Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Projekt 2

Alexander Schulz

Converting Reconfigurable Petri Nets to Maude

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Alexander Schulz

Converting Reconfigurable Petri Nets to Maude

Projekt 2 eingereicht im Rahmen der Projekte

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

Eingereicht am: 28. Sep. 2014

Abstract

Model checking is an important aim of the theoretical computer science. It enables the verification of a model with a set of properties such as liveness, deadlock or safety. One of the typical modelling techniques are Petri nets they are well understood and can be used for a model checking. Reconfigurable Petri nets are based on a Petri nets with a set of rules. These rules can be used dynamically to change the net.
Missing is the possibility to verify a reconfigurable net and properties such as deadlocks or liveness. This paper introduces a conversion from reconfigurable Petri net to Maude, that allows us to fill the gap. It presents a net transformation approach which is based on Maude's equation- and rewrite logic as well as the LTLR model checker.

Contents

List of Figures

1 Motivation

The first approach to convert reconfigurable Petri net to Maude (see [1]) 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*.
Hence, that the model-checking process has only one-step application of a rule may be wrong if a rule is used twice, because an error may occur after the second usage.
The new approach is based on the algebraic structure of reconfigurable Petri nets. The main task was to create a structure which can be read similar to the mathematical notation of a reconfigurable Petri net. It includes the possibility to simulate the net. This implies a solution for a transition that defines the activation and firing. Maude has been chosen as the appropriate language to implement this definition.
Moreover, the rules need to be implemented within the new structure. In contrast to the first approach it should be able to detect a match itself.
The following sections contain an overview of all relevant parts of this approach. The first section gives a short overview of the background for this work. The next section focuses the new modules, which contain the data-types for the resulting Maude specification of a converted reconfigurable Petri net. Finally, an evaluation shows the performance of the current implementation based on a test net.

2 Background

First we introduce reconfigurable Petri nets. They extend Petri nets with a set of rules, that can modify the net at runtime. Moreover, Maude is introduced since it is the result the conversion aims at. Finally, a short survey 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 [2].
A marked Petri net can be formally described as a tuple $N = (P, T, pre, post, M_0)$ where P is a set of places and T is a set of transitions. pre is used for all pre-conditions of transitions, which describes how many token are required for firing. On the other hand, $post$ holds all information of the post-conditions for all transitions. Finally, M_0 shows all initial tokens on the places for this net N [3, 4].
Further, based on a Petri net are reconfigurable Petri nets important because they can modify themselves with a set of rules [5–7]. A reconfigurable Petri net can be describe a tuple of a reconfigurable Petri net $RN = (N, \mathcal{R})$. This definition uses the Petri net tuple and a set $\mathcal{R}$ of rules, which are given by rule $r = (L \leftarrow K \rightarrow R)$ [8, 9]. L is the left-hand side (LHS), which needs a morphism to be mapped to a net N. K is an interface between L and R. R is the part which is inserted into the original net. To realise this replacement a matching algorithm needs to be defined that finds L within the source net N. 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 [10].
Reconfigurable Petri nets are also comprises capacities and labels for transitions/places [8]. A limitation for place is realised via a capacity, that contains a value which describes how much token can be stored on a place. The function *cap* : $P \rightarrow \mathbb{N}_+^w$ assigns for each place a natural number as capacity. Further, two label function (*pname* and *tname*) refer for each place or transition a name from a name space (*pname* : $P \rightarrow A_P$ and *tname* : $R \rightarrow A_T$).
ReConNet is shown in Figure 1 with an example net N_1 and rule $\mathcal{R}_1$. The configurations such as node names or markings as well as the control elements for firing and transformation are

presented in the upper region of the graphical interface. A graphical illustration of reconfigurable Petri nets is in the remaining interface. First, a net which models a cycle is composed of three places and transitions as well as one token. Wider, a rule which changes the arc direction of a transition is displayed under the net. In both editors are activated transitions marked as black transitions instead of grey normal transitions. Colours for places in rules are used to ensure the common bond.

