sub processes*: Mining process models from such smaller subsets of traces is more efficient than mining the whole log. The composition of the mined sub process models is an open task.

(iii) *Decomposition of the activities to distributed components of the process model*: This can easily be integrated to our algorithm by removing possible dependencies between activities from the inequation systems.

The second and third cases introduce modularity to the mining algorithm. They simplify the resulting process model by implementing a clear and concise structure. The modularity also improves the performance, since dealing with smaller problems is less costly. But not only behavioural information, also *additional state information* can be implemented to the process model. For example an information, that the state of the process after two different sequences of activities coincides, can easily be captured by adaption of the inequation systems.
Another possibility to account for special behavioural or structural information of the logged process, is to predefine certain components of the process model. In our case this means *predefining places*. Such places are not introduced to the mined net by the actual mining algorithm, but they are constructed beforehand. Examples of reasonable predefined places are initial or final places for processes, places defining known dependencies, resources or routings in the process and places given by some a-priori process model. Such an a-priori process model does not necessarily be defined as a Petri net, because most process modelling languages can be translated to Petri nets. It is possible to assume that certain places are correct. Alternatively, the mining algorithm can detect incorrect parts of an a-priori process model through a comparison to the mining result or to the traces in the event log. The mining algorithm can be used to improve the a-priori model, by *adding correct places* and by either *deleting or modifying incorrect places*. An a-priori model may be more abstract than the behaviour of system observed by the log. Then activities of the a-priori model need to be refined and the mining algorithm can use causal dependencies known from the a-priori model. Our algorithm can very well integrate predefined places and causal dependencies. The algorithm even benefits from having reasonable predefined places, because they can directly be integrated and exploited in the construction procedure of the Petri net.
There are also general procedures to *generate a more simple and concise net*. There are heuristical methods to intentionally introduce a simplified structure to the net. That means one tries to *avoid complicated routings*, to *introduce concurrency between activities* and to *ignore less significant places*. In particular highly branched places are filtered out (not allowed), simplified or decomposed. The overall number of places can be reduced by *ignoring certain wrong continuations* of some of the traces of the log (e.g. wrong continuations which are not specially defined as undesired behaviour). Some *information from the preprocessing phase also lead to simplified nets*, such as information on cycles of events, on independent or sparsely connected components or sub-processes, on split activities or on hidden

routing activities. Such information can mostly be integrated to our algorithm by adapting the considered inequation systems. A concrete example of a very promising general method to simplify the mined net is to only account for dependencies of a certain depth. That means, if one event occurs before another event in some trace, a dependency of the activities is only assumed, if there is at most a certain number of events in between the two events. If there are more events in between the activities, the activities are not allowed to be directly connected by a place. Such restriction of the dependency to a certain depth can easily be introduced to our mining algorithm, if we consider the variant considering flows of tokens [BDLM07, LBMD07]. Then we allow flows of tokens only of a certain depth. This maximal depth can also be varied, e.g. accounting for deeper dependencies in specially significant traces. This method is a very flexible method to abstract from certain dependencies (the α-algorithm [ADH+03] only considers dependencies of depth one). All these simplification methods in general lead to overapproximation. A restriction of the overapproximation can be introduced by a certain upper bound for the overapproximation defining some tolerance range. Such bound can be given by a maximal set or number of traces of the mined net or by a minimum value of some conformance measure, modelling the conformance of the process model and the log. To *control the degree of simplification*, stochastic methods based on frequencies or ratings (described in the pre-processing passage) are useful. An extensive control can be achieved by introducing *cost-functions*. Typical cost functions introduce *structural costs* for places, branching of places, arc weights, arc connections between certain distributed activities, arc connections to infrequent or low rated activities, depth of token flows respectively direct dependencies in the traces etc. and *behavioural costs* attached to certain traces or patterns not present in the log, explicitly unwanted behaviour, rare or low rated traces, etc. The structural costs ensure a concise model and the direction of overapproximation, while the behavioural costs restrict and control the overapproximation. Instead of introducing behavioural costs, it is also possible to fix an upper bound for the overapproximation. Also *fixed structural bounds* for the mined net, e.g. restricting the number of places, the maximal branching or the maximal arc weight, may be reasonable to guarantee a simple process model. Such costs and bounds can very well be integrated to our mining algorithm, because the algorithm introduces places step by step solving certain inequation systems. These steps may then turn into optimization problems or games with equilibria.

There are precise methods improving the compactness of the mined net and the performance of the mining algorithm. These methods are based on special characteristics of the mining algorithm. For a deeper insight to these methods we refer to [BDLM07]. Some of the methods are already explained in [BDLM07] in a more formal way. Remember, that places are constructed by solving certain inequation systems. The *choice of a solver* for these inequation systems is an important parameter of the algorithm. As described in [BDLM07] possible solvers are the Simplex algorithm, the Khachyan method or the method of Karmarkar and variants of these algorithms. The evolution of the inequation systems allows for example the application of an iterative Simplex variant, which is more efficient than the standard Simplex algorithm. The solvers differ in average as well as worst case runtime and may yield different solution places. The Simplex algorithm allows to define a *linear objective function* to influence the computed solution place. Typical objective functions may aim at *minimizing certain structural properties* of the net, e.g. arc weights, branching of places or conflicts. The definition of reasonable objective functions benefits from frequencies or ratings. To control the computed solution places, it is also possible to calculate the complete solution space, or at least several solutions, of the inequation systems. Then according to ones needs, one can choose one of these places to be added to the constructed net. There are several possibilities to choose an adequate place, e.g. (possibly non-linear) objective functions, places prohibiting many wrong continuations at once (thus reducing the overall number of constructed places), places respecting distributed components of the resulting net, etc. Beside the solver, the *order of the wrong continuations* is an important parameter. This order influences which inequation systems actually have to be solved. Therefore the set of constructed places, in particular the overall number of places, depends on this order. There are several possibilities to develop an order leading to a preferably concise set of places of the mined net. For example one may start with inequation systems associated with highly significant traces or one may solve several inequation systems in parallel and decide, which of the solutions should be integrated to the net first. As mentioned in the pre-processing passage, it is also possible, that in addition to the set of logged traces, there may be a *set of unwanted traces*. If simplifications are applied in the mining procedure, it may be necessary to explicitly exclude these traces in the actual mining algorithm. This can easily be done by regarding inequation systems excluding these traces, i.e. places prohibiting the unwanted traces are explicitly computed. Lastly it was mentioned in [BDLM07], that sometimes a direct wrong continuation of a trace of the log cannot be separated, but a follower wrong continuation can. *Prohibiting such follower wrong continuations (up to a certain depth)* for specially significant traces may improve the mining quality in some cases.

2.3 Post-processing

During the post-processing phase, the mined Petri net can be fine-tuned and a graphical representation can be built. The methods in this phase mostly focus on a final adaption of the mined net to the needs of the user. The most important challenge in practice is, that the model is readable and interpretable by the user. This challenging task requires well developed post-processing techniques, because, despite of several simplification methods in the pre-processing phase and the mining algorithm phase, the pure mined model can often still be quite complex.

A first possibility to a-posteriori simplify the mined model are *precise reduction techniques* for Petri nets. Such

techniques reduce the components of the net, while preserving its behaviour. Reduction techniques for Petri nets are well established. The most apparent method in our setting is identifying and deleting *redundant places*, also called implicit places. Basically it is possible to detect a minimal sized subset of the set of places of the mined model guaranteeing the same behaviour as the set of all places. Since detecting such set is very costly, more simple approaches to delete some redundant places have to be used. Many such approaches exist; some examples are explained in [BDLM07]. Besides deleting redundant places there are also more intricate reduction techniques, e.g. concerning symmetries, transitions, etc. Some of these more complex methods can also be useful for post-processing the mined net.

Furthermore there are non-precise simplification techniques. These are either pure heuristical methods or the methods exploit certain additional information. Such methods mostly aim for systematically *deleting or simplifying places*. Simplifying places means that connections to certain transitions are deleted to reduce branching or arc weights are reduced. Sometimes this requires decomposition of one place to several places, but sometimes places can also be combined. The methods are not only targeted on improving the readability of the mined net, but also on generating a clear structure of the net, e.g. divide the net into distributed components. Further *structural simplification techniques combine transitions*, *split transitions* (label splitting) or introduce *routing transitions*. Additionally, also *methods to simplify the behaviour* of the net are reasonable, e.g. *introducing additional concurrency, cycles or conflicts*. To guarantee that the model is not biased too much by such methods, the methods may be controlled by some conformance measure ensuring a certain degree of consistency between the model and the log.

For complicated, especially unstructured processes, the pure mined model may still be "spaghetti-like". The real process itself is often "spaghetti-like" and, despite of several simplification methods, an accurate process model reflecting such process is often hardly readable. Therefore *visualization techniques* for the mined models are a very important part of the post-processing phase. Examples of adequate basic visualization techniques are summarized in [AG07].

Finally there are also some post-processing methods solely trying to a-posteriori *improve the quality of the mined model*. In contrast to the previous methods, these methods are not tackling the readability of the model. Firstly, methods a-posteriori *checking correctness properties* of the mined net are of interest. These methods benefit from having a formal model, i.e. Petri nets, as the target language. There are many verification, validation and analysis techniques for Petri nets, that can be applied. Having detected certain inconsistencies of the model, it can be tried to fix them. Secondly, it is also possible to improve the quality of the mined model by a-posteriori editing the model to improve some *conformance measure*.

3 Conclusion

We proposed methods for our process mining approach to improve performance, to simplify the mined net and to overapproximate in a targeted direction. We moreover presented strategies to control the level of overapproximation and simplification. All these methods finally result in an appropriate adaption of the linear equation systems which are solved to compute places. Some methods need the application of an advanced region based synthesis method using token flows. The next steps are to implement these methods into VipTool and evaluate them using practical examples.

References

[ADH+03] Aalst, W. M. P. d. ; Dongen, B. F. ; Herbst, J. ; Maruster, L. ; Schimm, G. ; Weijters, A. J. M. M.: Workflow mining: A survey of issues and approaches. In: *Data Knowl. Eng.* 47 (2003), Nr. 2, S. 237–267

[AG07] Aalst, W. M. P. d. ; Günther, C. W.: Finding Structure in Unstructured Processes: The Case for Process Mining. In: *Proceedings of the 7th International Conference on Application of Concurrency to System Design (ACSD)*, 2007, S. 3–12

[BDLM07] Bergenthum, R. ; Desel, J. ; Lorenz, R. ; Mauser, S.: Process Mining Based on Regions of Languages. In: *Proceedings of the 5th International Conference on Business Process Management (BPM)*, 2007

[LBMD07] Lorenz, R. ; Bergenthum, R. ; Mauser, S. ; Desel, J.: Synthesis of Petri Nets from Finite Partial Languages. In: *Proceedings of the 7th International Conference on Application of Concurrency to System Design (ACSD)*, 2007, S. 157–166

[PE] ProcessMiningGroup ; EindhovenTechnicalUniversity: *ProM-Homepage*. – http://is.tm.tue.nl/ cgunther/dev/prom/

Strukturelle Analyse zyklenfreier Workflownetze

Jan Ortmann
Universität Hamburg, Department Informatik
Vogt-Kölln-Str. 30, D-22527 Hamburg
ortmann@informatik.uni-hamburg.de

***Abstract*— Es werden eine Reihe von Kritierien für die Korrektheit (Soundness) von S-überdeckbaren zyklenfreien WF-Netzen gegeben mit dem Ziel ein Workflownetz zu partionieren und dann jeden Teil des Netzes getrennt analysieren zu können.**

Keywords: Workflownetze, Analyse, Petrinetze

I. Einführung

Zur Modellierung und Analyse von Workflows werden Workflownetze (WF-Netze) heute vielfältig eingesetzt. Es fehlt bislang jedoch ein Editor, der es erlaubt, während der Eingabe direkt diejenigen Teile eines Modells ausfindig zu machen, die zu einem nicht-korrekten Workflow führen können. An dieser Stelle sollen S-überdeckbare Workflownetze betrachtet werden und in einer ersten Annäherung untersucht werden, welche Teile sich identifizieren lassen, die sich gesondert analysieren lassen. Auch wenn es sounde WF-Netze gibt, die nicht S-überdeckbar sind, so bilden S-überdeckbare WF-Netze doch eine gute Grundlage, da nicht S-überdeckbare sounde WF-Netze meist schwer zu verstehen sind und daher eher theoretischer Natur sind.

Weniger ausdrucksstark ist die Teilklasse der wohlstrukturierten WF-Netze. Für diese lassen sich jedoch leichter die gewünschten Ergebnisse zeigen, die dann unter bestimmten Bedingungen auf S-überdeckbare WF-Netze übertragbar sind. An dieser Stelle werden ausschließlich zyklenfreie S-überdeckbare WF-Netze behandelt, die Analyse von WF-Netzen mit Zyklen ist dann Gegenstand weiterer Arbeiten. Hintergrund ist hierbei die Möglichkeit, einen komplexen Workflow anhand der Handle in feinere Strukturen aufzuteilen, die sich leichter analysieren lassen, so dass bereits zur Designzeit überprüft werden kann, ob ein Workflow sound ist. Ausgenutzt werden kann hierbei, dass Änderungen in der Regel lediglich lokal stattfinden. Wird eine dieser Komponenten geändert, muss lediglich diese Komponente neu untersucht werden. Betrachtet werden im weiteren Verlauf lediglich endlich WF-Netze.

II. Notation und Begriffe

Definition 1 (Petrinetz)
Ein Petrinetz $\mathcal{N}$ ist ein Tripel $\mathcal{N} = (P, T, F)$, wobei

- P eine endliche Menge *Stellen* (auch Plätzen),
- T eine endliche Menge von *Transitionen* mit $P \cap T = \emptyset$ und
- $F \subseteq (P \times T) \cup (T \times P)$ eine *Flussrelation* darstellt.

Jedes $x \in P \cup T$ wird auch *Knoten* genannt. ♦

Definition 2 (Markierung, Aktiviertheit, Schalten, Folgemarkierung)
Eine *Markierung* $\mathbf{m}$ eines Petrinetzes $\mathcal{N}$ ist eine Abbildung $\mathbf{m} : P \to \mathbb{N}$, die jeder Stelle eine Anzahl von Marken zuordnet. Eine Stelle ist markiert gdw. $\mathbf{m}(p) > 0$. Es sei $[q]$ die Markierung, die nur genau der Stelle q eine Marke zuweist, d.h. $\forall p \in (P \setminus \{q\}).[q](p) = 0$ und $[q](q) = 1$. Eine Transition $\mathfrak{t} \in T$ ist *aktiviert* in einer Markierung $\mathbf{m}$ gdw. $\forall p \in {}^\bullet\mathfrak{t} : \mathbf{m}(p) \geq 1$. Eine aktivierte Markierung $\mathbf{m}$ kann *schalten*, was als $\mathbf{m} \xrightarrow{\mathfrak{t}} \mathbf{m}'$ notiert wird. In diesem Fall ist die Nachfolgemarkierung durch $\mathbf{m}'(p) \stackrel{\text{def}}{=} \mathbf{m}(p) - |F(p,\mathfrak{t})| + |F(\mathfrak{t},p)|$ definiert. ♦

Definition 3 (Schaltfolgen, erreichbare Markierungen)
Sei $\mathbf{m}$ eine Markierung des Petrinetzes $\mathcal{N}$ und sei $\mathbf{m} \xrightarrow{\mathfrak{t}_1} \mathbf{m}_1 \xrightarrow{\mathfrak{t}_2} \ldots \xrightarrow{\mathfrak{t}_n} \mathbf{m}_n$ das aufeinanderfolgende Schalten der Transitionen $\mathfrak{t}_1$ bis $\mathfrak{t}_n$, so wird $\sigma = \mathfrak{t}_1\mathfrak{t}_2 \ldots t_n$ eine Schaltfolge von $\mathbf{m}$ nach $\mathbf{m}_n$ genannt und als $M \xrightarrow{\sigma} M_n$ notiert. $[\mathbf{m}\rangle \stackrel{\text{def}}{=} \{\mathbf{m}' | \exists \sigma \in T^*.\mathbf{m} \xrightarrow{\sigma} \mathbf{m}'\}$ definiert die Menge aller von $\mathbf{m}$ aus erreichbaren Markierungen. ♦

Definition 4 (Workflownetz)
Ein Petrinetz $\mathcal{N} = (P, T, F)$ ist ein Workflownetz (WF-Netz) gdw.

1) es eine Anfangsstelle (engl. source place) $i \in P$ gibt, so dass ${}^\bullet i = \emptyset$,
2) es eine Senke (engl. sink) $o \in P$ gibt, so dass $o^\bullet = \emptyset$ und
3) jeder Knoten $x \in P \cup T$ auf einem Pfad von i nach o liegt.

