Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Master Thesis

Alexander Schulz

Model Checking of Reconfigurable Petri Nets

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Alexander Schulz

Model Checking of Reconfigurable Petri Nets

Master Thesis eingereicht im Rahmen der Masterprüfung

im Studiengang Master of Science
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Julia Padberg
Zweitgutachter: Prof. Dr. Michael Köhler-Bußmeier

Eingereicht am: 30. April 2015

Alexander Schulz

Thema der Arbeit
Model Checking of Reconfigurable Petri Nets

Stichworte
Rekonfigurierbare Petri-Netze, Maude, Bisimulation, Model Checking

Kurzzusammenfassung
Ein wichtiges Ziel der theoretischen Informatik ist die Entwicklung formaler Methoden, die es erlauben die Qualität der zu entwickelnden Software zu verbessern. Eigenschaften wie Lebendigkeit, Verklemmungen und Sicherheiten können für ein gegebenes Modell nachgewiesen werden. Hierfür eignet sich das Modellieren mit Petri-Netzen als eine etablierte wissenschaftliche Technik besonders gut. Basierend auf Petri-Netzen, erweitern rekonfigurierbare Petri-Netze die Netze um eine Menge von Regeln, die genutzt werden um das Netz dynamisch zu verändern.

Bisher fehlt die Möglichkeit der Verifizierung von rekonfigurierbaren Petri-Netzen. Diese Thesis beschreibt die Überführung von rekonfigurierbaren Petri-Netzen zu einem Maude Netz. Ziel dieser Master Thesis ist der Nachweis der Korrektheit des Maude Netzes.

Alexander Schulz

Title of the paper
Model Checking of Reconfigurable Petri Nets

Keywords
Reconfigurable Petri nets, Maude, bisimulation, Model checking

Abstract
One important aim of theoretical computer science is model checking to improve the software quality. Properties such as liveness, deadlock or safety can be verified for a given model. Modelling with Petri nets is a typical technique because it is well understood and can be used for model checking. Reconfigurable Petri nets are extending the concept of Petri nets with a set of rules that can be used dynamically to change the net.

The possibility to verify a reconfigurable Petri net and properties such as deadlocks or liveness is non-existent. The aim of this thesis is the proof of correctness of a Maude net.

Contents

List of Figures

Listings

List of Tables

List of definitions

List of lemmas and theorems

1 Introduction

The fundamentals of many daily routines are software-controlled devices used in the areas of public transport, security systems, and banking terminals. In this context, cost efficiency and fault resistance are the main challenges that must be solved. In order to prevent faults, a technique is needed to combine formalization, algorithms, and tools in comparison with other standard tests that show only the absence of faults. Model checking is a solution for computer-based systems that colligate a formal verification technique for the behavioural properties of a given system by inspecting all model states. Specifications given as formulae are proven against a model that is used as the base, where the correctness of the model is essential [1].

1.1 Aim of this Thesis

Reconfigurable Petri nets are well-established models for concurrent and non-deterministic behaviour models [4–6]. Their concurrent behaviour are suitable for complex systems that describe dynamic structures, including changes by rules at runtime. For users it is difficult to determine whether some properties emerge due to the non-deterministic and concurrent behaviour. Therefore, this thesis aims to develop an approach for the model checking of a reconfigurable Petri net. This approach includes the proof of model correctness so that the verification is applicable.

Maude's well-established theory of rewriting logic is suitable for reconfigurable Petri nets due to the unified model of concurrency, which is particularly interesting for the concurrent model of reconfigurable Petri nets [7, 8]. Definitions of P/T nets, coloured Petri nets, and algebraic Petri nets are defined in [9] in a manner that makes Maude a suitable basis for the definition of a Maude net that models the net and rules of a reconfigurable Petri net. Finally, Maude includes a implementation of model checking by its linear temporal logic of rewriting (LTLR) module[1].

[1] `http://maude.cs.illinois.edu/tools/tlr/`, 24 March 2015

Based on the algebraic model of the previous work presented in [3], this thesis ensures the correctness of the conversion as well as the bisimulation between a reconfigurable Petri net and a Maude net. The purpose is to guarantee that the given Maude net is applicable for the related reconfigurable Petri net so that the verification process can return valid results. This is achieved through definitions and proofs of the conversion between a reconfigurable Petri net and a Maude net, as well as the bisimulation between the reachability graph for the reconfigurable Petri net and the related search tree of the Maude net. Hence, the rewriting logic is formally defined and combined with the theory of reconfigurable Petri nets (for more technical point of views, see [3], or the Maude modules in appendix B). Those functions that convert all parts such as places, transitions, pre- and post-domains, or markings of a reconfigurable Petri net into a Maude net are defined. With this result, all the functions can be bundled into a main conversion by Theorem 1. Resting on both theories, Theorem 2 introduces the bisimulation for both transition systems derived from the related definitions in Def. 15 and Def. 3. Such bisimulation is based on the firing and transforming steps of a reconfigurable Petri net and the search tree of Maude, which are mapped with a function. The resulting proof of such bisimulation implies that both theories are behaviourally equivalent.

The algebraic approach of a Maude net leads to the proof of properties, such as deadlock freeness, by Maude's model checking with the LTLR-module. Additionally, the model enables the implementation of extension as, for example, the reachability graph. The Maude net implementation is thus used as a basis for more implementations which can be used to improve the usability of net designs and which abides by required properties.

1.2 Outline

The following chapters can be divided into the background with all relevant theories such as linear temporal logic and bisimulation. Next, the definition of a transition system, including bisimulation and Maude is given. This chapter ends with an introduction to reconfigurable Petri nets and related works such as fundamental Petri net writing in Maude or more complex – for example, coloured Petri nets. Then, the ReConNet model checker and the related reachability graph are introduced. Next, the chapter labelled transition systems contains the definition of related labelled transition systems for a reconfigurable Petri net and a Maude net. The chapter on the correctness of model checking for Maude contains the central content with all definitions and proofs for the correctness of the conversion as well as the bisimulation. Based on this, the evaluation chapter contains an example, including the results for a reconfigurable Petri net, that has been proven against Charlie[2]. Finally, this thesis provides the basis for future works such as graphically animated counterexamples, and a summary of all collected results and a differentiated conclusion is also given.

The appendices include detailed examples of the Maude search tree as well as the Maude source code of the example net and rules.

[2] `http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Charlie`, March 2015 24

2 Background

This chapter introduces the formal definition of reconfigurable Petri nets (with respect to well-known Petri nets). It also shows the corresponding implementation, which is called *ReConNet* [10], as well as the rewrite logic and the application *rMC*. *rMC* converts a reconfigurable Petri net into a Maude net (see [3, 11]). Finally, it includes related works, such as Petri net definition or more complex coloured Petri nets in Maude, as well as previous publications dealing with reconfigurable Petri nets.

2.1 Temporal Logic

Temporal logic is an extension by time for inference logic. It contains the logic for specifying and the techniques for reasoning time-based problems in fields of philosophy and computer science. The time-based relations between moments are used by the before-afterwards relationships so that „He is always happy!“ can be reasoned by a single path with linear time logic or branching paths with computation tree logic [12]. Both logics are used to verify computer-based systems against a given specification. An application of verification is implemented by model checking in Maude, where a state/event-based extension is used to verify the rewrite steps.

Model checking of reconfigurable Petri nets using Maude is defined in [3, 11]. Both papers use a conversion to transmit a net and a set of rules into a Maude net that can be used for linear temporal logic (LTL) model checking with the module LTLR[1]. The LTL model-checking module contains all the usual operators, such as *true, false, conjunction, disjunction* and *not*, and complex operators with the next-operator being written with $O\ \phi$ or the until-operator notated with $\psi\ U\ \phi$. Further, it supports release-operator statements, such as $\psi\ R\ \phi$ that are internally converted into $\phi\ until\ \psi$. Finally, it defines the future-operator that is written with $\Diamond\ \phi$ that ϕ is possible in the future, whereby the global-operator that is written with $\Box\ \phi$ claims that ϕ is true in all states. All the operators are summarized in the following graphic, using a suitable path description.

[1] Linear temporal logic for rewrite (LTLR) with extensions for rewrite rules and properties such as fairness: `http://maude.cs.illinois.edu/tools/tlr/`, 4 March 2015

$O\ \phi$: $\bullet \longrightarrow \phi \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet$

$\psi\ U\ \phi$: $\psi \longrightarrow \psi \longrightarrow \psi \longrightarrow \phi \longrightarrow \bullet \longrightarrow \bullet$

$\psi\ R\ \phi$: $\phi \longrightarrow \phi \longrightarrow \phi \rightarrow \phi, \psi \rightarrow \bullet \longrightarrow \bullet$

$\phi \longrightarrow \phi \longrightarrow \phi \longrightarrow \phi \longrightarrow \phi \longrightarrow \phi$

$\Diamond\ \phi$: $\bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \phi \longrightarrow \bullet$

$\Box\ \phi$: $\phi \longrightarrow \phi \longrightarrow \phi \longrightarrow \phi \longrightarrow \phi \longrightarrow \phi$

Figure 1 shows a subsumption of the main LTL verification process for a formula ϕ and a given reconfigurable Petri net written in Maude. Both parts are translated into Büchi automatons and combined, afterwards, into a product automaton that is finally proven against emptiness.

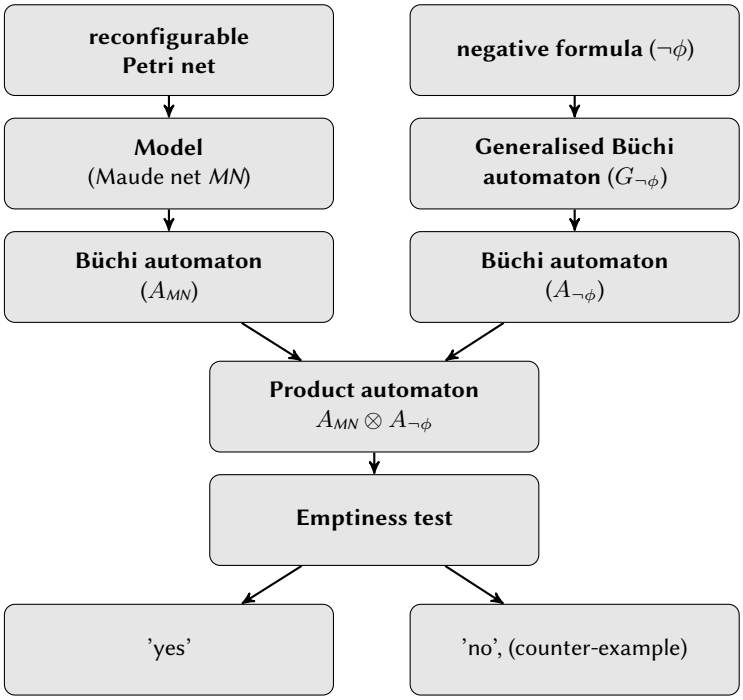


Figure 1: LTL model checking of a Maude net, adjusted from [1, P. 292]

2.2 Bisimulation of the Transition Systems

According to [2, 13, 14], bisimilar refers to the behaviour equivalence of two systems. In contradistinction to bisimilarity, can trace equivalence systems not determine decisions in the system. An example of two systems that demonstrate the issue of trace equivalence is shown in Figure 2. Initially, both systems can choose an a action. System S is now in state s_1 and can choose between b or c actions. System T is in state t_1 or t'_1, where no action can be selected. As a result, both systems exhibit trace equivalence but are not bisimilar due to the decision.

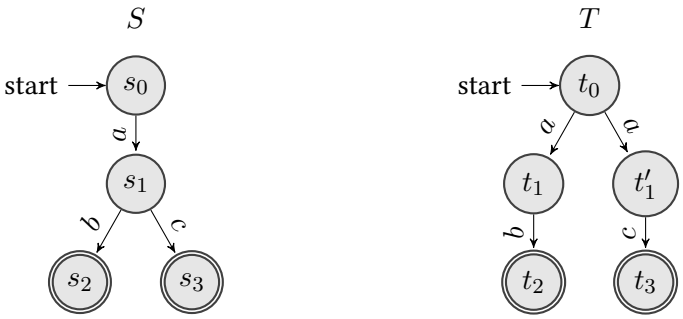


Figure 2: Two trace equivalent but not bisimilar systems S and T due to action a (adjusted from [2])

In formal terms the example in Figure 2 can be expressed as a transition system of Def. 1, where all states are combined in a set S for states, while transition relation *tr* combines two states and an action to a flow.

Definition 1 (Transition system (TS))**.** *A transition system (TS) consists of a three-tuple* $TS = (S, A, tr)$ *with a set* S *of states, a set* A *of actions and transition relations* $tr \subseteq S \times A \times S$.

Based on such a transition system, a bisimulation is defined by Def. 2. A relation B is used to combine two states of two transition systems if they satisfy both conditions for all outgoing actions.

Definition 2 (Action-based bisimulation of TS [14, 15]). *Given two TS_i with $i \in 1, 2$ and $TS_i = (S_i, A_i, tr_i)$ is an action-based bisimulation defined by a binary relation $\mathcal{B} \subseteq S_1 \times S_2$, which is constructed by:*

$$\forall (s_1, r_1) \in \mathcal{B} \text{ with action } a \in A$$
$$\forall s_2 \text{ with } s_1 \xrightarrow[1]{a} s_2 \Rightarrow \exists r_2 \in S_2 : r_1 \xrightarrow[2]{a} r_2 \wedge (s_2, r_2) \in \mathcal{B}$$
$$\forall r_2 \text{ with } r_1 \xrightarrow[2]{a} r_2 \Rightarrow \exists s_2 \in S_1 : s_1 \xrightarrow[1]{a} s_2 \wedge (s_2, r_2) \in \mathcal{B}$$

If B is a binary relation between TS_1 and TS_2, then there exists a bisimulation $TS_1 \leftrightarrows TS_2$.

2.3 Maude

Developed mainly at the Stanford Research Institute International (SRI International), Maude is a well-known implementation of equation and rewriting logic [16, 17]. As a base, it uses a powerful algebraic language for models of a concurrent state system. An extension of Maude is the linear temporal logic for rewrite (LTLR) module that can be used to test defined modules with LTL properties, such as deadlocks [1, 18, 19].
Implementations in Maude are based on one or many modules, where each is an abstract data type (ADT). Further, each module is based on types, which are declared with the keyword „sort“ for a single sort or for more with „sorts“. Hence, some types for a Petri net can be described with:

```
sorts Places Transitions Markings .
```

Depending on a given set of sorts, the operators can be defined. The operators describe all functions that are needed to work with the defined types so that, for example, a multiset of markings can be expressed with a *whitespace*-functor. Placeholders, denoted by a underscore, are used for the types after the double point, and finally, the return type is given by the type right to the arrow. The following example is based on the above type of declaration:

```
op _ _ : Markings Markings → Markings .
```

If an operator is associative or commutative, it can be written with keywords such as „assoc“ and „comm“. These keywords are defined at the end of a line so that a multiset of markings can be extended by such properties by:

```
op _ _ : Markings Markings → Markings [assoc comm] .
```

The axioms are expressed by the equation logic of Maude, which defines the validity for the given operators. For example, the initial marking of a Petri net can be exemplified with the `initial` operator. As the validity can be written with an equation, where, for example, a given Petri net of Figure 3 is defined with a token on „A“, these three lines follow:

```
op initial : → Markings .
ops A B : → Markings .
eq initial = A .
```

In summary, types defined are defined as `sort`, operators as functors `op`, and equations `eq` as the validity of the related operators. Based on such definitions, the rewrite rules can be used to replace one multiset with another. As usual in a functional language, all the terms are immutable so that a A-term can be replaced by a rule with a B-term:

```
rl [T] : A ⇒ B .
```

Based on the above-mentioned definitions, the example in Figure 3 is a graphical representation of the implementation. The rule implements the token game of Petri nets, where two multisets of the rule `T` can be seen as the pre- and $post$-set for a transition so that these rules describe a firing step.

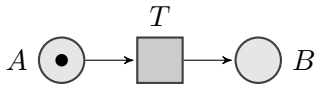


Figure 3: Example Petri net for the Maude introduction

The internal representation of Maude is shown in [20] as the labelled rewrite theory R. R is a four-tuple (Σ, E, L, R) with Σ being an alphabet of functions, a set of equations E over Σ, a set of labels L and a set of relation pairs $R \subseteq L \times (T_{\Sigma,E}(X)^2)$ that consists of a label and a pair of terms.

The rewrite rules of R can be understood as a labelled sequence with the notation $r : [t]_E \rightarrow [t']_E$. The semantics should be read as $[t]_E$ *becomes* $[t']_E$. Further, a rule can be extended with variables $\{x_1, \ldots, x_n\}$ for each term, so that r can be written with $r : [t(x_1, \ldots, x_n)]_E \rightarrow [t'(x_1, \ldots, x_n)]_E$ (or, in short, $r : [t(\bar{x}^n)]_E \rightarrow [t'(\bar{x}^n)]_E$). The following deduction rules in Def. 3 can be applied if R is defined.

Definition 3 (Rewrite theory of Maude [20]). *Deduction rules*

1. *Reflexivity, for each* $[t] \in T_{\Sigma,E}(X)$

$$\frac{}{[t] \to [t]}$$

2. *Congruence, for each* $f \in \Sigma_n, n \in \mathbb{N}$

$$\frac{[t_1] \to [t'_1] \dots [t_n] \to [t'_n]}{[f(t_1, \dots, t_n)] \to [f(t'_1, \dots, t'_n)]}$$

3. *Replacement, for each rewrite rule* $r : [t(x_1, \dots, x_n)] \to [t'(x_1, \dots, x_n)]$ *in* R

$$\frac{[w_1] \to [w'_1] \dots [w_n] \to [w'_n]}{[t(\bar{w}/\bar{x})] \to [t'(\bar{w}'/\bar{x})]}$$

For the sake of completeness, Def. 4 introduces all the defined rules of [20]. Transitivity and symmetry modulo equations are also part of the deduction rules defined for Maude.

Definition 4 (Rewrite theory of Maude, including equations [20]). *Extended deduction rules for equations*

4. *Transitivity*

$$\frac{[t_1] \to [t_2] \quad [t_2] \to [t_3]}{[t_1] \to [t_3]}$$

5. *Symmetry (modulo a set of axioms* E*)*

$$\frac{[t_1] \to [t_2]}{[t_2] \to [t_1]}$$

2.4 Reconfigurable Petri Net

One of the most important models for concurrent systems and some software engineering parts are Petri nets, which are based on Carl Adam Petri's dissertation [21]. Petri's thesis combines states and actions in one model that is exemplarily defined as a marked Petri net in Def. 5. An extended variety of a Petri net is a reconfigurable Petri net, as in Def. 6, which combines modification rules with a net.

Definition 5 (Marked Petri net N [22, 23]). *A marked Petri net N can be formally described as a tuple by*

$$N = (P, T, pre, post, cap, p_{name}, t_{name}, M):$$

- P *is a set of places*
- T *is a set of transitions*
- $pre : T \to P^{\oplus}$ *is a function used for all* pre*-domains of each transition*
- $post : T \to P^{\oplus}$ *is a function used for all* $post$*-domains of each transition*
- $cap : P \to \mathbb{N}_{+}^{w}$ *assigns for each place a natural number as capacity*
- $p_{name} : P \to A_P$ *is a label function for places*
- $t_{name} : T \to A_T$ *is a label function for transitions*
- M *is a set of tokens by* $M \in P^{\oplus}$

Remark 1. *The set of token is also defined as* $M : P \to \mathbb{N}$ *and the capacity by* $cap : P \to P^{\oplus}$

Reconfigurable Petri nets are based on Petri nets and are significant due to the fact that they can modify themselves with a set of rules [24–26]. Providing a base is the following Def. 6, which is used for the conversion process. For elaborate definitions, [6] contains definitions for negative application conditions (NACs) and functions that change labels with *rnw* and *tlb*.

