# Project 2

**Model Checking for reconfigurable Petri nets**

Alexander Schulz  
(alexander.schulz1@haw-hamburg.de)

Supervised by  
Prof. Dr. Padberg

October 1, 2014

## Contents

1. **Motivation**  
2. **Background**  
   2.1 Reconfigurable Petri nets  
   2.2 Maude  
   2.3 Related Work  
3. **Data type**  
   3.1 Activation and Firing  
   3.2 LTL Properties  
   3.3 Matching of Rules  
   3.4 Dangling-Condition  
   3.5 Multi-Set for used Identifiers  
4. **Transformation**  
   4.1 Architecture  
   4.2 Results  
5. **Evaluation**  
6. **Conclusion**  
7. **Summary**
Abstract

Model checking is an important part of the theoretical informatics. It enables the verification of a model with a set of properties such as liveness, deadlock or safety. Furthermore, Petri nets are well known and can be used for a model checking process. Wider, a reconﬁgurable Petri net is based on a pure Petri net with a set of rules. These rules can be used dynamically to change the net itself.

One missing part of these nets is the possibility to verify a given net with a set of rules and properties. This paper attempts to ﬁll this gap. It presents a transformation approach which is based on Maude’s equation- and rewrite logic as well as the LTLR model checker.

1 Motivation

The result of the ﬁrst approach (see [Schulz, 2014]) is designed as extension for ReConNet. It uses the implementation of ReConNet to get all possible matches for a set of rules and a given Petri net. This approach results in a dependency of the current net state and the algorithm of ReConNet.

So, the result arose that the model-checking process has only one-step application of a rule. Hence, the result is not expressive. The result can be wrong if a rule is used twice, because an error can occur after the second usage.

To solve this issue, a new approach was chosen for this project. The new idea is based on an algebraic structure for reconﬁgurable Petri nets. The main aim was to create a structure which can be read as a normal mathematical notation of a reconﬁgurable Petri net. It should also contain the possibility to simulate the net, which implies a solution for a transition that deﬁnes the activation and ﬁring. Further, the solution should be read as a mathematical deﬁnition. An appropriate language is necessary to implement this deﬁnition. Similar to the ﬁrst project, this solution is based on Maude and has a clearly readable and algebraic programming style.

Moreover, the rules should also be implemented within the new structure. In contrast to the ﬁrst solution it should be able to detect a match itself. Nevertheless, it should be an easy-to-read solution without needless hassle.

The following sections contain an overview of all relevant parts of this solution. The ﬁrst section contains a short overview of the background for this work. This section shows the background for reconﬁgurable Petri nets and gives also a short introduction to Maude. Furthermore, the next section focus the new modules, which contain the data-types for the resulting Maude speciﬁcation of a transformed reconﬁgurable Petri net. Eventually, an evaluation shows the performance of the current implementation based on a test net.
2 Background

This section gives an overview of the backgrounds of this paper. The first part contains the reconfigurable Petri nets. This model extends a Petri net with a set of rules, which can be useful for modifying the net at runtime. Moreover, Maude is introduced because it is used as a transformation result for the works aim. Finally, a short overview of related works is presented.

2.1 Reconfigurable Petri nets

One of the most important models for concurrent systems and some software engineering parts are the Petri nets, based on Carl Adam Petri’s dissertation [Petri, 1962]. Since his work many researchers evolved the model with many extensions like colours as types [Jensen, 1987] or token-as-net for agent based models [Valk, 1998].

A marked Petri net can be formally described as a tuple \( N = (P, T, \text{pre}, \text{post}, M_0) \) where \( P \) is a set of places and \( T \) is a set of transitions. \( \text{pre} \) is used for all pre-conditions of transitions, which describes how many token are required for firing. On the other hand, \( \text{post} \) holds all information of the post-conditions for all transitions. Finally, \( M_0 \) shows all initial tokens on the places for this net \( N \) [Meseguer a. Montanari, 1990; Juhas et al., 2007].

Further, based on a Petri net are reconfigurable Petri nets important because they can modify themselves with a set of rules [Ehrig et al., 2007; Prange et al., 2008; Kahloul et al., 2010]. Formally a reconfigurable Petri net can be describe a tuple of a reconfigurable Petri net \( RN = (N, R) \). This definition uses the Petri net tuple and a set \( R \) of rules, which are given by rule \( r = (L \leftarrow K \rightarrow R) \) [Padberg, 2012; Ehrig et al., 2009]. \( L \) is the left-hand side (LHS), which needs morphism to \( N \). \( K \) is an interface between \( L \) and \( R \). \( R \) is the part which is inserted into the original net. To realise this replacement it is necessary to define a matching algorithm that finds \( L \) within the original Petri net. This match includes a mapping between the elements in the Petri net and the left side of the rule (L). Basically, this algorithm finds the same structure (form \( L \)) within the Petri nets [Blumreiter, 2013].
2.2 Maude

Maude is recently developed at the Stanford Research Institute International (SRI International). The equation and rewriting logic, which supports a powerful algebraic language, is used as a base Escobar et al. [2009]; Clavel et al. [2002]. Based on these two kinds of logic the result is a concurrent state system which can be used for some semantic analysis such as deadlock discovery via LTL-model-checking-module Katoen a. Baier [2008]; Eker et al. [2004].

Maude consists of a core which is named „Maude Core“. On top of its core every other part is written in Maude itself. Actually, Maude is distributed in version 2.6 form the website Clavel et al. [2011].

A program in Maude rests on one or many modules where every part of the system looks like a clear to read abstract data type (ADT). A module contains a set of types, which are used with the „sort“-keyword. It is also possible to define more than one type with the plural form „sorts“. Each type describes a property for the module. For example types for a Petri net can be described with:

\[
\text{sort Places Transitions Markings .} \tag{1}
\]

Depending on some sorts it is also possible to define a set of operators. These operators describe all functors which are used to work with the defined types. For example a functor, which is used to write a multiset of markings, can be defined with a whitespace. This

\[\text{www.maude.cs.uiuc.edu/ Abruf 16.05.2014}\]
whitespace is surrounded with underlines, which is a placeholder for the types defined after the double point. The return type stands after the arrow, which is also from the markings sort. The following line shows this operator:

\[
\text{op } \_\_ : \text{Markings Markings } \rightarrow \text{Markings} . \tag{2}
\]

If it is also necessary that this operator has associative (in Maude with a short version: „assoc”) and commutative (short with: „comm”) property, Maude is able to define this in the end of this line. Hence we obtain a multiset of markings by this operator. The notation allows these properties in box brackets, so that it can be written as:

\[
\text{op } \_\_ : \text{Markings Markings } \rightarrow \text{Markings [assoc comm]} . \tag{3}
\]