♦

Definition 5 (Pfad, elementar, konfliktfrei)
Sei $\mathcal{N} = (P, T, F)$ ein Petrinetz. Eine nicht-leere Folge von Knoten $s = (x_0 \ldots x_k),\ k \in \mathbb{N}, x_i \in P \cup T$ wird (gerichteter) *Pfad* von x_0 nach x_k genannt gdw. $\forall i \in \{0, \ldots, k-1\} : (x_i, x_{i+1}) \in F$ gilt. Ein Pfad $(x_0 \ldots x_k)$ wird als *elementar* bezeichnet gdw. $\forall i, j \in \{0, \ldots, k\} : x_i = x_j \Rightarrow i = j$. Die Menge aller gerichteten Pfade von x_0 nach x_k wird als $X_d(x_0, x_k)$ notiert. Die Menge aller gerichteten elementaren Pfade von x_0 nach x_k wird als $E_d(x_0, x_k)$ notiert. ♦

Für eine Folge $s = (x_0 x_1 \ldots x_k)$ sei $\textsc{Alph}(s) \stackrel{\text{def}}{=} \{x_0, x_1, \ldots, x_k\}$ die Menge aller Elemente von s. In zyklen-

freien Petrinetzen ist jeder Pfad elementar, so das wir diesen Zusatz weglassen werden.

Definition 6 (Zyklus, Zusammenhängende Netze)
Ein elementarer Pfad von x_0 nach x_k wird als *Zyklus* bezeichnet gdw. $(x_k, x_0) \in F$. Ein Netz ist *zyklenfrei* gdw. es keine Zyklen enthält.
Ein Petrinetz $\mathcal{N} = (P, T, F)$ wird als *(schwach) zusammenhängend* bezeichnet gdw. $(x, y) \in (F \cup F^{-1})^*$ für je zwei Knoten $x, y \in (P \cup T)$ gilt. Ein Netz wird als *stark zusammenhängend* bezeichnet gdw. $(x, y) \in F^*$ für je zwei Knoten $x, y \in (P \cup T)$ gilt, d.h. es gibt für je zwei Knoten x und y einen Pfad von x nach y. ♦

Definition 7 (Tote, Lebendigkeit, Beschränktheit)
Sei $\mathcal{NS} = (\mathcal{N}, \mathbf{m}_0)$ ein Netzsystem. Eine Transition $\mathfrak{t}$ ist

- *tot* in der Markierung $\mathbf{m}$ gdw. $\neg\exists \mathbf{m}' \in [\mathbf{m}\rangle : \mathbf{m}' \xrightarrow{\mathfrak{t}}$,
- *lebendig* gdw. $\forall \mathbf{m} \in [\mathbf{m}_0\rangle \, \exists \mathbf{m}' : \mathbf{m} \xrightarrow{*} \mathbf{m}' \xrightarrow{\mathfrak{t}}$, d.h. für jede erreichbare Markierung $\mathbf{m}$ gibt es eine von $\mathbf{m}$ erreichbare Markierung $\mathbf{m}'$, die $\mathfrak{t}$ aktiviert.

Ein Netzsystem $\mathcal{NS} = (\mathcal{N}, \mathbf{m}_0)$ ist *lebendig* gdw. alle Transitionen $\mathfrak{t} \in T$ lebendig sind. Ein Netzsystem $\mathcal{NS} = (\mathcal{N}, \mathbf{m}_0)$ ist beschränkt gdw. $\forall p \in P \, \exists k \in \mathbb{N} \, \forall \mathbf{m} \in [\mathbf{m}_0\rangle : \mathbf{m}(p) < k$.Ein Netzsystem ist *sicher* gdw. $\forall p \in P \, \forall \mathbf{m} \in [\mathbf{m}_0\rangle : \mathbf{m}(p) < 1$. ♦

Definition 8 (Wohlgeformtheit eines Netzes)
Ein Petrinetz $\mathcal{N}$ ist wohlgeformt gdw. es eine Markierung $\mathbf{m}_0$ gibt, so dass $\mathcal{N}$ unter der Anfangsmarkierung $\mathbf{m}_0$ beschränkt und lebendig ist. ♦

Definition 9 (Rand, transitions- und stellenberandet)
Sei $\mathcal{N} = (P, T, F)$ ein Netz und $X \subseteq P \cup T$ eine Menge von Knoten. Die Menge $\text{BORDER}(X) \stackrel{\text{def}}{=} \{x \in X \mid \exists y \in (P \cup T) : y \notin X \wedge y \in ({}^\bullet x \cup x^\bullet)\}$ wird *Rand* von X genannt. X wird *transitionsberandet* genannt, wenn $\text{BORDER}(X) \subseteq T$ und *stellenberandet* genannt, wenn $\text{BORDER}(X) \subseteq P$. ♦

Definition 10 (Statemachine, S-Komponente, S-überdeckbar)
Gegeben ein Petrinetz $\mathcal{N} = (P, T, F)$. $\mathcal{N}$ besitzt die *Statemachine*-Eigenschaft, wenn für alle $\mathfrak{t} \in T$ gilt $|{}^\bullet\mathfrak{t}| = |\mathfrak{t}^\bullet| = 1$.
Sei $\mathcal{N}' = (P', T', F')$ ein Subnetz von $\mathcal{N}$, so dass $(P \cup T) \neq \emptyset$. $\mathcal{N}'$ ist eine *S-Komponente*, wenn $\mathcal{N}'$ stark zusammenhängend ist, die Statemachine-Eigenschaft besitzt und wenn für alle $p \in P'$ und alle $\mathfrak{t} \in T$ gilt $(p, \mathfrak{t}) \in F \implies (p, \mathfrak{t}) \in F'$ und $(\mathfrak{t}, p) \in F \implies (\mathfrak{t}, p) \in F'$. Ein Netz $\mathcal{N}$ ist S-überdeckbar, wenn es eine Menge von S-Komponenten $\mathcal{N}_j^S, j \in \mathbb{N}$ gibt, so dass jede Stelle in P Element einer S-Komponente ist. ♦

Definition 11 (Marked Graph, T-Komponente, T-überdeckbar)
Gegeben ein Petrinetz $\mathcal{N} = (P, T, F)$. $\mathcal{N}$ ist ein *Marked Graph*, wenn für alle $p \in P$ gilt $|{}^\bullet p| = |p^\bullet| = 1$.
Sei $\mathcal{N}' = (P', T', F')$ ein Subnetz von $\mathcal{N}$, so dass $(P \cup T) \neq \emptyset$. $\mathcal{N}'$ ist eine *T-Komponente*, wenn $\mathcal{N}'$ stark zusammenhängend ist, $\mathcal{N}'$ ein Marked Graph ist und wenn für alle $p \in P$ und alle $\mathfrak{t} \in T'$ gilt $(p, \mathfrak{t}) \in F \implies (p, \mathfrak{t}) \in F'$ und $(\mathfrak{t}, p) \in F \implies (\mathfrak{t}, p) \in F'$. Ein Netz $\mathcal{N}$ ist T-überdeckbar, wenn es eine Menge von T-Komponenten $\mathcal{N}_j^T, j \in \mathbb{N}$ gibt, so dass jede Transition in T Element einer T-Komponente ist. ♦

Definition 12 (Siphon, Trap)
Eine Menge von Stellen $S \subseteq P$ eines Petrinetzes $\mathcal{N} = (P, T, F)$ ist

- ein *Siphon* gdw. ${}^\bullet S \subseteq S^\bullet$. Ein Siphon wird als ordentlich (engl. proper) bezeichnet, wenn er nicht die leere Menge ist. Ein Siphon S ist minimal, wenn kein anderer Siphon S' mit $S' \subset S$ existiert.
- eine *Trap* (engl. trap) gdw. $S^\bullet \subseteq {}^\bullet S$. Eine Trap wird als ordentlich (engl. proper) bezeichnet, wenn sie nicht die leere Menge ist. Eine Trap S ist minimal, wenn keine andere Trap S' mit $S' \subset S$ existiert.

♦

Ein Siphon bleibt unmarkiert, wenn er einmal unmarkiert ist, eine Trap bleibt markiert, wenn sie einmal markiert ist.

Satz 13 (Lebendige Systeme besitzen kein unmarkiertes Sipon)
Jedes proper Siphon eines lebendigen Systems $(\mathcal{N}, \mathbf{m}_0)$ ist initial markiert.

Beweis:
Der Beweis findet sich in [1] als Beweis zu Proposition 4.10. □

Definition 14 (Soundness von Workflownetzen)
Ein Workflownetz (WF-Netz) $\mathcal{N} = (P, T, F)$ heißt sound gdw.

1) $\forall \mathbf{m} \in [[i]\rangle : \mathbf{m} \xrightarrow{*} [o])$,
2) $\forall \mathbf{m} \in [[i]\rangle : \mathbf{m} \geq [o] \Rightarrow (\mathbf{m} = [o])$,
3) $\forall \mathfrak{t} \in T \, \exists \mathbf{m}, \mathbf{m}' : [i] \xrightarrow{*} \mathbf{m} \xrightarrow{\mathfrak{t}} \mathbf{m}'$.

♦

Definition 15 (Abschluss eines Workflownetzes)
Der Abschluss eines WF-Netzes $\mathcal{N}$ ist definiert als Netz $\overline{\mathcal{N}} \stackrel{\text{def}}{=} (\overline{P}, \overline{T}, \overline{F})$, welches durch eine zusätzliche Transition $\mathfrak{t}^*$ entsteht, die o mit i verbindet, d.h. $\overline{\mathcal{N}} = (P, T \cup \{\mathfrak{t}^*\}, F \cup \{(o, \mathfrak{t}^*), (\mathfrak{t}^*, i)\})$. ♦

Satz 16 (Soundness, Lebendigkeit und Beschränktheit)
Ein Workflownetz $\mathcal{N}$ ist sound gdw. $(\overline{\mathcal{N}}, [i])$ lebendig und beschränkt ist.

Beweis:
Der Beweis findet sich in [5]. □

Lemma 17 (S-Überdeckbarkeit impliziert Sicherheit)
Sei $\mathcal{N}$ ein WF-Netz, so dass der Abschluss $\overline{\mathcal{N}}$ S-überdeckbar ist. Dann ist $\overline{\mathcal{N}}$ unter der Initialmarkierung $[i]$ sicher.

Beweis:
Der Beweis findet sich in [7], Beweis zu Theorem 4.3. □

Definition 18 (Handle, well-handled, wohlstrukturiert)
In einem Petrinetz $\mathcal{N} = (P, T, F)$ wird ein Knoten-Paar $(n_1, n_2) \in (P \cup T) \times (P \cup T)$ *Handle* genannt gdw. es zwei elementare Pfade von n_1 nach n_2 gibt, die lediglich die beiden Knoten n_1 und n_2 gemeinsam haben, d.h. $\exists s_0, s_1 \in E_d(n_1, n_2) : s_0 \neq s_1 \wedge \text{ALPH}(s_0) \cap \text{ALPH}(s_1) = \{n_1, n_2\}$. Für ein PP-Handle gilt $(n_1, n_2) \in P \times P$, für ein PT-Handle gilt $(n_1, n_2) \in P \times T$. Analog sind TT- und TP-Handle definiert. Bei einem Handle (n_1, n_2) bezeichnet n_1 den öffnenden Knoten und n_2 den schließenden Knoten. Die Knoten n_1 und n_2 müssen hierbei nicht verschieden sein. Ein Petrinetz ist *well-handled* gdw. es keine PT-handle und keine TP-handle besitzt. Ein WF-Netz $\mathcal{N}$ ist *wohlstrukturiert*dw. sein Abschluss $\overline{\mathcal{N}}$ well-handled ist. ♦

Lemma 19 (Korrekte wohlstrukturierte WF-Netze sind sicher)
Ein soundes wohlstrukturiertes WF-Netz ist (unter der Markierung $[i]$) sicher.

Beweis:
Der Beweis findet sich in [6, Beweis zu Lemma 3]. □

Lemma 20 (Well-handled und wohlgeformt)
Ein stark zusammenhängendes Netz, das well-handled ist, ist auch wohlgeformt.

Beweis:
Der Beweis findet sich in [2, Beweis zu Lemma 2]. □

III. Überdeckbarkeit zyklenfreier WF-Netze

Lemma 21 (T-Überdeckbarkeit impliziert Lebendigkeit)
Sei $\mathcal{N}$ ein zyklenfreies WF-Netz. Wenn der Abschluss $\overline{\mathcal{N}}$ T-überdeckbar ist, ist $\overline{\mathcal{N}}$ unter der Initialmarkierung $[i]$ lebendig.

Beweis:
Da $\overline{\mathcal{N}}$ lediglich einen Zyklus enthält, muss jede T-Komponente $\mathcal{N}_j^T = (P_j, T_j, F_j)$ ($j \in \mathbb{N}$) die Stelle i beinhalten. Nach Theorem 3.15 in [1] ist somit jede T-Komponente lebendig. Wir können nun über die Zahl der T-Komponenten k zeigen, dass $\overline{\mathcal{N}}$ lebendig ist. Für $k = 1$ ist die Aussage trivial. Sei die Aussage also für $k = n$ bewiesen und sei $k = n + 1$. Somit gibt es ein lebendiges Netz $\mathcal{N}' = (P', T', F')$ aus n T-Komponenten, das mit einer weiteren T-Komponente $\mathcal{N}_k^T = (P_k, T_k, F_k)$ zu einem WF-Netz $\mathcal{N}''$ verschmolzen wird (also $\mathcal{N}'' = (P' \cup P_k, T' \cup T_k, F' \cup F_k)$). Ist $T_k \subseteq T'$, ist auch $P_k \subseteq P'$ und $F_k \subseteq F'$ und die Aussage folgt sofort. Betrachten wir also eine T-Komponente, für die es eine Transition $\mathfrak{t} \in (T_k \setminus T')$ gibt. Wäre $\overline{\mathcal{N}}$ nicht lebendig, so müsste es durch die Verschmelzung einen nicht markierten Siphon geben können und somit ein PT-Handle $(p', \mathfrak{t}')$. Dann wäre $\mathfrak{t}'$ jedoch sowohl in T' wie auch in T_k vorhanden gewesen. Da dies für alle potentiellen PT-Handle gilt, gibt es einen Widerspruch und die Aussage ist belegt. □

Satz 22 (Überdeckbarkeit und Soundness)
Ein beschränktes zyklenfreies WF-Netz $\mathcal{N} = (P, T, F)$ ist sound gdw. $\overline{\mathcal{N}}$ T-überdeckbar ist.

Beweis:
($\Rightarrow$) Da $\mathcal{N}$ sound ist gibt es eine Schaltfolge σ, so dass $[i] \xrightarrow{\sigma} [i]$ in $\overline{\mathcal{N}}$ gilt. Dann ist der Parikh-Vector (der Vektor, der zu jeder Transition $\mathfrak{t}$ der Schaltfolge angibt, wie häufig $\mathfrak{t}$ schaltete) nach Proposition 2.37 in [1] eine T-Invariante. Sei $T_T = \text{ALPH}(\sigma)$ und sei $\mathcal{N}^{TC} = (P_T, T_T, F_T)$ die aus T_T resultierende T-Komponente, so dass P_T alle Stellen aus dem Vor- und Nachbereich von P_T umfasst. Wäre $\mathcal{N}^{TC}$ keine T-Komponente gäbe es zwei Pfade von einer Stelle zu einer Transition oder von einer Transition zu einer Stelle in $\mathcal{N}^{TC}$. In ersterem Fall wäre σ dann jedoch keine Schaltfolge, in letzterem Fall wäre $\mathcal{N}^{TC}$ nicht beschränkt. Da es für jede Transition $\mathfrak{t} \in T$ eine solche Schaltfolge σ gibt, existiert folglich für alle Transitionen eine T-Komponente und $\overline{\mathcal{N}}$ ist T-überdeckbar.
($\Leftarrow$) $\mathcal{N}$ ist T-überdeckbar und nach Lemma 21 somit auch lebendig. Da $\mathcal{N}$ sicher ist, ist es sound. □

IV. Wohlstrukturierte WF-Netze

Lemma 23 (Existenz von TT-Handles)
Sei $\mathcal{N}$ ein wohlstrukturiertes WF-Netz und $\overline{\mathcal{N}}$ der Abschluss von $\mathcal{N}$. Zu jeder Transition $\mathfrak{t} \in T$ mit $|\mathfrak{t}^\bullet| > 1$ existiert eine Transition $\mathfrak{t}' \in T$ mit $|{}^\bullet\mathfrak{t}| > 1$, so dass $(\mathfrak{t}, \mathfrak{t}')$ ein TT-Handle in $\overline{\mathcal{N}}$ bildet. Bilden $\mathfrak{t}$ und $\mathfrak{t}'$ keinen Zyklus, so ist das TT-Handle in $\mathcal{N}$.