Definition 6 (Reconfigurable Petri net *RPN* [6, 27]). *A reconfigurable Petri net can be described as a tuple of a reconfigurable Petri net* $RN = (N, \mathcal{R})$ *by*

- N *is a Petri net*
- $\mathcal{R}$ *is a set of rules*
- $r \in \mathcal{R}$ *is defined by* $r = (r_{name}, L \leftarrow K \rightarrow R)$*, where* L *is the left-hand side, which needs a morphism to be mapped to a net* N*.* K *is an interface between* L *and* R*.* R *is the part that is inserted into the original net and* $A_R = \bigcup\{r_{name}\}$ *with* $(r_{name}, L \leftarrow K \rightarrow R) \in R$*.*

Remark 2. *The initial state of a reconfigurable Petri net is given as* $(N_0, \mathcal{R})$ *with* $N = (N_0, M_0)$ *and* $N = (P_0, T_0, pre_0, post_0, M_0)$ *so that* $[(N_0, M_0)]$ *is the initial state in* LTS_{RPN}.

One possible action of a reconfigurable Petri net is the firing step by a transition t. Def. 7 defines that t is enabled if there are enough tokens on all *pre*-domain-related places and if the *post*-domain satisfies the capacity restriction for all related places. Then, the resultant marking can be calculated by the current marking minus the *pre*-domain, plus the *post*-domain, for the transition t.

Definition 7 (Transition firing [6])**.** *A transition* $t \in T$ *is enabled, if its pre-domain is less or equal than M and the resulting marking is less or equal than the capacity for each place. The resultant marking is calculated by the current marking minus the pre-domain plus the post-domain.*

$$pre^{\oplus}(t) \leq M$$
$$M + post^{\oplus}(t) \leq cap$$
$$M' = (M \ominus pre^{\oplus}(t)) \oplus post^{\oplus}(t).$$

A transformation step for a reconfigurable Petri net (N, M) to (N', M') is defined in Def. 8. It defines different states for isomorphic nets when there are varied markings or labels. As a result, there is no isomorphism between both nets in the example of Figure 13.

Definition 8 (A transformation step in RPN [6])**.** *A place-respecting transformation step in the reconfigurable Petri net is given by*

$$\begin{array}{ccccccc}
(L, m_L) & \xleftarrow{l} & (K, m_K) & \xrightarrow{r} & (R, m_R) & & \\
\downarrow{\scriptstyle o} & (PO) & \downarrow & (PO) & \downarrow & & \\
(N, m) & \xleftarrow{l'} & (D, m_D) & \xrightarrow{r'} & (O, m_O) & \xrightarrow{f} & (N', m')
\end{array}$$

Definition 9 (Isomorphism classes of nets)**.** *Isomorphism classes of nets:* $[(\overline{N}, \overline{M})] = \{(N, M) \mid (N, M) \cong (\overline{N}, \overline{M})\}$[2]

[2] The isomorphism class is compatible to firing and transformation steps

An example net N_1 is illustrated in Figure 4, which contains two places and transitions as well as two tokens. The net is enabled to fire with the initial marking. Both transitions T are activated and, after two fire steps, both tokens are moved to the other place *P*. From now on, the net is in a state of deadlock and cannot fire unless a rule is used. Such a rule r_1 is shown in the lower bar in Figure 7 and explicitly in Figure 11, where a transition is replaced with another with an inverse arc direction. After using r_1, the net is enabled and can fire again until the rule ceases to be applied.

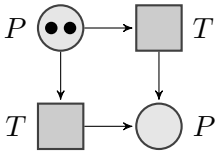


Figure 4: Example Petri net N_1

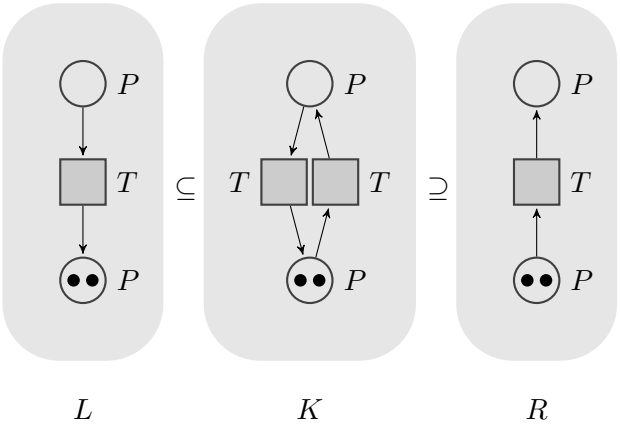


Figure 5: Example rule r_1, which changes the arc direction

Based on Def. 5, the net N_1 can be formally written as in Figure 6. Both places are in the set P and both transitions in T. All transitions are described with the *pre*- and *post*-domains, and the initial marking is defined with m and the capacity for each place with *cap*.

$$
\begin{aligned}
P &= \{P_1, P_2\} \\
T &= \{T_3, T_4\} \\
pre(T_3) &= P_1 \\
pre(T_4) &= P_1
\end{aligned}
\qquad
\begin{aligned}
post(T_3) &= P_2 \\
post(T_4) &= P_2 \\
M_0 &= P_1 + P_1 \\
cap(P_1) &= \omega \\
cap(P_2) &= \omega
\end{aligned}
$$

Figure 6: Formal description of a Petri net (for a graphical presentation, see also Figure 4)

ReConNet, as published in [10], is an implementation of the reconfigurable Petri net in Def. 6. An example net N_1 and a rule r_1 are shown in Figure 7, where N_1 contains two places and transitions. Both transitions convey a token from the upper place to that below. If all tokens are consumed, the net is in a deadlock that can only be solved with the rule r_1, which changes the direction of firing with a replacement of the transition.

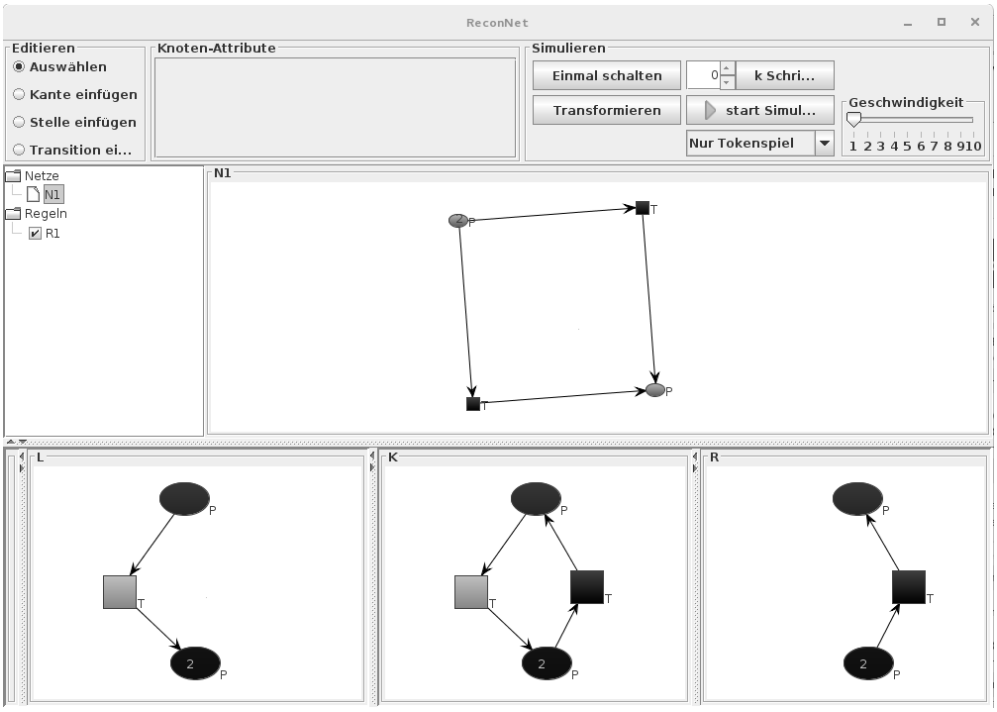
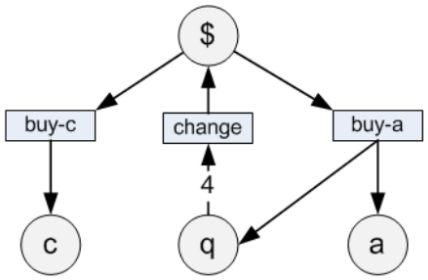


Figure 7: ReConNet: Graphical editor for reconfigurable Petri nets

2.5 Related Works

Part of the Maude documentation in [28] is a Petri net example that is graphically presented in Figure 8. Maude's term replacement system is used to model the firing steps of transitions, such as *buy-c, change* or *buy-a*. Based on this Maude structure, it is possible that add a model-checking possibility which can be used to verify a deadlock or safety properties.




```
1  mod PETRI-NET is
     sorts Place Marking .
3    subsort Place < Marking .
     op __ : Marking Marking ->
        Marking [assoc comm] .
5    ops $ q a c : -> Place .

7    rl [buy-c] : $ => c .
     rl [buy-a] : $ => a q .
9    rl [change] :
        q q q q => $ .
11 endm
```

Figure 8: Petri net example written in Maude[3]

High-level nets (coloured Petri net) are introduced in [29] with a conversion of the banker problem. The focus is on the soundness and correctness of the Maude structure. Since the aim is a formal definition of the model and the operators as well as the firing of a transition are given, it extends the previous approach of [28] with operators that contain details for the firing replacement rules pertaining to colours.

[30] shows an automatic mapping for UML models to a Maude specification that is similar to [3]. The three-step process of modelling, analysing and converting to Maude modules is used, where the first step focuses on subject-specific modelling within the UMLs' class, state or component diagrams. *AtoM* is used to convert the model into a Python-code representation that solves constraints inside the UML model. The final step is for the verification of properties, such as deadlocks, and contains the transfer to Maude.

In [31], Petri nets are also converted into several Maude modules, as seen in [3]. The base is an Input-Output Place/Transition net that is used for the conversion process. All components

[3] `http://maude.cs.uiuc.edu/maude1/manual/maude-manual-html/maude-manual_13.html`, 27 April 2015

are divided into special Maude modules for the net, which basically separate semantics and the initial marking [30].

[32] presents a graphical editor for CPNs, which uses Maude in the background to verify LTL properties. Specified Maude modules (similar to [29]) are defined, which contain one-step commands for the simulation.

A unifying Petri net framework is Snoopy2 published by the Brandenburg University of Technology Cottbus. It combines families of coloured and uncoloured Petri nets in one graphical user interface. The aim of Snoopy2 is the design and execution of Petri nets [33, 34]. Additionally, Charlie is a graphical verification tool published. It can import nets which are created by Snoopy2, and can be analysed by various static and dynamically properties such as invariants or a reachability graph [35].

GRaph- based Object-Oriented VErification (GROOVE) differs to tools like Snoopy2 and Charlie or ReConNet by the chosen model. The graph transformations are realized with states as snapshots, and transitions between the states are calculated by rule applications. A consequence of such model is that not only static models can be used but also models with evolutions can be modelled and proven with the included model-checking implementation [36, 37].

The public transport of Oslo as a case study is presented in [38]. The authors created a model of public transport using a Petri net that is converted into a Maude structure. The aim is to prove the freeness of deadlocks or liveness as well as performance tests that are presented in [39].

3 Model Checker for Reconfigurable Petri Net

This section introduces the implementation of ReConNet Model Checker (rMC). rMC is a Java-based tool that enables a user to convert a given reconfigurable Petri net to a Maude net. Further, such Maude nets can be executed and analysed by the tool. The analysis includes an export to a reachability graph, which is based on the returned output by the `search` command. The export includes an implementation that parses the output and generates graphical representations with an export to Graphviz[1].

3.1 ReConNet Model Checker (rMC)

The basis of this thesis is the internal model of *ReConNet*, which allows the user to create and simulate a reconfigurable Petri net. Due to the non-deterministic behaviour, it is difficult for a user to determine if some properties, such as deadlocks or liveness, are complied with. Therefore, an approach for the model checking of a reconfigurable Petri net is a common practice for ensuring such properties. The *ReConNet Model Checker* (rMC) in Figure 9 is a Java- and Maude-based approach that solved this gap by defining Maude modules for a reconfigurable Petri net. The modules contain the net and a set of rules as well as all mechanisms to fire a transition or transform the net with a rule [3].

[1] `http://www.graphviz.org/`, 26 April 2015

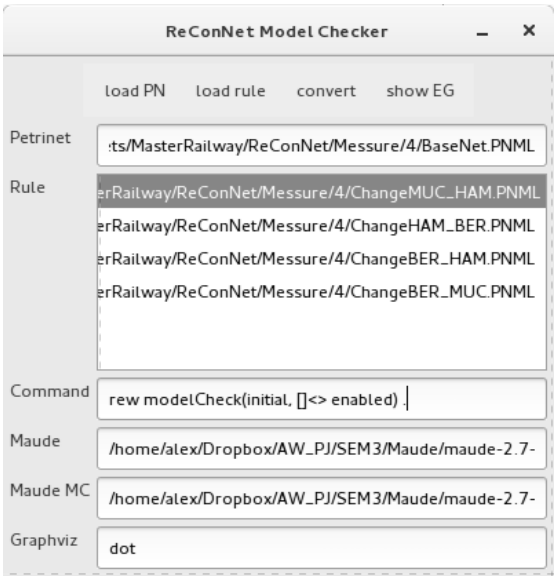


Figure 9: rMC: ReConNet Model Checker

Listing 1 shows, in comparison to Figure 6, the same net, written in the Maude modules defined in [3]. Each set is modelled similarly to a set of places and transitions. As a special feature, the capacity is directly defined for each place, so that a place is defined as `p(<label>|<identifier>|<capacity>)`. Meanwhile, transitions only consist of `t(<label>|<identifier>`. The *pre*- and *post*-domains are wrapped in a set with the `pre`- or `post`-operator. Finally, the initial marking is modelled as a set in the last line.

```
net(places{ p("P" | 3 | 2147483647) , p("P" | 2 | w) } ,
    transitions{ t("T" | 4) : t("T" | 5) } ,
    pre{ (t("T" | 4) --> p("P" | 3 | w)) ,
         (t("T" | 5) --> p("P" | 3 | w)) } ,
    post{ (t("T" | 4) --> p("P" | 2 | w)) ,
          (t("T" | 5) --> p("P" | 2 | w)) } ,
    marking{ p("P" | 3 | w) ;
             p("P" | 3 | w) } )
```

Listing 1: N_1 converted into Maude

Rule r_1 is shown in Listing 2 and Listing 3. One rule consists of two nets with the same structure, as presented in Listing 1. Only the left-hand side and right-hand side are relevant for the replacement rules, since the left-hand side is necessary to find a match and the right-hand side is necessary for the elements that replace the elements of the left-hand side and the gluing conditions of Def. 10 are fulfilled [3].

```
rule( l( net( places{ p("P" | 19 | w) ,
                      p("P" | 16 | w) } ,
              transitions{ t("T" | 23) } ,
              pre{ (t("T" | 23) -->
                    p("P" | 16 | w)) } ,
              post{ (t("T" | 23) -->
                     p("P" | 19 | w)) } ,
              marking{ p("P" | 19 | w) ;
                       p("P" | 19 | w) } ) ) ,
```

Listing 2: Left-hand side of r_1 converted into Maude

```
      r( net( places{ p("P" | 16 | w) ,
                      p("P" | 19 | w) } ,
              transitions{ t("T" | 25) } ,
              pre{ (t("T" | 25) -->
                    p("P" | 19 | w)) } ,
              post{ (t("T" | 25) -->
                     p("P" | 16 | w)) } ,
              marking{ p("P" | 19 | w) ;
                       p("P" | 19 | w) } ) ) )
```

Listing 3: Right-hand side of r_1 converted into Maude

The term replacement for firing of transitions is based on the Def. 7. A rule uses the *pre*-domain to determine if a transition is activated and observes the capacity for each place in the *post*-domain. Listing 4 contains the implementation of Definition 7, where each *pre*-domain condition is implemented inside the left side of a rule and the capacity condition is implemented as an if condition.

```
crl [fire] :
    net(P,
        transitions{T : TRest},
        pre{(T --> PreValue), MTupleRest1},
        post{(T --> PostValue), MTupleRest2},
        marking{PreValue ; M})
    Rules
    MaxID
    StepSize
    aid
    =>
    net(P,
        transitions{T : TRest},
        pre{(T --> PreValue), MTupleRest1},
        post{(T --> PostValue), MTupleRest2},
        calc(((PreValue ; M) minus PreValue)
                              plus PostValue))
    Rules
    MaxID
    StepSize
    aid
    if calc((PreValue ; M) plus PostValue) <=? PostValue
```

Listing 4: Firing term replacement rule written with Maude

Besides firing, the transformation rules of Def. 11 are a central part of a reconfigurable Petri net. Dynamical changes effected by rules enable the net to modify its structure on its own. An abstract implementation of such rules is illustrated in Listing 5. The rule consists of the pattern-matching algorithm of Maude, which ensures that the left-hand side is a subset of the current net state. Furthermore, takes the right-hand side by rule application effect, if the conditions are successfully proven. Conditions such as `freeOfMarking`[2] or `emptyNeighbourForPlace`[3] by Def. 10 test whether parts of the current net states satisfy requirements of the formal rule application. For example, a place can only be deleted if no transition is related to this place.