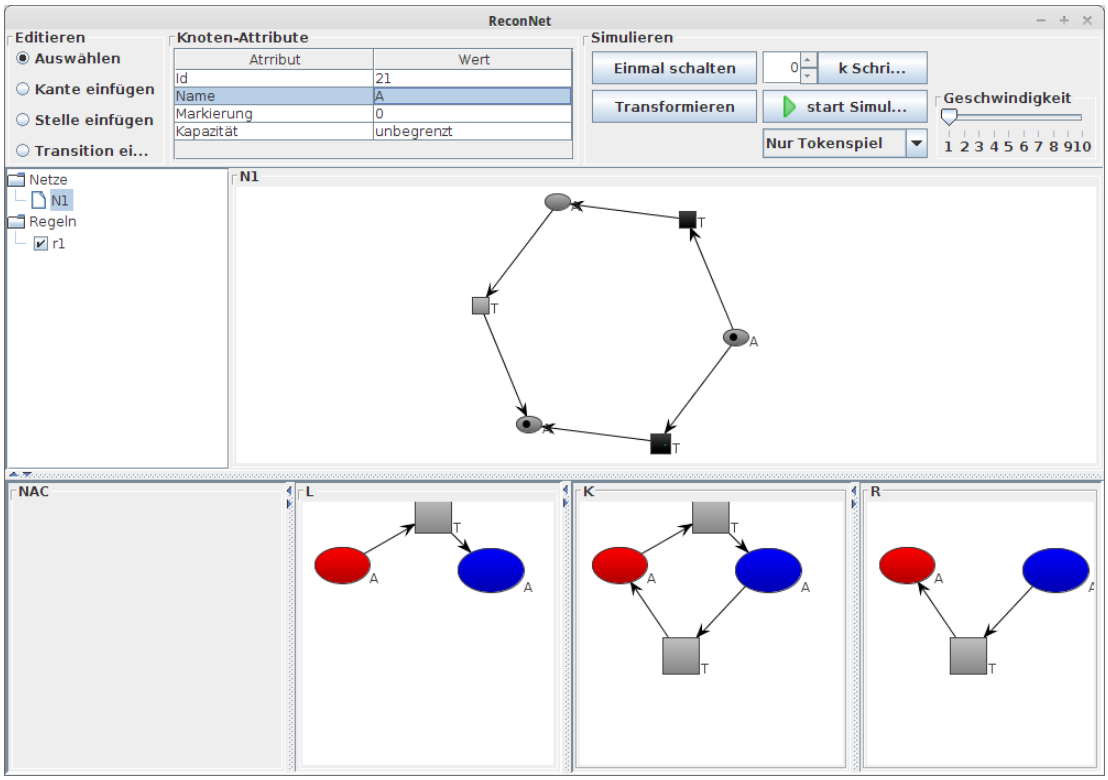

Figure 1: ReConNet - graphical editor for reconfigurable Petri nets

2.2 Maude

Maude has been developed at the Stanford Research Institute International (SRI International) for the last two decades. The equation and rewriting logic, which supports a powerful algebraic language, is used as a base [11, 12]. Based on these two kinds of logics Maude models a concurrent state systems that used for semantic analysis such as deadlock discovery via LTL-model-checking-module [13, 14].
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[1] [15]. A program in Maude is based 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:

$$sort\ Places\ Transitions\ Markings\ . \tag{2.1}$$

Depending on some sorts a set of operators needs to be defined. These operators describe all functors which are used to work with the defined types. For example a functor for writing a multiset of markings, can be expressed with a whitespace. This whitespace is surrounded with underscore, denoting a placeholder for the types defined after the double point. The return type right of the arrow, is of sort *markings*.

$$op\ \_\ \_\ :\ Markings\ Markings\ \rightarrow\ Markings\ . \tag{2.2}$$

If this operator has to be associative (in Maude with a short version: „assoc") and commutative (short with: „comm") properties, Maude defines 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:

$$op\ \_\ \_\ :\ Markings\ Markings\ \rightarrow\ Markings\ [assoc\ comm]\ . \tag{2.3}$$

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 marking 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 added. If we have a Petri net with only one marking with an „A" label we obtain these two lines:

$$op\ initial\ :\ \rightarrow\ Markings\ . \tag{2.4}$$

$$eq\ initial\ =\ A\ . \tag{2.5}$$

[1] `www.maude.cs.uiuc.edu/`, retrieval on 16/05/2014

Types are defined as „sort“, operators as functors and equations as the validity of operators. The rewrite rules can be used to replace one multiset with another multiset. So all terms are immutable as in many functional languages. A replacement rule consists of two multisets, where the first set is replaced with the second one. These two termsets are separated with a double arrow, as shown in the following example, where a term A is replaced with a new term B:

$$rl\ [T]\ :\ A\ \Rightarrow\ B\ . \tag{2.6}$$

Based on this example an implementation of the token game of Petri nets can be realised. The two multisets 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:

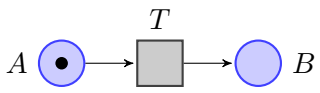

Figure 2: Example Petri net N for the Equation 2.6

2.3 Related Work

The basic example for a Petri net to Maude conversion uses a 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 [15].