Beweis:
Sei $\mathfrak{t} \in T$ mit $|\mathfrak{t}^\bullet| > 1$ eine Transition zu der keine Transition $\mathfrak{t}' \in T$ mit $|{}^\bullet\mathfrak{t}| > 1$ existiert. Da $\mathcal{N}$ ein WF-Netz ist, liegen alle Knoten und somit auch die Knoten aus $\mathfrak{t}^\bullet$ auf einem Pfad von i nach o. Da $|\mathfrak{t}^\bullet| \geq 2$ ist, gibt es für $p, p' \in \mathfrak{t}^\bullet$ mit $p \neq p'$ zwei elementare Pfade $s = (\mathfrak{t}\ p\ x_0 \ldots x_k\, o)$ und $s' = (\mathfrak{t}\ p'\ y_0 \ldots y_{k'}\, o)$ von $\mathfrak{t}$ nach o mit $s \neq s'$. Ist nun für einen der beiden Pfade $\mathfrak{t} = x_j$ oder $\mathfrak{t} = y_j$, so existiert ein Zyklus. Seien die beiden Pfade zyklenfrei. Da $p \neq p'$ ist, gibt es einen Teilpfad $s_\# = (\mathfrak{t}\ p\ x_0 \ldots x_i\ x), i < k$ und einen Teilpfad $s'_\# = (\mathfrak{t}\ p'\ y_0 \ldots y_{i'}\, x), i' < k'$, so dass $x \neq \mathfrak{t}$ und $\text{ALPH}(s_\#) \cap \text{ALPH}(s'_\#) = \{\mathfrak{t}, x\}$. Wäre $x \in P$ wäre $\mathcal{N}$ nicht wohlstrukturiert, somit gibt es ein TT-Handle $(\mathfrak{t}, x)$ in $\mathcal{N}$.

Gilt hingegen o.B.d.A. $\mathfrak{t} = x_j$ innerhalb s, so gibt es einen elementaren Pfad $(i \ldots x_l)$ mit $p \notin \text{ALPH}(i \ldots x_l)$ für ein x_l mit $l \leq j$ und $|{}^\bullet x_l| \geq 2$, da der Zyklus $(\mathfrak{t}\ p\ x_0 \ldots x_{j-1})$ andernfalls nicht von i erreichbar wäre (im Widerspruch zum strengen Zusammenhang). Folglich existiert im Abschluss ein elementarer Pfad $s_\# = (\mathfrak{t} \ldots o\ \mathfrak{t}^*\ i \ldots x_l)$ und es gibt zwei

elementare Pfade von $\mathfrak{t}$ nach x_l: $s_\#$ und $(t\ p\ x_0 \dots x_l)$. Ist x_l keine Transition, so ist das Netz $\mathcal{N}$ nicht wohlstrukturiert. □

Lemma 24 (Existenz von PP-Handles)
Sei $\mathcal{N}$ ein wohlstrukturiertes WF-Netz und $\overline{\mathcal{N}}$ der Abschluss von $\mathcal{N}$. Zu jeder Stelle $p \in P$ mit $|p^\bullet| > 1$ existiert eine Stelle $p' \in P$ mit $|{}^\bullet p| > 1$, so dass (p, p') ein PP-Handle in $\overline{\mathcal{N}}$ bildet. Bilden p und p' keinen Zyklus, so ist das PP-Handle in $\mathcal{N}$.

Beweis:
Der Beweis ist analog zu Lemma 23. □

Definition 25 (Handle-Subnetz)
Gegeben ein WF-Netz $\mathcal{N} = (P, T, F)$. Zu einem Handle (n_s, n_f) in $\mathcal{N}$ ist das von (n_s, n_f) erzeugte *Handle-Subnetz* von $\mathcal{N}$ das durch alle zwischen (n_s, n_f) existierenden Knoten beschriebene Netz $\mathcal{N}^H_{(n_s,n_f)} \stackrel{\text{def}}{=} (P_H, T_H, F_H)$ mit $P_H \stackrel{\text{def}}{=} P \cap X$, $T_H \stackrel{\text{def}}{=} T \cap X$ und $F_H \stackrel{\text{def}}{=} F \cap X \times X$, wobei $X \stackrel{\text{def}}{=} \bigcup_{s \in X_d(n_s,n_f)} \text{ALPH}(s)$. Hierbei stellt $X_d(n_s, n_f)$ die Menge aller gerichteten Pfade von n_s nach n_f dar. ♦

Lemma 26 (TT-Handle-Subnetze wohlstrukturierter WF-Netze sind transistionsberandet)
Sei $\mathcal{N}$ ein wohlstrukturiertes WF-Netz, $(\mathfrak{t}_s, \mathfrak{t}_f) \in T \times T$ ein beliebiges TT-Handle von $\mathcal{N}$ und $\mathcal{N}^{TT}_{(\mathfrak{t}_s,\mathfrak{t}_f)} = (P_T, T_T, F_T)$ das entsprechende TT-Handle-Subnetz. Dann gilt

1) Für jede Stelle $p \in P_T$ gilt ${}^\bullet p \subseteq T_T$.
2) Für jede Stelle $p \in P_T$ gilt $p^\bullet \subseteq T_T$.
3) $\mathcal{N}^{TT}$ ist transitionsberandet.

Beweis:
(1) Angenommen es existierten ein $p \in P_T$ und ein $\mathfrak{t} \in ((T \setminus T_T) \cap {}^\bullet p)$. Da der Abschluss des WF-Netzes $\mathcal{N}$ stark zusammenhängend ist, gibt es einen Pfad von $\mathfrak{t}_f$ nach $\mathfrak{t}$ im Abschluss $\overline{\mathcal{N}}$. Wenn nun ein $x \in P_T \cup T_T$, $x \neq \mathfrak{t}_f$ Teil dieses Pfades wäre, wäre $\mathfrak{t} \in T_T$, da es mit $(\mathfrak{t}_s \dots x \dots \mathfrak{t}\ p \dots \mathfrak{t}_f)$ einen Pfad von $\mathfrak{t}_s$ nach $\mathfrak{t}_f$ gäbe, der $\mathfrak{t}$ enthält. Folglich gilt für den Pfad $s_1 = (\mathfrak{t}_f \dots \mathfrak{t})$, dass $\text{ALPH}(s_1) \cap (P_T \cup T_T) = \{\mathfrak{t}_f\}$. Da $p \in P_T$ gilt, liegt p auf einem Pfad von $\mathfrak{t}_s$ nach $\mathfrak{t}_f$ und es gilt somit $|{}^\bullet p| \geq 2$ und $(\mathfrak{t}_s \dots \mathfrak{t}_\#\ p) \in E_d(\mathfrak{t}_s, p)$ für ein $\mathfrak{t}_\# \in T_T$. Für $s_2 = (\mathfrak{t}_s \dots \mathfrak{t}_\#, p)$ ist $\text{ALPH}(s_2) \subseteq (P_T \cup T_T)$. Da $(\mathfrak{t}_s, \mathfrak{t}_f)$ ein TT-Handle ist, gibt es zudem (mindestens) zwei elementare Pfade von $\mathfrak{t}_s$ nach $\mathfrak{t}_f$, von denen einer p enthält und (mindestens) einer nicht. Wenn wir nun zunächst einen Pfad ohne p von $\mathfrak{t}_s$ nach $\mathfrak{t}_f$ wählen und dann den Pfad s_1, so haben wir einen Pfad s_3 von t_s nach p für den gilt $\text{ALPH}(s_3) \cap \text{ALPH}(s_2) = \{\mathfrak{t}_s, p\}$. Somit gibt es zwei elementare Pfade von $\mathfrak{t}_s$ nach p (einen über $\mathfrak{t}_\# \in T_T$ und einen über $\mathfrak{t}_f$ und über $\mathfrak{t}$), die nur $\mathfrak{t}_s$ und p gemeinsam haben.
(2) Der Beweis ist analog zu (1). Angenommen es gäbe ein $p \in P_T$ und ein $\mathfrak{t} \in ((T \setminus T_T) \cap p^\bullet)$. Wiederum gäbe es dann einen Pfad von $\mathfrak{t}$ nach $\mathfrak{t}_s$ im Abschluss $\overline{\mathcal{N}}$. Wenn nun ein $x \in P_T \cup T_T$, $x \neq \mathfrak{t}_s$ Teil dieses Pfades wäre, wäre $\mathfrak{t} \in T_T$, da es mit $(\mathfrak{t}_s \dots p\ \mathfrak{t} \dots x \dots \mathfrak{t}_f)$ einen Pfad von $\mathfrak{t}_s$ nach $\mathfrak{t}_f$ gäbe. Folglich gilt für den Pfad $s_1 = (\mathfrak{t} \dots \mathfrak{t}_s)$, dass $\text{ALPH}(s_1) \cap (P_T \cup T_T) = \{\mathfrak{t}_s\}$. Da $p \in P_T$ gilt, liegt p auf einem Pfad von $\mathfrak{t}_s$ nach $\mathfrak{t}_f$ und es gilt somit $|p^\bullet| \geq 2$ und $(p\ \mathfrak{t}_\# \dots\ \mathfrak{t}_f) \in E_d(p, \mathfrak{t}_f)$ für ein $\mathfrak{t}_\# \in T_T$. Für $s_2 = (p\ \mathfrak{t}_\# \dots \mathfrak{t}_f)$ ist $\text{ALPH}(s_2) \subseteq (P_T \cup T_T)$. Wiederum lässt sich ein Pfad von $\mathfrak{t}_s$ nach $\mathfrak{t}_f$ wählen, der p nicht enthält, da $(\mathfrak{t}_s, \mathfrak{t}_f)$ ein TT-Handle ist. Wählen wir nun zunächst s_1 und dann diesen Pfad so haben wir einen Pfad s_3 für den gilt $\text{ALPH}(s_2) \cap \text{ALPH}(s_3) = \{p, \mathfrak{t}_f\}$.
(3) Dies folgt direkt aus (1) und (2). □

Lemma 27 (PP-Handle-Subnetze wohlstrukturierter WF-Netze sind stellenberandet)
Sei $\mathcal{N}$ ein wohlstrukturiertes WF-Netz, $(p_s, p_f) \in P \times P$ ein beliebiges PP-Handle von $\mathcal{N}$ und $\mathcal{N}^{PP}_{(p_s,p_f)} = (P_P, T_P, F_P)$ das entsprechende PP-Handle-Subnetz. Dann ist $\mathcal{N}^{PP}$ stellenberandet.

Beweis:
Der Beweis ist analog zu dem von Lemma 26 und soll hier daher nicht wiederholt werden. □

Definition 28 (Ersetzung eines TT-Handle-Subnetzes)
Sei $\mathcal{N} = (P, T, F)$ ein WF-Netz, $(\mathfrak{t}_s, \mathfrak{t}_f) \in T \times T$ ein TT-Handle von $\mathcal{N}$ und $\mathcal{N}^{TT}_{(\mathfrak{t}_s,\mathfrak{t}_f)} = (P_T, T_T, F_T)$ das entsprechende TT-Handle-Subnetz. Die Ersetzung von $\mathcal{N}^{TT}_{(\mathfrak{t}_s,\mathfrak{t}_f)}$ durch eine Transition $\mathfrak{t}^+$ in $\mathcal{N}$ ist gegeben durch das Netz $\mathcal{N}^R \stackrel{\text{def}}{=} (P_R, T_R, F_R)$ mit $P_R = P \setminus P_T$, $T_R = (T \cup \{\mathfrak{t}^+\}) \setminus T_T$ und

$$F_R = F \cap ((P_R \times T_R) \cup (T_R \times P_R)) \cup \{(p, \mathfrak{t}^+) \in P \times \{\mathfrak{t}^+\} | (p, \mathfrak{t}) \in (F \cap ((P \setminus P_T) \times T_T))\} \cup \{(\mathfrak{t}^+, p) \in \{\mathfrak{t}^+\} \times P | (\mathfrak{t}, p) \in (F \cap (T_T \times (P \setminus P_T)))\}$$

♦

Definition 29 (Ersetzung eines PP-Handle-Subnetzes)
Sei $\mathcal{N} = (P, T, F)$ ein WF-Netz, $(p_s, p_f) \in P \times P$ ein PP-Handle von $\mathcal{N}$ und $\mathcal{N}^{PP}_{(p_s,p_f)} = (P_P, T_P, F_P)$ das entsprechende PP-Handle-Subnetz. Die Ersetzung von $\mathcal{N}^{PP}_{(p_s,p_f)}$ durch eine Stelle p^+ in $\mathcal{N}$ ist gegeben durch das Netz $\mathcal{N}^R \stackrel{\text{def}}{=} (P_R, T_R, F_R)$ mit $P_R = (P \cup \{p^+\}) \setminus P_P$, $T_R = P \setminus T_P$ und

$$F_R = F \cap ((P_R \times T_R) \cup (T_R \times P_R)) \cup \{(\mathfrak{t}, p^+) \in T \times \{p^+\} | (\mathfrak{t}, p) \in (F \cap ((T \setminus T_T) \times P_T))\} \cup \{(p^+, \mathfrak{t}) \in \{p^+\} \times T | (p, \mathfrak{t}) \in (F \cap (P_T \times (T \setminus T_T)))\}$$

♦

Lemma 30 (Ersetzung transitionsberandeter TT-Handle-Subnetze)
Sei $\mathcal{N} = (P, T, F)$ ein WF-Netz und sei $(\mathfrak{t}_s, \mathfrak{t}_f)$ ein TT-Handle von $\mathcal{N}$, so dass das Handle-Subnetz $\mathcal{N}^{TT}_{(\mathfrak{t}_s,\mathfrak{t}_f)} = (P_T, T_T, F_T)$ transitionsberandet ist und außer $(\mathfrak{t}_s, \mathfrak{t}_f)$ keine Handle enthält. Sei $\mathcal{N}^R$ das durch die Ersetzung von $\mathcal{N}^{TT}_{(\mathfrak{t}_s,\mathfrak{t}_f)}$ durch $\mathfrak{t}^+$

hervorgegangene Netz. Dann ist $\mathcal{N}$ T-überdeckbar gdw. $\mathcal{N}^R$ T-überdeckbar ist und es ist $\mathcal{N}$ S-überdeckbar gdw. $\mathcal{N}^R$ S-überdeckbar ist.

Beweis:
(T-Überdeckbarkeit) ($\Rightarrow$) Da $\mathcal{N}^{TT}_{(\mathfrak{t}_s,\mathfrak{t}_f)}$ transitionsberandet ist und keine weiteren Handle enthält, müssen alle Knotne in $(P_T \cup T_T)$ in den selben T-Komponenten von $\mathcal{N}$ sein. Somit ist $\mathcal{N}^R$ ebenfalls T-überdeckbar. ($\Leftarrow$) Ist $\mathcal{N}^R$ T-überdeckbar so gibt es T-Komponenten, die $\mathfrak{t}^+$ beinhalten. Ersetzt man in diesen $\mathfrak{t}^+$ durch $\mathcal{N}^{TT}_{(\mathfrak{t}_s,\mathfrak{t}_f)}$ so erhält man wiederum T-Komponenten, die dann $\mathcal{N}$ überdecken.
(S-Überdeckbarkeit) ($\Rightarrow$) Wäre $\mathcal{N}^R$ nicht S-überdeckbar, so wäre durch die Ersetzung ein PT- oder ein TP-Handle in $\mathcal{N}^R$ entstanden, welches nicht in $\mathcal{N}$ ist. Angenommen es wäre ein PT-Handle $(p, \mathfrak{t})$ entstanden. Würde $\mathfrak{t}^+$ nicht Teil dieses Handles sein, so wäre es auch in $\mathcal{N}$ existent. Gäbe es zwei Pfade von $\mathfrak{t}_s$ nach $\mathfrak{t}_f$ und jeweils eine Transition $\mathfrak{t}_1, \mathfrak{t}_2$ auf jedem der Pfade, so dass es Pfade $(p \ldots \mathfrak{t}_1)$ und $(\mathfrak{t}_2 \ldots \mathfrak{t})$ sowie einen weiteren Pfad $(p \ldots \mathfrak{t})$, dann müsste es auch einen Pfad von $\mathfrak{t}_1$ nach $\mathfrak{t}$ in $\mathcal{N}$ geben, da es andernfalls einen Zyklus gibt. Dann wäre das Handle jedoch ebenfalls in $\mathcal{N}$ vorhanden gewesen. Für TP-Handle gilt eine ähnliche Argumentation mit der öffnenden Transition $\mathfrak{t}_s$. ($\Leftarrow$) Wäre $\mathcal{N}$ nicht S-überdeckbar, so wäre durch die Ersetzung ein PT- oder ein TP-Handle in $\mathcal{N}$ entstanden, welches nicht in $\mathcal{N}^R$ ist. Angenommen es wäre ein PT-Handle entstanden. Dann müssten von einer Stelle p zwei Pfade zu einer Stelle $\mathfrak{t}$ existieren. Ist $\mathfrak{t} \in T_T$, so hätte es das Handle auch mit t^+ gegeben. Andererseits aufgrund der Transitionsberandung auch sonst kein PT-Handle entstanden sein. Für TP-Handle gilt wiederum eine analoge Argumentation. □

Lemma 31 (Ersetzung stellenberandeter PP-Handle-Subnetze)
Sei $\mathcal{N} = (P, T, F)$ ein WF-Netz und sei (p_s, p_f) ein PP-Handle von $\mathcal{N}$, so dass das Handle-Subnetz $\mathcal{N}^{PP}_{(p_s,p_f)} = (P_P, T_P, F_P)$ stellenberandet ist und außer (p_s, p_f) keine Handle enthält. Sei $\mathcal{N}^R$ das durch die Ersetzung von $\mathcal{N}^{PP}_{(p_s,p_f)}$ durch p^+ hervorgegangene Netz. Dann ist $\mathcal{N}$ T-überdeckbar gdw. $\mathcal{N}^R$ T-überdeckbar ist und es ist $\mathcal{N}$ S-überdeckbar gdw. $\mathcal{N}^R$ S-überdeckbar ist.