[2] see line 78 in Listing 16 for the definition and implementation
[3] see line 102 in Listing 16 for the definition and implementation

Definition 10 (Gluing condition for rules [11]). *Each rule r application with $r = (r_{name}, L \leftarrow K \rightarrow R)$ in a reconfigurable Petri net satisfies the gluing conditions. The gluing condition is divided into the identification and the dangling condition, so that:*

- *the identification condition implies that no place or transition can be deleted and obtained at the same time. Transitions are secured by their isomorph mapping of pre- and post-domains and `freeOfMarking` is used for places to ensure that each deleted place applies $p \nsubseteq$ MRest (for the implementation, see Listing 16 line 78)*
- *the dangling condition implies that a place can be deleted only if there are no connections outside the rule. `emptyNeighbourForPlace` ensures for deleted places that p is not used in `pre`{MTupleRest1} or `post`{MTupleRest2} (for the implementation, see Listing 16 line 102)*

Definition 11 (Transforming with Maude net rules). *For each rule $r = (r_{name}, L \leftarrow K \rightarrow R)$ with $L = (P_L, T_L, pre_L, post_L, M_L)$ and $R = (P_R, T_R, pre_R, post_R, M_R)$ in a reconfigurable Petri net, there exists a term rewriting rule that handles the quest by Maude's pattern matching and conditions, such as preserving markings by deletion, or, in case of a deleted transition, the preservation of connections to places, so that*

- *there is a pattern match of the left-hand side to ensure that the left-hand side is a subset of the current net state*
- *the match satisfies the identification condition of Def. 10 to ensure that no transition or place is deleted and obtained at the same time*
- *the match satisfies the dangling condition of Def. 10 to ensure that no deleted place is connected to a transition outside of the match*

```
crl [<r_name>] :
    net(places{ P_L , PRest } ,
        transitions{ T_L : TRest } ,
        pre{ (∀t ∈ T_L : pre(t)_L) , MTupleRest1 } ,
        post{ (∀t ∈ T_L : post(t)_L) , MTupleRest2 } ,
        marking{ M_L ; MRest } )
    rule( l( net( places{ P_L } ,
                  transitions{ T_L } ,
                  pre{ (∀t ∈ T_L : pre(t)_L) } ,
```

```
10                  post{ (∀t ∈ T_L : post(t)_L) } ,
                    marking{ M_L } ) ) ,
12          r( net( places{ P_R } ,
                    transitions{ T_R } ,
14                  pre{ (∀t ∈ T_R : pre(t)_R) } ,
                    post{ (∀t ∈ T_R : post(t)_R) } ,
16                  marking{ M_R } ) ) )
    | RRest
18  MaxID
    StepSize
20  aid{ AidRest }
    =>
22  net( places{ P_R , PRest } ,
         transitions{ T_R : TRest } ,
24       pre{ (∀t ∈ T_R : pre(t)_R) , MTupleRest1 } ,
         post{ (∀t ∈ T_R : post(t)_R) , MTupleRest2 } ,
26       marking{ M_R ; MRest } )
    rule( l( net( places{ P_L } ,
28                  transitions{ T_L } ,
                    pre{ (∀t ∈ T_L : pre(t)_L) } ,
30                  post{ (∀t ∈ T_L : post(t)_L) } ,
                    marking{ M_L } ) ) ,
32          r( net( places{ P_R } ,
                    transitions{ T_R } ,
34                  pre{ (∀t ∈ T_R : pre(t)_R) } ,
                    post{ (∀t ∈ T_R : post(t)_R) } ,
36                  marking{ M_R } ) ) )
    | RRest
38  NewMaxID
    StepSize
40  aid{ AidRestNew }
    if  *** calculate new identifiers
42      AidRestNew := calculateAllIdentifiers /\
        *** ∀p ∈ P_L which are deleted; prove if they
44      ***         are part of MRest (identity condition)
```

```
        freeOfMarking( ( p(<label> | <identifier> |
46                          <capacity>) ) | MRest ) /\
        *** ∀p ∈ P_L which are deleted; prove if there
48      *** is a related transition (dangling condition)
        emptyNeighbourForPlace( p(<label> | <identifier>
50                                          | <capacity>) ,
                                pre{ MTupleRest1 } ,
52                              post{ MTupleRest2 } ) /\
        *** set new maximal identifier counter
54      NewMaxID := correctMaxID(MaxID | StepSize |
                                 |AidRestNew|) .
```

Listing 5: Transformation term replacement rule written with Maude

3.2 Reachability Graph

Based on the search command in Listing 7, Figure 11 shows the resulting state graph. All the states are calculated by all possible rewrite rules of Def. 3 and Def. 5. Finally, the `show search graph` command returns a text-based result of such states search as in Listing 7.

```
1   search initial =>! net(P:Places , T:Transitions ,
                           Pre:Pre , Post:Post ,
3                          Any:Markings )
                    Rules:Rule MaxID:Int StepSize:Int
5                   aidP:IDPool .
    show search graph .
```

Listing 6: Maude search commands for a Maude net

The `search` command output is exemplarily presented in Listing 7. At first, `No solution.` signifies that no final state, such as deadlocks, is found. The next line gives a general overview of the states found or the required time. Afterwards, the output is grouped by all states, such as `state 0`, and followed by the related outgoing arcs, as for example `arc 0 ===> state 1`.

```
search in NET : initial =>! net(P:Places, T:Transitions,
    Pre:Pre, Post:Post, Any:Markings) Rules:Rule MaxID:Int
StepSize:Int aidP:IDPool .

No solution.
states: 12  rewrites: 161 in 1ms cpu (0ms real) (161000
    rewrites/second)
state 0, Configuration: net(places{p("P" | 2 | w),p("P" |
    3 | w)}, transitions{t("T" | 4) : t("T" | 5)}, pre{(t("
    T" | 4) --> p("P" | 3 | w)),t("T" | 5) --> p("P" | 3 |
    w)}, post{(t("T" | 4) --> p("P" | 2 | w)),t("T" | 5)
    --> p("P" | 2 | w)}, marking{p("P" | 3 | w) ; p("P" | 3
     | w)}) rule(l(net(places{p("P" | 16 | w),p("P" | 19 |
    w)}, transitions{t("T" | 23)}, pre{t("T" | 23) --> p("P
    " | 16 | w)}, post{t("T" | 23) --> p("P" | 19 | w)},
    marking{ p("P" | 19 | w) ; p("P" | 19 | w)})), r(net(
    places{p("P" | 16 | w),p("P" | 19 | w)}, transitions{t
    ("T" | 25)}, pre{t("T" | 25) --> p("P" | 19 | w)}, post
    {t("T" | 25) --> p("P" | 16 | w)}, marking{p("P" | 19 |
     w) ; p("P" | 19 | w)}))) 25 10 aid
    {25,(26,(27,(28,(29,(30,(31,(32,(33,(34,(35)))))))))))}
arc 0 ===> state 1 (crl net(P:Places, transitions{
    T:Transitions : TRest:Transitions}, pre{
    MTupleRest1:MappingTuple, T:Transitions -->
    PreValue:Places}, post{MTupleRest2:MappingTuple,
    T:Transitions --> PostValue:Places}, marking{M:Markings
     ; PreValue:Places}) Rules:Rule MaxID:Int StepSize:Int
    aid:IDPool => net(P:Places, transitions{T:Transitions :
     TRest:Transitions}, pre{MTupleRest1:MappingTuple,
```

```
    T:Transitions --> PreValue:Places}, post{
    MTupleRest2:MappingTuple, T:Transitions -->
    PostValue:Places}, calc(((M:Markings ; PreValue:Places)
     minus PreValue:Places) plus PostValue:Places))
    Rules:Rule MaxID:Int StepSize:Int aid:IDPool if calc((
    M:Markings ; PreValue:Places) plus PostValue:Places)
    <=? PostValue:Places = true [label fire] .)
...
```

Listing 7: Example output for the search commands in Listing 7

Rule r_2 in Figure 10 is used as an example to show a deadlock. Its differences with r_1 has other arc directions so that it is possible that a rule application ends in a deadlock. Such deadlocks are visible as sinks in Figure 12 as state without outgoing arcs.

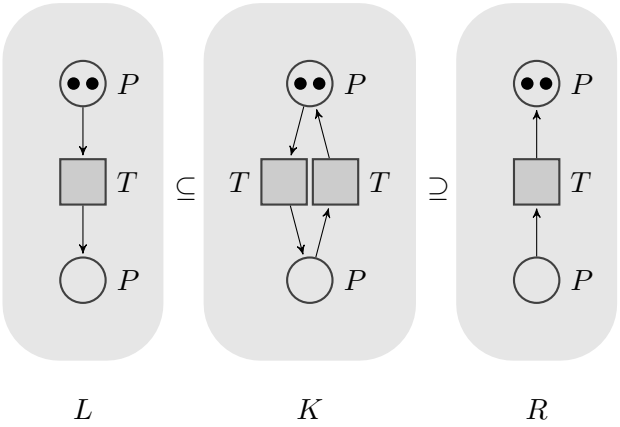


Figure 10: Example rule r_2 which changes the arc direction, can result in a deadlock

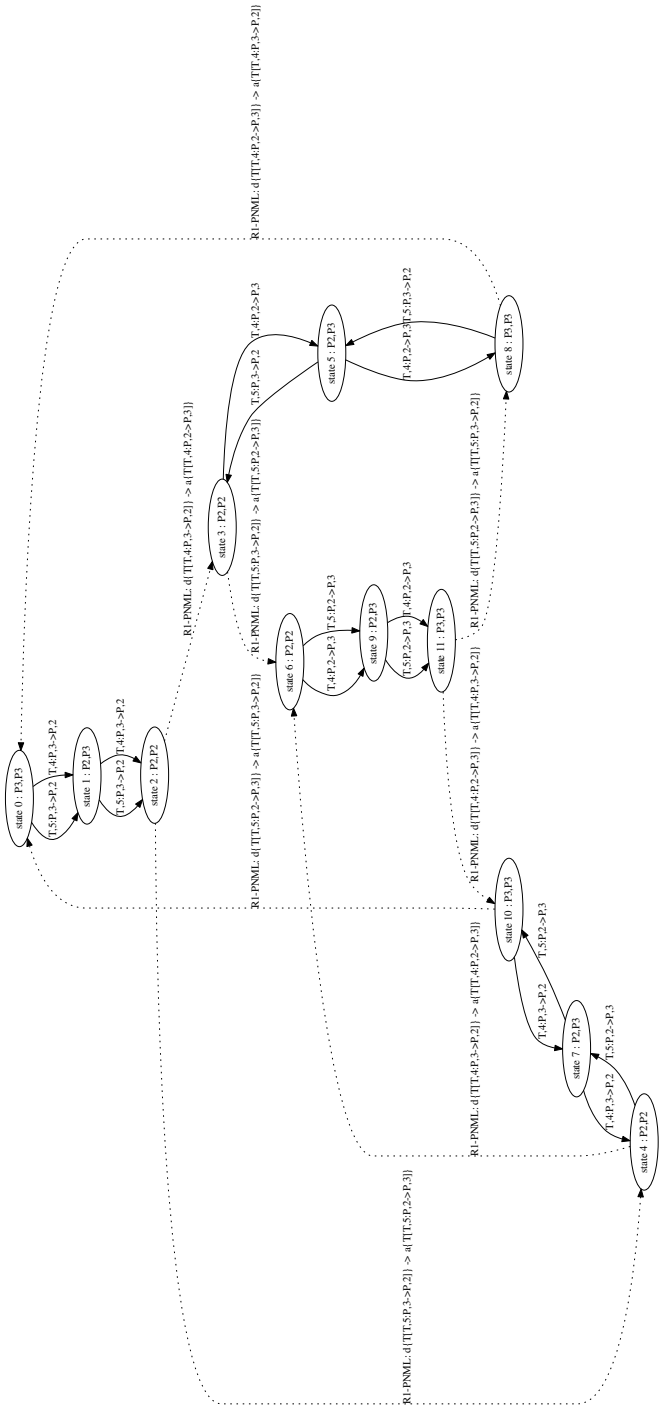


Figure 11: Abstract reachablility graph (ARG) for N_1 and r_1

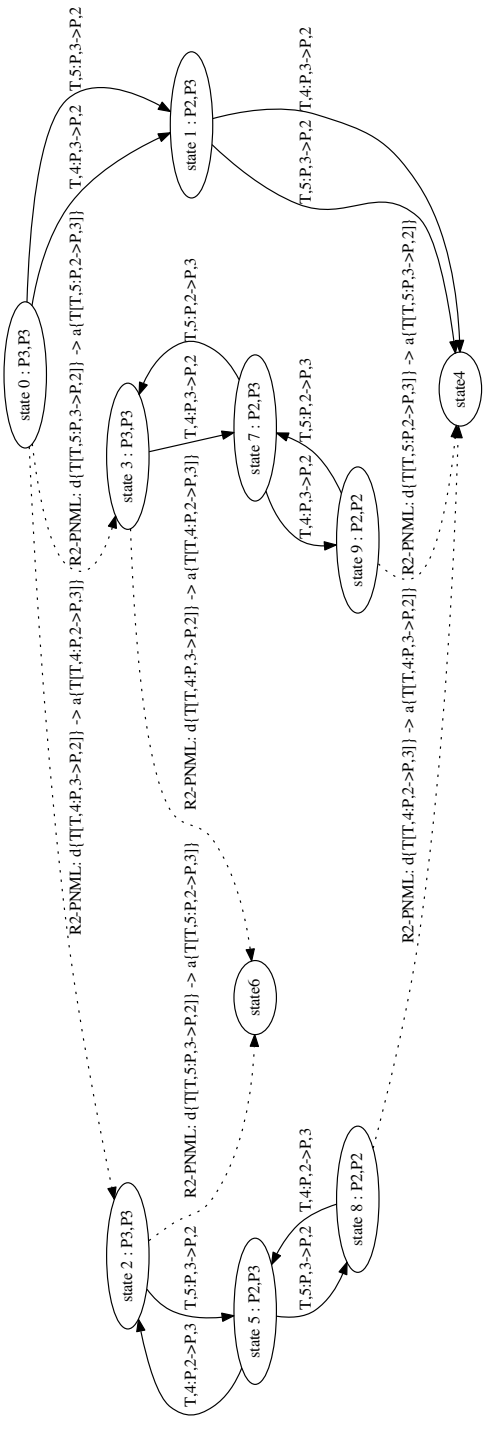


Figure 12: Abstract reachablility graph (ARG) for N_1 and r_2 with deadlock states *state 4* and *6*

4 Labelled Transition Systems

This chapter introduce the Maude net of [3] formally. The following labelled transition systems are defined in combination with Maude term replacement rules so that their labels are given for reconfigurable Petri nets (in Def. 15) as well as for a Maude net (in Def. 16). Further, both transition systems are used as a basis for the definition and proof of bisimulation in the following chapter 5.

4.1 Maude net

The main term of a reconfigurable Petri net is called a Maude net. It combines a net and a set of rules as well as metadata such as the highest identifier. The following section introduces the formal definition of the NET module that is introduced in [3]. Therefore, this section aims to define the NET module using a definition and the implementation of required sorts and operators.

The conversion of Theorem 1 is based on the reconfigurable Petri nets in Def. 5 and a Maude net configuration $\mathbb{T}_{Configuration}$ as a conversion target specified in Def. 14. A Maude net configuration in Def. 14 again includes a net of Def. 12 and a set of rules in Def. 13, and, therefore, specifies all parts of a reconfigurable Petri net.

First, it specifies Def. 12, which is a net including places, transitions, *pre*- and *post*-domains as well as markings through sorts for each term, identity elements and separation operators. Therefore, a net can be written as shown in Listing 1.

Definition 12 (Maude net and its term sets)**.** *A Maude net is provided by a Maude module NET (see Listing 18) by* `initial = net(P, T, Pre, Post, M)`*, hence the following well-formed conditions hold:*

- `P = places{ emptyPlace }` *or*
  `P = places{` $p_1, ..., p_n$ `}` *so that some* p_i *and* p_j *are pairwise disjoint:* $p_i = p_j \implies i = j$
- `T = transitions { emptyTransition }` *or*
  `T = transitions {` $t_1 : ... : t_m$ `}` *so that some* t_i *and* t_j *are pairwise disjoint:* $t_i = t_j \implies i = j$
- `Pre = pre { emptyMappingTuple }` *or*
  `Pre = pre {` $\texttt{mappingTuple}_1, ..., \texttt{mappingTuple}_o$ `}` *with*
  mappingTuple$_i$ $=$ `(`t_i`-->emptyPlace)` *or*
  mappingTuple$_i$ $=$ `(`t_i`-->{`$p_{i1}, ..., p_{is}$`})` *so that some* t_i *and* t_j *are pairwise disjoint:* $t_i = t_j \implies i = j$ *and* $p_{iu} \in \{p_1, ..., p_n\}$
- `Post = post { emptyMappingTuple }` *or*
  `Post = post {` $\texttt{mappingTuple}_1, ..., \texttt{mappingTuple}_u$ `}` *with*
  mappingTuple$_i$ $=$ `(`t_i`-->emptyPlace)` *or*
  mappingTuple$_i$ $=$ `(`t_i`-->{`$p_{i1}, ..., p_{is}$`})` *so that some* t_i *and* t_j *are pairwise disjoint:* $t_i = t_j \implies i = j$ *and* $p_{iu} \in \{p_1, ..., p_n\}$
- `M = marking { emptyMarking }` *or*
  `M = marking {` $p_1, ..., p_k$ `}` *with* $p_i \in \{p_1, ..., p_n\}$

Further, a Maude net rule is defined in Def. 13. It combines the left and right sides of a rule in $\mathcal{R}$ to form a Maude net rule. As a wrapper, the operator `rule` is used, which contains the `l` and the `r` operators for both sides. `l` and `r` both contain a net of Def. 12.

Definition 13 (Term sets for rules [3])**.** *After module NET gives a Maude net, the sorts* `Left-HandSide`*,* `RightHandSide` *and* `Rule` *describe term sets of rules (see Listing 16).*

Finally, the definition of the Maude net configuration in Def. 14 combines the Maude net term in Def. 12 and a set of rule terms in Def. 13 with metadata such as the highest identifier that is currently used, and a set of free identifiers wrapped in the `IDPool`.

Definition 14 (Maude net configuration and its term sets [3])**.** *A Maude net configuration is given by a Maude module NET as in Def. 12. Thereafter, a* `Configuration` *is defined by* `IDPool`*,* `Int` *and* `Configuration` *(see Listing 15).*

With respect to membership equation logic [40, 41], the specification for reconfigurable Petri nets is given by $RPN(N) = (\Sigma, V, E)$. That specification is given as Maude source code in the following pages through listings for `sorts`, as in Listing 8 or Listing 11 for `operators`. Therefore, a membership equation logic is defined by a triple (Σ, V, E), where a signature $\Sigma = (K, S, \Omega)$ is based on V that contains variables and axioms in E. Additionally, S contains all sorts that are listed in Listing 8, K contains all kinds over S and Ω includes all operators that are listed in Listing 10 and Listing 11.

As the base, the types are declared in Listing 8. `Net` and all included types such as `Places`, `Transitions`, etc. for the Def. 12 and `Rule, RightHandSide` and `LeftHand-Side` for the rule definition in Def. 13.

```
sort Net .                    sort Places .
sort Transitions .            sort Pre .
sort Post .                   sort MappingTuple .
sort Markings .               sort Omega .
sort Rule .                   sort RightHandSide .
sort LeftHandSide .
```

Listing 8: Sorts of a reconfigurable Petri net defined in Maude [3]

As defined in Def. 12, each term set has a separation operator for more than one term. For `Places` and `MappingTuple`, the „,“is used. „+“is defined as a separation operator for `Places` terms that describes the firing step, including the marking calculation. `Transitions` are separated using „:“and `Markings` using „;“.

```
op places{_} : Places -> Places .
op transitions{_} : Transitions -> Transitions .
op pre{_} : MappingTuple -> Pre .
op post{_} : MappingTuple -> Post .
op marking{_} : Markings -> Markings .

op _,_ : Places Places -> Places
[ctor assoc comm id: emptyPlace] .
op _+_ : Places Places -> Places
[ctor assoc comm id: emptyPlace] .
op _:_ : Transitions Transitions -> Transitions
[ctor assoc comm id: emptyTransition] .
op _,_ : MappingTuple MappingTuple -> MappingTuple
[ctor assoc comm id: emptyMappingTuple] .
op _;_ : Markings Markings -> Markings
[ctor assoc comm id: emptyMarking] .
```

Listing 9: Wrapping and grouping operators of a reconfigurable Petri net defined in Maude [3]

If a net does not contain nodes, such as places or transitions, their identity elements can be used. For each `sort`, an element in Listing 10 is defined. Hence, it is feasible to write `places{emptyPlace}` if there is no place.

```
op emptyPlace : -> Places .
op emptyTransition : -> Transitions .
op emptyMappingTuple : -> MappingTuple .
op emptyMarking : -> Markings .
op w : -> Omega .
op emptyRule : -> Rule .
```

Listing 10: Identity elements of a reconfigurable Petri net defined in Maude [3]

The definition of single places, transitions, pre- and post- domains, and rules are shown in Listing 11. A place consists of the sort `Places` and the `p(<label>|<identifier>|<capacity>)` operator, where the capacity is either a concrete integer or a „w“for omega. Transitions are defined with their `Transitions` sort while the `t(<label>|<identifier>)` operator combines a label and an identifier for each transition. `MappingTuple` is used to define the *pre-* and *post*-domains for each transition (and `Transitions` term). It consists of `(<transition> -- > <places>)`, where one `<transition>` is mapped to a set of `<places>`. Finally, Def. 13 that are based on the left-hand and right-hand sides of a `rule` are given by the `l` and `r` operators as well as by „|“, which is used as a separator.

```
*** p ( < label > | < id > | < capacity > )
op p(_|_|_) : String Int Int -> Places .
op p(_|_|_) : String Int Omega -> Places .
*** t ( < label > | < id > )
op t(_|_) : String Int -> Transitions .

*** t -> P^⊕
op (_-->_) : Transitions Places -> MappingTuple .

*** rule
op _|_ : Rule Rule -> Rule
[ctor assoc comm id: emptyRule] .
op l : Net -> LeftHandSide .
op r : Net -> RightHandSide .
op rule : LeftHandSide RightHandSide -> Rule .
```

Listing 11: Operator definitions of a reconfigurable Petri net defined in Maude [3]

Listing 12 concatenates the Def. 12 and the implementation above in Listings 8 to 11, and Def. 13 to a Maude net configuration in Def. 14. Additionally, a `IDPool` consists of its identity element `emptyIDSet`, the „,“as separator operator and `aid{...}` as wrapper for all free identities.

```
1 sort IDPool .
  op emptyIDSet : -> Int .
3 op _,(_) : Int Int -> Int [comm id: emptyIDSet] .
  op aid{_} : Int -> IDPool .
5
  sort Configuration .
7 *** READING: NET SET<RULE> MAXID STEP_SIZE ID
  op _____ : Net Rule Int Int IDPool -> Configuration .
```

Listing 12: Implementation of the Maude net configuration in Maude [3]

4.2 Labelled Transition System for Reconfigurable Petri Net

A labelled transition system for a reconfigurable Petri net is defined in Def. 15 with the included isomorphism class in Def. 9. All states that are reachable by a firing step of Def. 7 or the transforming steps of Def. 8 are consolidated by the isomorphism class if their labels are equal. The example in Figure 13 shows two of those isomorphic nets, where the label of the places are equal.