Maude uses the equation logic to define the validity for an operator (axioms). This can be exemplified with the initial marking from a Petri net. This initial marking consists of all tokens. It is a representation of the initial state of the Petri net. Based on this information we can define an operator that describes the initial state of a Petri net. After that, the validity with an equation can be added. If we have a Petri net with only one marking with an „A” label we get this two lines:

\[
\text{op initial } : \rightarrow \text{Markings} .
\]

\[
\text{eq initial } = A . \tag{5}
\]

It is possible to define types as „sort”, operators for functors and equations for the validity of operators. The rewrite rules can be used to replace a term-set with another term-set. So all terms are immutable as in many functional languages. A replacement rule consists of two term-sets, where the first set is replaced with the second one. These two sets are separated with a double arrow, which is shown with the following example, where a term \( A \) is replaced with a new term \( B \):

\[
\text{rl } [T] : A \Rightarrow B . \tag{6}
\]

Based on this example an implementation of the token game of Petri nets can be realised. The two term-sets can be seen as pre- and post-set of a transition. Hence, a rule can be used to describe a firing step with this two sets. This replacement rule can be modelled with the following graphical representation:

Figure 2: Example Petri net \( N \) for the Equation 6

2.3 Related Work

An automatic mapping process for UML-models to a Maude-specification is described in Chama et al. [2013] which is similar to the idea of transforming reconfigurable Petri nets.
In this paper the authors present three steps which are separated in modeling, analysing and transforming to Maude modules. The first step focuses on subject-specific modeling within UMLs class, state or components diagrams. After that step, the tool AtoM is used to convert the model into a Python-code representation. This code will be used to solve some constrains inside the UML-model components and some diagram specific parts. Lastly, the final step transfers all data into a Maude-specification, which can be used to verify some properties for example deadlocks.

Another work is published in Barbosa et al. [2011], where Petri nets are also converted into Maude-modules. As a base an Input-Output Place/Transition net (IOPT net) is used and saved in a PNML-file. These files are the origin for the transformation process. Further, PNML is used as a well-known markup-language for Petri nets. This process divides all components of a Petri net in special Maude-modules (net, semantic and initial markings) which can be used to verify in a same way as in Chama et al. [2013].

Continuing, the literature for Maude contains an example for a Petri net written with in Maude. The example uses a basic shop system, where a user can buy an apple or candies. The mapping into Maude uses the term replacement system to model the firing steps of this net. Based on this Maude-structure it is possible to add a model-checking possibility, which can be used to verify a deadlock or safety properties Clavel et al. [2011].

![Figure 3: Shop example with apple and candy](image)

Figure 3: Shop example with apple and candy
3 Data type

A reconfigurable Petri net $N_1$ consists of a tuple which is separated in a Petri net $N$ and a set of rules $R$. It can be written with $N_1 = (N,R)$. Furthermore, a Petri net can be formally described as a tuple $N = (P,T,pre,post,M,cap)$. Where $P$ is a set of places, $T$ is a set of transitions, $pre$ and $post$ are functions which maps $T \rightarrow P^\oplus$ and finally $M$ is the initial marking. Additionally, a function $cap : P \rightarrow N^\omega$ can be used to model a capacity of a place $P$ with a value $N^\omega$ [Padberg, 2012].

An example of a Petri net is shown in Figure 4. Each blue circle is a place and each rectangle is a transition. The arrows between these elements describe the arcs, which can connect a place with a transition and vice versa. The two black points are tokens which can be consumed by a transition.

![Figure 4: Example Petri net $N_1$](image)

A rule is made up of three Petri nets $L$, $K$ and $R$. Where, $L$ is the left-hand side (LHS), which should be found. The right-hand side $R$ would be inserted in the net, if this rule is used. Between $L$ and $R$, can $K$ be seen as an interface.

The example in Figure 5 shows a rule which changes the direction of an arc for a transition $T$. The change is realised by two steps. At first, the match of the left-hand side ensures that the rules can be applied. And finally, the right-hand side contains the information to be used.

This example contains a transition which connects the places in reverse order (arc colour black). However, the mapping net contains both transitions. The arc inversion is realised by a deleted and new transition.
\[
P = \{ A_2, A_3, A_4 \} \quad \text{post}(T_5) = A_3 \\
T = \{ T_5, T_6, T_7 \} \quad \text{post}(T_6) = A_4 \\
\text{pre}(T_5) = A_4 \quad \text{post}(T_7) = A_2 \\
\text{pre}(T_6) = A_2 \quad m = A_2 + A_4 \\
\text{pre}(T_7) = A_3 \quad \text{cap} = \omega A_2 + \omega A_3 + \omega A_4
\]

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

Figure 5: Example rule \( r_1 \), which switch the arc direction

The aim of this work is to create Maude modules, which provides the possibility to create a formal writing of a reconfigurable Petri net. The example net in Figure 4 can be formally indicated as:

Hence, the module needs a definition for places, transitions and markings as well as the definition of pre- and post-sets. It comprised the type hierarchy and the syntax as shown in Listing 2.

First, all sorts are defined. A sort can be understand as a type which contains the semantic. An example for a semantic is a Petri net place or transition. Furthermore, a type hierarchy can be formulated with the keyword \textit{subsort}. This is a membership equation logic feature of Maude. It enables Maude to use a mapping between different types. This made the possibilities to describe the type \textit{Places} as subset of the type \textit{Markings}.

Based on the types, all operators can be defined. A definition begins with the keyword \textit{op} followed by the operator syntax. This contains an operator-name and handover parameter. For all parameters a definition of the types are shown after the colon. If this multi-set of parameters is empty (an arrow stands after the colon) this operator is a constant. The return type stands after the arrow and finally a dot ends the statement. Additionally, special properties as associativity or commutativity can be added inside of box brackets. Further, the definition can be extended with equations.
Based on this definition, it is possible to write the net in Maude. The operator `net` is a wrapper-operator for a net and contains `Places, Transitions, Pre, Post` and `Markings`. The set of places is realised with the operator `places`. This operator contains a multi-set for places, which are separated with a comma. Further, the operator `transitions` is a set for transitions. It separates the elements with a colon. In addition, the `pre` and `post` operators describe the pre and post condition of a transition. Both operators contain a multi-set of `MappingTuple`, which are a mapping between a transition and a multi-set of places. Finally, the `marking`-operator contains a multi-set of `Places`. The content is separated with an additional symbol as the linear sum in the mathematics definition in Figure 6. The example in Figure 6 can be written in Maude as in the Listing 2. It separates all
operators and the included multi-sets with comma’s. Each multi-set are wrapped with curved brackets.

```
net(places{ p("A" | 3 | 2147483647) , p("A" | 4 | 2147483647) },
    p("A" | 2 | 2147483647) },
    transitions{ t("T" | 7) : t("T" | 5) : t("T" | 6) },
    pre{ (t("T" | 7) --> p("A" | 3 | 2147483647)) ,
         (t("T" | 5) --> p("A" | 4 | 2147483647)) ,
         (t("T" | 6) --> p("A" | 2 | 2147483647)) },
    post{ (t("T" | 7) --> p("A" | 2 | 2147483647)) ,
          (t("T" | 5) --> p("A" | 3 | 2147483647)) ,
          (t("T" | 6) --> p("A" | 4 | 2147483647)) },
    marking{ p("A" | 3 | 2147483647) ; p("A" | 4 | 2147483647) })
```