Beweis:
Der Beweis ist analog zum Beweis in Lemma 30 □

Besitzt ein TT-Handle weitere PP- oder TT-Handle, so können diese Ersetzungen natürlich wiederholt angewandt werden, so dass die Ergebnisse übertragen werden können. Da Handle wiederum eine workflow-artige Struktur besitzen, werden wir hier ebenfalls von Soundness sprechen. Das entsprechende Workflownetz entsteht durch Hinzufügen einer Stelle i bzw. einer Stelle o und entsprechender Kanten. Der Abschluss ist dann definiert wie der Abschluss für WF-Netze.

Satz 32 (Rand von well-handled Subnetzen)
Sei $\mathcal{N} = (P, T, F)$ ein WF-Netz und sei $(\mathfrak{t}_s, \mathfrak{t}_f)$ ein TT-Handle von $\mathcal{N}$, so dass das Handle-Subnetz $\mathcal{N}^{TT}_{(\mathfrak{t}_s,\mathfrak{t}_f)} = (P_T, T_T, F_T)$ well-handled ist. Dann ist $\mathcal{N}^{TT}_{(\mathfrak{t}_s,\mathfrak{t}_f)}$ transitionsberandet.
Sei (p_s, p_f) ein PP-Handle von $\mathcal{N}$, so dass das Handle-Subnetz $\mathcal{N}^{PP}_{(p_s,p_f)} = (P_P, T_P, F_P)$ well-handled ist. Dann ist $\mathcal{N}^{PP}_{(p_s,p_f)}$ stellenberandet.

Beweis:
Da $\mathcal{N}^{TT}_{(\mathfrak{t}_s,\mathfrak{t}_f)}$ well-handled ist, ist der Abschluss $\overline{\mathcal{N}^{TT}_{(\mathfrak{t}_s,\mathfrak{t}_f)}}$ wohlstrukturiert und somit ist $\mathcal{N}^{TT}_{(\mathfrak{t}_s,\mathfrak{t}_f)}$ nach Lemma 26 transitionsberandet.

Ebenso ist $\mathcal{N}^{PP}_{(p_s,p_f)}$ well-handled und der Abschluss $\overline{\mathcal{N}^{PP}_{(p_s,p_f)}}$ ist wohlstrukturiert und somit ist $\mathcal{N}^{PP}_{(p_s,p_f)}$ nach Lemma 27 stellenberandet. □

Lemma 33 (Überdeckbarkeit zyklenfreier wohlstrukturierter WF-Netze)
Ein zyklenfreies wohlstrukturiertes WF-Netze $\mathcal{N}$ ist T-überdeckbar, S-überdeckbar und sound.

Beweis:
(T-Überdeckbarkeit) Wir zeigen dies über die Zahl der PP-Handle. Besitzt $\mathcal{N}$ keine PP-Handle, so ist es ein Marked Graph. Sei die Aussage für n PP-Handle bewiesen und besitze $\mathcal{N}$ $n+1$ PP-Handle. Wir wählen ein PP-Handle (p_s, p_f), das keine weiteren PP-Handle besitzt. Entfernen wir nun alle außer einem Pfad von p_s nach p_f, so erhalten wir ein Netz mit n PP-Handle. Nehmen wir nun von diesem Netz eine T-Komponente T_C, in der sich p_s und p_f befinden und fügen für jeden Pfad von p_s nach p_f eine getrennte T-Komponente hinzu, die ansonsten gleich T_C ist erhalten wir eine T-Überdeckung von $\mathcal{N}$.
(P-Überdeckbarkeit) Der Beweis ist analog zur T-Überdeckbarkeit.
(Soundness) Diese folgt direkt aus der Überdeckbarkeit. □

V. PT- und TP-Handle in WF-Netzen

Wir führen an dieser Stelle den Begriff der kontrollierten Handle ein. Diese haben die Eigenschaft, dass sie nicht die Sicherheits- bzw. die Soundness-Eigenschaft verletzen. Es ist leicht ersichtlich, dass Definition 34 und Definition 35 dual zueinander sind. Abbildung 1 zeigt jeweils ein kontrolliertes TP- und ein kontrolliertes PT-Handle.

Definition 34 (Kontrollierte PT-Handle-Subnetze)
Sei $\mathcal{N}$ ein S-überdeckbares WF-Netz, $(p_s, \mathfrak{t}_f) \in P \times T$ ein PT-Handle von $\mathcal{N}$ und $\mathcal{N}^{PT}_{(p_s,\mathfrak{t}_f)} = (P_H, T_H, F_H)$ das entsprechende PT-Handle-Subnetz, so dass $\mathcal{N}^{PT}_{(p_s,\mathfrak{t}_f)}$ weder transitionsberandete TT-Handle noch stellenberandete PP-Handle noch Zyklen enthält. $\mathcal{N}^{PT}_{(p_s,\mathfrak{t}_f)}$ heißt *kontrolliert* gdw. für alle $\mathfrak{t} \in p_s^\bullet$ und alle $p \in {}^\bullet\mathfrak{t}_f$ eine Transition $\mathfrak{t}_{\mathfrak{t},p}$ mit $|\mathfrak{t}^\bullet_{\mathfrak{t},p}| \geq 2$ existiert, so dass es einen Pfad $(\mathfrak{t} \ldots \mathfrak{t}_{\mathfrak{t},p} \ldots p)$ gibt und für alle Pfade $s = (p_s \ldots \mathfrak{t}_{\mathfrak{t},p})$ gilt $\mathfrak{t} \in \textsc{Alph}(s)$. ♦

Definition 35 (Kontrollierte TP-Handle-Subnetze)
Sei $\mathcal{N}$ ein S-überdeckbares WF-Netz, $(\mathfrak{t}_s, p_f) \in T \times P$ ein TP-Handle von $\mathcal{N}$ und $\mathcal{N}^{TP}_{(\mathfrak{t}_s,p_f)} = (P_H, T_H, F_H)$ das entsprechende TP-Handle-Subnetz, so dass $\mathcal{N}^{TP}_{(\mathfrak{t}_s,p_f)}$ weder transitionsberandete TT-Handle noch stellenberandete PP-Handle noch Zyklen enthält. $\mathcal{N}^{TP}_{(\mathfrak{t}_s,p_f)}$ heißt *kontrolliert* gdw. für alle $\mathfrak{t} \in {}^\bullet p_f$ und alle $p \in \mathfrak{t}_s^\bullet$ eine Transition $\mathfrak{t}_{\mathfrak{t},p}$ mit $|{}^\bullet\mathfrak{t}_{\mathfrak{t},p}| \geq 2$ existiert, so dass es einen Pfad $(p \ldots \mathfrak{t}_{\mathfrak{t},p} \ldots \mathfrak{t})$ gibt und für alle Pfade $s = (\mathfrak{t}_{\mathfrak{t},p} \ldots p_f)$ gilt $\mathfrak{t} \in \textsc{Alph}(s)$. ♦

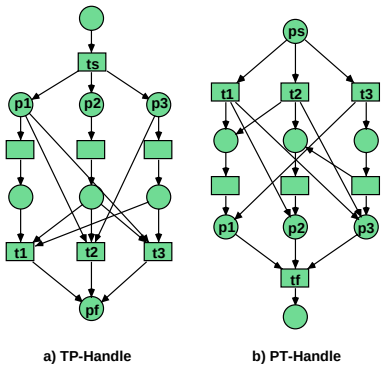

Fig. 1. Beispiele für S-überdeckbare und kontrollierte PT- und TP-Handle.

Satz 36 (Soundness kontrollierter PT-Handle-Subnetze)
Sei $\mathcal{N}$ ein sicheres WF-Netz, $(p_s, \mathfrak{t}_f) \in P \times T$ ein PT-Handle von $\mathcal{N}$ und $\mathcal{N}^{PT}_{(p_s,\mathfrak{t}_f)} = (P_H, T_H, F_H)$ das entsprechende PT-Handle-Subnetz, so dass für alle $p \in P_H$ $|p^\bullet| \geq 2 \implies p = p_s$ gilt und für alle $\mathfrak{t} \in T_H$ $|{}^\bullet\mathfrak{t}| \geq 2 \implies \mathfrak{t} = \mathfrak{t}_f$. $\mathcal{N}^{PT}_{(p_s,\mathfrak{t}_f)}$ ist T-überdeckbar (also sound) gdw. $\mathcal{N}^{PT}_{(p_s,\mathfrak{t}_f)}$ kontrolliert ist.

Beweis:
($\Rightarrow$) Da $\mathcal{N}^{PT}_{(p_s,\mathfrak{t}_f)}$ T-überdeckbar ist, sind alle $p \in {}^\bullet\mathfrak{t}_f$ stets Teil der selben T-Komponenten wie $\mathfrak{t}_f$. Da jede T-Komponente $\mathfrak{t}_f$ umfassen muss, gilt dies auch für die T-Komponenten von $\mathfrak{t} \in p_s^\bullet$. Somit ist muss es von jeder dieser Transitionen einen Pfad zu allen Stellen $p_l \in {}^\bullet\mathfrak{t}_f$ geben, so dass für alle $\mathfrak{t} \in p_s^\bullet$ und alle $p \in {}^\bullet\mathfrak{t}_f$ eine Transition $\mathfrak{t}_{\mathfrak{t},p}$ mit $|\mathfrak{t}^\bullet_{\mathfrak{t},p}| \geq 2$ existiert und es einen Pfad $(\mathfrak{t} \ldots \mathfrak{t}_{\mathfrak{t},p} \ldots p)$ gibt. Wäre nun nicht für alle Pfade $s = (p_s \ldots \mathfrak{t}_{\mathfrak{t},p})$ auch $\mathfrak{t} \in \textsc{Alph}(s)$, so gäbe es zwei Pfade von p_s nach $\mathfrak{t}_{\mathfrak{t},p}$ und folglich könnte eine Stelle im Nachbereich von $\mathfrak{t}$ zweimal markiert werden und das Netz wäre nicht sicher.
($\Leftarrow$) Sei $\mathcal{N}^{PT}_{(p_s,\mathfrak{t}_f)}$ kontrolliert. Angenommen $\overline{\mathcal{N}^{PT}_{(p_s,\mathfrak{t}_f)}}$ sei nicht T-überdeckbar. Gäbe es von jeder Transition $\mathfrak{t} \in p_s^\bullet$ einen Pfad zu $\mathfrak{t}_f$, so ließen sich entsprechende S-Komponenten bilden, indem alle Knoten zwischen $\mathfrak{t}$ und $\mathfrak{t}_f$ Teil der Komponente wären. Gäbe es keinen solchen Pfad, dann wäre $\mathcal{N}^{PT}_{(p_s,\mathfrak{t}_f)}$ nicht kontrolliert. □

Wie in Abbildung 2 zu sehen ist, gibt es sounde zyklenfreie WF-Netze, die nicht S-überdeckbar sind. Fügt man dem Netz jedoch die Stelle px hinzu, so wird es S-überdeckbar, ist aber nicht länger zyklenfrei. In Abbildung 3 ist ein S-überdeckbares Netz dargestellt, welches nicht sound ist. Folglich ist es nicht möglich, von einer dieser beiden Eigenschaften auf die andere zu schließen.

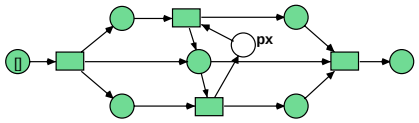

Fig. 2. Ein nicht S-überdeckbares zyklenfreies soundes WF-Netz. Wird die Stelle px hinzugefügt ist das Netz S-überdeckbar, aber nicht länger zyklenfrei.

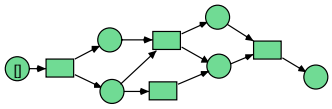

Fig. 3. Ein S-überdeckbares zyklenfreies WF-Netz, das nicht sound ist.

VI. Ausblick

In [6] werden verschiedene Eigenschaften angegeben, die erhalten bleiben, wenn WF-Netze komponiert werden. Dies erlaubt es, in einem WF-Netz zunächst die TT- und PP-Handle zu identifizieren und durch Transitionen bzw. Stellen zu ersetzen. Hiernach lassen sich die PT-Handle analysieren. Sind diese sicher und gilt, dass ihr Rand lediglich aus der Eingangsstelle und der Ausgangstransition besteht, so lassen sich diese ebenfalls durch eine Kante ersetzen. Es ist nun die Hoffnung, dass zusammen mit den Ersetzungsregeln aus [4] ein zyklenfreies S-überdeckbares WF-Netz, das sound ist, soweit reduziert werden kann, bis es lediglich aus einer Stelle besteht. Ist dies nicht möglich, so ist das WF-Netz entweder nicht zyklenfrei, nicht S-überdeckbar oder nicht sound. Dasjenige Handle, an dem die Reduktion scheitert, kann dann an den Nutzer zurückgekoppelt und bearbeitet werden.

Als nächstes wird die Identifikation von Zyklen und die Implementation eines Algorithmusses angestrebt, der die Handle des Netzes identifiziert und überprüft. Hierbei kann auf Resultate aus der Compiler-Optimierung [3] zurückgegriffen werden.

References

[1] Jörg Desel and Javier Esparza. *Free Choice Petri nets.* Number 40 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge(UK), 1995.

[2] Javier Esparza and Manuel Silva. Circuits, handles, bridges and nets. In Grzegorz Rozenberg, editor, *Applications and Theory of Petri Nets*, volume 483 of *Lecture Notes in Computer Science*, pages 210–242. Springer-Verlag, 1989.

[3] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[4] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[5] Wil M. P. van der Aalst. Verification of workflow nets. In Pierre Azéma and Gianfranco Balbo, editors, *Application and Theory of Petri Nets 1997*, number 1248 in Lecture Notes in Computer Science, pages 407–426, Berlin u.a., 1997. Springer-Verlag.

[6] Wil M. P. van der Aalst. Workflow verification: Finding control-flow errors using petri-net-based techniques. In Jörg Desel and Andreas Oberweis, editors, *Business Process Management, Models, Techniques, and Empirical Studies*, pages 161–183, Berlin u.a., 2000. Springer-Verlag.

[7] Henricus M.W. Verbeek. *Verification of WF-nets*. PhD thesis, Technische Universiteit, Eindhoven, 2004. WWW-Dokument: alexandria.tue.nl/extra2/200411300.pdf.

Finite Automata Controlled by Petri Nets

Matthias Jantzen[1], Manfred Kudlek[1], and Georg Zetzsche[1]

[1] Department Informatik, MIN-Fakultät, Universität Hamburg, DE
email: {jantzen,kudlek,3zetzsch}@informatik.uni-hamburg.de

Abstract. We present a generalization of finite automata using Petri nets as control. Acceptance is defined by final markings of the Petri net. The class of languages obtained by λ-free concurrent finite automata contains both the class of regular sets and the class of Petri net languages defined by final marking.

1 Introduction

In classical finite automata, the input is read by one head that moves across one symbol in every step. In order to investigate the impact of concurrency on automata accepting languages, our generalization of finite automata allows arbitrarily many heads that can move concurrently. These heads are distributed across the input and in particular, different parts of the input can be processed at the same time.

A similar model, that applies the idea of multiple independent heads to Turing machines instead of finite automata is investigated in [FJKRZ] where also a detailed bibliography can be found.

The concurrency is achieved by using a Petri net that describes the movement of the heads. For every position on the input, there is a multiset of heads, which is interpreted as a marking of the Petri net. If a transition fires at some position of the input, the corresponding preset is removed from that position and the postset is added at the next position. In contrast to multi-head automata, there is no global state, since the ability to fire only depends on the heads at the respective position. Therefore, different parts of the word can be processed concurrently.

It turns out that this model is equivalent to an automaton which, for every symbol on the input, solves a system of algebraic equations and applies some homomorphism to the solution to obtain the next configuration. These two definitions are presented in section 2. Section 3 and 4 give an overview of the results obtained so far concerning the languages accepted by CFA.