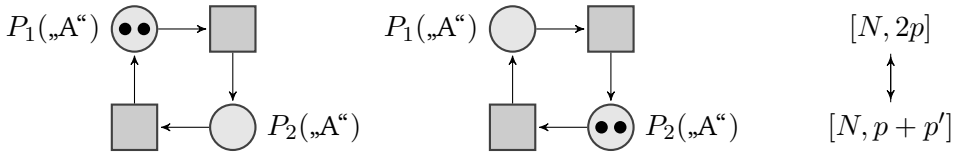


Figure 13: Two isomorphic nets by Def. 9 and the related labelled transition system

In the context of a reconfigurable Petri net, a transition system is defined by $LTS_{RPN} = (S_{RPN}, A_{RPN},$
$tr_{RPN})$, where S_{RPN} is a non-empty set that contains all states $s = (N, M) \in S_{RPN}$. A_{RPN} contains two kinds of arc labels, such as firing and transforming, which are defined by $A_{RPN} = A_T \bigcup A_R$. Transition relations are based on $tr_{RPN} \subseteq S_{RPN} \times A_{RPN} \times S_{RPN}$, so that a transition relation defines the flow by a combination of two states and one label.

Definition 15 (Labelled transition system for reconfigurable Petri net)**.** *Given a reconfigurable Petri net* $((N_0, M_0), \mathcal{R})$*, the definition of a labelled transition system* $LTS_{RPN} = (S_{RPN}, A_{RPN}, tr_{RPN})$ *is based on the isomorphism classes of nets:*

1. *Initial states by Def. 9:*
   $[(N_0, M_0)] \in S_{RPN}$
2. *Firing steps:*
   If $[(\overline{N}, \overline{M})] \in S_{RPN} \wedge (N, M) \in [(\overline{N}, \overline{M})] \wedge M[t\rangle M'$ *in* N *then:* $[(N, M')] \in S_{RPN}$*,* $t_{name}(t) \in A_{RPN}$ *and* $[(N, M)] \xrightarrow{t_{name}(t)} [(N, M')] \in tr_{RPN}$
3. *Transformation steps:*
   If $[(\overline{N}, \overline{M})] \in S_{RPN} \wedge (N, M) \in [(\overline{N}, \overline{M})] \wedge (N, M) \overset{(r,o)}{\Longmapsto} (N', M')$ *for some rule* $r = (r_{name}, L \leftarrow K \rightarrow R) \in \mathcal{R}$ *and some occurrence* $o : L \rightarrow N$ *then:*
   $[(N', M')] \in S_{RPN}$*,* $r_{name} \in A_{RPN}$ *and* $[(N, M)] \xrightarrow{r_{name}} [(N', M')] \in tr_{RPN}$
4. *Finally:*
   $S_{RPN}, A_{RPN}, tr_{RPN}$ *are the smallest sets satisfying the above conditions.*

4.3 Labelled Transition System for Maude

A transition system for a Maude net is defined by $LTS_{MNC} = (S_{MNC}, A_{MNC}, R_{MNC})$, where S_{MNC} is a non-empty set that contains all states of a Maude breadth-first search tree. Maude's deduction rules of Def. 3 and Def. 4 are used to execute all known rules, such as firing or transformations, with rules in the RULES module. Such a state $s \in S_{MNC}$ consists of a `Net` term as current state. A_{MNC} is defined with $A_{MNC} = A_T \bigcup A_R$ and contains the labels of rewrite rules, such as the firing of a transition or transforming. tr_{MNC} is defined as a set of transition relations that is based on $tr_{MNC} \subseteq S_{MNC} \times A_{MNC} \times S_{MNC}$. Therefore, two terms of S_{MNC} are connected with a label of a rewrite rule in A_{MNC}.

Definition 16 (Labelled transition system for a Maude net)**.** *Given the Maude module NET, a labelled transition system* $LTS_{MNC} = (S_{MNC}, A_{MNC}, tr_{MNC})$ *is defined with respect to the term sets over the equation conditions of the Maude modules by:*

1. *Initial:*
   $initial \in S_{MNC}$

2. *Firing steps:*
   If $s \in S_{MNC}$ *and* $s \rightarrow s'$ *is a replacement for a rewrite rule* `[fire]` *of Listing 4 with the third rule of Def. 3 so that*
   $s =$ `net(P,`
   `transitions{`$t(\overline{label}|identifier) : \texttt{TRest}\}$,
   `pre{`$t(\overline{label}|identifier) -- > PreValue, \texttt{MTupleRest1}\}$,
   `post{`$t(\overline{label}|identifier) -- > PostValue, \texttt{MTupleRest2}\}$,
   `marking{`$PreValue; M\}$)
   is used as left-hand side of Listing 4, then $s' \in S_{MNC}$, $t(\overline{label}) \in A_{MNC}$ *and* $t \xrightarrow{t(\overline{label})} s' \in tr_{MNC}$

3. *Firing step for a transition with an empty pre:*
   If $s \in S_{MNC}$ *and* $s \rightarrow s'$ *are a replacement for a rewrite rule* `[fire-emptyPre]` *with the third rule of Def. 3 in the Maude module RPN and*
   $s =$ `net(P,`
   `transitions{`$t(\overline{label}|identifier|capacity) : \texttt{TRest}\}$,
   `pre{`$t(\overline{label}|identifier|capacity) -- > \texttt{emptyPlace , MTupleRest1}\}$,
   `post{`$t(\overline{label}|identifier|capacity) -- > PostValue, \texttt{MTupleRest2}\}$,
   `marking{`$M\}$)
   then $s' \in S_{MNC}$, $t(\overline{label}) \in A_{MNC}$ *and* $t \xrightarrow{t(\overline{label})} s' \in tr_{MNC}$

4. *Transformation steps:*
   If $s \in S_{MNC}$ *and* $s \rightarrow s'$ *is a replacement for a rewrite rule* $[r_{name}]$ *in the Maude module RULE and*
   $s = \texttt{net(places}\{P_L, \texttt{PRest}\},$
   $\texttt{transitions}\{T_L : \texttt{TRest}\},$
   $\texttt{pre}\{\texttt{Pre}_L, \texttt{MTupleRest1}\},$
   $\texttt{post}\{\texttt{Post}_L, \texttt{MTupleRest2}\},$
   $\texttt{marking}\{M_L; M\})$
   $\texttt{rule(l(}\ P_L, T_L, \texttt{Pre}_L, \texttt{Post}_L, M_L\ \texttt{)}\ , \texttt{r(R)}\ \texttt{)}$
   then is:
   $s' \in S_{MNC}$, $r \in A_{RPN}$ *and* $s \xrightarrow{r_{name}} s' \in tr_{MNC}$

5. *Finally:*
   $S_{MNC}, A_{MNC}, tr_{MNC}$ *are the smallest sets satisfying the above conditions.*

4.4 Résumé of the Formalisation

The purpose of this chapter is the definition of a Maude net and the labelled transition systems for both nets. To clarify the definition of [3], a Maude net is defined by a formal definition in Def. 12 for nets, Def. 13 for rules and Def. 14 for a configuration itself. Selected code snippets are based on the definition listed for sorts in Listing 10, for operators in Listing 11 and Listing 11 as well as for the configuration in Listing 12.

Def. 15 contains the definition of a labelled transition system for reconfigurable Petri nets. It is based on an isomorphism class for net states and firing as well as transforming steps that are used for the transition relations. A construction of a similarly labelled transition system for a Maude net is defined in Def. 16. All reachable states and their associated actions, such as firing and transforming rewrite rules, are used to define the labelled transition system.

5 Correctness of Model Checking for Maude

The content of this chapter is graphically represented by Figure 14. Theorem 1 is defined as a conversion between the reconfigurable Petri net $(N, \mathcal{R})$ and the Maude net NET. Further, labelled transition systems are calculated by the related nets and inference rules. For reconfigurable Petri nets, the transforming steps of Def. **??** and firing steps of Def. 7 are used. Equivalent rules are defined for a Maude net so that the firing or transforming steps are transferred into Maude rewriting rules. Formally, the deduction of such rewriting rules is defined by Def. 3, which are extended by Def. 4. Finally, the labelled transition systems for both nets are introduced by LTS_{RPN} using Def. 15 and LTS_{MNC} applying Def. 16.

LTS_{RPN} as labelled transition systems for reconfigurable Petri nets is given by all reachable states using firing and transforming steps, whereby the states are pooled by the isomorphism of Def. 9. Furthermore, LTS_{MNC} is given for a Maude net, where all rewriting rule applications of firing and transforming steps are summarized.

Theorem 2 defines the bisimulation between these two labelled transition systems, whereby Def. 2 is used to prove the bisimilarity of both systems, outgoing from the initial states by Theorem 1. Bisimilarity for each reachable state implies the compliance of equality actions for each pair in the relation. Such a relation is defined with *map* by Def. 18 between LTS_{RPN} and LTS_{MNC}. Regarding the introduction of Section 2.2, the aim of Theorem 2 is to be able to decide whether both systems are behaviourally equivalent.

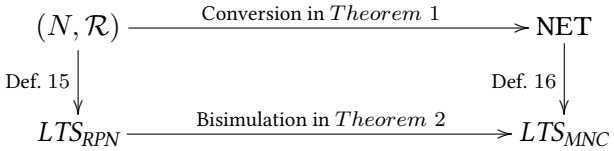


Figure 14: Correctness of conversion

The following chapter contains both theorems of Figure 14. Theorem 1 defines the conversion of a reconfigurable Petri net to a Maude net. Therefore, several lemmas are defined, where all parts such as places, transitions, etc. are converted separately. Theorem 2 defines the bisimulation on the basis of the labelled transition systems.

5.1 Syntax Conversion

Towards Theorem 1, the following injective functions in Lemma 1-6 are used to convert all parts of a reconfigurable Petri net into a NET- and a RULES-module. The theorem itself specifies the conversion for a given reconfigurable Petri net to a Maude net that implies theses functions so that

- *buildPlace* in Lemma 1 defines the conversion for places
- *buildTransition* in Lemma 2 defines the conversion for transitions (similar to *buildPlace*)
- *buildPre* in Lemma 3 defines the conversion for each *pre*(t) with $t \in T^{\oplus}$
- *buildPost* in Lemma 4 defines the conversion for each *post*(t) with $t \in T^{\oplus}$ (similar to *buildPre*)
- *buildNet* in Lemma 5 defines the conversion of a net
- *buildRule* in Lemma 6 defines the conversion of rules in $\mathcal{R}$

Def. 17 contains functions for the mapping of identifiers and capacities leading to lemmas for the conversion of places and transitions. The identifiers are used as unique keys for nodes such as places or transitions due to the use by *pre*- and *post*-domains. Furthermore, the capacities are used to define a limit of tokens that can be stored on a place.

Definition 17 (Injective identity mapping of id_P and $\overline{cap}$)**.** *Given a reconfigurable Petri net* $(N, \mathcal{R})$ *and an injective identity mapping* $id_P : P \to \mathbb{N}$ *for the places, for transitions analogously, and a capacity function* $\overline{cap} : P \to \mathbb{N} \bigcup \{\omega\}$ *is defined by:*

$$\overline{cap}(q) = \begin{cases} \omega & \text{if } cap(q) = \omega \\ cap(q) & \text{else} \end{cases}$$

The conversion of places is defined in Lemma 1 where $P^{\oplus}$ of net N is used as the source for the induction. The identity element `emptyPlace` is used in the induction basis, if $P^{\oplus}$ is empty. Each new place p_{n+1} is inductively converted by the induction step in the definition of a Maude net place, whereby the conversion uses the identifier and capacity function to convert a place by the `p(<label>|<identifier>|<capacity>)` operator.

Lemma 1 (buildPlace). *Given a set of places P together with an identity function id_P (see Def. 17), a capacity function $\overline{cap}$ (see Def. 17) and a labelling function p_{name} (see Def. 5), then there is a injective function buildPlace : $P^{\oplus} \to \mathbb{T}_{Places}$.*

Proof of Lemma 1. *buildPlace* is defined inductively over $|P|$ by:

- for $P = \emptyset$, $P^{\oplus} = \{0\}$ and $buildPlace(0) = \texttt{emptyPlace}$
- for $P' = P \uplus \{p'\}$ there is a $buildPlace' : P' \to \mathbb{T}_{Places}$ defined by

$$buildPlace'(s) = \begin{cases} buildPlace(s) \text{ if } s \in P^{\oplus} \\ buildPlace(s'), \mathrm{p}(p_{name}(p_{n+1,1})|id_P(p_{n+1,1})|\overline{cap}(p_{n+1,1})), ..., \\ \qquad \mathrm{p}(p_{name}(p_{n+1,k})|id_P(p_{n+1,k})|\overline{cap}(p_{n+1,k})) \\ \qquad \text{if } s = s' + kp_{n+1},\ k \geq 1 \text{ and } s' \in P^{\oplus} \setminus P'^{\oplus} \end{cases}$$

buildPlace is injective, given some set of places $P = \{p_1, ..., p_n\}$ with id_P, $\overline{cap}$ and p_{name}, so that $s = \sum_{1 \leq i \leq n} \lambda_i p_i \neq \sum_{1 \leq i \leq n} \mu_i p_i = s'$, then there is some $1 \leq i \leq n$ with $\lambda_i \neq \mu_i$, and hence

$$\begin{aligned} buildPlace(s) = &\mathrm{p}(p_{name}(p_1), 1), \ldots, \mathrm{p}(p_{name}(p_1), \lambda_1), \ldots, \\ &\mathrm{p}(p_{name}(p_i), 1), \ldots, \mathrm{p}(p_{name}(p_i), \lambda_i), \ldots, \\ &\mathrm{p}(p_{name}(p_n), 1), \ldots, \mathrm{p}(p_{name}(p_i), \lambda_n) \\ &\neq \\ &\mathrm{p}(p_{name}(p_1), 1), \ldots, \mathrm{p}(p_{name}(p_1), \mu_1), \ldots, \\ &\mathrm{p}(p_{name}(p_i), 1), \ldots, \mathrm{p}(p_{name}(p_i), \mu_i), \ldots, \\ &\mathrm{p}(p_{name}(p_n), 1), \ldots, \mathrm{p}(p_{name}(p_n), \mu_n) = buildPlace(s') \end{aligned}$$

is id_p injective.

The inverse function $buildPlace^{-1}$ is defined for well-formed terms of sort `Places` (see Def. 12) by

$$
\begin{aligned}
&buildPlace^{-1}(\texttt{emptyPlace}) = 0 \textit{ and}\\
&buildPlace^{-1}(\mathrm{p}(p_{name}(p_1), 1), \ldots, \mathrm{p}(p_{name}(p_1), \lambda_1), \ldots,\\
&\qquad \mathrm{p}(p_{name}(p_i), 1), \ldots, \mathrm{p}(p_{name}(p_i), \lambda_i), \ldots,\\
&\qquad \mathrm{p}(p_{name}(p_n), 1), \ldots, \mathrm{p}(p_{name}(p_n), \lambda_n)) = \sum_{1 \leq i \leq n} \lambda_i p_i
\end{aligned}
$$

□

The proof of Lemma 2 is similar to Lemma 1. It differs in the part of the conversion, where the definition of a transition as well as the identity element is used instead of the place definition. Transitions are defined by the `t(<label>|<identifier>)` operator that contains only the identifier derived by the identity function id_T to get a unique identifier for this transition from the source data.

Lemma 2 (buildTransition)**.** *Given a set of transitions T with t_{name} by Def. 5 and id_T by Def. 17, then there is a injective function by buildTransition : $T^{\oplus} \rightarrow \mathbb{T}_{Transitions}$*

Proof of Lemma 2. Similar induction to the proof of Lemma 1, due to the „:“constructor in line 44 of module NET in Listing 15. This includes differences by the definition in Def. 12:

- „:“as separation instead of „„
- `emptyTransition` instead of `emptyPlace`
- `t(<label>|<identifier>)` operator instead of the place operator

□

Conversions of *pre*- and *post*-domains requires special definitions due to the *MappingTuple*. Def. 12 introduces the concept of both domains that are based on the sort *MappingTuple*. It is a mapping between a transition and a set of places so that it is suitable for *buildPre* in Lemma 3 as well as *buildPost* in Lemma 4.

The proof itself is realised with an induction over the set of transitions. Each new transition is a mapping to a `MappingTuple`-defined term. Such term consists of a transition that is mapped with an arrow, which is written in a Maude net with „$--\,>$“, to a set of `places`-terms that are returned from $pre(t_n)$[1]. The related set of places is converted by *buildPlace* of Lemma 1.

[1] Empty *pre*- or *post*-domains are special cases that are solved by the identity element for place `emptyPlace`

Lemma 3 (buildPre)**.** *Given a set of transitions T with buildPlace of Lemma 1 and buildTransition of Lemma 2, then there is a injective function by buildPre : $T^{\oplus} \to \mathbb{T}_{MappingTuple}$*

Proof of Lemma 3. *buildPre* is defined inductively over $|T|$ by:

- for $T = \emptyset, T^{\oplus} = \{0\}$ and *buildPre*$(0) =$`emptyMappingTuple`
- for $T' = T \bigcup \{t'\}$ there is a *buildPre*$' : T' \to \mathbb{T}_{MappingTuple}$ defined by:

$$buildPre'(t') = \begin{cases} buildPre(t') \; if \; t' \in T^{\oplus} \\ buildPre(t''), (buildTransition\{t_{n+1,1}\} - - > buildPlace\{pre(t_{n+1,1})\}), ..., \\ \qquad (buildTransition\{t_{n+1,k}\} - - > buildPlace\{pre(t_{n+1,k})\}) \\ \qquad if \; t' = t'' + kt_{n+1}, k \geq 1 \text{ and } t'' \in T^{\oplus} \setminus T'^{\oplus} \end{cases}$$

The proof of injection and the inverse function *buildPre*$^{-1}$ for *buildPre* are analogously to Lemma 1.

