# Can I Execute My Scenario In Your Net? VipTool Tells You!

Robin Bergenthum[1], Jörg Desel[1], Gabriel Juhás[2], and Robert Lorenz[1]

[1]Lehrstuhl für Angewandte Informatik
Katholische Universität Eichstätt-Ingolstadt, Eichstätt, Germany
e-mail: name.surname@ku-eichstaett.de

[2]Faculty of Electrical Engineering and Information Technology
Slovak University of Technology, Bratislava, Slovakia
e-mail: gabriel.juhas@stuba.sk

**Abstract.** This paper describes the verification module (the VipVerify Module) of the VipTool [4]. VipVerify allows to verify whether a given scenario is an execution of a system model, given by a Petri net. Scenarios can be graphically specified by means of Labeled Partial Orders (LPOs). A specified LPO is an execution of a Petri net if it is a (partial) sequentialization of an LPO generated by a process of the net. We have shown in [2] that the executability of an LPO can be tested by a polynomial algorithm. The VipVerify Module implements this algorithm. If the test is positive, the corresponding process is computed and visualized. If the test is negative, a maximal executable prefix of the LPO is computed and visualized, together with a corresponding process and the set of those following events in the LPO which are not enabled to occur after the occurrence of the prefix. Further, the VipVerify Module allows to test in polynomial time whether a scenario equals an execution with minimal causality. A small case study illustrates the verification of scenarios w.r.t. business process models.

## 1 Introduction

Specifications of distributed systems are often formulated in terms of scenarios. In other words, it is often part of the specification that some scenarios should or should not be executable by the system. Given the system, a natural question is whether a scenario can be executed. Answering this question can help to uncover system faults or requirements, to evaluate design alternatives and to validate the system design.

There are basically two possibilities to express single executions of distributed systems, namely as sequences of actions (that means as totally ordered sets of action names) or as partially ordered sets of action names. Since sequences lack any information about independence and causality between actions, we consider executions (and scenarios) as partially ordered sets of action names in case of distributed systems.

There exist several software packages, developed at universities or software companies, which support the design and verification of distributed systems based on scenarios. Some of them allow to compute the unfolding of a distributed system (given as a Petri net, a communicating automaton or a process algebra) in order to run LTL and

CTL model checking algorithms on this unfolding (the tool PEP and the Model Checking Kit, [13–15]). Other tools use message sequence charts (MSCs) or their extension to live sequence charts (LCSs) to describe scenario-based requirements. These are used to guide the system design (the tool Mesa or the Playengine, [16–19]), for test generation and validation (the tool TestConductor integrated into Rhapsody, [20, 21]), or for the synthesis of SDL or statecharts models (the tool MSC2SDL, [22, 23]). In [24, 25] a verification environment is described in which LSCs are used to express requirements that are verified against a statemate model implementation, where the verification is based on translating LSCs into automata.

Up to now, there exists no tool support to verify a given scenario to be an execution of a distributed system. One reason might be that there were no efficient verification algorithms so far for this problem. In case a scenario is given as a labeled partial order (LPO) over the set of possible actions (events) and the distributed system is given as a (place/transition) Petri net, we presented in [2] a polynomial algorithm.

The notion of *executions* of Petri nets is based on their non-sequential semantics given by occurrence nets and processes [11, 12]. Abstracting from the conditions in a process gives an LPO, called *run*. Runs capture the causal ordering of events. Events which are independent can occur sequentially in any order. Thus, adding order to a run still leads to a possible execution. For example, occurrence sequences of transitions (understood as labeled total orders) sequentialize runs. Generalizing this relationship, an LPO which (partially) sequentializes a run is an execution of the net. The process represented by such a run is called *corresponding to the specified LPO* in the following.

If a specified LPO is an execution of a given Petri net, the mentioned algorithm computes a process corresponding to the LPO. In the negative case a maximal executable prefix of the LPO is computed as well as the set of those following events in the LPO, which are not enabled to occur after the occurrence of the prefix. We further deduced a polynomial algorithm to test if a specified LPO precisely matches a process w.r.t. causality and concurrency of the events in the specification, if this process represents a minimal ordering of events among all processes.