2 Definitions

Definition 1. *A set M with an operation $+ : M \times M \to M$ is a* monoid*, if the operation is associative and there is a neutral element $0 \in M$ for which $0 + x = x + 0 = x$ for every $x \in M$. M is called* commutative*, if $x + y = y + x$ for all $x, y \in M$. For $x, y \in M$, let $x \sqsubseteq y$ iff there is a $z \in M$ with $y = x + z$.*

For every set A, we have the set $A^\oplus$ of mappings $\mu : A \to \mathbb{N}$. The elements of $A^\oplus$ are called multisets *over A. We identify every element $a \in A$ with the corresponding multiset μ_a with $\mu_a(a) = 1$ and $\mu_a(a') = 0$ for every $a' \in A \setminus \{a\}$. With the operation $\oplus$, defined by $(\mu \oplus \nu)(a) = \mu(a) + \nu(a)$, $A^\oplus$ becomes a commutative monoid with the neutral element $\mathbf{0}$, $\mathbf{0}(a) := 0$ for every $a \in A$. In the case $\mu \sqsubseteq \nu$ we can define $(\nu \ominus \mu)(a) := \nu(a) - \mu(a)$ for all $a \in A$. If A is finite, let $|\mu| := \sum_{a \in A} \mu(a)$. These definitions are carried over to $\mathbb{N}^k$ by noting that $\mathbb{N}^k \cong \{a_1, \ldots, a_k\}^\oplus$.*

A concurrent finite automaton is given by the following data.

Definition 2. *A* CFA *is a sextuple $C = (\Sigma, N, \sigma, \mu_0, \mathcal{F}, \#)$, where*

- *Σ is an alphabet,*
- *$N = (P, T, \partial_0, \partial_1)$ is a Petri net, where P (T) is the set of places (transitions) and $\partial_0, \partial_1 : T^\oplus \to P^\oplus$ are homomorphisms that specifiy the pre- and post-multisets. Furthermore, $\partial_0(t) \neq \mathbf{0}$ for every $t \in T$. In the case that $\partial_1(t) \neq \mathbf{0}$ for every $t \in T$, C is called* non-erasing*.*
- *$\sigma : T \to \Sigma \cup \{\lambda\}$ defines the corresponding symbol for every transition. A transition $t \in T$ is called* λ-transition*, if $\sigma(t) = \lambda$. C is called* λ-free*, if it does not contain λ-transitions.*
- *$\mu_0 \in P^\oplus$ is the* initial marking*,*
- *$\mathcal{F}$ is a finite set of* final markings*,*
- *$\# \notin \Sigma$ is the* end marker *symbol.*

Now we give two definitions of the accepted language that are equivalent, that is, the same language classes result from these definitions. The first definition directly describes the firing of the transitions in the underlying Petri net.

Definition 3. *Let $C = (\Sigma, N, \sigma, \mu_0, \mathcal{F}, \#)$ a CFA, where $N = (P, T, \partial_0, \partial_1)$. A* configuration *is a tuple $(\nu_0, a_1, \nu_1, \ldots, a_n, \nu_n, \#, \nu_{n+1})$, where $\nu_0, \ldots, \nu_{n+1} \in P^\oplus$ and $a_1, \ldots, a_n \in \Sigma$. On the set of configurations of C, we define the binary relation $\xrightarrow[C]{}$. It describes the firing of one transition. Let*

$$(\nu_0, a_1, \nu_1, \ldots, a_n, \nu_n, \#, \nu_{n+1}) \xrightarrow[C]{} (\nu'_0, a_1, \nu'_1, \ldots, a_n, \nu'_n, \#, \nu'_{n+1})$$

iff there are $t \in T, j \in \{1, \ldots, n+1\}$ and one of the following conditions holds:

- *$\sigma(t) = a_j$ (where $a_{n+1} = \#$), $\nu'_{j-1} = \nu_{j-1} \ominus \partial_0(t)$, $\nu'_j = \nu_j \oplus \partial_1(t)$, and $\nu'_i = \nu_i$ for every $i \neq j$.*
- *$\sigma(t) = \lambda$ and $\partial_0(t) \sqsubseteq \nu_j$, $\nu'_j = \nu_j \ominus \partial_0(t) \oplus \partial_1(t)$ and $\nu'_i = \nu_i$ for every $i \neq j$.*

So the firing of a non-λ-transition moves tokens across one symbol whereas λ-transitions work like ordinary Petri net transitions on the multiset at one position. Then for $w \in \Sigma^*$, $w = a_1 \cdots a_n$, $a_1, \ldots, a_n \in \Sigma$, it is $w \in L_1(C)$ iff

$$(\mu_0, a_1, \mathbf{0}, \ldots, a_n, \mathbf{0}, \#, \mathbf{0}) \xrightarrow[C]{*} (\mathbf{0}, a_1, \mathbf{0}, \ldots, a_n, \mathbf{0}, \#, \mu)$$

for some $\mu \in \mathcal{F}$, where $\xrightarrow[C]{*}$ denotes the reflexive transitive closure of $\xrightarrow[C]{}$. Thus, a word is accepted if all the heads are on the rightmost position and a final marking has been reached.

One possible variant is to accept a word also if the final marking is reached in a distributed manner. So for $w \in \Sigma^*$, $w = a_1 \cdots a_n$, $a_1, \ldots, a_n \in \Sigma$, let $w \in L_2(C)$ iff

$$(\mu_0, a_1, \mathbf{0}, \ldots, a_n, \mathbf{0}, \#, \mathbf{0}) \xrightarrow[C]{*} (\nu_0, a_1, \nu_1, \ldots, a_n, \nu_n, \#, \nu_{n+1}),$$

where $\nu_0 \oplus \cdots \oplus \nu_{n+1} \in \mathcal{F}$ and $\nu_{n+1} \neq \mathbf{0}$.

Now we present another definition, where the automaton solves a system of algebraic equations and obtains the next configuration by applying a homomorphism to the solution.

Definition 4. *Using the notation $T_x := \{t \in T \mid \sigma(t) = x\}$ for $x \in \Sigma \cup \{\lambda\}$, we define for every CFA C the binary relation $\Longrightarrow_C$ on $\Sigma^* \times P^\oplus$ by*

$$(w, \mu) \Longrightarrow_C (wa, \mu') \text{ if } \exists v \in T_a^\oplus : \partial_0(v) = \mu \wedge \partial_1(v) = \mu',$$

for $w \in (\Sigma \cup \{\#\})^*$, $a \in \Sigma$, $\mu, \mu' \in P^\oplus$ and

$$(w, \mu) \Longrightarrow_C (w, \mu') \text{ if } \exists t \in T_\lambda : \partial_0(t) \sqsubseteq \mu \wedge \mu' = \mu \ominus \partial_0(t) \oplus \partial_1(t),$$

for $w \in (\Sigma \cup \{\#\})^*$ and $\mu, \mu' \in P^\oplus$. If $\overset{*}{\Longrightarrow}_C$ denotes the reflexive transitive closure of $\Longrightarrow_C$, then the accepted language of C is

$$L_3(C) := \{w \in \Sigma^* \mid \exists \mu \in \mathcal{F} : (\lambda, \mu_0) \overset{*}{\Longrightarrow}_C (w\#, \mu)\}.$$

Now it is not hard to see that $L_3(C) = L_1(C)$ for every CFA C. Furthermore, it can be shown that the following definitions of language classes accepted by different types of CFA do not depend on whether one chooses $L_1(C)$ or $L_2(C)$ as the language of C.

By $\mathcal{C}_0$ we denote the class of languages accepted by non-erasing λ-free CFA. $\mathcal{C}_0'$ is the class of languages accepted by λ-free CFA, $\mathcal{C}_0^\lambda$ the class of languages accepted by non-erasing CFA, and $\mathcal{C}_0'^\lambda$ the class of languages accepted by arbitrary CFA. **REG**, **CF**, **CS**, **RE** denotes the class of regular, context-free, context-sensitive, recursively enumerable languages, respectively. Then let $\mathcal{L}_0^\lambda$, ($\mathcal{L}_0$) be the class of Petri net languages generated by (λ-free) Petri nets with final marking, as defined in [Jan]. For alphabets Σ, Γ, a homomorphism $h : \Sigma^* \to \Gamma^*$ is called *coding*, if $h(a) \in \Gamma$ for all $a \in \Sigma$. For a language class $\mathcal{L}$, let $\hat{\mathcal{H}}(\mathcal{L})$ ($\mathcal{H}^{cod}(\mathcal{L})$) be the class of all languages $h(L)$ with $L \in \mathcal{L}$, where h is an arbitrary homomorphism (coding).

3 Relations To Other Language Classes

Erasing transitions can be simulated by λ-transitions by basically adding some token to the postset of every transition and delete these tokens with λ-transitions. Therefore, we obtain the

Theorem 1. *$\mathcal{C}_0' \subseteq \mathcal{C}_0^\lambda$ and $\mathcal{C}_0^\lambda = \mathcal{C}_0'^\lambda$.*

Furthermore, it is easy to see that (λ-free) Petri nets can be simulated by (λ-free) CFA. This inclusion is even proper.

Theorem 2. *$\mathcal{L}_0 \subset \mathcal{C}_0$ and $\mathcal{L}_0^\lambda \subset \mathcal{C}_0^\lambda$.*

While it is still open whether all context-free languages are accepted by CFA, the application of codings allows a construction similar to one used for push-down-automata.

Theorem 3. *For every context-free language L, there is a coding h and a non-erasing λ-free CFA C such that $L = h(L(C))$. Thus, $\mathbf{CF} \subseteq \mathcal{H}^{cod}(\mathcal{C}_0)$.*

Applying arbitrary homomorphisms to languages accepted by CFA leads to the whole class of recursively enumerable languages.

Theorem 4. *For every recursively enumerable language L, there is a (possibly erasing) homomorphism h and a non-erasing λ-free CFA C such that $L = h(L(C))$. Thereby, h and C are effectively constructible. Thus, $\mathbf{RE} = \hat{\mathcal{H}}(\mathcal{C}_0)$.*

In particular, the emptiness problem is undecidable even for non-erasing λ-free CFA. In contrast, the word problem is decidable for every type of CFA and therefore we have the proper inclusion $\mathcal{C}_0 \subset \mathbf{RE}$. When we restrict to λ-free CFA, the word problem becomes much simpler. For every CFA, there is an algorithm accepting the language in linear space and quadratic time.

Theorem 5. *$\mathcal{C}_0' \subseteq \mathrm{NTimeSpace}(n^2, n)$.*

Since the inclusion of $\mathrm{NTimeSpace}(n^2, n)$ in **CS** is a proper one, we get the following

Corollary 1. *$\mathcal{C}_0' \subset \mathbf{CS}$.*

The diagram in figure 1 shows the known relations between the languages accepted by CFA and other language classes.

4 Closure Properties

An easy consequence from $\mathcal{C}_0 \subseteq \mathcal{C}_0' \subseteq \mathcal{C}_0^\lambda \subset \mathbf{RE}$ and Theorem 4 is the following

Theorem 6. *$\mathcal{C}_0$, $\mathcal{C}_0'$, $\mathcal{C}_0^\lambda$ are not closed under arbitrary homomorphisms.*

Nevertheless, at least the class $\mathcal{C}_0^\lambda$ is closed under non-erasing homomorphisms. This fact and Theorem 3 imply

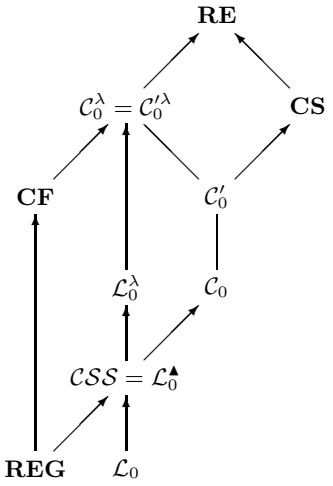

Fig. 1. Relations to other language classes.

Theorem 7. $\mathbf{CF} \subseteq \mathcal{C}_0^\lambda$.

It is not difficult to prove the closure against union and intersection. $\mathcal{C}_0$ and $\mathcal{C}_0^\lambda$ are also closed under concatenation. However, it is still open whether this is also true for $\mathcal{C}_0'$.

Theorem 8. *$\mathcal{C}_0$, $\mathcal{C}_0'$ and $\mathcal{C}_0^\lambda$ are closed under union and intersection. $\mathcal{C}_0$ and $\mathcal{C}_0^\lambda$ are closed unter concatenation.*

In order to prove that the CFA-languages are closed unter inverse homomorphisms, a lemma about finitely generated monoids was needed. It states that a certain property, called *quasi-invertibility*, of submonoids of finitely generated commutative monoids is sufficient for the submonoid to be also finitely generated.

Definition 5. *Let M be a commutative monoid. A subset $S \subseteq M$ is called* quasi-invertible *iff for every $a, b \in M$, $a \in S$ and $a + b \in S$ imply $b \in S$.*

For example, every kernel of a homomorphism is a quasi-invertible submonoid. Or, slightly more general, if $h : M \to M'$ is a homomorphism of commutative monoids and $S \subseteq M'$ is quasi-invertible, then $h^{-1}(S) \subseteq M$ is quasi-invertible as well.

Lemma 1. *Let M be a finitely generated commutative monoid and $S \subseteq M$ a quasi-invertible submonoid. Then S is finitely generated.*

For the application of this lemma, the algebraic view on CFA becomes useful.

Theorem 9. *$\mathcal{C}_0$ and $\mathcal{C}_0'$ are closed under inverse homomorphisms.*

An overview of the closure properties is given in figure 2. Thereby, $\cap R$, $L_1 \cdot L_2$, $\cup$, $\cap$, h^{-1}, h, λ-free h stand for intersection with regular languages, concatenation, union, intersection, inverse homomorphism, arbitrary homomorphism and non-erasing homomorphism, respectively.

Operator	$\mathcal{C}_0$	$\mathcal{C}_0'$	$\mathcal{C}_0^\lambda$
$\cap R$	+	+	+
$L_1 \cdot L_2$	+	?	+
$\cup$	+	+	+
$\cap$	+	+	+
h^{-1}	+	+	?
h	-	-	-
λ-free h	?	?	+

Fig. 2. Closure Properties

References

[FJKRZ] B. Farwer, M. Jantzen, M. Kudlek, H. Rölke, G. Zetzsche: *On Concurrent Finite Automata*, to be published, 2007.

[Jan] M. Jantzen : *On the Hierarchy of Petri Net Languages.* RAIRO **13**, no. 1, pp. 19-30, 1979.

Faster Unfolding of General Petri Nets

Robin Bergenthum, Robert Lorenz, Sebastian Mauser

Lehrstuhl für Angewandte Informatik
Katholische Universität Eichstätt-Ingolstadt, Eichstätt, Germany
e-mail: {robin.bergenthum, robert.lorenz, sebastian.mauser}@ku-eichstaett.de

Abstract

We propose two new unfolding semantics for general Petri nets based on the concept of prime event structures. The definitions of the two unfolding models are motivated by algorithmic aspects. We develop a construction algorithm for both unfolding models. The unfolding models employ the idea of token flows developed in [JLD05] and are much smaller than the standard unfolding model. We show that they still represent the complete partial order behaviour of the given net. Since the models are smaller, they can be constructed much faster.

1 Introduction

Non-sequential Petri net semantics can be coarsely classified into unfolding semantics, process-oriented semantics and algebraic semantics [MMS94]. While the latter is not widely known and process models do not provide semantics of a net as a whole, but specify only single, deterministic computations, unfolding models are a popular approach to describe the complete behaviour of nets and thus account for the fine interplay between concurrency and nondeterminism. To study the behaviour of Petri nets primarily two models for unfolding semantics were retained: labelled occurrence nets and event structures. In this work we focus on algorithmic aspects of unfolding semantics. We propose two new event structure-based unfoldings of general Petri nets and compare them to the existing unfolding semantics. For both new unfolding models, we sketch a construction algorithm and argue, that they can be constructed in many cases very efficient compared to the standard unfolding approach.

The standard unfolding semantics for general Petri nets (p/t-nets) was developed in [BD87, MMS97, MMS94]. This approach generalizes the unfolding semantics originally introduced in [NPW81] for safe nets, and extended in [Eng91] to initially one marked p/t-nets without arc weights by viewing the tokens as individualized "coloured" entities. In contrast to [Eng91], the unfolding approach for p/t-nets even individualizes tokens having the same "history". This guarantees that one can analogously as in [Eng91] define a single occurrence net (or more precisely a branching process), called the unfolding of the system, capturing the complete behaviour of the p/t-net (see Figure 2). The unfolding construction is well analyzed and several algorithmic improvements are proposed in literature. By restricting the relations of causality and conflict of the resulting occurrence net to events, one obtains a labelled prime event structure (with binary conflict relation) representing the causality of the unfolding (see Figure 3). This labelled prime event structure modelling the behaviour of a p/t-net is called PES0 in the following. In figures of prime event structures, e.g. Figure 3, the colouring of the events illustrates the sets of consistent events, i.e. events not being in conflict. A set of consistent events of a prime events structure defines a partially ordered *run*.