□

The proof of Lemma 4 is similar to Lemma 3, which is realized inductively over the set of transitions. It differs only in the used functions so that the same proof can be realized with *post*(t_n) instead of *pre*(t_n).

Lemma 4 (buildPost)**.** *Given a set of transitions T with buildPlace of Lemma 1 and buildTransition of Lemma 2, then there is a injective function by buildPost : $T^{\oplus} \to \mathbb{T}_{MappingTuple}$*

Proof of Lemma 4. Similar induction to proof of Lemma 3, due to the equal construction in line 48 of module NET in Listing 15. □

In Lemma 5 is a injective function defined, which combines all parts of a Petri net to Maude `net`-term. A `net` operator wraps the places, transitions, *pre*-and *post*-domains as well as markings into one net. *buildNet* is a injection as all functions are injections.

Lemma 5 (buildNet)**.** *Given a net N with buildPlace of Lemma 1, buildTransition of Lemma 2, buildPre of Lemma 3 and buildPost of Lemma 4, then there is a injective function by buildNet : $(N, M) \to \mathbb{T}_{Net}$ with $N = (P, T, pre, post, M)$*

Proof of Lemma 5. *buildNet* The proof is separated into all parts of a Maude net and hence a `Net` is given by injective functions:

$$\begin{aligned}&\texttt{net (places}\{\textit{buildPlaces}(P)\},\\ &\quad\texttt{transitions}\{\textit{buildTransition}(T)\},\\ &\quad\texttt{pre}\{\textit{buildPre}(T)\},\\ &\quad\texttt{post}\{\textit{buildPost}(T)\},\\ &\quad\texttt{marking}\{\textit{buildPlaces}(M)\}\texttt{)}\end{aligned}$$

□

Rules are converted by Lemma 6 where a set of reconfigurable Petri net rules is used to create `rule`-terms in Maude. The identity element `emptyRule` is returned if no rule is defined. Each rule consists of two Petri nets with left-hand side for the left-hand side and right-hand side for the right-hand-side so that the *buildNet* function can be used to convert the nets.

Lemma 6 (buildRule)**.** *Given a rule r by* $r = (r_{name}, L \leftarrow K \rightarrow R)$ *and buildNet by Lemma 5, then there is a injective function by buildRule :* $\mathcal{R} \rightarrow \mathbb{T}_{Rule}$

Proof of Lemma 6. *buildRule*

Basis: $|\mathcal{R}| = 0$ so that $\mathbb{T}_{Rule} = \{emptyRule\} \cong \{0\}$

Induction hypothesis: $|\mathcal{R}| = n$ then there exists a injective function

$$buildRule : \mathcal{R}' \rightarrow \mathbb{T}_{Rule}$$

Induction step ($n \rightarrow n+1$**):** Is $\mathcal{R}' = \mathcal{R} \uplus \{r_{n+1}\}$ with $r' = (r_{name}, L \leftarrow K \rightarrow R)$ then there is by the induction hypothesis a injection with $buildRule' : \mathcal{R}' \rightarrow \mathbb{T}_{Rule}$ which is defined by:

$$buildRule'(r') = \begin{cases} buildRule(r') \text{ if } r' \in \mathcal{R} \\ buildRule(r'') \mid \{\texttt{rule(l(}buildNet(L)\texttt{), r(}buildNet(R)\texttt{))}\} \\ \qquad \text{if } r' = \{r''\} \biguplus \{r_{n+1}\} \\ \qquad \text{with } r'' \in \mathcal{R} \setminus \mathcal{R}' \end{cases}$$

□

Finally, Theorem 1 introduces the conversion of one given reconfigurable Petri net into a Maude net. The first part of the proof is based on the Def. 14 that defines all parts of a reconfigurable Petri net in Maude's term algebra. A Maude net consists of a net that is included in the `net`-operator, a set of rules that are defined by the `rule`-operator as well as metadatas such the highest identifier. The second part presented the RULES module by rewrite rules for each rule in $\mathcal{R}$.

Theorem 1 (Syntactic conversion of a reconfigurable Petri net to a Maude net configuration)**.** *For each reconfigurable Petri net $(N, \mathcal{R})$, there is a well-formed Maude NET and RULES module.*

Proof of Theorem 1. By *buildPlace* of Lemma 1, *buildTransition* of Lemma 2, *buildPre* of Lemma 3, *buildPost* of Lemma 4 and *buildRule* of Lemma 6 is for each reconfigurable Petri net $(N, \mathcal{R})$, with $N = (P, T, pre, post, M)$ and $\mathcal{R} = \{(r_{name_i}, L_i \leftarrow K_i \rightarrow R_i) | 1 \leq i \leq n\}$, a well-formed Maude NET and RULES module (as in appendix Listing 18) given so that:

```
eq initial = buildNet(N)
             buildRule(R)
             metadata
```

In addition, there is the Maude module RULES with the rewrite rules of Def. 11 and the implementation of Listing 5. Thus we have for each rewrite rule $r \in \mathcal{R}$ with $r = (r_{name_i}, L_i \leftarrow K_i \rightarrow R_i)$:

```
crl [r_name] : net( places{buildPlaces(P_Li), PRest},
                    transitions{buildTransition(T_Li): TRest},
                    pre{buildPre(T_Li), MTupleRest1},
                    post{buildPost(T_Li), MTupleRest2},
                    marking{buildPlaces(M_Li); MRest})
                    buildRule(R)
                    metadata
                    =>
               net( places{buildPlaces(P_Ri), PRest},
                    transitions{buildTransition(T_Ri): TRest},
                    pre{buildPre(T_Ri), MTupleRest1},
                    post{buildPost(T_Ri), MTupleRest2},
```

marking{*buildPlaces*(M_{Ri}); MRest})
buildRule($\mathcal{R}$)
new metadata
if *** for deleted places
freeOfMarking($\forall p \in P_{Li}$ | MRest) $\wedge$
*** for places of deleted transitions
emptyNeighbourForPlace($\forall p \in P_{Li} \setminus P_{Ri}$ |
pre{MTupleRest1} | post{MTupleRest2})$\wedge$
calculate new metadata .

□

5.2 Equivalence by Bisimulation

Labelled transition systems are defined for reconfigurable Petri nets by LTS_{RPN} and a Maude net by LTS_{MNC}. Both labelled transition systems are comprised in this section by equivalence through bisimulation. A *map* function is defined in Lemma 8 that linked a state $s \in LTS_{MNC}$ to a state $r \in LTS_{RPN}$. Hence, the function *map* inverses the direction of Theorem 1 and enables the proof of Theorem 2, since bisimulation requires linked states in both directions.

Remark 3. *To distinguish the states of the respective labelled transition systems, the variables s for state in LTS_{MNC} and r for state in LTS_{RPN} are used*

The Theorem 2 is the major aim of this thesis and defines the behaviour equivalence of both transition systems. The proof implies that for each pair $(s_n, r_n) \in$ *map* with $n \geq 0$, all outgoing actions are equal. Furthermore, the proof ensures that the reached states s_{n+1} and r_{n+1} are also in *map*, so that $(s_{n+1}, r_{n+1}) \in$ *map* is inductively required.

Preliminary to Theorem 2 defines Def. 18, a reversed conversion of Theorem 1. It maps all parts of a Maude net back to a reconfigurable Petri net so that two states of the related labelled transition system can be associated with each other.

Definition 18 (Surjective mapping between states of LTS_{RPN} and LTS_{MNC})**.** *Given a reconfigurable Petri net* $(N_0, \mathcal{R})$ *with* $N_0 = (P_0, T_0, pre_0, post_0, p_{name0}, t_{name0}, cap_0, M_0)$ *and* $\mathcal{R}$ *as in Def. 6 and the corresponding Maude modules NET and RULE as in Theorem 1. Further, the label transition system is given by* LTS_{RPN} *with Def. 15 and* LTS_{MNC} *by Def. 16. So that there is a mapping* $map : S_{MNC} \to S_{RPN}$ *for some states* $s \in S_{MNC}$ *of Def. 12 with* $s =$ `net(Places, Transitions, Pre, Post, Markings) | Rule Int Int IDPool` *by*
$map(s) = [(N, M)]$ *and*

- $P = \{p | p$ *is an atomic element in buildPlace*$^{-1}$`(Places)`$\}$
- $T = \{t | t$ *is an atomic element in buildTransition*$^{-1}$`(Transitions)`$\}$
- $pre : T \to P^{\oplus}$ *defined by pre(t)* $=$ *buildPlace*$^{-1}$`(place)` *; if*
  `Transitions` $=$ `transitions{T : t(`t_{name} `| x)}` *and*
  `Pre` $=$ `pre{MT, (t(`t_{name} `| x)` $\to$ `place)}`
- $post : T \to P^{\oplus}$ *defined by post(t)* $=$ *buildPlace*$^{-1}$`(place)` *; if*
  `Transitions` $=$ `transitions{T : t(`t_{name} `| x)}` *and*
  `Post` $=$ `post{MT, (t(`t_{name} `| x)` $\to$ `place)}`
- $p_{name} : P \to A_P$ *defined by* $p_{name}(p) =$ `label` *; if*
  `Places` $=$ `places{P , p(label | x | x)}`
- $t_{name} : T \to A_T$ *defined by* $t_{name}(t) =$ `label` *; if*
  `Transitions` $=$ `transitions{T : t(label | x)}`
- $cap : P \to \mathbb{N}$ *defined by* $cap(p) =$ `capacity` *; if*
  `Places` $=$ `places{P , p(str | x | capacity)}`
- $cap : P \to \omega$ *defined by* $cap(p) =$ `w` *; if*
  `Places` $=$ `places{P , p(str | x | w)}`
- $M = \{m | m$ *is an atomic element in buildMarking*$^{-1}(Markings)\}$

Based on Def. 18, the following Lemma 8 and Lemma 9 define the linked states of both transition systems. Lemma 8 is used to link states from S_{MNC} to S_{RPN}, whereby the Lemma 9 link states the inverted direction from S_{RPN} to S_{MNC}.

Lemma 7 (map of the initial state)**.** $(N, M_0) \in$ *map*($\texttt{initial}$) *is given by map as defined in Def. 18*

Proof of Lemma 7. Given `initial` as defined by Def. 12 as *initial = net(Places, Transitions, Pre, Post, Markings)*, then there is a $[(N_0, M_0)]$ as defined by Def. 6 with $N_0 = (P_0, T_0, pre_0, post_0, p_{name0}, t_{name0}, cap_0, M_0)$ so that:

- P_0 is defined by the inverse function of Lemma 1 with $buildPlace^{-1}$ so that $P_0 = \{p | p \text{ is a atomic element in } buildPlace^{-1}(\texttt{Places})\}$
- T_0 is defined by the inverse function of Lemma 2 with $buildTransition^{-1}$ so that $T_0 = \{t | t \text{ is a atomic element in } buildTransition^{-1}(\texttt{Transitions})\}$
- pre_0 is defined by the inverse function of Lemma 3 with $buildPre^{-1}$ so that $pre_0 : T \to P^{\oplus}$ defined by $pre(t) = buildPlace^{-1}(\texttt{place})$; if
  $\texttt{Transitions} = \texttt{transitions}\{\texttt{T : t(}t_{name}\texttt{ | x)}\}$ and
  $\texttt{Pre} = \texttt{pre}\{\texttt{MT}, \texttt{(t(}t_{name}\texttt{ | x)} \to \texttt{place)}\}$
- $post_0$ is defined by the inverse function of Lemma 4 with $buildPost^{-1}$ so that $post_0 : T \to P^{\oplus}$ defined by $post(t) = buildPlace^{-1}(\texttt{place})$; if
  $\texttt{Transitions} = \texttt{transitions}\{\texttt{T : t(}t_{name}\texttt{ | x)}\}$ and
  $\texttt{Post} = \texttt{post}\{\texttt{MT}, \texttt{(t(}t_{name}\texttt{ | x)} \to \texttt{place)}\}$
- p_{name0} is defined by Def. 5 with $p_{name} : P \to A_P$ so that $p_{name}(p) = \texttt{label}$; if
  $\texttt{Places} = \texttt{places}\{\texttt{P , p(label | x | x)}\}$
- t_{name0} is defined by Def. 5 with $t_{name} : T \to A_T$ so that $t_{name}(t) = \texttt{label}$; if
  $\texttt{Transitions} = \texttt{transitions}\{\texttt{T : t(label | x)}\}$

- cap_0 is defined by Def. 5 with $cap : P \to \mathbb{N}_+^w$ so that:
  - $cap(p) =$ `capacity` ; if
    `Places =places{P , p(str | x | capacity)}`
  - $cap(p) =$ `w` ; if
    `Places =places{P , p(str | x | w)}`
- M_0 is defined by the inverse function of Lemma 1 with $buildPlace^{-1}$ so that $M = \{m | m$ is a atomic element in $buildMarking^{-1}(\mathbb{T}_{\texttt{Markings}})\}$

□

Lemma 8 (map as function)**.** *$map : S_{MNC} \to S_{RPN}$ is a function given by map as defined in Def. 18*

Proof of Lemma 8. For each $s \in S_{MNC}$ there is one $r \in S_{RPN}$ with $map(s) = r$.

Basis: For the initial $s_0 \in S_{MNC}$ exists by Lemma 7 an initial state $r_0 \in S_{RPN}$

Induction hypothesis: Let be given a state $s_n \in S_{MNC}$ with $s_n =$ `net(Places, Transitions, Pre, Post, Markings) | Rule Int Int IDPool`, so that $map(s_n) = r_n = [(N, M)]$ with $N = (P, T, pre, post, p_{name}, t_{name}, cap, M)$

Induction step ($n \to n+1$): For each follower state $s_{n+1} \in S_{MNC}$ with $s_n \xrightarrow{l} s_{n+1} \in tr_{MNC}$ there is a $r_{n+1} \in S_{RPN}$ with $r_n \xrightarrow{l} r_{n+1} \in tr_{RPN}$ and $map(s_{n+1}) = r_{n+1}$ so that l can be applied by:

- Firing by $s_n \xrightarrow{t_{name}(t_s)} s_{n+1}$ as in Def. 16 with $s_{n+1} =$ `net(Places, Transitions, Pre, Post, Markings′) | Rule Int Int IDPool` and by the isomorphism class of Def. 9 there is also a $r_n \xrightarrow{t_{name}(t_r)} r_{n+1}$ as in Def. 15 with $r_{n+1} = [(N, M')]$ so that
  - *Activation:*
    If `marking{PreValue ; M}` can be rewritten by the rewrite rule `[fire]` defined in Def. 7 and Listing 4, then the `PreValue` for t_s is less or equal than the marking of s_n. Hence, is $pre^{\oplus}(t_r) \leq M_r$ (line one of Def. 7) and $r_n \xrightarrow{t_{name}(t_r)} r_{n+1} \in tr_{RPN}$ due to $M[t_r\rangle M'$ in N with $r_{n+1} = [N, M']$, $t_{name}(t_s) = t_{name}(t_r)$ and $t_{name}(t_r) \in A_{RPN}$.
  - *Capacity limitation:*
    If `(PreValue ; M) plus PostValue` can be rewritten by the

rewrite rule `[fire]` defined in Def. 7 and Listing 4, then the `PostValue` for each place used by t_s is less or equal than the capacity and $M + \mathit{post}^{\oplus}(t_r) \leq \mathit{cap}$ for t_r (line two of Def. 7)

- *New marking:*
  If `calc(((PreValue ; M) minus PreValue) plus PostValue)` can be rewritten by the rewrite rule `[fire]` defined in Def. 7 and Listing 4, then the following marking `Markings'` is given and the marking for r_{n+1} is calculated by $M' = (M_r \ominus \mathit{pre}^{\oplus}(t_r)) \oplus \mathit{post}^{\oplus}(t_r)$. (line three of Def. 7)

- Transformation by $s_n \xrightarrow{r_{name}(r_s)} s_{n+1}$ as in Def. 16 with $s_{n+1} =$ `net(Places, Transitions, Pre, Post, Markings') | Rule Int Int IDPool` and the isomorphism class of Def. 9 there is also a $r_n \xrightarrow{r_{name}(r_r)} r_{n+1}$ as in Def. 15 with $r_{n+1} = [(N', M')]$ so that
  - *match:*
    If s_n can be rewritten by the rewrite rule `[r`$_{name}$`]` defined in Def. 11 and Listing 5, then is the L a subset of s_n. Hence, there is an occurrence $o : L \to N$ defined in Def. 15 by r_r and $r_n \xrightarrow{r_{name}(r_r)} r_{n+1} \in \mathit{tr}_{RPN}$ as well as $r_{name}(r_s) = r_{name}(r_r)$.
  - `freeOfMarking` applies for each deleted place $p \notin$ `MRest`, as defined in Def. 11 by the identification condition in Def. 10
  - `emptyNeighbourForPlace` applies for each deleted place p no occurrence in `Pre` and `Post`, as defined in Def. 11 by the dangling condition in Def. 10

□

Lemma 9 (map as surjective function)**.** *map* : $S_{RPN} \to S_{MNC}$ *is a surjective function given by map as defined in Def. 18*

Proof of Lemma 9. For each $r \in S_{RPN}$ there is one $s \in S_{MNC}$ with $map(s) = r$

Basis: For the initial $r_0 \in S_{RPN}$ exists by Theorem 1 an initial state $s_0 \in S_{MNC}$

Induction hypothesis: Let be given a state $r_n \in S_{RPN}$ with $r_n = [N, M]$ so that there is a $s_n \in S_{MNC}$ with $map(s_n) = r_n = [(N, M)]$ of Def. 18 and $N = (P, T, pre, post, p_{name}, t_{name}, cap, M)$.

Induction step ($n \to n+1$)**:** For each follower state $r_{n+1} \in S_{RPN}$ with $r_n \xrightarrow{l} r_{n+1} \in tr_{RPN}$ there is a $s_{n+1} \in S_{MNC}$ with $s_n \xrightarrow{l} s_{n+1} \in tr_{MNC}$ and $map(s_{n+1}) = r_{n+1}$ so that l can be applied by:

- Firing by $r_n \xrightarrow{t_{name}(t_r)} r_{n+1} \in tr_{RPN}$ as in Def. 15 with $r_{n+1} = [(N, M')]$ there is by Def. 16 also a $s_n \xrightarrow{t_{name}(t_s)} s_{n+1} \in tr_{MNC}$ with $s_{n+1} =$ `net(Places, Transitions, Pre, Post, Markings′) | Rule Int Int IDPool` so that
  - *Activation:*
    If $pre^{\oplus}(t_r) \leq M_r$ (line one of Def. 7) and $r_n \xrightarrow{t_{name}(t_r)} r_{n+1} \in tr_{RPN}$ due to $M[t_r\rangle M'$ in N with $r_{n+1} = [N, M']$, then t_r is activated. Hence, `marking{PreValue ; M}` can be rewritten by the rewrite rule `[fire]` defined in Def. 7 and Listing 4, so that `PreValue` for t_s is the less or equal than the marking of s_n as well as $t_{name}(t_s) = t_{name}(t_r)$ and $t_{name}(t_r) \in A_{RPN}$.
  - *Capacity limitation:*
    If $M + post^{\oplus}(t_r) \leq cap$ for t_r, then the `PostValue` is less or equal than the capacity for each place used by t_s (line two of Def. 7). Hence, `(PreValue ; M) plus PostValue` can be rewritten by the rewrite rule `[fire]` defined in Def. 7 and Listing 4,
  - *New marking:*
    If the following marking for r_{n+1} is calculated by $M' = (M_r \ominus pre^{\oplus}(t_r)) \oplus post^{\oplus}(t_r)$ (line three of Def. 7), then `calc(((PreValue ; M) minus PreValue) plus PostValue)` can be rewritten as the following marking `Markings′` by the rewrite rule `[fire]` defined in Def. 7 and Listing 4.