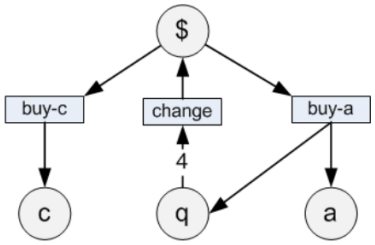

Figure 3: Shop example with apple and candy

A more complex example of this modelling (see Figure 3) is shown with high level nets in [16]. It presents a conversion of the banker problem where two credits are handled by the net structure. The focus of the work lays on the soundness and correctness for this conversion

approach. A formal definition of the model and the operators as well as the firing of a transition is given. Further, the paper presents an approach for a coloured Petri net (CPN). It extends the previous approach with more complex operators for the firing replacement rules. The conversion of transitions is realised via Maudes replacement rules and operators which contain the coloured tokens.
Automatic mapping for UML-models to a Maude-specification (see [17]) is similar to this paper's idea of converting reconfigurable Petri nets. In [17] the authors present three steps modeling, analysing and converting 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.
In [18] 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 conversion 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 [17].
A use case for the resulting Maude structure is presented in [19]. The authors modelled the public transport of Oslo with a Petri net, which is converted into a Maude structure. The aim is to proof the net with Maude's LTL module and properties such as deadlock freedom or liveness. An evaluation shows that the simulation can handle a net with 2067 places and 3572 transitions as well up to 32 tokens (which models the trains) [20]. Test measurements for one train used 1614 rewrites and for 32 trains 35630 rewrites in 3.62 seconds.
[21] presents a graphical editor for CPNs. It uses Maude in the background to verify properties such as liveness and deadlock freedom. Therefore, it converts a net into specified Maude modules (similar to [16]) which are simulations with one step commands. After one step the tool is capable to present the results.

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 $\mathcal{R}$. It can be written with $N_1 = (N, \mathcal{R})$. 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^{ω} [8].
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.

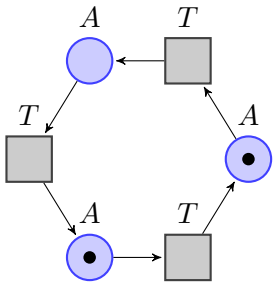

Figure 4: Example Petri net N_1

A rule is consists of three Petri nets L, K and R. 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. K is the interface between L and R.
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 deleting one transition and adding a new transition with reversed arcs.

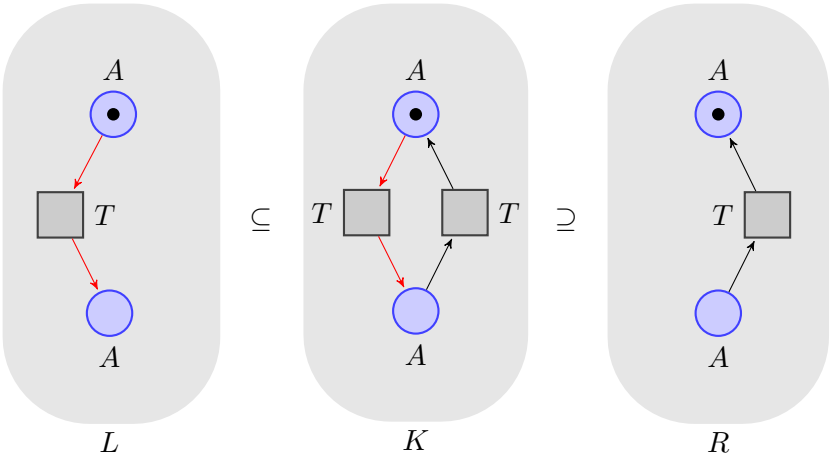

Figure 5: Example rule r_1, which switch the arc direction

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:

$$
\begin{aligned}
P &= \{A_2, A_3, A_4\} & post(T_5) &= A_3 \\
T &= \{T_5, T_6, T_7\} & post(T_6) &= A_4 \\
pre(T_5) &= A_4 & post(T_7) &= A_2 \\
pre(T_6) &= A_2 & m &= A_2 + A_4 \\
pre(T_7) &= A_3 & cap &= \omega A_2 + \omega A_3 + \omega A_4
\end{aligned}
$$

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

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 1.

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 *subsort*. This is a membership equation logic feature of Maude. It enables Maude to use a mapping between different types. This facilitated to describe the type *Places* as subset of the type *Markings*.