Listing 2: Maude module for a Petri net

3.1 Activation and Firing

A transition \( t \) is formally activated, written by \( m[t] \), when the following two conditions are satisfied. The first condition consists on the pre-set of this transition. The net marking has to contain greater or equal tokens for this transition, as described in the pre-set (see Equation 7). Furthermore, all post places have to be satisfy the capacity-condition. It is not possible to add more tokens as a place can store (see Equation 8).

If both conditions are complied, the transition \( t \) can fire. One firing step is written with \( m[t]m' \), where \( m \) is the current marking and \( m' \) is the following marking. The calculation of \( m' \) is described in Equation 9. Firstly, the pre-set is deducted from the current marking. Now the post-set of \( t \) can be added to the result.

\[
\begin{align*}
pre^@(t) &\leq m \\
m + post^@(t) &\leq \text{cap} \\
m' &= (m \ominus pre^@(t)) \oplus post^@(t).
\end{align*}
\]

A transformation of the first condition (see Equation 7) is shown in Listing 3. The rewrite rule contains two pivotal points for the condition. Firstly, \( T \rightarrow \text{PreValue} \) models \( pre^@(t) \). And \( \leq m \) is transformed into \( \text{marking}[\text{PreValue} ; M] \). Hence, the formal definition is implied.

The pre-set of a transition is a part of the marking multi-set, or the rule is not enabled for firing. This implementation is based on the matching algorithm from Maude. It is able to determine when the set contains this condition. In summary, a rule uses one transition from the \textit{net}-tuple and tests the existing in the \textit{pre}-set in the current marking.

Furthermore, the Equation 8 condition is written in the condition of the Maude rule. Hence, the sum of the current marking plus the post-set for the transition is less or equal than the capacity of each place. The addition of the current marking and the post-set is written after the \textit{if} in the last line of Listing 3. The result is used with the \textit{<=} which requires on the right side a multi-set of places. Further, detailed information can be found in Listing 4.
Further, the rule result contains a function which calculates the resulting set of markings.

\[
\text{calc}(((\text{PreValue} ; M) \text{ minus } \text{PreValue}) \text{ plus } \text{PostValue})
\]

It can be read as the formal definition in Equation 9. Where \(\text{PreValue} ; M\) describe \(m\) and \(\text{PreValue}\) is the pre-value of the transition \(v\). The place holder \(\text{PostValue}\) contains all of the post-set of \(t\).

```
crl [fire] :
  net(P,
    transitions{T : TRest},
    pre{(T --> PreValue), MTupleRest1},
    post{(T --> PostValue), MTupleRest2},
    marking{PreValue ; M})
Rules
  I
  =>
  net(P,
    transitions{T : TRest},
    pre{(T --> PreValue), MTupleRest1},
    post{(T --> PostValue), MTupleRest2},
    calc(((PreValue ; M) \text{ minus } \text{PreValue}) \text{ plus } \text{PostValue}))
Rules
  I
  if calc((PreValue ; M) \text{ plus } \text{PostValue}) \leq? \text{PostValue} .
```

Listing 3: Activation and firing of a transition

The operator \(\leq?\) (in words \textit{smallerAsCap}) is used to map the capacity condition in this Maude module. The aim is to return \textit{true} if the marking is less or equal than the capacity of each places in the post-set of a transition.

The source code consists of the helper method \(\text{loeqth with }\). This operator tests the capacity of a place multi-set and a single place. The third parameter is used to count the occurrence of a token in the multi-set.
3.2 LTL Properties

One aim of this work is to verify properties as deadlocks, liveness or reachability for a reconfigurable Petri net. To realise this Maude’s LTLR implementation is used. It is based on an implementation of the linear temporal logic (LTL). The implementation itself used a Kripke structure, which is realised on the basis of the equation and rewriting logic, basically a finite transition system [Eker et al., 2004].

The following examples in Listing 15 ff. are using the operators defined in Listing 5. It
contains an operator for the reachability of a marking. The enabled-operator includes the activation of a transition as well as the ability to apply a rule. Finally, the last three lines in Listing 5 include a standard equation, which is used when no other equation can be used. The implementation starts with a sub sorting of the Configuration-type. This is necessary, because the Kripke structure is based on these information. It means that all Configuration-objects are relevant for the construction of the states of the Kripke structure. In terms of this work a Configuration-object contains a snapshot of a reconfigurable Petri net. At the beginning it includes the initial marking and the primal state of the net without any transformation with rules. All other following conditions include the further interactions of the net (in this case the Configuration with firing steps and rule treatments).
subsort Configuration < State.

op reachable : Markings -> Prop.
eq net(P, T, Pre, Post,
    marking{ M ; MRest })

Rules I
|= reachable(M) = true.

op t-enabled : -> Prop.
eq net(P, T,
    pre{ (T1 --> PreValue) , MappingTuple },
    Post,
    marking{ PreValue ; MRest })

Rules I
|= t-enabled = true.
eq C |= t-enabled = false [wise].

op enabled : -> Prop.
eq net(P, T,
    pre{ (T1 --> PreValue) , MappingTuple },
    Post,
    marking{ PreValue ; MRest })

Rules I
|= enabled = true.
eq net(places{ p("A" | Irule2017 | 2147483647),
    p("A" | Irule2020 | 2147483647), P },
    transitions{ t("T" | Irule2024) : T },
    pre{ (t("T" | Irule2024) --> p("A" | Irule2017 | 2147483647)),
    MTupleRest1 },
    post{ (t("T" | Irule2024) --> p("A" | Irule2020 | 2147483647)),
    MTupleRest2 },
    marking{ p("A" | Irule2017 | 2147483647); M })

Rules I
|= enabled = true.

var C : Configuration.
var Prop : Prop.
eq C |= Prop = false [wise].