As explained in [HKT96, MMS94, CPW04] this standard unfolding approach for p/t-nets, in particular the concept of coloured tokens, yields several problems. Consequently there have been many attempts to define alternative unfolding semantics for p/t-nets: using so called local event structures [HKT96] instead of prime event structures, allowing arbitrarily valued and non-safe occurrence nets [CPW04], grouping conditions into families [MMS97] (decorated occurrence nets), translating general nets into safe nets by introducing places for reachable markings [Haa98] or

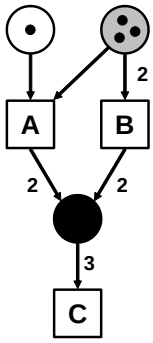

Figure 1: Example net N.

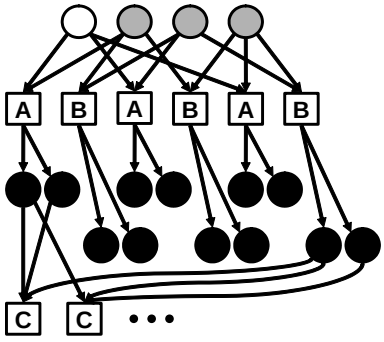

Figure 2: Standard unfolding of N.

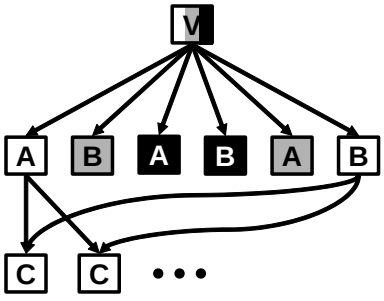

Figure 3: Prime event structure PES0 of N.

considering a swapping equivalence to introduce a collective token view [BD87]. All of these works focus on algebraic and order-theoretic properties, but the approaches are not to of algorithmic interest.

In this paper we are interested in algorithms to efficiently construct unfoldings. With regard to this, the standard unfolding procedure for p/t-nets has the drawback, that the individualized tokens cause multiple conditions referring to the same place in the net and having the same pre-event (or being initial events), e.g. the three initial grey conditions in Figure 2. This artificially blows up the size of the unfolding and the size of PES0. The individualized tokens in the worst case increases the number of events exponentially for every step of depth of the unfolding. For example one has to regard all three possibilities of the transition b to consume two of the three individualized tokens in the grey place (compare the three b events in Figure 2). That means the individualized tokens (condition) cause PES0 to contain multiple instances of so-called *identical* events, which have the same label, the same pre-events (pre-events are events from which the considered event is causally dependant in a prime event structure) and are in conflict (e.g. the three a- or b-events in Figure 3 are identical, also the two c-events in Figure 4).[1] In other words two events in PES0 are identical, if the pre-sets (of conditions) of the two events in the respective unfolding have the same pre-set of events and share at least one condition. Two cases of identical events can be distinguished. First the two presets of the identical events in the unfolding can contain for each place label the same number of conditions having this label and a common pre-event (see the identical a-events in Figure 2 respectively 3). This phenomenon of generating identical events we abbreviate as (IDENT1). But it is also possible, that there are identical events in PES0, whose presets in the unfolding contain for some place label a different number of conditions having this label and a common pre-event. For example one of the two depicted c-events in Figure 2 has two black pre-conditions generated by the most left a-event and one black pre-condition generated by the most right b-event, while the other depicted c-event has two black pre-conditions generated by the most right b-event and one black pre-condition generated by the most left a-event. This phenomenon is called (IDENT2). The described problem (IDENT1) causes the standard unfolding to be an inefficient representation of the set of Goltz-Reisig processes [GR83] of the given p/t-net. The unfolding includes several copies of the same process, because each (IDENT1)-identical event introduces additional isomorphic processes (e.g. in Figure 2, all three conflict-free combinations of a and b events yield isomorphic processes). Moreover the calculation of the processes from the unfolding is inefficient. In contrast to (IDENT1)-identical events, (IDENT2)-identical events may (usually but not always) constitute different Goltz-Reisig processes (the two identical c-events in Figure 4 define two different processes). But for a compact unfolding representation of the net behaviour, it is also not necessary to consider these multiple instances of (IDENT2)-identical events, because such events produce isomorphic partially ordered runs.

Altogether individualizing tokens causes the standard unfolding and the associated labelled prime event structure PES0 to become unnecessary large. PES0 contains huge numbers of identical events, although identical events are never necessary to capture the causal behaviour of a system by a labelled prime event structure. Unfolding models ensuring less nodes could significantly decrease the construction time, because a construction algorithm in some way always has to test all co-sets of events or conditions of the so-far constructed model to append further events. For example, in each algorithm to construct the standard unfolding and PES0, all co-sets of constructed conditions have to be checked (e.g. in [DJL03]). Since the number of conditions in the unfolding is linearly dependant on the number of events, reducing the number of events could significantly improve the runtime of a construction algorithm. In this paper we propose two unfolding approaches reducing the number of events in contrast to the standard unfolding by neglecting identical events.

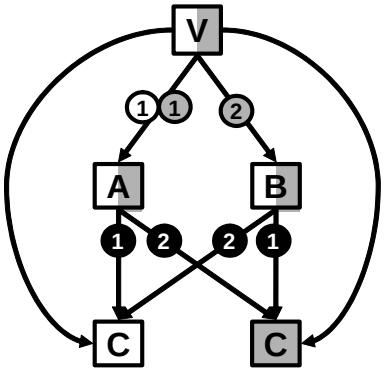

Figure 4: Prime event structure PES1 of N.

We show two methods to directly unfold a p/t-net to a labelled prime event structure using the concept of *token flow* presented in [JLD05]. Token flows were developed for a compact representation of process nets. Namely, token flows abstract from the individuality of conditions of the process net and encode the flow relation of the process by natural numbers. For each place a natural number is assigned to the edges of the partial order relation between events defined by the process net. Such a natural number assigned to an edge (v, v') represents the number of tokens produced by the transition occurrence v and consumed by the transition occurrence v' in the respective place. This idea is generalized to unfoldings in this paper. Since we abstract from the individuality of conditions, both presented unfolding methods generate event structures having significantly less events than PES0. To append events in a construction algorithm generating directly an event structure, co-sets of events or more precisely prefixes (defined by a co-set) of consistent events have to be checked. Minimizing the number of events, the two proposed approaches are candidates to yield fast construction algorithms.

The first method constructs a labelled prime event structure enriched with the full token flow information from a p/t-net. The resulting event structure, called PES1 in the following, basically coincides with PES0, except (IDENT1) being resolved, i.e. respective identical events are neglected (compare Figure 3 and 4). That means the number of

[1] Note that multiple instances of concurrent events with the same label and the same pre-events are necessary to model the causalities of the net, but identical events are not necessary.

events is usually a lot smaller, while the token flow information is preserved. Note here that omitting identical events disables the possibility of applying prime event structures with a binary conflict relation. We use the general prime event structures with a consistency set as introduced in [Win86] (in a slightly simplified version). That means for each process in the unfolding we have to store a set of respective events in a so called consistency set. Because the number of processes represented by an unfolding might be exponential, this could lead to some performance problems of the construction algorithm. On the one hand it could lead to a higher memory consumption for saving the consistency set, although the number of stored events is decreased. On the other hand this also leads to some minor runtime problems explained later on, so that it is not clear in advance, in which cases the method is faster than the standard unfolding method. It remains to mention that the token flow information of PES1 guarantees that the set of Goltz-Reisig processes of the p/t-net is still completely reflected in PES1, but the number of isomorphic processes is a lot smaller than in the standard unfolding. Isomorphic processes occur in PES1 only in specific auto-concurrency situations. The set of processes can directly be deduced from PES1 by the consistency sets. We implemented an algorithm to construct PES1.

The second method constructs a labelled prime event structure PES2 from a p/t-net, which additionally to PES1 resolves (IDENT2), i.e. respective identical events are also neglected (the two (IDENT2)-identical c-events in Figure 4 do not occur in Figure 5). That means PES2 contains no two identical events at all. Therefore it is the smallest labelled prime event structure capturing the complete behaviour of a p/t-net. Notice, that the token flow information of PES2 is not any more "complete" (in contrast to PES1). That means not each possible token flow distribution is represented in PES2. Instead "example token flows" are stored for each partially ordered run. Consequently, while the construction algorithm deals with smaller event structures and thus less possibilities to append further events, the procedure of appending one event is more complicated, because the token flow eventually has to be recalculated. There is an efficient method for this recalculation based on the ideas in [JLD05]. Finally, note that the set of Goltz-Reisig processes is of course not completely represented in PES2, since there are different processes generating the same run, and every run exists exactly once in PES2. For each run associated to some process, one example process generating the run is given by a respective example token flow in PES2. We did not implement a construction algorithm for PES2 so far.

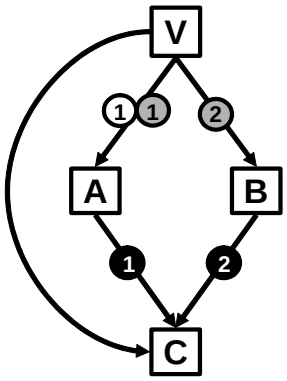

Figure 5: Prime event structure PES2 of N.

Lastly the question for the practical usefulness of the two event structures PES1 and PES2 has to be answered. First they can in many cases be constructed more efficient than the standard unfolding. A comparison of runtime and size of constructing PES0 and PES1 follows in Section 4. In particular the new methods may lead to a more efficient computation of the set of processes of a p/t-net. Furthermore they can still be applied almost analogously as the standard unfolding and PES0: they form a compact representation of the complete behaviour of a p/t-net accounting at the same time for concurrency and nondeterminism and they offer a possibility to calculate and represent the set of Goltz-Reisig processes. Most of the model checking algorithms working on standard unfoldings and PES0 can also be adapted to PES1 and PES2. The two event structures PES1 and PES2 are even more compact than the standard unfolding. Consequently applying PES1 and PES2 could further improve the efficiency of such automatic verification methods. In this context it is also important that the concepts of creating a finite complete prefix of the standard unfolding of a bounded p/t-net using adequate partial orders and cut-off events can analogously be transferred to PES1 and PES2. Having this in mind we do not discuss these application topics in this paper any further, but concentrate only on the runtime of construction algorithms.

The two prime event structure unfolding methods will be explained and discussed in detail in the following two sections. In particular they will be compared with each other and with the standard unfolding method on a conceptual level. We omit technical definitions or proofs in this workshop paper. In Section 4 we finally compare the runtime of our implementation of a construction algorithm for PES1 and of the standard unfolding algorithm used in [DJL03] with experimental results.

2 Algorithm 1

PES1: The first method constructs a general labelled prime event structure PES1 enriched with fixed token flow tuples for every edge of the structure (see Figure 4). The consistency set of the prime event structure stores sets of events forming a run (or process) of the p/t-net. The algorithm will only consider so called left-closed sets of consistent events (forming prefixes of the prime event structure), since only such sets represent runs. Moreover only maximal (w.r.t. set inclusion) runs are stored having in mind that prefixes of runs are also runs. Within one run, i.e. one set of consistent events, the ordering relation of the prime event structure models the causal dependencies of the events. That means ordered events of one run necessarily occur in the timeline of the respective computation in this ordering, i.e. the first event occurs before the second one, while unordered events of one run occur concurrently in the respective computation. The token flow tuples assigned to the edges modelling the causal dependency relation of the

prime event structure have one non-negative integer entry for every place in the given p/t-net (compare Figure 4). These values represent the numbers of tokens produced by the event at the beginning of the edge and consumed by the event at the end of the edge in the respective place. Such flow of tokens is responsible for the causal dependencies of events in a p/t-net. Thus such tuple assigned to an edge of the skeleton of the event structure is never the zero-tuple, since a dependency between events always comes with a non-zero flow of tokens or it is caused by the transitivity of the dependency relation. A zero flow of tokens means, that there is no direct dependency between events. In particular, there are no token flows between concurrent events of the event structure. The token flows also determine the construction of PES1. PES1 contains one copy of each run of the net for each possible token flow distribution within the run. As mentioned in the introduction identical events w.r.t. (IDENT1) having the same ingoing flows of tokens are not duplicated. But for each possible distribution of the ingoing token flows, one event is generated, i.e. there are identical events in PES1, but the token flows assigned to the ingoing edges of the events differ (e.g. the two c-events in Figure 4). Since in the standard unfolding, there may be exponentially many identical events having the same ingoing token flows, PES1 may have significantly less events than PES0.

Construction algorithm: We implemented a construction algorithm for PES1. It starts introducing an initial event v representing the initial marking of the p/t-net. This event constitutes the first consistency set, and the residual token flow of this event within the consistency set coincides with the initial marking. The residual token flow of an event within a consistency set represents the numbers of tokens produced by the event and not consumed by some other event in the consistency set. Transition occurrences are step by step appended in a fixed order. Given a consistency set and residual token flows for each event within this consistency set, one can append events consuming less tokens than given by the sum of the residual token flows. For each appended transition occurrence, all possibilities to distribute the residual token flows onto the ingoing token flow of this transition occurrence are considered. This is a combinatoric problem. Having solved this problem yields the candidates of events together with their ingoing token flows, that have to be appended to PES1 within the consistency set. Each such event leads to a new consistency set with new residual token flows for each event. The new residual token flows can easily be computed from the old ones. The only remaining challenge here is that some event, that should be appended in some stage of the algorithm, was possibly already added in some previous stage. Then it must not be duplicated. This has to be checked and leads to a recalculation of some events.

Performance: Altogether the algorithm should have a significantly faster runtime than the standard unfolding algorithm in many cases, because the number of events is a lot smaller (the difference can be of exponential size) and the procedure of appending events is similar: The stored token flows of a run determine the residual token flows, that can be used to append an event. The computation of the residual token flows is a simple summation operation. It is approximately as costly as the stepwise calculation of co-sets of conditions, which is necessary in the standard unfolding. Given the residual token flows, computing all possibilities to append an event is a combinatoric problem. This problem is of similar complexity as the procedure of testing all computed co-sets of conditions in order to append an event in the standard unfolding. But the construction algorithm for PES1 may also have a worse runtime than the standard unfolding in some special cases, because the application of a consistency set in the prime event structure PES1 causes the construction algorithm to recalculate some events (as already mentioned above), that do not have to be recalculated in the standard unfolding algorithm working with a binary conflict relation. That means the absence of a local binary conflict relation in PES1 disables the possibility to directly classify an appended event to runs. Therefore some events have to be considered by the algorithm more than once to determine a classification of the events to certain runs. Altogether there are two opposite effects concerning the runtime of the construction algorithm of PES1 in comparison to the standard unfolding: Less events are considered, but some events have to be recalculated, whereas the first effect seems to be more significant.

For the sake of completeness, it is not exactly true that PES1 constructed in this way, coincides with PES0 except for neglecting (IDENT1)-identical events (as mentioned in the introduction). There is also a very specific auto-concurrency situation, in which PES1 neglects some further events compared to PES0. But these events only generate isomorphic processes. Therefore it is even desirable to neglect these events. It is algorithmically possible not to neglect these events, but this is more costly and therefore makes no sense. On the contrary, it is an interesting extension approach to neglect all events generating isomorphic processes. We considered such extension of our construction algorithm for PES1, but not implemented it yet. The removal procedure for respective events is relatively costly, but in some cases it may significantly reduce the size of the constructed prime event structure. Note that this topic only plays a role in certain auto-concurrency situations.

3 Algorithm 2

PES2: The second method constructs a labelled prime event structure PES2 enriched with example token flow tuples for every edge of the structure (see Figure 5). The intuition behind the causal dependency relation and the consistency set of the prime event structure as well as the token flows assigned to edges of the structure is the same as for PES1. In contrast to PES1, PES2 does not contain identical events. That means, only one event having a certain set of pre-events is introduced (except for concurrent events in one run), although there are different possible distributions of the token flows on ingoing edges of the event (compare the c-events in Figure 4 and Figure 5). Only one example distribution of these possible token flows is stored. That means, that abstracting from the token flows, each run occurs

only once in PES2, while PES1 contains one instance of a run for each possible token flow distribution of the run. This drastically reduces the number of events of PES2 in contrast to PES1, the difference may even be of exponential size. Moreover as in PES1, PES2 only contains events, for which a token flow distribution being positive on each ingoing skeleton arc exists.