Actually, we implemented the above described algorithms as parts of the new VipVerify Module of the VipTool [3, 4, 7]. The algorithms are based on computing the maximal flow in a flow network [8]. While the maximal flow algorithm presented in [8] is only pseudo-polynomial in general, there came up strict polynomial algorithms running in cubic time (see e.g. [9]) and also faster (see [10] for an overview) during the last decades. Since the basic algorithm from [8] turns out to be strict polynomial (running in cubic time) in our special case, we started with an implementation of the algorithms based on this basic algorithm. Moreover, we added a graphical interface (the VipLpoEditor module) which allows the user to graphically specify scenarios of a given Petri net in terms of LPOs over the set of transition names of the Petri net.

The paper is further organized as follows: In Section 2 we present a description of the new modules of the VipTool. A simple case study illustrates the new functionalities in Section 3. Then, in section 4, we briefly describe how the new functionalities additionally fit into the existing validation and verification concept for business process models the VipTool supports. In Section 5 we present some performance results for the implemented algorithms. Finally, the conclusion outlines the future development.

## 2 Description of the New VipTool Modules

To support the new functionalities, the VipEditor provides three graphical submodules: In the existing VipNetEditor Petri net models of distributed systems can be designed. In the new VipLpoEditor the user can specify scenarios in terms of LPOs. Finally, processes (computed by the VipVerify Module) are visualized in the existing VipProEditor.

The VipNetEditor is only slightly revised compared to the last version of VipTool. Very briefly, it has the following main functionalities: Drawing and painting features can be used analogously as by any standard Windows application. Size, colors, fonts can by easily changed by the user for all draw elements such as places, transitions, arcs, labels etc. Furthermore, all standard editing features such as select, move, copy, paste etc. are implemented. Beyond that, for example automatic alignment and click-and-drag-points of net arcs are supported. Usual token game simulation is also a part of the VipNetEditor. Figure 1 shows a screen-shot of the VipNetEditor with an example of a simple Petri net model of a business process.

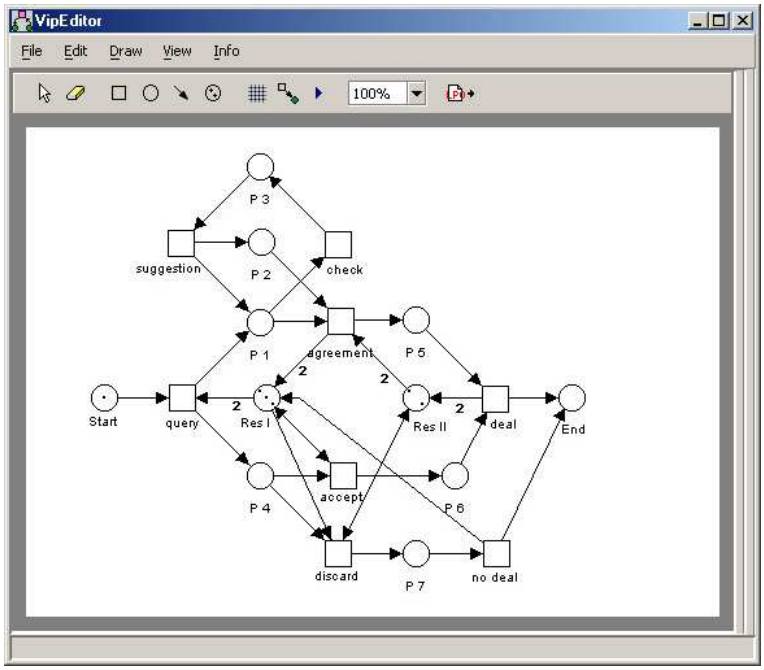


**Fig. 1.** Screenshot of the VipNetEditor, including an example net, which is explained later in a case study.

Given a Petri net in the VipNetEditor, the user may take advantage of the VipLpoEditor. Clicking the appropriate button splits the screen and the VipLpoEditor is available. A grid makes drawing the LPO easy. An arc is automatically added between two nodes

arranged on top of each other. For each added node, the related transition is chosen over a popup menu. For clarity, only the skeleton arcs of an LPO are drawn. Figure 2 shows a screen-shot of the VipEditor consisting of the VipNetEditor and the VipLpoEditor.