- Transformation by $r_n \xrightarrow{r_{name}(r_r)} r_{n+1} \in tr_{RPN}$ as in Def. 15 with $r_{n+1} = [(N', M')]$ there is by Def. 16 also a $s_n \xrightarrow{r_{name}(r_s)} s_{n+1} \in tr_{MNC}$ with $s_{n+1} =$ `net(Places, Transitions, Pre, Post, Markings′) | Rule Int Int ID-Pool` so that
  - *match:*
    If there is an occurrence $o : L \to N$ defined in Def. 15 by r_r and $r_n \xrightarrow{r_{name}(r_r)} r_{n+1} \in tr_{RPN}$, then s_n can be rewritten by the rewrite rule $[\mathtt{r}_{name}]$ defined in Def. 11 and Listing 5 by $\mathrm{L} \subseteq s_n$.
  - `freeOfMarking` applies for each deleted place $p \notin$ `MRest`, as defined in Def. 11 by the gluing condition
  - `emptyNeighbourForPlace` applies for each deleted place $p \notin$ `Pre`$\wedge$ $p \notin$ `Post` as defined in Def. 11 by the gluing condition

□

Remark 4. *The function map in Lemma 9 is not injective due to the isomorphism class in Def. 9.*

Theorem 2 (Bisimulation of LTS_{RPN} and LTS_{MNC})**.** *LTS_{RPN} and LTS_{MNC} are bisimilar as defined in Def. 2 by map in Def. 18*

Proof of Theorem 2. For each relation defined by *map* of Def. 18, which consists of $s \in S_{MNC}$ and $r \in S_{RPN}$ with $map(s) = r = [N, M]$, we have:

- $s \to s'$: For each $a \in A_{MNC}$ there is $map(s) = r$ and $r \xrightarrow{a} r' \in tr_{RPN}$, due to $s \xrightarrow{a} s' \in tr_{MNC}$ and the mapping of Lemma 8 and Lemma 9 there is $map(s') = r'$ by Lemma 8 and Lemma 9.
- $r \to r'$: For each $a \in A_{MNC}$ there is $map(s) = r$ and $s \xrightarrow{a} s' \in tr_{MNC}$, due to $r \xrightarrow{a} r' \in tr_{RPN}$ and the mapping of Lemma 8 and Lemma 9 there is $map(s') = r'$ by Lemma 8 and Lemma 9.

So that a bisimulation between LTS_{RPN} and LTS_{MNC} is defined by the map function. □

5.3 Résumé by the Correctness of Model Checking for Maude

The previous chapter introduced the formal argumentation for the implementation correctness of the Maude net defined in [3]. The correctness is reasoned by the conversion in Theorem 1, which converts a reconfigurable Petri net to a Maude net as well as the bisimulation in the proof of Theorem 2.

The bisimulation of Theorem 2 proves the behaviour equivalence for both labelled transition systems by the definition in Def. 2. As results, the proof of Theorem 1 shows that there is the possibility of a conversion from a reconfigurable Petri net into a Maude net. Further, the proof of Theorem 2 clarified that there exists a behaviour equivalence for both labelled transition systems.

6 Evaluation

This chapter evaluates the performance of the approach written with Maude in version 2.7, including LTLR in version 1.0[1] against the established tool Charlie version 2.0[2].

A Petri net with equal semantics, as a reconfigurable Petri net, is used as an example in Figure 15. This net is used to perform a comparative analysis, including a transfer into a flat Petri net, where all transformation steps are modelled as separated nets that include the transformation results. The nets, as shown in Figure 16, are created with Snoopy2 in version 1.13 [3] based on the reachability graph that contains all firing steps as usual arcs and transformation steps as dotted arcs[4]. The Figure 15 models the flight routes between Hamburg, Berlin, and Munich. For each *fly*-transition there exists a rule that handles flight route changes which are modelled as direction alters. A route change can occur when an aircraft is expected at another airport, where it will serve another flight. This behaviour is modelled by the replacement of a transition, including a switching of the *pre*- and *post*-domains (cf. Figure 5).

[1] `http://maude.cs.illinois.edu/tools/tlr/`, 11 March 2015

[2] `http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Charlie`, 11 March 2015

[3] `http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Snoopy`, 11 March 2015

[4] The nets modelled in Snoopy2 are manual created. An automation is part of the future work, where extend benchmarks are challenged.

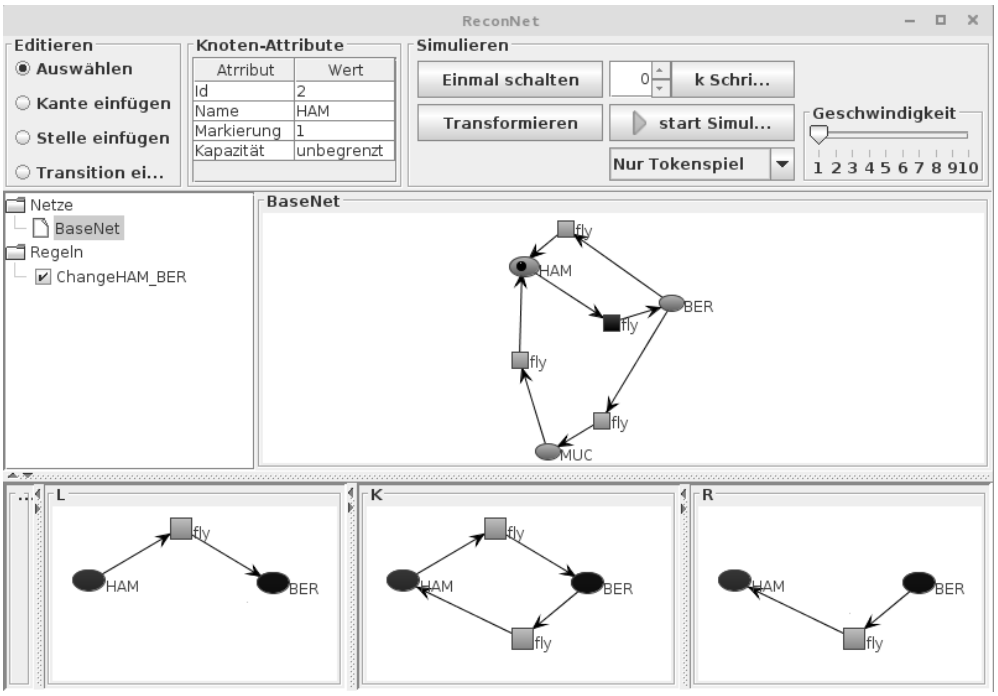


Figure 15: Flight routes net for evaluation tests

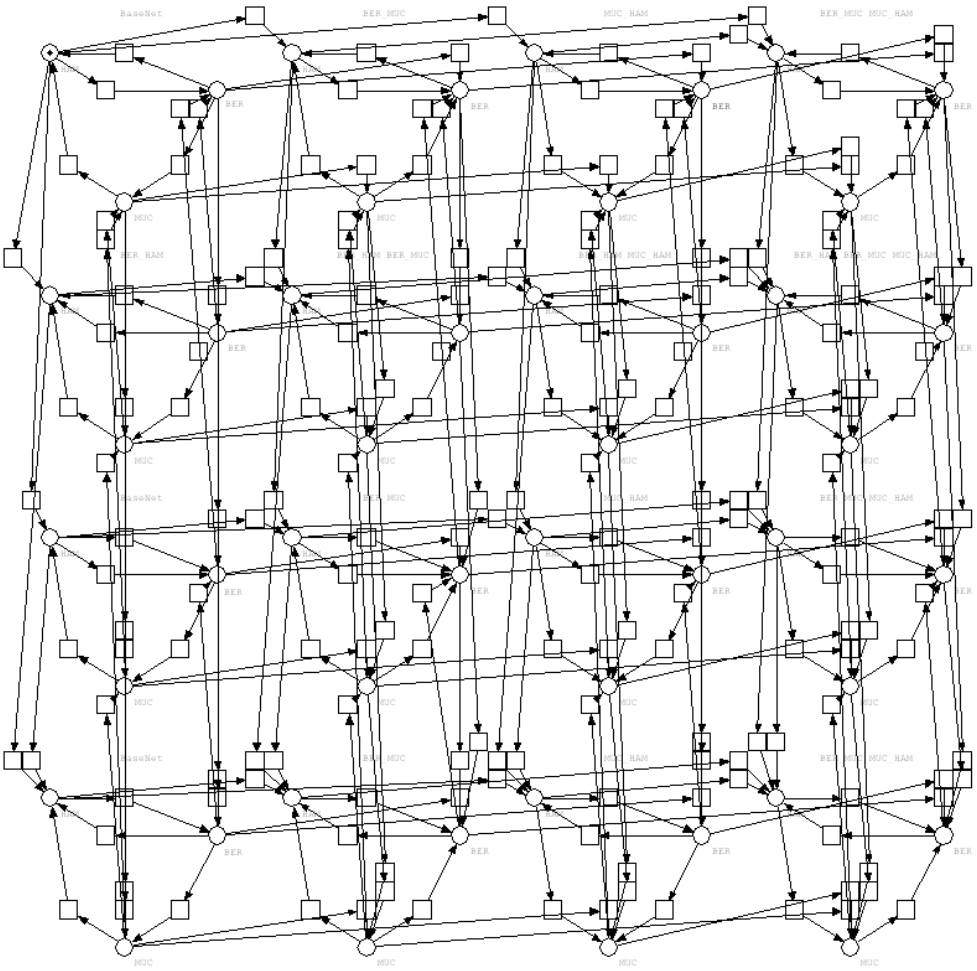


Figure 16: Snoopy net of Figure 15 and all rules: HAM-BER, BER-MUC, MUC-HAM and BER-HAM

Table 1 contains all evaluation results of the reachability graph construction by Charlie and Maude. All testcases are based on the previous test case (denoted by '* +') so that the reachability graph is extended step by step with new rules. Furthermore, information about states and edges are carried together for each testcase. For better reading, Figure 17 shows the data from Table 1.

	Charlie	States	Edges	Maude	States	Edges
* + HAM-BER	125,2	6	11	51,5	6	11
* + BER-MUC	125,3	12	28	51,9	12	28
* + MUC-HAM	131,4	24	68	51,8	24	68
* + BER-HAM	147,1	48	428	60,8	48	208

Table 1: Evaluation results of reachability graph analysis between Charlie and Maude (in milliseconds)

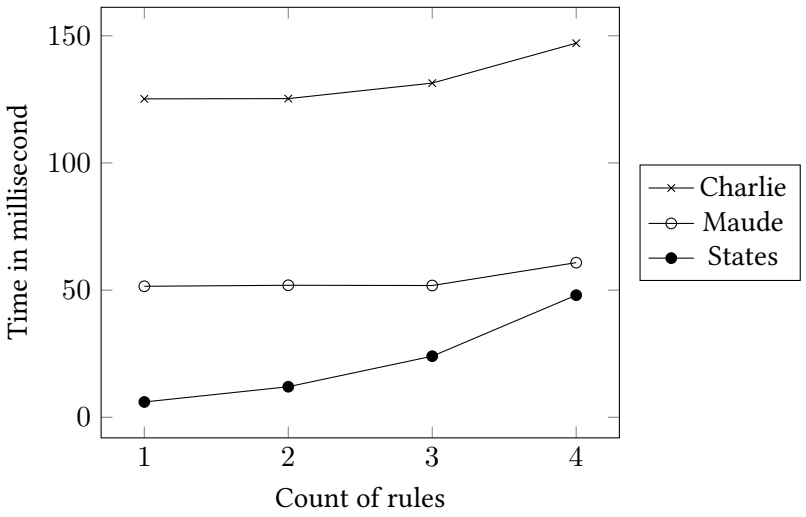


Figure 17: Compare collected data as graph

Besides the reachability graph generation, metadata is calculated for both systems. Maude reveals deadlocks with `Solution` during the tracing of the state generation, and Charlie has a metadata overview window. The Table 2 connects all results for both systems with equal results.

	Charlie	Maude
* + HAM-BER	yes (at HAM)	yes (at HAM)
* + BER-MUC	yes (at HAM)	yes (at HAM)
* + MUC-HAM	no	no
* + BER-HAM	no	no

Table 2: Evaluation results of the reachability graph analysis between Charlie and Maude (in milliseconds)

7 Future work

An integration of *rMC* in *ReConNet*, including the workflow that converts a reconfigurable Petri net into Maude modules, is advantageous. Executions for analysis are also important for a helpful representation of the received information that can be used as assistance for the net designer. Due to the complexity of a net as well as a set of rules, it is necessary to show the reachability graph (e.g., Figure 11 or Figure 12). Information on deadlocks or circular dependencies is simpler to understand, if presented in a graphical form.

Evolution of the reachability graph by using a coverability graph is required as well as the general improvement of the Maude net. Reducing the size of the scope and states of the reachability graph is an important challenge due to the performance. For example, it reduces the loops of the coverability graph, which are created through the insertion of a transition infinite times by a rule. Further, the implementation of the identifier pool becomes obsolete if such rules are put in place.

The implementation of more complex sort-structures are useful to prevent Maude net structure issues. Currently, it is possible to add two place wrappers into each other such as `places{ places{...} }`. An example approach, which solves this issue, is shown in Listing 13. It inserts a new `sort` beside the existing sort `Places` and extends it by `Place`. `Place` is used as a sort for single `place`-terms. The wrapper `places` uses `Place`, instead of `Places` so that it only takes terms such as `p(<label>|<identifier>|<capacity>)`.

```
sorts Places Place .
op p(_|_|_) : String Int Int -> Place .
op _,_ : Place Place -> Place .
op places : Place -> Places .
```

Listing 13: Prevent Maude net sort structure issues

Enhancements of LTL properties are useful for simplified spellings of formulae such as the reachability of a set of nodes by name. *prop.maude* actually contains operators that enable users to ask for the reachablility of places by label, identifier, and capacity with the `reachable`-operator. Another example is the deadlock-freeness of a Maude net by the *enabled*-operator (including only transition activation by *t-enabled*). Operators that use only the label of a place make more sense since identifiers can be changed by rule applications. A sample implementation is presented in Listing 14, where only one label is used to prove whether a place is reachable.

```
op reachable : String -> Prop .

eq net(P, T, Pre, Post,
       marking{ p(L | I | Cap) ; MRest } )
       Rules MaxID StepSize aid
       |= reachable(L) = true .
```

Listing 14: Example of an extending operator for the LTL formulae

An implementation of special features such as negative application conditions (*NACs*) (see [42]) or decorations (see [10]) is useful. *NACs* are used as net states for rules which should not occur. Decoration includes the definition and implementation of *tlb* and *rnw* functions in Maude. The *tlb* function maps a transition label to a specific transition. Further, those labels can be changed with the *rnw* function so that it is possible to add functions into a transition. Some example applications are executable calculations such as counting an integer.

An interactive front end, which displays the collected LTL results, is a necessary extension for the graphical implementation in *ReConNet*. Traces of the printed LTL-paths can help a user to understand problems of the net and rules. If, for example, a deadlock occurs, then an animation can show all actions that lead to the dead state. Such an animation can present all actions, such as firing and transformation steps, which end in a dead state where no action can be used. Currently, the functionality of the front end supports only the generation of Maude modules or reachability graphs in PDF- or SVG-files.

Finally, advanced benchmarks are useful to compare tools as *Charlie* against *rMC*. The automatization of testcases, including a database of different net structures and rules, provides differentiated results. The example in Figure 15 contains an exponential growing state space that is magnificently handled by Maude net, but the net is minor due to the count of places and transitions. A database of different net structures can express an abstracted result in contrast to the specific example net. Further, the testcases can be extended to other tools like Groove[1], which uses graph grammars to express rules.

[1] `http://groove.cs.utwente.nl/`, 24 April 2015

8 Summary and Conclusion

The model checking for computer-based systems such as Maude's linear temporal logic of rewriting by term replacement algebra is a well understood technique to verify the behavioural properties of a given system. Maude's intuitive writing and the logical model are suitable for the aim of this thesis, as they define a model for reconfigurable Petri nets in term algebra.

The concurrent and distributed model of reconfigurable Petri nets can be written with term algebra by a conversion of Theorem 1. Each part of such a reconfigurable Petri net is converted into a Maude net. Furthermore, actions such as firing or transforming are defined so that labelled transition systems can be defined for both models and their related inference rules.

Bisimulation of labelled transition systems requires behaviour equivalence for each state mapping in a relation defined between both systems. Theorem 2 and the related *map* function define such a bisimilarity for a given reconfigurable Petri net and a Maude net.

The aim of this thesis is summarized in Figure 18, where the required conversion is introduced by Theorem 1 and the bisimulation by Theorem 2. A given reconfigurable Petri net $((N, M), \mathcal{R})$ is converted with several functions into a Maude net term. The labelled transition systems are derived by the inference rules of both nets. Finally, the Theorem 2 defines the bisimulation between both labelled transition systems by a behaviour equivalence.

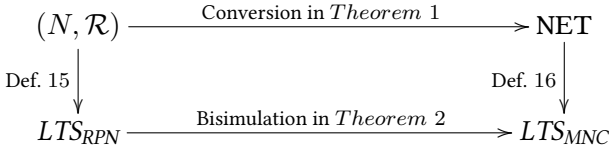


Figure 18: Correctness of conversion

The result of the proof for Theorem 1 clarified that a Maude net of Def. 14 is a valid representation of a formally defined reconfigurable Petri net. A conversion is formally defined and proven so that it is possible to implement a conversion that transmits all parts of such model. The project before this thesis (see [3]) contains such a conversion, which uses a PNML-file as well as a XSL implementation.

Further, the proof of bisimilarity shows that both systems are behaviourally equivalent. The states and actions of a labelled transition system, derived from a reconfigurable Petri net, show that a Maude net calculates behaviourally equivalent states. Therefore, is a *map*-function by Def. 18 defined as relation, which maps states of both transition system.

Consequently, the proofs show that the verification of reconfigurable Petri nets and Maude nets are correct using model checking by Maude's LTLR. For this purpose, shows the evaluation that properties such as deadlocks can be detected by the implementation of the model and special operators for the LTL process. Further, Maude generates a text-based reachability graph for a finite state space. As side effect is such state space generated by *rMC* as graphic (e.g. Figure 11).

Bibliography

[1] Christel Baier and Joost-Pieter Katoen. Principles of model checking, 2008.

[2] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. ISBN 3-540-10235-3. doi: 10.1007/3-540-10235-3. URL http://dx.doi.org/10.1007/3-540-10235-3.

[3] Alexander Schulz. Converting reconfigurable Petri nets to maude. *CoRR*, abs/1409.8404, 2014. URL http://arxiv.org/abs/1409.8404.

[4] Hartmut Ehrig and Julia Padberg. Graph grammars and Petri net transformations. In *Lectures on Concurrency and Petri Nets*, pages 496–536. Springer Science, 2004. doi: 10.1007/978-3-540-27755-2_14. URL http://dx.doi.org/10.1007/978-3-540-27755-2_14.