Listing 5: LTL Properties: deadlocks, liveness and reachability
### 3.3 Matching of Rules

A reconfigurable Petri net consists of a Net and a Set of Rules $\mathcal{R}$. Each rule contains three sub-nets, which contains a net $L$ for matching, a net $R$ for the replacement and a net $K$ that maps between the two nets.

The first project [Schulz, 2014] based on ReConNet. This tool provides the capability to find non-deterministic matches for a net and a set of rules [Blumreiter, 2013]. The aim of the first project is an extension that enables ReConNet to verify a net with a given set of rules.

The verification process is realised with a transformation to a Maude specification. In order to realise this an interface is designed to use ReConNet to find a match. This constellation results in the state, that only the initial state of a net and all rules can be verified with an LTL-process.

Based on this situation the new aim is to find a way where the Maude specification is able to find the match itself. This implies a possibility to define a rule in this specification as well as the dangling-condition (see section 3.4). Further, the meta configuration should be adapted a net and a set of rules.

The definition of a rule and the meta configuration can be found in [Listing 6]. Firstly, the sorts Rule, LeftHandSide and RightHandSide are defined. This models the two sides of a rule. The mapping net $K$ is not included, because it is not relevant for matching of a rule.

Further, the Configuration consist on a net, a multi-set of rules and a global ID-count. The net contains all information as places, transitions, markings and pre as well as post-set. Each rule multi-set entry contains a left and right side. At last the ID-count is used for each insertion step, where a new transition or a place is going to be added.

```plaintext
*** Rule R = (l_net, r_net)
2 sort Rule .
3 sorts LeftHandSide RightHandSide .
4 op emptyRule : -> Rule .
6 op l : Net -> LeftHandSide .
7 op r : Net -> RightHandSide .
8 op rule : LeftHandSide RightHandSide -> Rule .

*** Configuration
9 sort Configuration .
10 op ___ : Net Rule Int -> Configuration .
```

Listing 6: Definition of a rule and the configuration (net, multi-set of rules and global ID-count)

The rule $r_1$ is shown in two Listings [7] and [8]. It based on the definition in [Listing 6]. Where the left-hand side provides all information which are necessary for finding a match. The right-hand side contains all information for the result of the transformation. It contains as an example the new elements (here a transition) and the related ID.

The whole rule is written as condition replacement. If the condition is satisfied with the
current net, the rule can be applied. Relating to the replacement it means, that the left-hand side of the rule is replaced with the right-hand side. Furthermore, a rule is working with a configuration object. Hence, it use a net, multi-set of rules as well as the ID-count. The example in Listing 7 shows the left side of the rule in Figure 5. The net which should be found, consist of two places with the label A. Further, it contains a transition T which connects the two places. A description of the arcs can be found in the pre- and post-set. At last it contains a marking on the place A. All these elements are part of the net in the rule. It only differ with the ID’s, which are not given as concrete numbers. Each $\text{rule<number>}$ are variables that are used to find a structural match. If the net had the same structure, but other ID’s for the places and transitions, this rule is also activated. This is possible, because Maude maps the ID’s internal with each variable. Moreover, contains each set a variable for a possible rest of elements. For example the set places contains in this net more than two places. The remaining places are mapped into the variable $\text{PRest}$, so this rule is still activated. If a set have no more elements the id of the definition above is used (see Listing 2). The id is the empty-set constant operator for each set.
The right-hand side differs to the left-hand side. The part of the net now contains the structure from the right-hand side.

In the example in Listing 8 is the new transition $T$ added. The calculation of the new identifier is detailed described in subsection 3.5.
nl(places\{p("A" | Irule1017 | 2147483647) ,
    p("A" | Irule1020 | 2147483647) , PRest} ,
transitions\{t("T" | AidT1) : TRest} ,
pre\{(t("T" | AidT1) --> p("A" | Irule1017 | 2147483647)) ,
    MTupleRest1} ,
post\{(t("T" | AidT1) --> p("A" | Irule1020 | 2147483647)) ,
    MTupleRest2} ,
marking\{p("A" | Irule1017 | 2147483647) ; MRest}\})
rule(l
nl(places\{p("A" | Irule2017 | 2147483647) ,
    p("A" | Irule2020 | 2147483647)} ,
transitions\{t("T" | Irule2024)} ,
pre\{(t("T" | Irule2024) -->
    p("A" | Irule2020 | 2147483647))} ,
post\{(t("T" | Irule2024) -->
    p("A" | Irule2017 | 2147483647))} ,
marking\{p("A" | Irule2017 | 2147483647)}
))
, r
nl(places\{p("A" | Irule3017 | 2147483647) ,
    p("A" | Irule3020 | 2147483647)} ,
transitions\{t("T" | Irule3026)} ,
pre\{(t("T" | Irule3026) -->
    p("A" | Irule3017 | 2147483647))} ,
post\{(t("T" | Irule3026) -->
    p("A" | Irule3020 | 2147483647))} ,
marking\{p("A" | Irule3017 | 2147483647)}
))
NewMaxID
StepSize
aidPlace\{AidPRest\}
aidTransition\{AidTRest2\}
if AidTRest1 := addOldID(AidTRest | Irule1024) /
    AidT1 := getAid(AidTRest1 | MaxID | StepSize) /
    AidTRest2 := removeFirstElement(AidTRest1 | MaxID | StepSize) /
    NewMaxID := correctMaxID(MaxID | StepSize | 1) .

Listing 8: Example of rule r1 written with Maude (right-hand side)
3.4 Dangling-Condition

A special part of a rule matching is the gluing condition. This condition is separated into the identification and dangling condition. The identification condition requires that no place or transition is cleared and obtained in the same time. Further, the dangling condition defines that a place can only be deleted if there are no connecting arcs outside the rule. A transition is not relevant, because it is isomorph mapped \cite{Blumreiter2013}. The example net \( N_2 \) in Figure 7 shows a small example, where a place \( A \) (the red place) should be deleted with rule \( r_3 \) in Figure 8. The rule has only one match, because the bottom part of the net differs with an addition transition. This transition injured the dangling condition and the rule can not be used at this point.