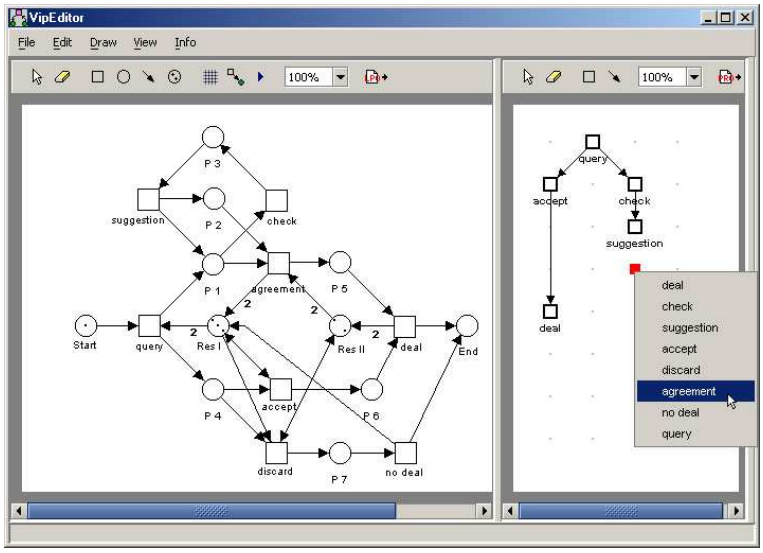


**Fig. 2.** Screenshot of the VipLpoEditor, showing the popup menu for adding a new node.

The last building block appears by starting the VipVerify Module to calculate if the LPO drawn in the VipLpoEditor is executable in the Petri net given in the VipNetEditor. The VipVerify Module distinguishes between the two following cases:

- If the LPO is executable, then a process corresponding to the LPO is computed by the VipVerify Module and is visualized in the VipProEditor. Moreover, then the VipVerify Module tests whether the LPO is minimal executable, i.e. whether there is another LPO with strictly less order between events which is also executable. If there is such another LPO, there are arcs in the given LPO representing an unnecessary ordering between events. Such arcs are highlighted.
- If the LPO is not executable, then the VipVerify Module computes a maximal executable prefix of the LPO and a process corresponding to this prefix. The process is visualized in the VipProEditor. In the VipLpoEditor, the prefix and the set of those events which are not concurrently enabled to occur after the occurrence of the prefix are highlighted by different colors.

Both cases will by described more precisely in the case study. The processes are visualized using the existing VipVisualizer module, which is based on the Sugiyama graph-drawing algorithm accommodated in [7]. Besides that, the objects of the visualized processes remain movable.

## 3 Functionality of the New VipTool Modules: A Case Study

In this section we briefly illustrate the functionality of the VipVerify Module and the VipLpoEditor by a simple case study.

The Petri net model of Figure 1 represents a possible business process of some company. The company handles their tasks with two resources (places *Res I* and *Res II*). After a prospective customer asks for a product (transition *query*) the business process divides into two concurrent sub-processes. In the upper one the company first checks on their offers or special conditions (transition *check*) and then makes offerings to the customer (transition *suggestion*). Yet they may agree (transition *agreement*). In the lower sub-process a parallel decision, for example on the solvency of the customer, has to take place. Only if that decision is positive, the customer and the company close a bargain (transition *deal*).

Figure 3 shows an LPO drawn in the VipLpoEditor, which represents a scenario that should be supported by the business process model. For this simple example it is easy to see that the specified scenario is minimal executable. By checking on the executability, a process corresponding to the scenario is computed and visualized in the VipProEditor. All nodes of the LPO are marked green.