[5] M. Llorens and J. Oliver. Structural and dynamic changes in concurrent systems: reconfigurable Petri nets. *IEEE Trans. Comput.*, 53(9):1147–1158, sep 2004. doi: 10.1109/tc.2004.66. URL http://dx.doi.org/10.1109/tc.2004.66.

[6] Julia Padberg. Abstract interleaving semantics for reconfigurable Petri nets. *ECEASST*, 51, 2012. URL http://journal.ub.tu-berlin.de/eceasst/article/view/775.

[7] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, apr 1992. doi: 10.1016/0304-3975(92)90182-f. URL http://dx.doi.org/10.1016/0304-3975(92)90182-f.

[8] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude as a metalanguage. *Electr. Notes Theor. Comput. Sci.*, 15:147–160, 1998. doi: 10.1016/S1571-0661(05)82557-5. URL http://dx.doi.org/10.1016/S1571-0661(05)82557-5.

[9] Mark-Oliver Stehr, José Meseguer, and Peter Csaba Ölveczky. Rewriting logic as a unifying framework for Petri nets. In *Unifying Petri Nets*, pages 250–303. Springer Science, 2001. doi: 10.1007/3-540-45541-8_9. URL http://dx.doi.org/10.1007/3-540-45541-8_9.

[10] Julia Padberg, Marvin Ede, Gerhard Oelker, and Kathrin Hoffmann. Reconnet: A tool for modeling and simulating with reconfigurable place/transition nets. volume 54, 2012. URL http://journal.ub.tu-berlin.de/eceasst/article/view/774.

[11] Julia Padberg and Alexander Schulz. Towards model checking reconfigurable Petri nets using maude. *ECEASST*, 68, 2014. URL http://journal.ub.tu-berlin.de/eceasst/article/view/953.

[12] Leslie Lamport. What good is temporal logic? In *IFIP Congress*, pages 657–668, 1983.

[13] David Michael Ritchie Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science, 5th GI-Conference, Karlsruhe, Germany, March 23-25, 1981, Proceedings*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981. doi: 10.1007/BFb0017309. URL http://dx.doi.org/10.1007/BFb0017309.

[14] Robin Milner. *Communication and concurrency*, volume 84. Prentice hall New York etc., 1989. ISBN 0-13-115007-3.

[15] Davide Sangiorgi and Jan Rutten, editors. *Advanced Topics in Bisimulation and Coinduction*. Cambridge University Press, 2011. ISBN 9780511792588. URL http://dx.doi.org/10.1017/CBO9780511792588. Cambridge Books Online.

[16] Santiago Escobar, José Meseguer, and Ralf Sasse. Variant narrowing and equational unification. *Electr. Notes Theor. Comput. Sci.*, 238(3):103–119, 2009. doi: 10.1016/j.entcs.2009.05.015. URL http://dx.doi.org/10.1016/j.entcs.2009.05.015.

[17] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002. doi: 10.1016/S0304-3975(01)00359-0. URL http://dx.doi.org/10.1016/S0304-3975(01)00359-0.

[18] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The maude LTL model checker. *Electr. Notes Theor. Comput. Sci.*, 71:162–187, 2002. doi:

10.1016/S1571-0661(05)82534-4. URL http://dx.doi.org/10.1016/S1571-0661(05)82534-4.

[19] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The maude LTL model checker and its implementation. 2648:230–234, 2003. doi: 10.1007/3-540-44829-2_16. URL http://dx.doi.org/10.1007/3-540-44829-2_16.

[20] José Meseguer. *A Logical Theory of Concurrent Objects*. ACM, 1990. doi: 10.1145/97945.97958. URL http://doi.acm.org/10.1145/97945.97958.

[21] Carl Adam Petri. Kommunikation mit Automaten. 1962.

[22] José Meseguer and U. Montanari. Petri nets are monoids: a new algebraic foundation for net theory. pages 155–164, 1988. doi: 10.1109/LICS.1988.5114.

[23] Gabriel Juhás, Fedor Lehocki, and Robert Lorenz. Semantics of Petri nets: a comparison. In Shane G. Henderson, Bahar Biller, Ming-Hua Hsieh, John Shortle, Jeffrey D. Tew, and Russell R. Barton, editors, *Proceedings of the Winter Simulation Conference, WSC 2007, Washington, DC, USA, December 9-12, 2007*, pages 617–628. WSC, 2007. doi: 10.1145/1351542.1351661. URL http://doi.acm.org/10.1145/1351542.1351661.

[24] Hartmut Ehrig, Kathrin Hoffmann, Julia Padberg, Ulrike Prange, and Claudia Ermel. Independence of net transformations and token firing in reconfigurable place/transition systems. In Jetty Kleijn and Alexandre Yakovlev, editors, *Petri Nets and Other Models of Concurrency - ICATPN 2007, 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, ICATPN 2007, Siedlce, Poland, June 25-29, 2007, Proceedings*, volume 4546 of *Lecture Notes in Computer Science*, pages 104–123. Springer, 2007. doi: 10.1007/978-3-540-73094-1_9. URL http://dx.doi.org/10.1007/978-3-540-73094-1_9.

[25] Ulrike Prange, Hartmut Ehrig, Kathrin Hoffmann, and Julia Padberg. Transformations in reconfigurable place/transition systems. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lecture Notes in Computer Science*, pages 96–113. Springer, 2008. doi: 10.1007/978-3-540-68679-8_7. URL http://dx.doi.org/10.1007/978-3-540-68679-8_7.

[26] Laïd Kahloul, Allaoua Chaoui, and Karim Djouani. Modeling and analysis of reconfigurable systems using flexible Petri nets. In Jing Liu, Doron Peled, Bow-Yaw Wang, and Farn Wang,

editors, *4th IEEE International Symposium on Theoretical Aspects of Software Engineering, TASE 2010, Taipei, Taiwan, 25-27 August 2010*, pages 107–116. IEEE Computer Society, 2010. doi: 10.1109/TASE.2010.28. URL http://dx.doi.org/10.1109/TASE.2010.28.

[27] Hartmut Ehrig, Frank Hermann, and Ulrike Prange. Cospan DPO approach: An alternative for DPO graph transformations. *Bulletin of the EATCS*, 98:139–149, 2009.

[28] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and Carolyn Talcott. Maude manual (version 2.6). *University of Illinois, Urbana-Champaign*, 1(3):4–6, 2011.

[29] Mark-Oliver Stehr, José Meseguer, and Peter Csaba Ölveczky. Rewriting logic as a unifying framework for Petri nets. In *Unifying Petri Nets, Advances in Petri Nets*, pages 250–303, 2001. doi: 10.1007/3-540-45541-8_9. URL http://dx.doi.org/10.1007/3-540-45541-8_9.

[30] W. Chama, R. Elmansouri, and A. Chaoui. Using graph transformation and maude to simulate and verify UML models. In *Technological Advances in Electrical, Electronics and Computer Engineering (TAEECE), 2013 International Conference on*, pages 459–464, May 2013. doi: 10.1109/TAEECE.2013.6557318.

[31] Paulo E. S. Barbosa, João Paulo Barros, Franklin Ramalho, Luís Gomes, Jorge Figueiredo, Filipe Moutinho, Anikó Costa, and André Aranha. Sysveritas: A framework for verifying IOPT nets and execution semantics within embedded systems design. In *Technological Innovation for Sustainability - Second IFIP WG 5.5/SOCOLNET Doctoral Conference on Computing, Electrical and Industrial Systems, DoCEIS 2011, Costa de Caparica, Portugal, February 21-23, 2011. Proceedings*, pages 256–265, 2011. doi: 10.1007/978-3-642-19170-1_28. URL http://dx.doi.org/10.1007/978-3-642-19170-1_28.

[32] Noura Boudiaf and Abdelhamid Djebbar. Towards an automatic translation of colored Petri nets to maude language. *International Journal of Computer Science & Engineering*, 3 (1), 2009.

[33] Monika Heiner, Mostafa Herajy, Fei Liu, Christian Rohr, and Martin Schwarick. Snoopy - A unifying Petri net tool. In *Application and Theory of Petri Nets - 33rd International Conference, PETRI NETS 2012, Hamburg, Germany, June 25-29, 2012. Proceedings*, pages 398–407, 2012. doi: 10.1007/978-3-642-31131-4_22. URL http://dx.doi.org/10.1007/978-3-642-31131-4_22.

[34] Monika Heiner, Ronny Richter, and Martin Schwarick. Snoopy: a tool to design and animate/simulate graph-based formalisms. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops, SimuTools 2008, Marseille, France, March 3-7, 2008*, page 15, 2008. doi: 10.4108/ICST.SIMUTOOLS2008.3098. URL http://dx.doi.org/10.4108/ICST.SIMUTOOLS2008.3098.

[35] J Wegener, M Schwarick, and M Heiner. A plugin system for charlie. In *Proc. International Workshop on Concurrency, Specification, and Programming (CS&P 2011)*, ISBN: 978-83-62582-06-8, pages 531–554. Biaystok University of Technology, September 2011. URL http://csp2011.mimuw.edu.pl/proceedings/index.html.

[36] Arend Rensink. The GROOVE simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers*, pages 479–485, 2003. doi: 10.1007/978-3-540-25959-6_40. URL http://dx.doi.org/10.1007/978-3-540-25959-6_40.

[37] Arend Rensink, Ákos Schmidt, and Dániel Varró. Model checking graph transformations: A comparison of two approaches. In *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings*, pages 226–241, 2004. doi: 10.1007/978-3-540-30203-2_17. URL http://dx.doi.org/10.1007/978-3-540-30203-2_17.

[38] Joakim Bjørk. Executing large scale colored Petri nets by using maude. *Hovedfagsoppgave, Department of Informatics, Universitetet i Oslo*, 2006.

[39] Joakim Bjørk and Anders M Hagalisletto. Challenges in simulating railway systems using Petri nets.

[40] José Meseguer. Membership algebra as a logical framework for equational specification. In Francesco Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1997. doi: 10.1007/3-540-64299-4_26. URL http://dx.doi.org/10.1007/3-540-64299-4_26.

[41] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. *Theor. Comput. Sci.*, 236(1-2):35–132, 2000.

doi: 10.1016/S0304-3975(99)00206-6. URL http://dx.doi.org/10.1016/S0304-3975(99)00206-6.

[42] Alexander Rein, Ulrike Prange, Leen Lambers, Kathrin Hoffmann, and Julia Padberg. Negative application conditions for reconfigurable place/transition systems. *ECEASST*, 10, 2008. URL http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/140.

Appendices

A Evaluation nets for Snoopy

This section contains all Snoopy nets, for example, the evaluation flight route. The evaluation is constructed in steps so that Figure 19 contains the initial example net, which results with rule *HAM-BER*. Figure 20 extends the net in Figure 19 by rule *BER-MUC*. *MUC-HAM* are added in Figure 21, and finally, the Figure 16 contains all four rules with rule *BER-HAM*.

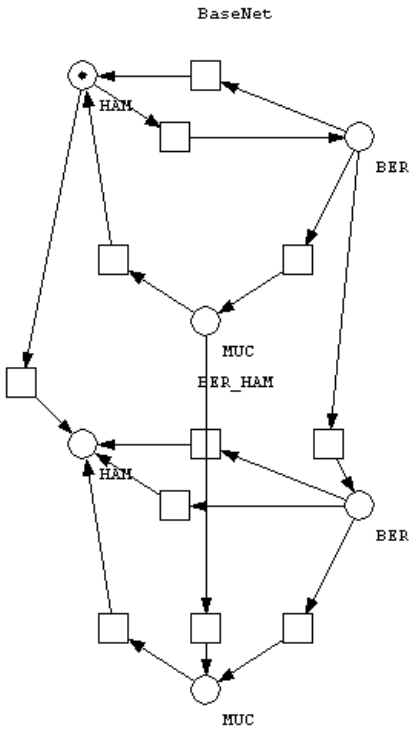


Figure 19: Snoopy net of Figure 15 and rule HAM-BER

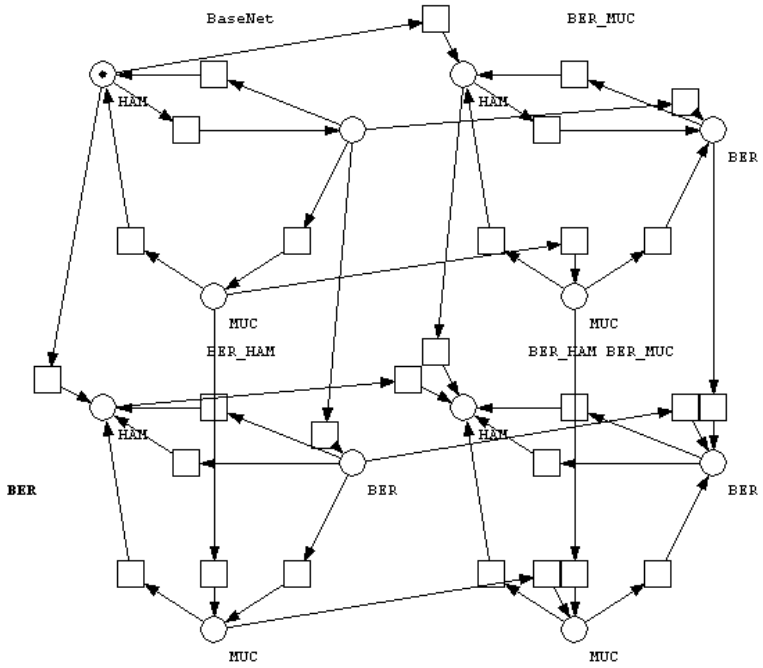


Figure 20: Snoopy net of Figure 15 and rules: HAM-BER and BER-MUC

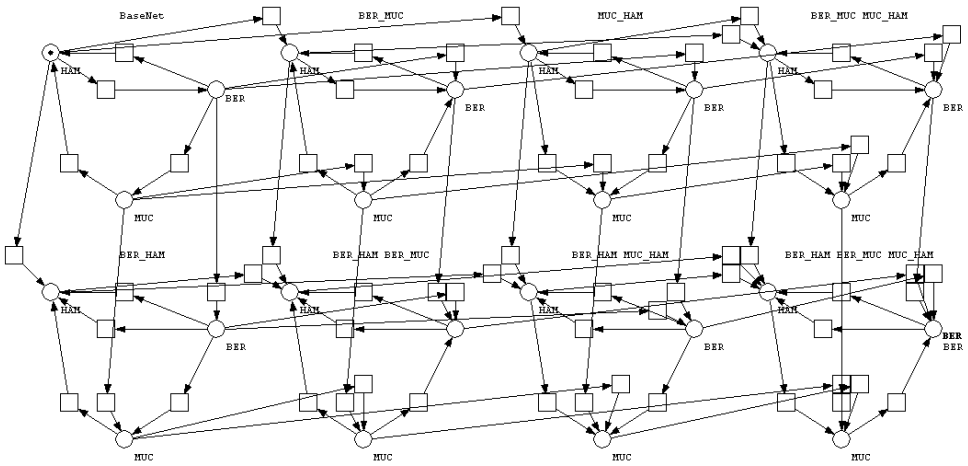


Figure 21: Snoopy net of Figure 15 and rules: HAM-BER, BER-MUC and MUC-HAM

B Extended Example of a Maude net

This section contains all Maude modules for N_1 and r_1. At first, Listing 15 presents the implementation of the NET module for the Maude net defined in Def. 14 and the Maude net in Def. 12. Listing 16 contains the implementation of the RULES module defined in Def. 13, including the rewrite rule that is generated for r_1 to perform such a transformation step, including Maude's pattern matching to detect a match. Listing 17 shows the implementation of the PROP module with all necessary LTL properties. Finally, Listing 18 defines the NET module with the initial state.

```
mod RPN is
  protecting INT .
  protecting STRING .

*** ####################################################
*** Local VARs
*** ####################################################

var Str Str1 Str2 : String .
var I I1 I2 I3 Cap Cap1 Cap2 Counter MaxID StepSize :
    Int .
var P PRest PSet PSetL PSetR MTupleValue PreValue
    PostValue P1 P2 : Places .
var T TRest TSet TSetL TSetR : Transitions .
var Pre PreL PreR : Pre .
var Post PostL PostR : Post .
var MTupleRest MTupleRest1 MTupleRest2 : MappingTuple .
var M M1 M2 MNew MRest MRest1 MRest2 MFollow : Markings
    .
var R RRest Rules : Rule .
var aid : IDPool .


*** ####################################################
*** Petrinet N = (P, T, Pre, Post, M_0)
```

```
    *** ##################################################
24
    sort Net .
26  sort Places .
    sort Transitions .
28  sort Pre .
    sort Post .
30  sort MappingTuple .
    sort Markings .
32  sort Omega .

34  subsort Places < Markings .

36  op emptyPlace : -> Places .
    op emptyTransition : -> Transitions .
38  op emptyMappingTuple : -> MappingTuple .
    op emptyMarking : -> Markings .
40  op w : -> Omega .

42  op _,_ : Places Places -> Places [ctor assoc comm id:
       emptyPlace] .
    op _+_ : Places Places -> Places [ctor assoc comm id:
       emptyPlace] .
44  op _:_ : Transitions Transitions ->
              Transitions [ctor assoc comm id:
                 emptyTransition] .
46  op _,_ : MappingTuple MappingTuple ->
              MappingTuple [ctor assoc comm id:
                 emptyMappingTuple] .
48  op _;_ : Markings Markings -> Markings [ctor assoc comm
       id: emptyMarking] .

50  *** READING: Pname | ID | Cap
    op p(_|_|_) : String Int Int -> Places .
52  op p(_|_|_) : String Int Omega -> Places .
```

```
    op t(_|_) : String Int -> Transitions .
54
    op (_-->_) : Transitions Places -> MappingTuple .
56
    op places{_} : Places -> Places .
58  op transitions{_} : Transitions -> Transitions .
    op pre{_} : MappingTuple -> Pre .
60  op post{_} : MappingTuple -> Post .
    op marking{_} : Markings -> Markings .
62
    *** Petrinet-tuple
64  op net : Places Transitions Pre Post Markings -> Net .

66
    *** ##################################################
68  *** Firing of N = (P, T, Pre, Post, M_0)
    *** ##################################################
70
    op calc : Markings -> Markings .
72  op _plus_ : Markings Markings -> Markings .
    op _minus_ : Markings Markings -> Markings .
74
    *** ################################
76  *** Enable AND Calc
    *** ################################
78
    op _ leeqth _ with _ : Places Places Int -> Bool .
80  op _ <=? _ : Markings Places -> Bool .

82  *** Impl - lowerEqualThan ##########

84  *** place multiset is empty
    ceq emptyMarking leeqth p(Str | I | Cap1) with Counter
86      = true if Counter <= Cap1 .
```

```
88  *** Cap-counter is too big
    eq (p(Str | I | Cap2) ; MRest) leeqth p(Str | I | Cap2)
       with (Cap2 + 1)
90     = false .

92  *** found same place
    ceq (p(Str | I | Cap2) ; MRest) leeqth p(Str | I | Cap2)
        with Counter
94      = true
        if (MRest leeqth p(Str | I | Cap2) with (Counter +
           1)) .
96
    *** del another place
98  ceq (p(Str | I | Cap1) ; MRest) leeqth p(Str2 | I2 |
       Cap2) with Counter
        = true
100     if I =/= I2 /\
           (MRest leeqth p(Str2 | I2 | (Cap2)) with Counter)
                .
102 ceq (p(Str | I | w) ; MRest) leeqth p(Str2 | I2 | Cap2)
       with Counter
        = true
104     if I =/= I2 /\
           (MRest leeqth p(Str2 | I2 | (Cap2)) with Counter)
                .
106
    *** otherwise
108 eq M leeqth P with I = false [owise] .

110 *** Impl - smallerAsCap #############

112 eq marking{ PSet } <=? emptyPlace = true .

114 eq marking{M} <=? (p(Str | I | w) , emptyPlace)
        = true .
```

```
116
    ceq marking{M} <=? (P , emptyPlace)
118     = true
        if M leeqth P with 0 .
120
    ceq marking{M} <=? (p(Str | I | w) , PRest)
122     = true
        if PRest =/= emptyPlace /\
124        marking{M} <=? PRest .

126 ceq marking{M} <=? (P , PRest)
        = true
128     if M leeqth P with 0 /\
           PRest =/= emptyPlace /\
130        marking{M} <=? PRest .

132 eq M <=? P = false [owise] .

134 *** Impl - fire ####################

136 crl [fire-emptyPre] :
         net(P,
138          transitions{T : TRest},
             pre{(T --> emptyPlace), MTupleRest1},
140          post{(T --> PostValue), MTupleRest2},
             marking{M})
142      Rules
         MaxID
144      StepSize
         aid
146      =>
         net(P,
148          transitions{T : TRest},
             pre{(T --> emptyPlace), MTupleRest1},
150          post{(T --> PostValue), MTupleRest2},
```

```
            calc(M plus PostValue))
152         Rules
            MaxID
154         StepSize
            aid
156         if calc(M plus PostValue) <=? PostValue .

158     crl [fire] :
            net(P,
160             transitions{T : TRest},
                pre{(T --> PreValue), MTupleRest1},
162             post{(T --> PostValue), MTupleRest2},
                marking{PreValue ; M})
164         Rules
            MaxID
166         StepSize
            aid
168         =>
            net(P,
170             transitions{T : TRest},
                pre{(T --> PreValue), MTupleRest1},
172             post{(T --> PostValue), MTupleRest2},
                calc(((PreValue ; M) minus PreValue) plus
                   PostValue))
174         Rules
            MaxID
176         StepSize
            aid
178         if calc((PreValue ; M) plus PostValue) <=?
               PostValue .

180     *** ################################
        *** Execute Calc
182     *** ################################
```

```
184   eq [execute-step-minusStepEmptyPlace] :
          calc((M minus emptyPlace) plus PostValue) =
186       calc(M plus PostValue) .

188   eq [execute-step-minusEnd-single] :
          calc((p(Str | I | Cap) minus (p(Str | I | Cap)))
             plus PostValue) =
190       marking{PostValue} .
      eq [execute-step-minusEnd-single] :
192       calc((p(Str | I | w) minus (p(Str | I | w))) plus
             PostValue) =
          marking{PostValue} .
194
      eq [execute-step-minusStep] :
196       calc(((p(Str | I | Cap) ; MRest1) minus (p(Str | I |
              Cap) + MRest2))
               plus PostValue) =
198       calc((MRest1 minus MRest2) plus PostValue) .
      eq [execute-step-minusStep] :
200       calc(((p(Str | I | w) ; MRest1) minus (p(Str | I | w
             ) + MRest2))
               plus PostValue) =
202       calc((MRest1 minus MRest2) plus PostValue) .

204   eq [execute-step-plusEnd] :
          calc(M plus PostValue) =
206       marking{M ; PostValue} .

208   *** ##################################################
      *** Rule R = (l_net, r_net)
210   *** ##################################################

212   sort Rule .
      sorts LeftHandSide RightHandSide .
214
```

```
    op emptyRule : -> Rule .
216
    op _|_ : Rule Rule -> Rule [ctor assoc comm id:
        emptyRule] .
218
    op l : Net -> LeftHandSide .
220 op r : Net -> RightHandSide .

222 op rule : LeftHandSide RightHandSide -> Rule .

224

    *** ######################################################
226 *** ID Pool
    *** ######################################################
228
    sort IDPool .
230
    op emptyIDSet : -> Int .
232 op _,(_) : Int Int -> Int [comm id: emptyIDSet] .

234 op aid{_} : Int -> IDPool .

236 *** ######################################################
    *** Configuration
238 *** ######################################################

240 sort Configuration .

242 *** READING: NET SET<RULE> MAXID STEP_SIZE PID TID
    op _____ : Net Rule Int Int IDPool -> Configuration .
244
endm
```