![Figure 7: Example net \( N_2 \)](image)

The associated Maude code can be found in Listing 9. It is separated in two parts. Firstly, the \textit{equalMarking}-operator test the current net marking. This condition ensures, that for all locations that are to be deleted, the marking in the net and the rule are the same. Here the marking of the net is given with

\[
\text{marking} \{ p(\_A | Irule1019 | 2147483647) ; MRest \}
\]

It contains the token \( A_{Irule1019} \) from the rule and also the variable \( MRest \). The second marking from the net (another \( A \) token) is in this multi-set of markings. Further, the second multi-set containing all places that should be deleted. In the example this is the other place called \( A \). Hence, the following line contains all relevant information

\[
\text{marking} \{ p(\_A'' | Irule2019 | 2147483647) \}
\]

The result is true, if the markings are for all places in the second set are equal than the marking in the first set. Furthermore, for each deleted place is a \textit{emptyNeighbourForPlace}-operator defined. This operator tests the dangling condition with the remaining pre and post multisets. In the example this two sets are defined as \( MTupleRest1 \) as well as \( MTupleRest2 \). All operator definitions can be found in Listing 10.
if equalMarking(
    ( marking{ p("A" | Irule1019 | 2147483647) ; MRest } )
    =?=
    marking{ p("A" | Irule2019 | 2147483647) } ) /
emptyNeighbourForPlace(
    p("A" | Irule1018 | 2147483647) ,
    pre{ MTupleRest1 }
    post{ MTupleRest2 } ) .

Listing 9: Dangling-Condition if a place should be deleted

All operators, which are needed for the dangling condition are defined in Listing 10. The equalMarking-operator checks whether the second multi-set is a subset of the first multi-set. The first parameter is a multi-set of the current net marking. And the second parameter contains the marking of the rule. The result is true, only if the first multi-set contains the same marking for each place as the second multi-set. Therefore, the equations are separated into five case differentiation. The first equation includes the situation where two identical marks are to be compared. If this situation occurs the result is true. The second case contains a recursion. It consumes two markings of each multi-set and returns true, if the recursion call also return true. The following equation differs only case distinction. It returns true including a recursion, when the markings is not in the multi-sets. Based on this equation the following lines includes the situation, where the second set only contains one element. If the first remaining multi-set do not contain more than one of these markings it returns true. Finally, if no other case can be used, this case is the „otherwise case“ (Maude’s keyword with brackets: [owise]). For all other state it returns false and ends the recursion.

Further, the emptyNeighbourForPlace-operator can be used for a test, that examines the neighbours of a place. It returns true if the place does not have any arcs outside the rule. This means exactly, that all arcs at the place are included in the rule.

The implementation uses three equations. They are separated into the first two situations where a pre or a post exists. And finally the otherwise case, where no arc exists and the result is true.
op contains(_, _): Places Places -> Bool.

eq contains(p(Str | I | Cap) | (p(Str | I | Cap), PRest)) = true.

eq contains(P | PNet) = false [owise].

*** READING: NET-MARKING, RULE-MARKING

op equalMarking(_, _): Places Places -> Bool.

eq equalMarking(p(Str | I | Cap) =?= p(Str | I2 | Cap)) = true.

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

ceq equalMarking((p(Str | I | Cap), MNet) =?= (p(Str | I | Cap), MRest)) = true if not(contains((p(Str | I | Cap)) | MNet)) /
not(contains((p(Str | I | Cap)) | MRest)) /
(MRest =/= emptyMarking) /
equalMarking(MNet =?= MRest).

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

eq equalMarking(PNet) =?= (PRule) = false [owise].

*** READING: PLACE, PRE, POST

op emptyNeighbourForPlace(_, _, _): Places Pre Post -> Bool.

eq emptyNeighbourForPlace(P, pre{ (T --> P, PRest), MTupleRest }, Post) = false.

eq emptyNeighbourForPlace(P, Pre, post{ (T --> P, PRest), MTupleRest }) = false.

eq emptyNeighbourForPlace(P, Pre, Post) = true [owise].

Listing 10: Dangling-Condition helper methods
3.5 Multi-Set for used Identifiers

One problem of Maude is the missing garbage collection. This can result in an overflow if a rule insert a node (place or transition). Because each new node gets an identifier. To solve this problem a multi-set of free identifiers are used. It requires a modification of the Configuration definition which introduced in Listing 6. Now it contains an integer for the maxID as well as for the defined step size. Further, it has two sets for the place and transition identifiers. The implementation can be found in Listing 11.

```
*** READING: NET SET<RULE> MAXID STEP_SIZE PID TID
    op ________ : 
   1 Net Rule Int Int IDPoolPlace IDPoolTransition 
   2 ->
   3 Configuration .
```

Listing 11: Extending the Configuration with the identifier multi-set

The usage of the defined fields in Listing 11 are useful when a rule deletes or adds a node as a place or a transition. An example of the implementation for a rule is shown in Listing 12.

The Transition t("X" — Irule1031) is to be deleted here. The identifier of this node is contained in the left side of the rule. The variable Irule1031 holds the current value which are reused in line 21, where the id is added into the new identifier multi-set AidTRest1. The next step uses this multi-set to get a new identifier for the new transition with the identifier AidT1. The getter-operator in line 22 sets the value for the new transition. Further the new identifier must be deleted from the old identifier multi-set. The last step sets the maxID to its new value, if the max-step is overrun.
crl \texttt{[R1-PNML]} :

\begin{verbatim}
... transitions{ t("X" | \texttt{Irul}e1031) : TRest } ,
... MaxID StepSize aidPlace{ AidPRest } aidTransition{ AidTRest } => ...
transitions{ t("X" | \texttt{AidT}1) : TRest } ,
pre{ (t("X" | \texttt{AidT}1) --> p("A" | Irule1013 | 2147483647)) , 
    MTupleRest1 },
post{ (t("X" | \texttt{AidT}1) --> p("A" | Irule1016 | 2147483647)) , 
    MTupleRest2 }, ...
NewMaxID StepSize aidPlace{ AidPRest } aidTransition{ AidTRest2 }
if AidTRest1 := addOldID(AidTRest | Irule1031) /\
    AidT1 := getAid(AidTRest1 | MaxID | StepSize) /\
    AidTRest2 := removeFirstElement(AidTRest | MaxID | StepSize) /\
    NewMaxID := correctMaxID(MaxID | StepSize | 1) .
\end{verbatim}

Listing 12: Save old identifier and get a new from the identifier multi-sets

Every operator which are used in the \texttt{Listing 12} are shown in \texttt{Listing 13}. This listing contains the implementation of the fill-operator which generates new identifiers if the set is empty. Further, it has operators which can be used to get or set an identifier to a multi-set of identifiers. It is not necessary to differ between places and transitions, because the operators can be generic programmed. Each operator has parameters which can take both multi-sets. So provides the \texttt{getAid}-operator the function to get the first element of a multi-set. Otherwise, the \texttt{removeFirstElement}-operator can delete this first element of a given multi-set. Moreover, the \texttt{addOldID}-operator puts an element into a multi-set and the \texttt{correctMaxID}-operator defines a new \texttt{maxID} if it is necessary.
Listing 13: Identifier multiset implementation

4 Transformation

This section includes the architecture of the transformation process as well as the results of some model checking formula.
4.1 Architecture

The output base of ReConNet consists of PNML. PNML is a xml-based standard for the Petri net export. The graphical editor ReConNet used this standard for the persistence of developed nets. In addition to the pure PNML-standard, a rule is stored with its three networks in a PNML file.