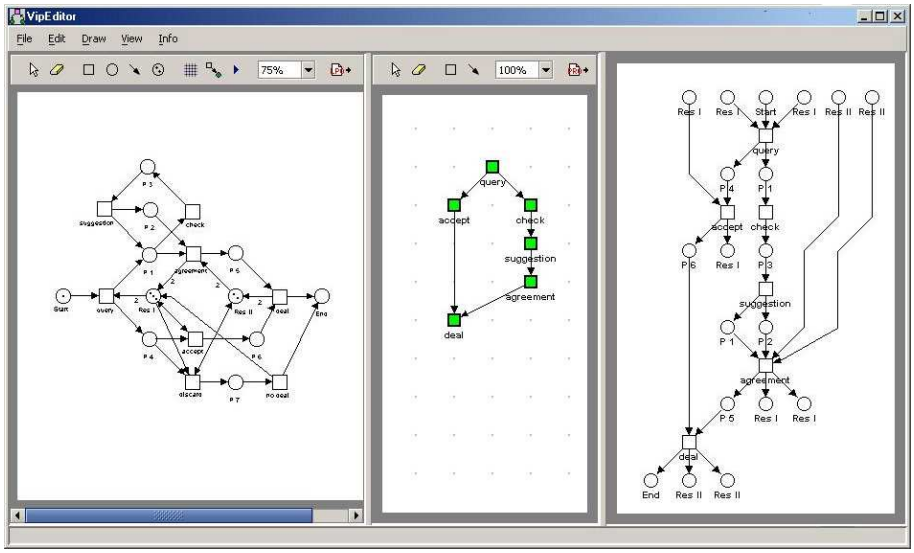


**Fig. 3.** The VipLpoEditor shows an executable LPO, and the VipProEditor shows a corresponding process of the Petri net.

Figure 4 shows another LPO which represents a scenario that should also be supported by the business process model. It is not an execution. In such a case, the VipVerify Module computes four other helpful contributions:

– A *maximal* executable prefix of the LPO. It is highlighted green and consists of the events *query*, *check* and *suggestion* in the example.
– The successor events which fail to be concurrently enabled after the occurrence of this prefix. They are highlighted red and are the events *agreement* and *discard* in the example.
– The place in the Petri net which does not carry enough tokens after the occurrence of the prefix to enable the red highlighted transitions. It is the place *Res II* in the example and highlighted red.
– A process corresponding to the executable prefix, which is visualized in the VipProEditor.

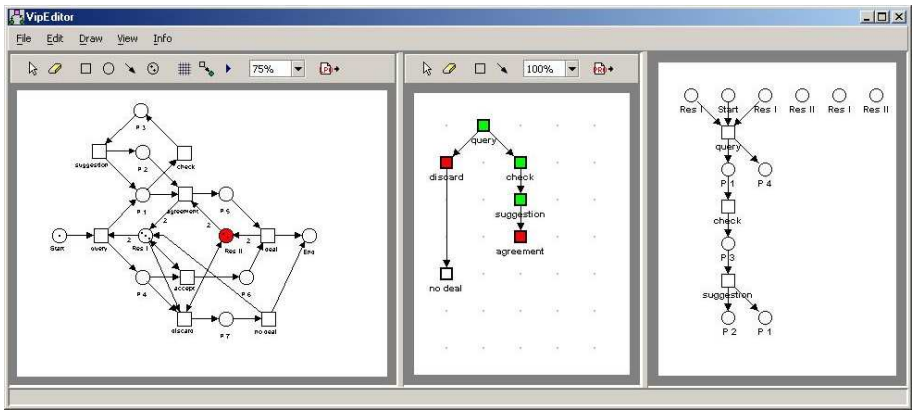


**Fig. 4.** Screenshot showing an LPO which is not executable in the given Petri net.

For the interpretation of this information we have to describe the verification algorithm more precisely. For this, we say an LPO *is an execution w.r.t. a place* $p$, if this LPO is an execution of the given Petri net restricted to the place set containing only $p$. An inductive procedure verifies separately for every place $p$ of the Petri net model, if the given LPO is an execution w.r.t. $p$. This is done by considering prefixes of the LPO increasing according to a calculated order over the set of nodes respecting the LPO. If for a place $p$ this procedure stops before all nodes were considered, a prefix of the LPO is computed

– which is an execution w.r.t. $p$, and
– whose extension by the set of its direct successor nodes is not an execution w.r.t. $p$.