Construction algorithm: A construction algorithm for PES2 is as follows: Introducing an initial event v, defining the first consistency set, constitutes the starting point of the algorithm. Events are then step by step appended in a certain order. The procedure to append an event is difficult: Given a consistency set, all prefixes of the set are candidates to append an event labelled by a certain transition. Methods from flow theory can be used to calculate, if an event having certain pre-events can be appended (see [JLD05]). These methods calculate an appropriate token flow, which is a quite complex procedure. The methods utilize an existing example token flow of the consistency set and yield a new token flow, which replaces the example token flow for further calculations. A further difficulty, that has to be regarded, is that skeleton arcs are not allowed to have a zero token flow in all possible token flow distributions. Finally having computed an event that should be appended, there exists the same problem already described in the construction algorithm of PES1, that a respective event was possibly already added in some previous stage of the algorithm. Then the event must not be duplicated. Therefore some events are calculated more than once by the algorithm.

Performance: As explained, in this algorithm it is obviously possible that the stored example distribution of the token flows is not appropriate to append some event to some prefix of a run, but some other possible token flow distribution would allow to append this event. Therefore, to append an event in the construction algorithm of PES2, first a prefix of some run has to be chosen and then the token flows of this run have to be recalculated. Choosing a prefix is a similar even slightly less costly operation as the combinatorial problems occurring in the construction algorithms of PES1 and PES0. But an elaborate recalculation of token flows is not necessary in the other two approaches. Fortunately, there are efficient methods for the recalculation. In [JLD05] a polynomial method to test, if there exists an appropriate token flow to append an event, is shown. This method works without any additional information, but having already stored an example token flow distribution of the run significantly improves the runtime of this method. The already existing example token flows can then be redistributed to extend the token flows to the additional edges connected with the new event. It is not clear if this method is more or less efficient than the standard unfolding algorithm or the first presented method. On the one hand the number of events of PES2 is significantly smaller than the number of events of PES1 (and therefore a lot smaller than the number of events of PES0), but on the other hand appending one event requires a redistribution of the token flows and is therefore very costly compared to the standard unfolding or PES1. Moreover using a consistency set instead of a binary conflict relation in PES2 yields the same runtime problem of the construction algorithm as explained in the last section for PES1: Some events have to be recalculated. Since the reduction of the number of events may be exponential and the method for redistributing the token flows is polynomial, the construction of PES2 should at least in some cases be more efficient than constructing PES1 or the standard unfolding.

4 Comparison of the Algorithms and Experimental Results

In this section we experimentally test our implementation of the construction algorithm of PES0 having the standard unfolding algorithm constructing PES1 as a benchmark. To construct PES1, we use an adapted version of the unfolding algorithm in [DJL03]. When interpreting the results, one has to pay attention that this unfolding algorithm is not completely runtime optimized, but the remaining improvement potential should be very limited. We compare the runtime and the size of the resulting event structures. The upper table in Figure 7 shows a test of the parameterized version of the example net of Figure 1 shown in Figure 6. We computed PES0 and PES1 for several arc weights. The lower table in Figure 7 shows a test of the lower net in Figure 6 modelling for example a coffee automata. In this net the initial marking is varied.

The experimental results indicate that our new unfolding approach is superior to the standard approach. For the tested examples, the runtimes and the sizes of the resulting structures are a lot better in our new algorithm. It is clear, that PES1 is at least as big as PES0, but usually a lot smaller, if the net contains arc weights or a non-safe initial marking. In these cases our new algorithm is also significantly faster than the standard algorithm. We did not check memory consumption. While the size of PES1 argues for the new algorithm, the need to store consistency sets increases the memory consumption of the new algorithm. This still has to be tested.

5 Conclusion

In this paper we propose two new unfolding semantics for p/t-nets based on the concept of prime event structures. The definitions of the two unfolding models are motivated by algorithmic aspects. We develop a construction algorithm for both unfolding models. We show that there are many cases in which our implemented algorithms is significantly more efficient than standard unfolding methods for p/t-nets. We also justify the applicability of the two newly developed unfolding methods.

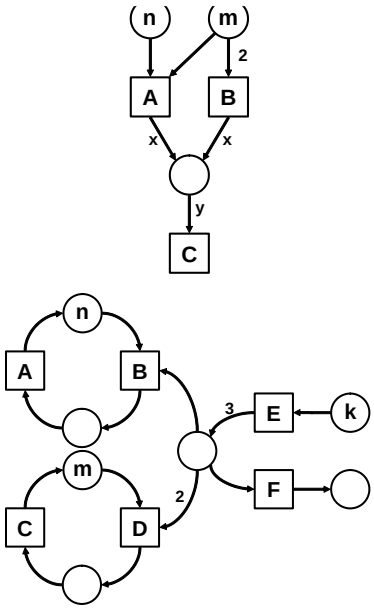

Figure 6: Parameterized test nets.

				PES0			PES1		
n	m	x	y	E	P	time	E	P	time
1	3	2	3	18	12	0,36s	4	2	0,08s
2	4	2	3	266	133	10,99s	14	6	0,05s
1	3	4	2	90	315	14,20s	10	3	0,04s
1	3	4	3	173	840	157,20s	8	6	0,05s
3	4	2	3	798	612	813,04s	21	10	0,06s
3	4	5	3	-	-	-	44	104	3,45s
	n	m	k						
	1	1	1	40	22	0,20s	12	6	0,09s
	1	2	1	46	28	0,63s	12	6	0,09s
	2	1	1	70	58	2,67s	16	9	0,13s
	2	2	1	76	67	4,63s	16	9	0,13s
	1	1	2	-	-	-	178	150	1,92s
	1	2	2	-	-	-	182	174	2,13s
	2	1	2	-	-	-	234	385	13,69s
	2	2	2	-	-	-	239	413	15,81s

Figure 7: Experimental results: E shows the number of events and P the number of processes in the constructed structure.

It could be interesting to adapt the two prime event structure unfolding methods presented in this paper in such a way that the nondeterminism can still be described by a binary conflict relation. Throughout this paper we discussed some reasons justifying that such approach could lead to an improvement of runtime as well as space consumption.

References

[BD87] Best, E. ; Devillers, R.R.: Sequential and Concurrent Behaviour in Petri Net Theory. In: *Theoretical Computer Science* 55 (1987), Nr. 1, S. 87–136

[CPW04] Couvreur, J.-M. ; Poitrenaud, D. ; Weil, P.: Unfoldings for General Petri Nets. In: *http://www.labri.fr/perso/weil/publications/depliage.pdf* University de Bordeaux I (Talence, France), University Pierre et Marie Curie (Paris, France) (2004)

[DJL03] Desel, J. ; Juhás, G. ; Lorenz, R.: *VipTool-Homepage.* 2003. – http://www.informatik.ku-eichstaett.de/projekte/vip/

[Eng91] Engelfriet, J.: Branching Processes of Petri Nets. In: *Acta Informatica* 28 (1991), Nr. 6, S. 575–591

[GR83] Goltz, U. ; Reisig, W.: The Non-Sequential Behaviour of Petri Nets. In: *Information and Control* 57 (1983), Nr. 2/3, S. 125–147

[Haa98] Haar, S.: Branching Processes of general S/T-Systems and their properties. In: *Electr. Notes Theor. Comput. Sci.* 18 (1998)

[HKT96] Hoogers, P.W. ; Kleijn, H.C.M. ; Thiagarajan, P.S.: An Event Structure Semantics for General Petri Nets. In: *Theoretical Computer Science* 153 (1996), Nr. 1&2, S. 129–170

[JLD05] Juhás, G. ; Lorenz, R. ; Desel, J.: Can I Execute My Scenario in Your Net?. In: Ciardo, G. (Hrsg.) ; Darondeau, P. (Hrsg.): *ICATPN* Bd. 3536, Springer, 2005 (Lecture Notes in Computer Science), S. 289–308

[MMS94] Meseguer, J. ; Montanari, U. ; Sassone, V.: On the Model of Computation of Place/Transition Petri Nets. In: Valette, R. (Hrsg.): *Application and Theory of Petri Nets* Bd. 815, Springer, 1994 (Lecture Notes in Computer Science), S. 16–38

[MMS97] Meseguer, J. ; Montanari, U. ; Sassone, V.: On the Semantics of Place/Transition Petri Nets. In: *Mathematical Structures in Computer Science* 7 (1997), Nr. 4, S. 359–397

[NPW81] Nielsen, M. ; Plotkin, G.D. ; Winskel, G.: Petri Nets, Event Structures and Domains, Part I. In: *Theoretical Computer Science* 13 (1981), S. 85–108

[Win86] Winskel, G.: Event Structures. In: Brauer, W. (Hrsg.) ; Reisig, W. (Hrsg.) ; Rozenberg, G. (Hrsg.): *Advances in Petri Nets* Bd. 255, Springer, 1986 (Lecture Notes in Computer Science), S. 325–392

Synthesizing Petri nets from LTL specifications – An engineering approach

Dirk Fahland*

Humboldt-Universität zu Berlin, Institut für Informatik,
Unter den Linden 6, 10099 Berlin,Germany,
fahland@informatik.hu-berlin.de

Abstract. In this paper we present a pattern-based approach for synthesizing truly distributed Petri nets from a class of LTL specifications. The synthesis allows for the automatic, correct generation of humanly conceivable Petri nets, thus circumventing a manual construction of nets, or the use of Büchi automata which are not distributed and often less intuitive to understand.

1 Introduction

Coming up with an initial operational model of a distributed system that fulfils all requirements of a (temporal-logic) specification is fairly difficult and time-consuming. Even if the specification language is well-founded and suits the application domain, like the *declarative service flow language*, DecSerFlow [9], that is based on LTL, the problem persists: The available synthesis algorithms, like those used in DecSerFlow, yield a correct but hardly conceivable, and non-distributed model [1].

A faithful modeling of workflows as mathematical abstractions of actual work procedures requires a notion of local tasks, local resources, and their causal and temporal dependencies. Such notions naturally occur in models of distributed computations like process algebras or Petri nets, which are successfully applied in this domain [3, 10]. A crucial aspect of a workflow model is that it can be understood by a person looking at, typically, its graph representation. This requirement enables experts in the domain of the workflow – like business processes or disaster management – to build and refine these models. This paper deals with the challenge of synthesizing such an understandable model from a temporal-logic specification.

We present a structurally compositional approach to synthesize Petri nets from a chosen class of LTL formulae. This class is generated from a set of predefined LTL constraints and closed under conjunction; as an example, we will use the constraints provided by DecSerFlow. Following the 'divide et impera' principle, we propose a behaviorally equivalent Petri net pattern for each such LTL constraint. A conjunction of constraints translates to a merge of the corresponding Petri net patterns.

* The author's work is funded by the DFG-Graduiertenkolleg 1324 "METRIK".

Due to the local nature of each pattern, the synthesized Petri net provides truly concurrent transitions and places wherever the LTL specification makes no statement about their ordering. The constraint-implementing Petri net patterns are very small and each net element has a meaningful interpretation in terms of the application domain. Therefore, the resulting Petri net is humanly understandable and can apprehensibly be refined and analyzed. We will explain our approach in more detail in Sect. 2. We conclude in Sect. 3 together with giving some hints on analysis, and discussing ways to generalize our approach.

2 Pattern-based synthesis of Petri nets from LTL specifications

We will use this section to explain the underlying principles of our synthesis algorithm for Petri nets from LTL specifications (or more precisely DecSerFlow specifications), and provide arguments for its correctness. Beforehand, we give a brief and rather informal introduction the domain of our synthesis, a class of LTL specifications, and we recall the codomain, an extended class of Petri nets.

LTL specifications. LTL is a modal logic that allows to specify the future of paths in terms of a set *Prop* of atomic propositions. The set $\mathcal{F}_{\mathrm{LTL}}(\mathit{Prop})$ of LTL formulas is the least set containing *Prop* and that is closed under boolean connectives, the unary temporal operators $\bigcirc$ (next-state), $\Box$ (always), $\Diamond$ (eventually) and the binary operator $\mathcal{U}$ (until). [7, 5]

We confine ourselves to a special subclass $\mathcal{F}_{\mathrm{WF}}(\mathit{Tasks})$ of LTL: Firstly, let *Prop* contain only propositions $\mathit{task} = A$ where A is the name of a task in a workflow; let *Tasks* be a finite and fixed set of such names. The proposition $\mathit{task} = A$ holds in a state if A is the next task to be executed. Secondly, let $\mathcal{F}_{\mathrm{WF}}$ be a finite set of parameterized LTL formulae, called *constraint templates* in the following. Each template $\psi \in \mathcal{F}_{\mathrm{WF}}$ comes with a vector $\mathbf{x}$ of parameters that can be instantiated, $\psi[\mathbf{x} \mapsto \mathbf{v}]$, to values $\mathbf{v}$ like task names or numbers. Our class $\mathcal{F}_{\mathrm{WF}}(\mathit{Tasks})$ of formulae under consideration is generated by instantiating constraint templates and is closed under conjunction. We evaluate a formula $\phi = \bigwedge_{i=1}^{n} \psi_i[\mathbf{x}_i \mapsto \mathbf{v}_i] \in \mathcal{F}_{\mathrm{WF}}(\mathit{Tasks})$ only on *feasible* paths in which in each state holds exactly one proposition. The *behavior* that is specified by ϕ is the set of all feasible paths that satisfy ϕ.

Petri nets. We found that a faithful synthesis of such formulae benefits from *inhibitor reset Petri nets*, IR nets for short [2]. (We assume the reader to be familiar with Petri nets. A detailed introduction can, for instance, be found in [8].) An IR net $N = (P, T, F, I, R, m^0)$ is a place/transition net that contains two additional kinds of arcs: If a place $p \in P$ is connected to a transition $t \in T$ by a *reset arc*, firing t removes all tokens from p while p does not restrict the firing of t. An *inhibitor arc* from p to t disables t if p is marked, firing t has no effect on p. In the following, the *behavior* of a Petri net is the set of firing sequences it generates.

Principles of the synthesis. The result of our synthesis operationalizes each task $A \in \mathit{Tasks}$ as a transition task_A. Their firing will be constrained according to the specification using a *parameterized Petri net pattern* $N(\psi)$ for each likewise parameterized constraint template $\psi \in \mathcal{F}_{\mathrm{WF}}$. For the moment assume such an behaviorally equivalent $N(\psi)$ for each template ψ. Instantiating a constraint, $\phi = \psi[\mathbf{x} \mapsto \mathbf{v}]$, then translates to instantiating the corresponding net, $N(\phi) = N(\psi)[\mathbf{x} \mapsto \mathbf{v}]$. We operationalize the conjunction of two formulae, $\phi_1 \wedge \phi_2$, by merging the nets, $N(\phi_1) \bowtie N(\phi_2)$, where the *merge* operator $\bowtie$ is formally the union of places, transitions, arcs and initial markings. The synthesis algorithm therefore computes for a given specification $\phi = \bigwedge_{i=1}^{n} \psi_i[\mathbf{x}_i \mapsto \mathbf{v}_i] \in \mathcal{F}_{\mathrm{WF}}(\mathit{Tasks})$ the net $N(\psi_1)[\mathbf{x}_1 \mapsto \mathbf{v}_1] \bowtie N(\psi_2)[\mathbf{x}_2 \mapsto \mathbf{v}_2] \cdots \bowtie N(\psi_n)[\mathbf{x}_n \mapsto \mathbf{v}_n]$.

In the remainder of this section, we will cover the three remaining issues of our approach: (1) a proper correspondence between the behavior given by an LTL formula and the behavior of a Petri net, (2) the implementation of the empty specification, and (3) the sound definition of $N(\psi)$ for each constraint template $\psi \in \mathcal{F}_{\mathrm{WF}}$.

Finite and infinite behavior, and liveness properties. The correspondence between the behavior of Petri nets and LTL formulae is straight forward: A state in which $\mathit{task} = A$ holds corresponds to the firing of transition task_A. The correspondence of a path and a firing sequence then follows.

In principle, LTL formulae are evaluated over infinite paths. However, LTL formulae can be divided into two classes: Firstly, the class of formulae that, if evaluated to *true* on an infinite path π, also evaluates to *true* on a finite prefix of π; this class includes safety properties and finite liveness properties. And secondly the class of formulae that are satisfied on infinite paths only; this class includes infinite liveness and infinite fairness properties that, for instance, demand infinitely many occurrences of a task. For the scope of this paper we restrict $\mathcal{F}_{\mathrm{WF}}$ to the first class. Then we may define the behavior of a formula $\phi \in \mathcal{F}_{\mathrm{WF}}(\mathit{Tasks})$ to be the set of all finite, feasible paths satisfying ϕ.

This allows us to define an *acceptance criterion* in the synthesized Petri net that accepts a finite firing sequence iff the corresponding path satisfies the original LTL formula. Therefore, we add the Petri net pattern N_{end} containing an empty place final together with its only pre-transition end, see Fig. 1(a). A firing sequence will be accepting iff the marking that is reached by this firing sequence has a token on final. The Petri net patterns $N(\psi)$ for $\psi \in \mathcal{F}_{\mathrm{WF}}$ will restrict the firing of end; for instance, a liveness property $\Diamond\, \phi$ will result in a pre-place of end that becomes marked once ϕ has been satisfied.