Based on PNML this work use XSL to realise the transformation. The result uses the Maude modules which are defined before the transformation. Previously defined are the sorts for Places, Transitions and net itself. And further the logic of firing or the identification of the dangling condition (see also the definition of all modules in the listings above).

The XSL process is designed with the separation of the global types as places, transitions, pre or post. And further, it has the specific sub xsl-templates for the transformations as in the net, rules, prop or rpn. The structure is summarized in Figure 9. The global types are over the specific modules that are grouped together in separate packages.

![Diagram of stylesheet structure for transformation]

Figure 9: Structure of the stylesheets for the transformation

The difference to the first project is that this approach is independent of ReConNets implementation. The first project used the persistence-module of ReConNets to load a PNML-files (net or rule) [Schulz, 2014]. This approach is superior because it is built directly on the PNML data. The transformation process is written with XSL, which provides a well known language. The interface between this approach and ReConNet is realised through export to the PNML files that can be used on a net or a few rules.

4.2 Results

Based on the result of the transformation process it is possible to use Maude’s LTL implementation. The prop.maude-module includes all necessary code for the LTL process (see Listing 14). It subsorts the Configuration-typ under state, which is required (for the Configuration-typ see Listing 6).

including SATISFACTION.

subsort Configuration < State.

Listing 14: Sub sorting of Configuration with State

In the first example, the deadlock freedom of example Figure 4 will be shown. The formula based on the box- and diamond-operator. Together it describes the semantic of liveness. It means that a property is globally (box) repeatedly (diamond) is true. To write the liveness property for the reconfigurable Petri net modules, the following line can be used

rewmodelCheck(initial, [] <> enabled).

It uses the Maude modelCheck-operator with the initial configuration (net, marking, rules and global variables) and the formula []<> enabled. The formula is based on the three operators [] and <> and the enabled-operator (for the definition see Listing 5).

In terms of the example the following output in Listing 15 results if the trace is enabled.
Maude> rew modelCheck(initial, []<> enabled ).
rewrite in NET : modelCheck(initial, []<> enabled ).
rewrites: 197 in 0ms cpu (0ms real) (270064 rewrites/second)
result ModelCheckResult: counterexample(
...

{net(places{p("A" | 2 | 2147483647),p("A" | 3 | 2147483647),
p("A" | 4 | 2147483647)},
  transitions{t("T" | 6) : t("T" | 7) : t("T" | 26)},
  pre{(t("T" | 6) --> p("A" | 2 | 2147483647)),
    (t("T" | 7) --> p("A" | 3 | 2147483647)),
    t("T" | 26) --> p("A" | 3 | 2147483647)},
  post{(t("T" | 6) --> p("A" | 4 | 2147483647)),
     (t("T" | 7) --> p("A" | 2 | 2147483647)),
     t("T" | 26) --> p("A" | 4 | 2147483647)},
  marking{p("A" | 4 | 2147483647) ; p("A" | 4 | 2147483647)})
rule1{net(places{p("A" | 17 | 2147483647),
p("A" | 20 | 2147483647)},
  transitions{t("T" | 24)},
  pre{t("T" | 24) --> p("A" | 17 | 2147483647)},
  post{t("T" | 24) --> p("A" | 20 | 2147483647)},
  marking(p("A" | 17 | 2147483647))},
rule2{net(places{p("A" | 17 | 2147483647),p("A" | 20 | 2147483647),
  transitions{t("T" | 26)},
  pre{t("T" | 26) --> p("A" | 20 | 2147483647)},
  post{t("T" | 26) --> p("A" | 17 | 2147483647)},
  marking(p("A" | 17 | 2147483647))},deadlock)}

Listing 15: Counterexample of a deadlock

The meaning of this counterexample, is that the rule consist of the marking. It is possible
that all tokens are on one place. Furthermore, this place has only incoming arc, which
results in a deadlock. The state of the net with this deadlock is modelled in Figure 10.

Figure 10: State of N_1 with an deadlock (r_1 can not be applied)

In the assumption that the markings were changed on the places within the rule (see
Figure 11). Based on this rule, the same LTL-formal as in the example above (Listing 15)
returns the result in [Listing 16](#). It shows, that the net \( N1 \) and rule \( r2 \) are deadlock free. The new rule prevents the situation in [Figure 10](#), where the marking can be located on one place which has only incoming arcs. Hence, two situations are possible. Firstly, when a marking is at a place where all arcs are starting, the rule is not enabled. But the net itself can fire. On the other hand, a marking will be placed on a place, where one or more arcs are incoming. For this case, the rule can be used. The result is that a transition is enabled now and will also continue to be used for the token game. In either situation, an operator of [Listing 5](#) is enabled. Since the \textit{enabled}-operator is defined for the firing and transformation step.

**Figure 11**: Example rule \( r2 \), which change the direction of the arc (different marking)

Now the result of the formula is true and Maude prints some information the rewrite count.

```
Maude> rew modelCheck(initial, [\]<> enabled ) .
2 rewrite in NET : modelCheck(initial, [\]<> enabled ) .
Debug(1)> rew [1] modelCheck(initial, [\]<> enabled ) .
4 rewrite [1] in NET : modelCheck(initial, [\]<> enabled ) .
rewrites: 6575268 in 20240ms cpu (20243ms real) (324865 rewrites/second)
6 result Bool: true
```

Listing 16: Counterexample for a reachable test

In addition, the enable-test can be realised only for the transitions with the \textit{t-enabled}-operator. It consists only on the transitions and no rules. In contrast to the \textit{enabled}-operator returns the \textit{t-enabled}-operator for the example a negative result which is shown in [Listing 17](#)
Maude> rew [1] modelCheck(initial, []<> t-enabled).
rewrite [1] in NET : modelCheck(initial, []<> t-enabled).
rewrites: 114 in 0ms cpu (0ms real) (198952 rewrites/second)
result ModelCheckResult: counterexample(
...