This prefix serves as input-LPO of this procedure for the next place. Thus, if there are more places preventing the execution of the LPO, the VipVerify Module computes that place (highlighted red in the VipNetEditor) with the smallest corresponding executable prefix among all places (resp. one of them). Those direct successor nodes of

the prefix representing transitions which consume tokens from $p$ are highlighted red in the VipLpoEditor (and therewith all events preventing the executability of the LPO w.r.t. $p$). Those direct successor nodes of the prefix representing transitions which do not consume tokens from $p$ are highlighted yellow. Observe that possibly several of the red highlighted events are enabled after the occurrence of the prefix, but not all of them concurrently. Extending the prefix by such events would result in a bigger executable prefix, but with less information about the non-executability. The executable prefix corresponding to a place $p$ depends on the calculated total ordering of the LPO-nodes. Nevertheless it is maximal w.r.t. the red highlighted nodes: A prefix corresponding to $p$ computed w.r.t. another total ordering, which contains a red highlighted node, can not include the first prefix. Finally, a process corresponding to the prefix is visualized, showing a possible distribution of tokens among the pre- and post-conditions of the events of the prefix. In the example, that means:

- The prefix consisting of the events *query*, *check* and *suggestion* is an execution w.r.t. all places.
- After the occurrence of the events *query*, *check* and *suggestion*, the events *agreement* and *discard* are not concurrently enabled, since place *Res II* carries not enough tokens.
- Each of the events *agreement* and *discard* is enabled on its own and could be used to construct a bigger executable prefix. But then the user would get only the information that the event *agreement* (or, resp. *discard*) consumes too much tokens from place *Res II*, and not the information that the *combination of both events* needs too many tokens.

This gives the user clear information about how to change the model in order to support the scenario given by the LPO (namely how to reorganize the distribution of the resources), resp. about how the given distribution of the resources restricts the desired behavior.

## 4 Relating Old and New Functionalities of VipTool

In this section, we discuss, how the new implemented functionalities described in the previous two sections additionally fit into the existing validation and verification concept supported by VipTool. For this we briefly introduce this concept, but omit a detailed motivation, discussion and comparison to other approaches (here we refer to several publications from the last years ([3, 5–7]).

VipTool was originally developed at the University of Karlsruhe within the research project VIP [1] as a tool for modeling, simulation, validation and verification of business processes using Petri nets. It was implemented in the scripting language Python [7]. In [3] we presented a completely new and modular implementation in Java (using standard object oriented design) that allows to add extensions in a more flexible way.

The paper [6] proposes the following iterative validation procedure of Petri net models:

[1] Verification of Information systems by evaluation of Partially ordered runs

1. A requirement to be implemented is identified and formalized in terms of the graphical language of the model.
2. This formal specification is validated by distinction of those process nets that satisfy the specification from all other process nets. This way, the question "*what behavior is excluded by the specification?*" gets a clear and intuitive answer. The specification is changed until it precisely matches the intended property.
3. The valid specification is implemented, i.e. new elements are added to the model such that the extended model matches all previous and the new specifications. Obviously this step requires creativity and cannot be automated. However, again by generation and analysis of process nets it can be tested whether the extended model satisfies the specifications (actually, when all runs are constructed, this test can be viewed as a verification). At this stage, other verification methods can be applied as well.
4. If some requirements are still missing, we start again with the first item, until all specifications are validated and hold for the designed model.