Listing 15: rpn.maude of N_1 and r_1 generated by rMC

```
mod RULES is
  including RPN .

  var Str : String .
  vars I I1 I2 IRest IRest2 Cap Cap1 Cap2 StepSize : Int .
  vars P PRest PRest1 PRest2 PNet PRule : Places .
  vars T TRest TSet TSetL TSetR : Transitions .
  vars Pre PreL PreR : Pre .
  vars Post PostL PostR : Post .
  vars MTupleRest MTupleRest1 MTupleRest2 : MappingTuple .
  vars M M1 M2 MRest MRest1 MRest2 MNet MRule MFollow :
     Markings .
  vars R RRest : Rule .
  var N : Net .

  *** ####################################################
  *** Rule-Conditions
  *** ####################################################

  op checkContainingMarkings(_ || _) : Places Markings ->
     Bool .

  eq checkContainingMarkings(p(Str | I1 | Cap) || (p(Str |
      I1 | Cap) ; MRest)) = true .
  eq checkContainingMarkings(P || MNet) = false [owise] .

  op contains(_ || _) : Places Places -> Bool .

  eq contains(p(Str | I1 | Cap) || (p(Str | I1 | Cap),
     PRest)) = true .
  eq contains(P || PNet) = false [owise] .

  *** equalMarking returns true if:
```

```
    *** each marking of the rule is in the marking multiset
        of the net
31  *** and the marking set has no more markings of each
        marking
    ***     inside the rule set
33  *** READING: NET-MARKING, RULE-MARKING
    op equalMarking(_ =?= _) : Places Places -> Bool .
35
    eq equalMarking(
37      M =?= marking{ emptyMarking }
        ) = true .
39
    eq equalMarking(
41      M =?= M
        ) = true .
43
    ceq equalMarking(
45      marking{ p(Str | I1 | Cap) ; MNet } =?= marking{ p(
            Str | I2 | Cap) ; MRest }
        ) = true
47      if equalMarking(marking{ MNet } =?= marking{ MRest }
            ) /\
           (MRest =/= emptyMarking) .
49
    ceq equalMarking(
51      marking{ p(Str | I1 | Cap) ; MNet } =?= marking{ p(
            Str | I2 | Cap) } ) = true
        if not(contains((p(Str | I2 | Cap)) || MNet)) .
53
    eq equalMarking(
55      (MNet) =?= (MRule)
        ) = false [owise] .
57
***(
59  eq equalMarking(
```

```
      P =?= P
61    ) = true .

63  ceq equalMarking(
      (p(Str | I1 | Cap) , MNet) =?= (p(Str | I2 | Cap) ,
        MRest)
65    ) = true
      if equalMarking(MNet =?= MRest) /\
67      (MRest =/= emptyMarking) .

69  ceq equalMarking(
      (p(Str | I1 | Cap) , MNet) =?= (p(Str | I2 | Cap)))
        = true
71    if not(contains((p(Str | I2 | Cap)) || MNet)) .

73  eq equalMarking(
      (PNet) =?= (PRule)
75    ) = false [owise] .
)
77
  op freeOfMarking(_|_) : Places Markings -> Bool .
79
  *** there is no place
81  eq freeOfMarking(emptyPlace | M) = true .

83  *** there is no marking
  eq freeOfMarking(P | emptyMarking) = true .
85
  *** there is just one place left
87  ceq freeOfMarking(p(Str | I1 | Cap) | M) = true
      if not(checkContainingMarkings((p(Str | I1 | Cap))
        || M)) .
89
  *** normal case: test place and call rest
91  ceq freeOfMarking(p(Str | I1 | Cap), PRest | M) = true
```

```
      if not(checkContainingMarkings(p(Str | I1 | Cap) ||
         M)) /\
93        freeOfMarking(PRest | M) .

95  *** default case - false
    eq freeOfMarking(PRest | M) = false [owise] .
97


99

    *** emptyNeighbourForPlace returns true if no pre/post
       contains this place
101 *** READING: PLACE, PRE, POST
    op emptyNeighbourForPlace(_, _, _) : Places Pre Post ->
       Bool .
103
    eq emptyNeighbourForPlace(P,
105       pre{ (T --> P , PRest) , MTupleRest },
          Post) = false .
107
    eq emptyNeighbourForPlace(P,
109       Pre,
          post{ (T --> P , PRest) , MTupleRest }) = false .
111
    eq emptyNeighbourForPlace(P, Pre, Post) = true [owise] .
113
    *** ##################################################
115 *** RULE ID Pool getter
    *** ##################################################
117
    vars MaxID NewMaxID SetOfInts Count AidPRestSet
       AidPRestNewSet ISet : Int .
119


121 *** Helper, which fill the set of IDs
    *** READING: IDSET MAXID COUNTER INTERNAL-VAR
```

```
123
    op fill(_|_|_|_) : Int Int Int Int -> Int .
125 eq fill(I | MaxID | 0 | Count) = I .
    ceq fill(IRest | MaxID | Count | I)
127       = fill((MaxID + I, (IRest)) | MaxID | (Count - 1) |
              (I - 1))
          if I >= Count .
129 eq fill(I1 | MaxID | I2 | Count ) = I1 [owise] .

131

133 *** Getter for the new ID (place or transition)
    *** READING: CURRENT_SET MAXID STEP_SIZE
135
    op getAid(_|_|_) : Int Int Int -> Int .
137
    ceq getAid(I1 , (IRest) | MaxID | StepSize) = I1 if I1
       =/= emptyIDSet .
139 eq getAid(SetOfInts | MaxID | StepSize)
          = getAid(fill(SetOfInts | MaxID | StepSize |
              StepSize)
141                 | MaxID + MaxID | StepSize) [owise] .

143

145 *** Remove the first element from this multiset
    *** READING: CURRENT_SET MAXID STEP_SIZE
147
    op removeFirstElement(_|_|_) : Int Int Int -> Int .
149
    eq removeFirstElement(emptyIDSet | MaxID | StepSize) =
151       fill(emptyIDSet | MaxID | StepSize | StepSize) .
    ceq removeFirstElement(I1 , (IRest) | MaxID | StepSize)
       = IRest
153       if I1 =/= emptyIDSet [owise] .
```

```
155
157   *** Add unused ID
      *** READING: CURRENT_SET OLD_ID
159
      op addOldID(_|_) : Int Int -> Int .
161
      eq addOldID(SetOfInts | I) = I, (SetOfInts) .
163
165
      *** Correct the MaxID if new IDs are generated
167   *** READING: MAXID STEP_SIZE NEW_ID_COUNT

169   op correctMaxID(_|_|_) : Int Int Int -> Int .

171   ceq correctMaxID(MaxID | StepSize | Count)
            = correctMaxID(MaxID + StepSize | StepSize | Count
                - StepSize)
173         if Count > StepSize .
      eq correctMaxID(MaxID | StepSize | Count) = MaxID .
175
      *** ####################################################
177   *** RULE IMPLEMENTATION
      *** ####################################################
179
      vars Irule103 Irule102 Irule104 Irule105 Irule1019
         Irule1016 Irule1023 Irule2019 Irule2016 Irule2023
         Irule3016 Irule3019 Irule3025 : Int .
181
      vars Aid2 Aid1 AidRest2 AidRest1 AidRest : Int .
183
      crl [R1-PNML] :
185     net(
```

```
    places{ p("P" | Irule1019 | w) , p("P" | Irule1016 | w
        ) , PRest } ,
187 transitions{ t("T" | Irule1023) : TRest } ,
    pre{ (t("T" | Irule1023) --> p("P" | Irule1016 | w)) ,
         MTupleRest1 } ,
189 post{ (t("T" | Irule1023) --> p("P" | Irule1019 | w))
        , MTupleRest2 } ,
    marking{ p("P" | Irule1019 | w) ; p("P" | Irule1019 |
        w) ; MRest }
191 )
    rule(
193 l( net(
    places{ p("P" | Irule2019 | w) , p("P" | Irule2016 | w
        ) } ,
195 transitions{ t("T" | Irule2023) } ,
    pre{ (t("T" | Irule2023) --> p("P" | Irule2016 | w)) }
         ,
197 post{ (t("T" | Irule2023) --> p("P" | Irule2019 | w))
        } ,
    marking{ p("P" | Irule2019 | w) ; p("P" | Irule2019 |
        w) }
199 ) ) ,
    r( net(
201 places{ p("P" | Irule3016 | w) , p("P" | Irule3019 | w
        ) } ,
    transitions{ t("T" | Irule3025) } ,
203 pre{ (t("T" | Irule3025) --> p("P" | Irule3019 | w)) }
         ,
    post{ (t("T" | Irule3025) --> p("P" | Irule3016 | w))
        } ,
205 marking{ p("P" | Irule3019 | w) ; p("P" | Irule3019 |
        w) }
    ) ) )
207 | RRest
    MaxID
```

```
209     StepSize
        aid{ AidRest }
211     =>
        net(
213     places{ p("P" | Irule1016 | w) , p("P" | Irule1019 | w
            ) , PRest } ,
        transitions{ t("T" | Aid1 ) : TRest } ,
215     pre{ (t("T" | Aid1 ) --> p("P" | Irule1019 | w)) ,
            MTupleRest1 } ,
        post{ (t("T" | Aid1 ) --> p("P" | Irule1016 | w)) ,
            MTupleRest2 } ,
217     marking{ p("P" | Irule1019 | w) ; p("P" | Irule1019 |
            w) ; MRest }
        )
219     rule(
        l( net(
221     places{ p("P" | Irule2019 | w) , p("P" | Irule2016 | w
            ) } ,
        transitions{ t("T" | Irule2023) } ,
223     pre{ (t("T" | Irule2023) --> p("P" | Irule2016 | w)) }
            ,
        post{ (t("T" | Irule2023) --> p("P" | Irule2019 | w))
            } ,
225     marking{ p("P" | Irule2019 | w) ; p("P" | Irule2019 |
            w) }
        ) ) ,
227     r( net(
        places{ p("P" | Irule3016 | w) , p("P" | Irule3019 | w
            ) } ,
229     transitions{ t("T" | Irule3025) } ,
        pre{ (t("T" | Irule3025) --> p("P" | Irule3019 | w)) }
            ,
231     post{ (t("T" | Irule3025) --> p("P" | Irule3016 | w))
            } ,
```

```
    marking{ p("P" | Irule3019 | w) ; p("P" | Irule3019 |
        w) }
233 ) ) )
    | RRest
235 NewMaxID
    StepSize
237 aid{ AidRest2 }
    if          AidRest1 := addOldID(AidRest | Irule1023) /\
239     Aid1 := getAid(AidRest1 | MaxID | StepSize) /\
        AidRest2 := removeFirstElement(AidRest1 | MaxID |
            StepSize) /\        NewMaxID := correctMaxID(
            MaxID | StepSize | 2) .
241
endm
```

Listing 16: rules.maude of N_1 and r_1 generated by rMC

```
mod PROP is
  protecting STRING .
  including RULES .

  including SATISFACTION .

  subsort Configuration < State .

  var L : String .
  var I Cap MaxID StepSize : Int .
  var Pre : Pre .
  var Post : Post .
  vars P P1 P2 PRest PreValue : Places .
  vars T T1 T2 TRest : Transitions .
  vars MappingTuple MTupleRest1 MTupleRest2 : MappingTuple
      .
  vars Any M MRest : Markings .
  var Rules : Rule .
  var aid : IDPool .

  op reachable : Markings -> Prop .

  eq net(P ,
         T ,
         Pre ,
         Post ,
         marking{ M ; MRest } )
     Rules
     MaxID
     StepSize
     aid
     |= reachable(M) = true .

  eq net(P ,
```

```
34          T ,
            Pre ,
36          Post ,
            marking{ p(L | I | Cap) ; MRest } )
38      Rules
        MaxID
40      StepSize
        aid
42      |= reachable(p(L | I | Cap)) = true .

44  op t-enabled : -> Prop .

46  eq net(P ,
            T ,
48          pre{ (T1 --> PreValue) , MappingTuple } ,
            Post ,
50          marking{ PreValue ; MRest } )
        Rules
52      MaxID
        StepSize
54      aid
        |= t-enabled = true .
56  eq C |= t-enabled = false [owise] .

58  op enabled : -> Prop .

60  eq net(P ,
            T ,
62          pre{ (T1 --> PreValue) , MappingTuple } ,
            Post ,
64          marking{ PreValue ; MRest } )
        Rules
66      MaxID
        StepSize
68      aid
```

```
    |= enabled = true .
70
  vars Irule2019 Irule2016 Irule2023 : Int .
72
  eq net(places{ p("P" | Irule2019 | w) , p("P" |
     Irule2016 | w) , P } ,
74         transitions{ t("T" | Irule2023) : T } ,
           pre{ (t("T" | Irule2023) --> p("P" | Irule2016 |
              w)) , MTupleRest1 } ,
76         post{ (t("T" | Irule2023) --> p("P" | Irule2019 |
               w)) , MTupleRest2 } ,
           marking{ p("P" | Irule2019 | w) ; p("P" |
              Irule2019 | w) ; M } )
78         Rules
           MaxID
80         StepSize
           aid
82         |= enabled = true .

84  var C : Configuration .
    var Prop : Prop .
86  eq C |= Prop = false [owise] .

88 endm
```

Listing 17: prop.maude of N_1 and r_1 generated by rMC

```
mod NET is
  including PROP .
  including LTLR-MODEL-CHECKER .

  ops initial : -> Configuration .

  eq initial =
  net(
    places{ p("P" | 3 | w) , p("P" | 2 | w) } ,
    transitions{ t("T" | 4) : t("T" | 5) } ,
    pre{ (t("T" | 4) --> p("P" | 3 | w)) , (t("T" | 5) -->
        p("P" | 3 | w)) } ,
    post{ (t("T" | 4) --> p("P" | 2 | w)) , (t("T" | 5)
       --> p("P" | 2 | w)) } ,
    marking{ p("P" | 3 | w) ; p("P" | 3 | w) }
    )
  rule(
    l( net(
    places{ p("P" | 19 | w) , p("P" | 16 | w) } ,
    transitions{ t("T" | 23) } ,
    pre{ (t("T" | 23) --> p("P" | 16 | w)) } ,
    post{ (t("T" | 23) --> p("P" | 19 | w)) } ,
    marking{ p("P" | 19 | w) ; p("P" | 19 | w) }
    ) ) ,
    r( net(
    places{ p("P" | 16 | w) , p("P" | 19 | w) } ,
    transitions{ t("T" | 25) } ,
    pre{ (t("T" | 25) --> p("P" | 19 | w)) } ,
    post{ (t("T" | 25) --> p("P" | 16 | w)) } ,
    marking{ p("P" | 19 | w) ; p("P" | 19 | w) }
    ) ) )
    25
    10
```

```
32      aid{ (25 , (26 , (27 , (28 , (29 , (30 , (31 , (32 ,
            (33 , (34 , (35))))))))))) }
        .
34
    endm
```

Listing 18: net.maude of N_1 and r_1 generated by rMC

C CD Content

The CD content is organized by:

```
/
├── Schulz_1937371.pdf
├── implementation
│   └── eclipse project
│       ├── src
│       ├── lib
│       └── PNML_Files
├── bibliography
│   └── all PDF's named by <citeCount>_<author>_<year>.pdf
├── Documentation
└── Maude modules
```

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 30. April 2015 Alexander Schulz