{ net(places{p("A" | 2 | 2147483647), p("A" | 3 | 2147483647),
p("A" | 4 | 2147483647)},
  transitions{t("T" | 5) : t("T" | 6) : t("T" | 27)},
  pre{t("T" | 5) --> p("A" | 3 | 2147483647),
    t("T" | 6) --> p("A" | 4 | 2147483647),
    t("T" | 27) --> p("A" | 4 | 2147483647)},
  post{t("T" | 5) --> p("A" | 2 | 2147483647),
    t("T" | 6) --> p("A" | 3 | 2147483647),
    t("T" | 27) --> p("A" | 2 | 2147483647)},
  marking{p("A" | 2 | 2147483647) ; p("A" | 2 | 2147483647)})
rule(
  l(net(places{p("A" | 17 | 2147483647),
p("A" | 20 | 2147483647)},
    transitions{t("T" | 24)},
    pre{t("T" | 24) --> p("A" | 17 | 2147483647),
      post{t("T" | 24) --> p("A" | 20 | 2147483647)},
      marking{p("A" | 17 | 2147483647)}),
  r(net(places{p("A" | 17 | 2147483647), p("A" | 20 | 2147483647)},
    transitions{t("T" | 26)},
    pre{t("T" | 26) --> p("A" | 20 | 2147483647),
      post{t("T" | 26) --> p("A" | 17 | 2147483647)},
      marking{p("A" | 17 | 2147483647)}))) 28, deadlock })

Listing 17: Counterexample for a t-enabled test

The reachable-operator is designed to find a given marking. The formal definition of this operator is

\[ \text{op \ reachable : Markings} \rightarrow \text{Prop.} \]

Therefore, it is possible to test one or many tokens. In the example below (see Listing 18) two tokens \(A_3\) and \(A_3\) are searched. Maude's result describes a situation, where it found a deadlock with two tokens on \(A_2\).
An example where the LTL-formal is true, can be found in Listing 21. This is a basic test, where the formal verified that the initial marking is reachable. Maude reached true after six steps, because the marking already contain the search parameter.

A more complex example can be found in Listing 20. The example verified, that two
markings ($A_4$ and $A_4$) are reachable from the initial marking. The formula differ from the formula in the example above (see Listing 21). It negated the formula, so that the result contains a path to this marking, or otherwise it returns true.

```maude
Maude> rew [1] modelCheck(initial, (enabled ->
  ( <> reachable(p("A" | 4 | 2147483647) ; p("A" | 4 | 2147483647))))).
rewrite [1] in NET : modelCheck(initial, enabled ->
  ( <> reachable(p("A" | 4 | 2147483647) ; p("A" | 4 | 2147483647))).
rewrites: 174 in 0ms cpu (0ms real) (~ rewrites/second)
result ModelCheckResult: counterexample(
...
Listing 20: Counterexample for a reachable of the initial marking
```

Based on the reachable-operator it is possible to describe the liveness-condition. Which means that a marking can be reached again. This includes all possible firing steps or rule applying.
In the example net $N_1$ will the test return a situation where the net stops in a deadlock. All tokens are placed on $A_4$ which has only incoming arcs. Hence, no rule (in this example $r_1$) can be applied (for a graphical representation of the result see also Figure 10).

```
Maude> rew modelCheck(initial, []<> reachable(p("A" | 3 | 2147483647)))
   rewrite in NET : modelCheck(initial, []<> reachable(p("A" | 3 | 2147483647)))
   rewrites: 197 in 0ms cpu (0ms real) (285094 rewrites/second)
result ModelCheckResult: counterexample(
  ...[net(places{p("A" | 2 | 2147483647), p("A" | 3 | 2147483647),
    p("A" | 4 | 2147483647)},
    transitions{t("T" | 6) : t("T" | 7) : t("T" | 26)},
    pre{(t("T" | 6) --> p("A" | 2 | 2147483647)),
      (t("T" | 7) --> p("A" | 3 | 2147483647)),
      t("T" | 26) --> p("A" | 3 | 2147483647)},
    post{(t("T" | 6) --> p("A" | 4 | 2147483647)),
     (t("T" | 7) --> p("A" | 2 | 2147483647)),
     t("T" | 26) --> p("A" | 4 | 2147483647)},
    marking{p("A" | 4 | 2147483647) ; p("A" | 4 | 2147483647)})
  rule{l(net(places{p("A" | 17 | 2147483647), p("A" | 20 | 2147483647)},
               transitions{t("T" | 24)},
               pre{t("T" | 24) --> p("A" | 17 | 2147483647)},
               post{t("T" | 24) --> p("A" | 20 | 2147483647)},
               marking{p("A" | 17 | 2147483647)}),
          r(net(places{p("A" | 17 | 2147483647), p("A" | 20 | 2147483647)},
                transitions{t("T" | 26)},
                pre{t("T" | 26) --> p("A" | 20 | 2147483647)},
                post{t("T" | 26) --> p("A" | 17 | 2147483647)},
                marking{p("A" | 17 | 2147483647)}),deadlock))
Listing 21: Counterexample for the liveness-condition for $A_3$
```

5 Evaluation

This section tries to present the performance of this approach. The evaluation is based on two steps. At first, a net is transformed into the Maude modules. And after that step it is tested with the liveness formula (see the first formula in section 4.2). The transformation uses a net which is build as a circle and the rule $r_1$ (see Figure 5). Further, it contains one token at place $P_1$. The structure connects a place with two transitions (one for the pre and vice versa). Hence, it is possible to build a test which shows the performance of a net which can be scaled with the size of nodes (places and transitions).

For this work are four net sizes used, which allow to make a meaningful statement. Each
net has the same semantic and should return true. Hence, only the runtime metadata such as rewrite count and time are different. The transformation process runs in each case with nearly the same time (see Table 1). Further, the rewrites are grown linear with the size of nodes. It takes 109 rewrites for 10 places and transitions. If the net has twice as many nodes as in the first example, it takes 209 rewrites. Only the used time grows exponentially. It changes from 13 ms to 23994 ms. If the net receives only 4 new nodes, it grows to 189939 ms (see Table 2).

All tests are realised on a Thinkpad X230 with an Intel® Core™ i5-3320M CPU with 4 cores (2.60GHz) and 16 GB RAM. It was implemented on a Ubuntu 12.04 which is build up the 3.14.17-031417-generic kernel.