VipTool supports all these four steps. In particular, process nets representing partially ordered runs of Petri net models are generated. They are visualized, employing particularly adopted graph-drawing algorithms. Specifications can be expressed on the system level by graphical means. Process nets are analyzed w.r.t. these specified properties. The distinction of process nets that satisfy a specification is supported. For the test phase, the simulation stops when an error was detected.

The new functionalities now complement the second and third step of the above described validation procedure as follows: First, for complex Petri nets it can be (too) time consuming to construct all processes. In such cases it is helpful to have the possibility not to check on all the processes by unfolding the Petri net, but to directly test a particular scenario to be a possible execution of the Petri net or not. Second, the user now can specify concurrency of events. If an LPO representing a desired behavior turns out to be not executable, then the user gets detailed information about the reasons by visualizing the first state of the system which does not enable a set of concurrent events of the LPO. This facilitates the creative step of changing the specification in the second step as well as changing the model in the third step. Finally, in the third step the user now can verify particular concurrent runs directly.

## 5 Experimental results

In this Section we test the performance of the presented algorithms by means of experimental results for the example instances $(N_{1,n}, lpo_{1,n})$ and $(N_{2,n}, lpo_{2,n})$ shown in Figure 5 with $n \in \mathbb{N}$ increasing. All experiments were performed on a Windows PC with 256 MByte of RAM and 1 GHz Intel Pentium III CPU. The times are measured in seconds. Table 5 shows the results for the following four procedures executed by the VipVerify Module: (A) Translation into the associated flow network, (B) Test whether $lpo_{i,n}$ is an execution of $N_{i,n}$, $i \in \{1, 2\}$, (C) Test whether $lpo_{i,n}$ is a minimal execution of $N_{i,n}$, $i \in \{1, 2\}$, and (D) Computation and Visualization of the corresponding process. The results show that the algorithms work better for LPOs with much concurrency between events. The weaker performance for LPOs with little concurrency is

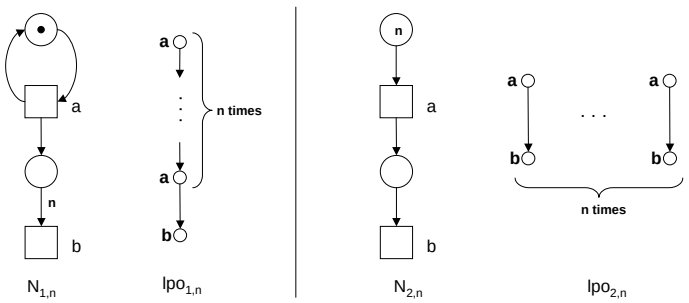


**Fig. 5.** Two place/transition Petri nets $N_{i,n}$, each together with an executable LPO $lpo_{i,n}$, $i = 1, 2$, dependent on the parameter $n \in \mathbb{N}$.

partially due to the quite general construction of the associated flow network, which could be further optimized.

| | $(N_{1,n}, lpo_{1,n})$ | | | | $(N_{2,n}, lpo_{2,n})$ | | | |
|---|---|---|---|---|---|---|---|---|
| **n** | **10** | **50** | **100** | **500** | **10** | **50** | **100** | **500** |
| **(A)** | 0.001 | 0.008 | 0.037 | 3.139 | 0.001 | 0.001 | 0.001 | 0.013 |
| **(B)** | 0.004 | 0.679 | 8.564 | 959.205 | 0.001 | 0.008 | 0.031 | 0.534 |
| **(C)** | 0.027 | 5.063 | 55.356 | – | 0.039 | 0.198 | 1.273 | 78.413 |
| **(D)** | 0.476 | 170.719 | – | – | 0.008 | 0.297 | 0.591 | 7.396 |

**Table 1.** Results of the executability test

## 6 Conclusion

We have presented the new VipVerify Module of the VipTool supporting scenario based verification of Petri net models of distributed systems and described its functionality within a small case study. VipVerify fits in the existing functionalities of the VipTool of supporting the step-wise design of business process models, employing validation of non-sequential specifications and verification of the model in each step. The further development of VipTool includes the following tasks:

- We plan to implement more efficient maximal flow algorithms underlying the presented verification algorithms.
- At present, VipTool only supports low level Petri nets. We plan to extend its functionalities to an appropriate restricted kind of predicate/transition nets.
- We are currently working on the synthesis of place/transition nets from given sets of LPOs. In [1] we present the first theoretical results which we plan to adapt for practical use and then to implement into VipTool.