Implementing the empty specification. The empty LTL specification allows arbitrary behavior; more precisely, the empty specification is satisfied by any path evaluating atomic propositions from *Prop*. This means in our case that the empty specification is satisfied by arbitrary occurrences of tasks from the set *Tasks*. The Petri net that exhibits the same behavior is the net that contains,

for each $A \in \mathit{Tasks}$, a transition task_A together with a marked place pre_A being pre- and post-place of task_A. Let $N(\mathit{Tasks})$ denote this net, see Fig. 1(b).

The net $N(\mathit{Tasks}) \bowtie N_{\mathsf{end}}$ will be the 'canvas' for our synthesis algorithm. It provides the same behavior as the empty specification according to our acceptance criterion. Conjoining an LTL constraint ϕ translates in the Petri net to merging with $N(\phi)$ that restricts the firing of the transitions of our canvas according to ϕ. Hence, given a specification $\phi = \bigwedge_{i=1}^{n} \psi_i[\mathbf{x}_i \mapsto \mathbf{v}_i] \in \mathcal{F}_{\mathrm{WF}}(\mathit{Tasks})$, our synthesis computes the Petri net $N(\phi) =_{\mathrm{df}} N(\mathit{Tasks}) \bowtie N_{\mathsf{end}} \bowtie N(\psi_1)[\mathbf{x}_1 \mapsto \mathbf{v}_1] \bowtie N(\psi_2)[\mathbf{x}_2 \mapsto \mathbf{v}_2] \cdots \bowtie N(\psi_n)[\mathbf{x}_n \mapsto \mathbf{v}_n]$ in linear time.

Deriving Petri net patterns. The remaining issue in our synthesis algorithm is the definition of an implementing Petri net pattern $N(\psi)$ for each constraint template $\psi \in \mathcal{F}_{\mathrm{WF}}$. The patterns depend on the chosen class of template formulae and each needs to be derived manually – this is the engineering part in our approach. We will explain the definition of three patterns from DecSerFlow [9] and provide arguments for their correctness.

A safety property. We start with the constraint $\mathsf{precedence}(A, B) =_{\mathrm{df}} (\Diamond\ \mathit{task} = B) \rightarrow [\neg(\mathit{task} = B)\ \mathcal{U}\ \mathit{task} = A]$ which formalizes that if B is ever going to occur, it must be preceded by an A. An implementing Petri net pattern for this safety property must allow any non-violating path and prevent any violating path.

Fig. 1. The nets (a) N_{end}, (b) the building block of $N(\mathit{Tasks})$, and (c) $N(\mathsf{precendence})(A, B)$.

Our chosen solution, pattern $N(\mathsf{precedence})(A, B)$ in Fig. 1(c), relates the transitions task_A and task_B: We add an empty pre- and post-place $\mathsf{precedence}^B_A$ to task_B, to disable B initially. By making $\mathsf{precedence}^B_A$ a post-place of task_A, the occurrence of A enables B. From the logic's perspective, $\mathsf{precedence}^B_A$ gets marked iff the right-hand side of the implication has been satisfied. This gives rise to the *frame condition* of $N(\mathsf{precedence})(A, B)$ that no other pattern may change the tokens on $\mathsf{precedence}^B_A$. Please note that $\mathsf{precedence}^B_A$ is unbounded; we will discuss this aspect at the end of this section.

A liveness property. Our example of a liveness property is $\mathsf{response}(A, B) =_{\mathrm{df}} \Box\ (\mathit{task} = A \rightarrow (\Diamond\ \mathit{task} = B))$ which requires that any occurrence of A is followed by an occurrence of B. An implementing Petri net cannot prevent violating behavior of this liveness property (an occurrence of A) but must allow all behavior and provide a check for satisfaction (a subsequent occurrence of B).

The pattern $N(\mathsf{response})(A, B)$ in Fig. 2(a) does the job: We add an empty post-place $\mathsf{response}^B_A$ to task_A and add a reset arc (dashed arc) from $\mathsf{response}^B_A$

to task_B. Firing task_A will add a token to that place, firing task_B will remove all tokens. Logically, the place $\mathsf{response}_A^B$ is marked iff the implication evaluates to *false*. An inhibitor arc (arc with a black dot) from $\mathsf{response}_A^B$ to end allows end to fire only if the implication (and hence the entire constraint) is satisfied; this meets our acceptance criterion defined above. Again, the pattern's frame condition requires that no other pattern changes tokens on $\mathsf{response}_A^B$.

Counting and induction. The patterns $\mathsf{ex}_i(A) =_{\mathrm{df}} \Diamond\,(\mathit{task} = A \wedge \bigcirc\, \mathsf{ex}_{i-1}(A))$ and $\mathsf{ex}_0(A) = \mathit{true}$ request that task A occurs at least i times. An implementing pattern needs to represent the levels of recursion. Since it is a liveness property, it has to extend the pre-set of end.

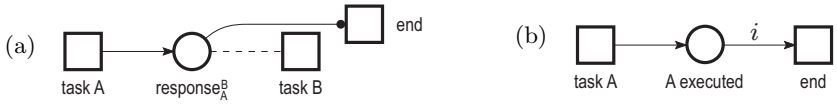

Fig. 2. The nets (a) $N(\mathsf{response})(A, B)$, and (b) $N(\mathsf{ex})_i(A)$.

We have chosen to add an empty post-place $\mathsf{executed}_A$ to task_A together with an arc of weight i from that place to end as shown in $N(\mathsf{ex})_i(A)$ in Fig. 2(b). We assume the frame condition that no other transition modifies the number of tokens on $\mathsf{executed}_A$. Then the number of tokens on that place is the number of occurrence of A and is also the number of recursion steps taken in the formula. Hence i firings of task_A yield i tokens which satisfies the constraint and enables end.

IR nets and unboundedness. The patterns that we explained above introduce unbounded places and use inhibitor and reset arcs. This is a consequence of our minimalistic approach to use exactly one transition per task. While it prohibits a direct application of standard analysis techniques, the synthesized Petri net is a structurally minimal blue-print for the implementation. By removing arcs like the loop in $N(\mathsf{precedence})(A, B)$ of Fig. 1(c), or by adding further net elements, the system can apprehensibly be refined. In [4], we defined a bisimilar place/transition net pattern for each IR net pattern; the synthesized Petri net can then be analyzed using standard techniques.

3 Conclusion and outlook

We sketched a linear-time synthesis algorithm for Petri nets from a well-defined subclass of LTL formulae. The synthesis merges small Petri nets, each implementing an LTL constraint that is a conjunct in a system specification. The synthesized Petri net is equivalent to the specification in the sense that each accepting firing sequence that marks the dedicated place final satisfies the specification and vice versa. The result of the synthesis is the starting point for refinement, analysis, and a correct implementation of the specification in terms

of a distributed system. We are not aware of another result that solves this problem. The work that comes closest to ours restricts a Petri net's behavior through merging with annotated Petri net patterns that implement constraints [6].

In [4], we demonstrate in a case study the automatic synthesis of a Petri net workflow model (11 transitions, 29 places) from a DecSerFlow specification and its analysis using model-checking. We also demonstrate how the analysis results can be related back to the specification level for refining the specification. This is made possible by our compositional approach: Each net element of the synthesized net was introduced by some Petri net pattern due to a specific constraint.

So far, we explained an engineering approach to synthesis as the modeler needs to provide a correct Petri net pattern for each LTL constraint that occurs in the specification. We have shown some patterns for constraints from the specification language DecSerFlow. The arguments for the correctness of the patterns suggest that our approach might carry further than on pre-chosen constraints and patterns only. There is a tight relation between transitions, places and markings on one side, and propositions and logical operators on the other side. We are currently searching for further correspondences between the logical connectives, and operations on Petri nets. Though, we expect that the synthesis might work only on a subclass of formulae, or may need further Petri net constructs, or that LTL is not the appropriate domain for this synthesis in general.

References

1. C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.
2. C. Dufourd, A. Finkel, and Ph. Schnoebelen. Reset nets between decidability and undecidability. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP'98*, volume 1443 of *LNCS*, pages 103–115, 1998.
3. S. Dustdar, J.L. Fiadeiro, and A.P. Sheth, editors. *Proceedings of BPM 2006*, volume 4102 of *LNCS*. Springer, 2006.
4. D. Fahland. Towards analyzing declarative workflows. In J. Koehler, M. Pistore, A.P. Sheth, P. Traverso, and M. Wirsing, editors, *Autonomous and Adaptive Web Services*, number 07061 in Dagstuhl Seminar Proceedings. IBFI, Schloss Dagstuhl, Germany, 2007.
5. D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, New York, NY, USA, 1980. ACM Press.
6. N. Lohmann, P. Massuthe, and K. Wolf. Behavioral constraints for services. In *Proceedings of BPM 2007*, LNCS. Springer-Verlag, September 2007. accepted.
7. A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.
8. W. Reisig. *A Primer in Petri Net Design.* Springer Compass International, 1992.
9. W.M.P. van der Aalst and M. Pesic. DecSerFlow: Towards a truly declarative service flow language. In *WS-FM*, pages 1–23, 2006.
10. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems.* MIT Press, 2002.

Bisher erschienen

Arbeitsberichte aus dem Fachbereich Informatik
(http://www.uni-koblenz.de/fb4/publikationen/arbeitsberichte)

Stephan Philippi, Alexander Pinl: Proceedings 14. Workshop 20.-21. September 2007 Algorithmen und Werkzeuge für Petrinetze 25/2007

Ulrich Furbach, Markus Maron, Kevin Read: CAMPUS NEWS – an Intelligent Bluetooth-based Mobile Information Network, Arbeitsberichte aus dem Fachbereich Informatik 24/2007

Ulrich Furbach, Markus Maron, Kevin Read: CAMPUS NEWS - an Information Network for Pervasive Universities, Arbeitsberichte aus dem Fachbereich Informatik 23/2007

Lutz Priese: Finite Automata on Unranked and Unordered DAGs Extented Version, Arbeitsberichte aus dem Fachbereich Informatik 22/2007

Mario Schaarschmidt, Harald F.O. von Kortzfleisch: Modularität als alternative Technologie- und Innovationsstrategie, Arbeitsberichte aus dem Fachbereich Informatik 21/2007

Kurt Lautenbach, Alexander Pinl: Probability Propagation Nets, Arbeitsberichte aus dem Fachbereich Informatik 20/2007

Rüdiger Grimm, Farid Mehr, Anastasia Meletiadou, Daniel Pähler, Ilka Uerz: SOA-Security, Arbeitsberichte aus dem Fachbereich Informatik 19/2007

Christoph Wernhard: Tableaux Between Proving, Projection and Compilation, Arbeitsberichte aus dem Fachbereich Informatik 18/2007

Ulrich Furbach, Claudia Obermaier: Knowledge Compilation for Description Logics, Arbeitsberichte aus dem Fachbereich Informatik 17/2007

Fernando Silva Parreiras, Steffen Staab, Andreas Winter: TwoUse: Integrating UML Models and OWL Ontologies, Arbeitsberichte aus dem Fachbereich Informatik 16/2007

Rüdiger Grimm, Anastasia Meletiadou: Rollenbasierte Zugriffskontrolle (RBAC) im Gesundheitswesen, Arbeitsberichte aud dem Fachbereich Informatik 15/2007

Ulrich Furbach, Jan Murray, Falk Schmidsberger, Frieder Stolzenburg: Hybrid Multiagent Systems with Timed Synchronization-Specification and Model Checking, Arbeitsberichte aus dem Fachbereich Informatik 14/2007

Björn Pelzer, Christoph Wernhard: System Description:“E-KRHyper“, Arbeitsberichte aus dem Fachbereich Informatik, 13/2007

Ulrich Furbach, Peter Baumgartner, Björn Pelzer: Hyper Tableaux with Equality, Arbeitsberichte aus dem Fachbereich Informatik, 12/2007

Ulrich Furbach, Markus Maron, Kevin Read: Location based Informationsystems, Arbeitsberichte aus dem Fachbereich Informatik, 11/2007

Philipp Schaer, Marco Thum: State-of-the-Art: Interaktion in erweiterten Realitäten, Arbeitsberichte aus dem Fachbereich Informatik, 10/2007

Ulrich Furbach, Claudia Obermaier: Applications of Automated Reasoning, Arbeitsberichte aus dem Fachbereich Informatik, 9/2007

Jürgen Ebert, Kerstin Falkowski: A First Proposal for an Overall Structure of an Enhanced Reality Framework, Arbeitsberichte aus dem Fachbereich Informatik, 8/2007

Lutz Priese, Frank Schmitt, Paul Lemke: Automatische See-Through Kalibrierung, Arbeitsberichte aus dem Fachbereich Informatik, 7/2007

Rüdiger Grimm, Robert Krimmer, Nils Meißner, Kai Reinhard, Melanie Volkamer, Marcel Weinand, Jörg Helbach: Security Requirements for Non-political Internet Voting, Arbeitsberichte aus dem Fachbereich Informatik, 6/2007

Daniel Bildhauer, Volker Riediger, Hannes Schwarz, Sascha Strauß, „grUML – Eine UML-basierte Modellierungssprache für T-Graphen“, Arbeitsberichte aus dem Fachbereich Informatik, 5/2007

Richard Arndt, Steffen Staab, Raphaël Troncy, Lynda Hardman: Adding Formal Semantics to MPEG-7: Designing a Well Founded Multimedia Ontology for the Web, Arbeitsberichte aus dem Fachbereich Informatik, 4/2007

Simon Schenk, Steffen Staab: Networked RDF Graphs, Arbeitsberichte aus dem Fachbereich Informatik, 3/2007

Rüdiger Grimm, Helge Hundacker, Anastasia Meletiadou: Anwendungsbeispiele für Kryptographie, Arbeitsberichte aus dem Fachbereich Informatik, 2/2007

Anastasia Meletiadou, J. Felix Hampe: Begriffsbestimmung und erwartete Trends im IT-Risk-Management, Arbeitsberichte aus dem Fachbereich Informatik, 1/2007

„Gelbe Reihe“

(http://www.uni-koblenz.de/fb4/publikationen/gelbereihe)

Lutz Priese: Some Examples of Semi-rational and Non-semi-rational DAG Languages. Extended Version, Fachberichte Informatik 3-2006

Kurt Lautenbach, Stephan Philippi, and Alexander Pinl: Bayesian Networks and Petri Nets, Fachberichte Informatik 2-2006

Rainer Gimnich and Andreas Winter: Workshop Software-Reengineering und Services, Fachberichte Informatik 1-2006

Kurt Lautenbach and Alexander Pinl: Probability Propagation in Petri Nets, Fachberichte Informatik 16-2005

Rainer Gimnich, Uwe Kaiser, and Andreas Winter: 2. Workshop "Reengineering Prozesse" – Software Migration, Fachberichte Informatik 15-2005

Jan Murray, Frieder Stolzenburg, and Toshiaki Arai: Hybrid State Machines with Timed Synchronization for Multi-Robot System Specification, Fachberichte Informatik 14-2005

Reinhold Letz: FTP 2005 – Fifth International Workshop on First-Order Theorem Proving, Fachberichte Informatik 13-2005

Bernhard Beckert: TABLEAUX 2005 – Position Papers and Tutorial Descriptions, Fachberichte Informatik 12-2005

Dietrich Paulus and Detlev Droege: Mixed-reality as a challenge to image understanding and artificial intelligence, Fachberichte Informatik 11-2005

Jürgen Sauer: 19. Workshop Planen, Scheduling und Konfigurieren / Entwerfen, Fachberichte Informatik 10-2005

Pascal Hitzler, Carsten Lutz, and Gerd Stumme: Foundational Aspects of Ontologies, Fachberichte Informatik 9-2005

Joachim Baumeister and Dietmar Seipel: Knowledge Engineering and Software Engineering, Fachberichte Informatik 8-2005

Benno Stein and Sven Meier zu Eißen: Proceedings of the Second International Workshop on Text-Based Information Retrieval, Fachberichte Informatik 7-2005

Andreas Winter and Jürgen Ebert: Metamodel-driven Service Interoperability, Fachberichte Informatik 6-2005

Joschka Boedecker, Norbert Michael Mayer, Masaki Ogino, Rodrigo da Silva Guerra, Masaaki Kikuchi, and Minoru Asada: Getting closer: How Simulation and Humanoid League can benefit from each other, Fachberichte Informatik 5-2005

Torsten Gipp and Jürgen Ebert: Web Engineering does profit from a Functional Approach, Fachberichte Informatik 4-2005

Oliver Obst, Anita Maas, and Joschka Boedecker: HTN Planning for Flexible Coordination Of Multiagent Team Behavior, Fachberichte Informatik 3-2005

Andreas von Hessling, Thomas Kleemann, and Alex Sinner: Semantic User Profiles and their Applications in a Mobile Environment, Fachberichte Informatik 2-2005

Heni Ben Amor and Achim Rettinger: Intelligent Exploration for Genetic Algorithms – Using Self-Organizing Maps in Evolutionary Computation, Fachberichte Informatik 1-2005