<table>
<thead>
<tr>
<th>p x t</th>
<th>time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10x10</td>
<td>1701</td>
</tr>
<tr>
<td>20x20</td>
<td>1737</td>
</tr>
<tr>
<td>22x22</td>
<td>1704</td>
</tr>
<tr>
<td>23x23</td>
<td>1737</td>
</tr>
</tbody>
</table>

Table 1: Transformation from PNML to Maude

<table>
<thead>
<tr>
<th>p x t</th>
<th>rew</th>
<th>time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10x10</td>
<td>109 in</td>
<td>13</td>
</tr>
<tr>
<td>20x20</td>
<td>209 in</td>
<td>23994</td>
</tr>
<tr>
<td>22x22</td>
<td>229 in</td>
<td>189939</td>
</tr>
<tr>
<td>23x23</td>
<td>239 in</td>
<td>834358</td>
</tr>
</tbody>
</table>

Table 2: Verification of liveness

6 Conclusion

This paper presents an approach which enables LTL model checking for reconfigurable Petri nets. The intention was to use Maude with the equation- and rewrite logic for the transformation result. Maude itself includes modules for an on-the-fly model checking, based on the state defined by the new modules of this work.

The resulting modules can be used for the modelling as the formal definition of the reconfigurable Petri nets. This means that the modules include a structure of sorts and with associated operators which allow a tool or user to write a clear formal definition of the net as well as a set of rules. Furthermore, conditions as for example the dangling condition are also included by the modules defined by this paper.

Finally, the evaluation shows that the defined modules have problems with the size of the net. One problem is that a rule can add a place or transition. Hence, it is necessary to get new identifiers for this elements. If a rule inserts a new node and this new node receives a new identifier, the problem results in an infinite behaviour. Currently this problem is not solved. Unless it exists a countable reuse of identifiers. For a example it is possible that a rule deletes one element and inserts a new one. The old unused identifier can be recycled. For this special formula the model checking process returns. If the rule creates infinite nodes (each use inserts a new transition) it has no chance to receive a result for this formula.
7 Summary

The aim of this work is to allow LTL for reconfigurable Petri nets. The tool ReConNet is the base, which makes it possible to create a reconfigurable Petri net. Further, it includes a possibility to export a net and a set of rules as PNML-files. This files are the origin for this approach.

The approach realises Maude modules which can be used in a LTL process. Maude includes a module for an on-the-fly model checking process. The new modules consist of 4 separated parts. First a module contains the definition for an algebraic reconfigurable Petri net. The aim of the module is to allow a writing of a net and a set of rules as it is provided by the mathematical notation. Furthermore, it supports an activation and firing of a net. The next module contains the rule definitions. Rewrite rules are used to design a rule which uses the pattern matching of Maude for the possibility to use this rule. Each rule ensures that the dangling condition is maintained. Further, an identifier multi-set is used to cache unused identifier for places and transitions. This caching allows the process to verify a formula, if the number of identifiers is limited. Next, a module contains the definition for operators which are necessary for the LTL formulae. For example the enabled-operator can be used for the liveness condition. At last, a module includes the initial definition of a net and a set of rules. The initial state embodies the initial marking, places and transitions.

Based on each module LTL formulae can be verified with Maude. It returns true, or a counterExample with an example for an error case of this formula.
## List of Figures

1. ReConNet - graphical editor for reconfigurable Petri nets ........................................ 4
2. Example Petri net $N$ for the Equation 6 .................................................................. 5
3. Shop example with apple and candy ........................................................................ 6
4. Example Petri net $N_1$ ............................................................................................ 7
5. Formal description of a Petri net (for a graphical presentation see also Figure 4) .... 8
6. Example rule $r_1$, which switch the arc direction .................................................... 8
7. Example net $N_2$ ..................................................................................................... 19
8. Example rule $r_3$, which deletes the place $A$ ........................................................ 19
9. Structure of the stylesheets for the transformation ................................................... 25
10. State of $N_1$ with an deadlock ($r_1$ can not be applied) ........................................ 27
11. Example rule $r_2$, which change the direction of the arc (different marking) ...... 28
<table>
<thead>
<tr>
<th>Listings</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Maude module for a Petri net</td>
<td>9</td>
</tr>
<tr>
<td>2 Maude module for a Petri net</td>
<td>10</td>
</tr>
<tr>
<td>3 Activation and firing of a transition</td>
<td>11</td>
</tr>
<tr>
<td>4 Capacity proof of each place in the post-set</td>
<td>12</td>
</tr>
<tr>
<td>5 LTL Properties: deadlocks, liveness and reachability</td>
<td>14</td>
</tr>
<tr>
<td>6 Definition of a rule and the configuration (net, multi-set of rules and global ID-count)</td>
<td>15</td>
</tr>
<tr>
<td>7 Example of rule $r_1$ written with Maude (left-hand side)</td>
<td>17</td>
</tr>
<tr>
<td>8 Example of rule $r_1$ written with Maude (right-hand side)</td>
<td>18</td>
</tr>
<tr>
<td>9 Dangling-Condition if a place should be deleted</td>
<td>20</td>
</tr>
<tr>
<td>10 Dangling-Condition helper methods</td>
<td>21</td>
</tr>
<tr>
<td>11 Extending the Configuration with the identifier multi-set</td>
<td>22</td>
</tr>
<tr>
<td>12 Save old identifier and get a new from the identifier multi-sets</td>
<td>23</td>
</tr>
<tr>
<td>13 Identifier multi-set implementation</td>
<td>24</td>
</tr>
<tr>
<td>14 Sub sorting of Configuration with $State$</td>
<td>26</td>
</tr>
<tr>
<td>15 Counterexample of a deadlock</td>
<td>27</td>
</tr>
<tr>
<td>16 Counterexample for a reachable test</td>
<td>28</td>
</tr>
<tr>
<td>17 Counterexample for a $t$-enabled test</td>
<td>29</td>
</tr>
<tr>
<td>18 Counterexample for a reachable test</td>
<td>30</td>
</tr>
<tr>
<td>19 Counterexample for a reachable of the initial marking</td>
<td>31</td>
</tr>
<tr>
<td>20 Counterexample for a reachable of the initial marking</td>
<td>31</td>
</tr>
<tr>
<td>21 Counterexample for the liveness-condition for $A_3$</td>
<td>32</td>
</tr>
</tbody>
</table>
References


[Clavel et al. 2002] Clavel, Manuel; Durán, Francisco; Eker, Steven; Lincoln, Patrick; Martí-Oliet, Narciso; Meseguer, José; Quesada, José F: Maude: Specification and programming in rewriting logic. In: Theoretical Computer Science 285 (2002), Nr. 2

[Clavel et al. 2011] Clavel, Manuel; Durán, Francisco; Eker, Steven; Lincoln, Patrick; Martí-Oliet, Narciso; Meseguer, José; Talcott, Carolyn: Maude manual (version 2.6). In: University of Illinois, Urbana-Champaign 1 (2011), Nr. 3, S. 4–6