We acknowledge the work of all other members of the VipTool development team, namely Niko Switek and Sebastian Mauser.

## References

1. Robert Lorenz and Gabriel Juhás. *Towards Synthesis of Petri nets from Scenarios*. Accepted for ICATPN 2006.
2. Gabriel Juhás, Robert Lorenz and Jörg Desel. *Can I Execute my Scenario in your Net?*. LNCS 3536, pages 289-308, 2005.
3. Jörg Desel, Gabriel Juhás, Robert Lorenz and Christian Neumair. *Modeling and Validation with VipTool*. LNCS 2678, pages 380-389, 2003.
4. VipTool-Homepage. *http://www.informatik.ku-eichstaett.de/projekte/vip/*.
5. J. Desel. *Validation of Process Models by Construction of Process Nets*. In [**?**], pages 110-128.
6. J. Desel. *Model Validation - A Theoretical Issue?*. LNCS 2360, pages 23-42, 2002.
7. T. Freytag. *Softwarevalidierung durch Auswertung von Petrinetz-Abläufen*. Dissertation, Karlsruhe, 2001.
8. L.R. Ford, Jr. and D.R. Fulkerson. *Maximal Flow Through a Network*. Canadian Journal of Mathematics 8, pages 399–404, 1955.
9. A.V. Karzanov. *Determining the Maximal Flow in a Network by the Method of Preflows*. Soviet Math. Doc. 15, pages 434–437, 1974.
10. A. Goldberg and S. Rao. *Beyond the Flow Decomposition Barrier*. Journal of the ACM 45/5, pages 783–797, 1998.
11. U. Goltz and W. Reisig. *The Non-Sequential Behaviour of Petri Nets*. Information and Control 57(2-3), pages 125-147, 1983.
12. U. Goltz and W. Reisig. *Processes of Place/Transition Nets*. LNCS 154, pages 264-277, 1983.
13. C. Schröter, S. Schwoon and J. Esparza *The Model-Checking Kit*. LNCS 2676, pages 463-472, 2003.
14. http://theoretica.informatik.uni-oldenburg.de/ pep/
15. http://www.fmi.uni-stuttgart.de/szs/tools/mckit/
16. D. Harel, H. Kugler and A. Pnueli. *Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements*. LNCS 3393, pages 309-324, 2005.
17. http://www.wisdom.weizmann.ac.il/ playbook/
18. http://www.inf.uni-konstanz.de/soft/tools_en.php?sys=1
19. H. Ben-Abdallah and S. Leue. *MESA: Support for Scenario-Based Design of Concurrent Systems*. LNCS 1384, pages 118-135, 1998.
20. M. Lettrari and J. Klose. *Scenario-Based Monitoring and Testing of Real-Time UML Models*. LNCS 2185, pages 317-328, 2001.
21. http://www.osc-es.de/products/en/testconductor.php
22. F. Khendek and X.J. Zhang. *From MSC to SDL: Overview and an Application to the Autonomous Shuttle Transport System*. LNCS 3466, pages 228-254, 2005.
23. N. Mansurov. *Automatic synthesis of SDL from MSC and its applications in forward and reverse engineering*. Comput. Lang. 27/1, pages 115-136, 2001.
24. W. Damm and J. Klose. *Verification of a Radio-Based Signaling System Using the STATEMATE Verification Environment*. Formal Methods in System Design 19/2, pages 121-141, 2001.
25. J. Klose and H. Wittke. *An Automata Based Interpretation of Live Sequence Charts*. LNCS 2031, pages 512-527, 2001.