Based on the types, all operators can be defined. A definition begins with the keyword *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.

```
1  sorts Net Places Transitions Pre Post MappingTuple
       Markings .

3  subsort Places < Markings .

5  op emptyPlace : -> Places .
   op emptyTransition : -> Transitions .
7  op emptyMappingTuple : -> MappingTuple .
   op emptyMarking : -> Markings .
9
   op _,_ : Places Places -> Places [ctor assoc comm id:
       emptyPlace] .
11 op _+_ : Places Places -> Places [ctor assoc comm id:
       emptyPlace] .
   op _:_ : Transitions Transitions ->
13            Transitions [ctor assoc comm id:
                  emptyTransition] .
   op _,_ : MappingTuple MappingTuple ->
15            MappingTuple [ctor assoc comm id:
                  emptyMappingTuple] .
   op _;_ : Markings Markings -> Markings [ctor assoc comm
       id: emptyMarking] .
17
   *** READING: Pname | ID | Cap
19 op p(_|_|_) : String Int Int -> Places .
   op t(_|_) : String Int -> Transitions .
21 op (_-->_) : Transitions Places -> MappingTuple .

23 op places{_} : Places -> Places .
```

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

29  *** Petrinet-tuple
    op net : Places Transitions Pre Post Markings -> Net .
```

Listing 1: Maude module for a Petri net

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 conditions 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 Listing 2. It separates all operators and the included multi-sets with commas. Each multi-set is wrapped with curved brackets. A place is modelled as tuple with *p(<label> | <identifier> | <capacity>)*. Labels are defined as a string, identifier and capacity as numbers. Transitions are based on the tuple *t(<label> | <identifier>)*. Each type has the same type as for the place.

```
net(places{ p("A" | 3 | 2147483647) , p("A" | 4 |
    2147483647) ,
2             p("A" | 2 | 2147483647) } ,
     transitions{ t("T" | 7) : t("T" | 5) : t("T" | 6) } ,
4    pre{ (t("T" | 7) --> p("A" | 3 | 2147483647)) ,
           (t("T" | 5) --> p("A" | 4 | 2147483647)) ,
6          (t("T" | 6) --> p("A" | 2 | 2147483647)) } ,
     post{ (t("T" | 7) --> p("A" | 2 | 2147483647)) ,
8           (t("T" | 5) --> p("A" | 3 | 2147483647)) ,
            (t("T" | 6) --> p("A" | 4 | 2147483647)) } ,
10   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 activated, written by $m[t\rangle$, when the following two conditions are satisfied. The first condition consists of the pre-set of this transition. The net marking has to contain at least as many tokens, as described it in the pre-set (see Equation 3.1). Furthermore, all post places have to satisfy the capacity condition. Adding more tokens than a place can store is not possible (see Equation 3.2).
If both conditions are satisfied, the transition t can fire. One firing step is written with $m[t\rangle m'$, where m is the current marking and m' is the following marking. The calculation of m' is described in Equation 3.3. First, the pre-set is deducted from the current marking. Now the post-set of t can be added to the result.

$$pre^{\oplus}(t) \leq m \tag{3.1}$$

$$m + post^{\oplus}(t) \leq cap \tag{3.2}$$

$$m' = (m \ominus pre^{\oplus}(t)) \oplus post^{\oplus}(t). \tag{3.3}$$

A conversion of the first condition (see Equation 3.1) is shown in Listing 3. The rewrite rule contains two parts for the condition. First, *T -> PreValue* models $pre^{\oplus}(t)$. Second, $\leq m$ is converted into *marking*{*PreValue ; M*}. Hence, the formal definition is implemented. Either the pre-set of a transition is a part of the marking multi-set, or the rule is not enabled for firing. This implementation uses the matching algorithm from Maude to find possible cases of applications. It is able to determine when the termset contains this condition. In summary, a rule uses one transition from the *net*-tuple and tests the existing in the *pre*-set in the current marking.
Furthermore, the Equation 3.2 is expressed by 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 *if* in the last line of Listing 3. The addition result is used with the <=? which requires a multi-set of places on the right side. Details can be found in Listing 4.
Further, the rule result contains a function which calculates the resulting set of markings.

calc(((*PreValue* ; *M*) *minus PreValue*) *plus PostValue*)

And may be read as the formal definition in Equation 3.3, where *PreValue ; M* describes *m* and *PreValue* is the pre-value of the transition *v*. The place holder *PostValue* represents the post-domain of the transition *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) minus PreValue) plus
          PostValue))
```

```
    Rules
16  I
    if calc((PreValue ; M) plus PostValue) <=? PostValue
       .
```

Listing 3: Activation and firing of a transition

The operator <=? (in words *smallerAsCap*) maps the capacity condition in this Maude module. The aim is to return *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 _ *leeqth* _ *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. In case that the counter is bigger than the capacity it returns false. All other cases results with true.

```
1   op _ <=? _ : Markings Places -> Bool .
    op _ leeqth _ with _ : Places Places Int -> Bool .
3
    *** Impl - smallerAsCap #############
5   eq marking{ PSet } <=? emptyPlace = true .
    ceq marking{M} <=? (P , emptyPlace)
7       = true
        if M leeqth P with 0 .
9   ceq marking{M} <=? (P , PRest)
        = true
11      if M leeqth P with 0
           /\ PRest =/= emptyPlace
13         /\ marking{M} <=? PRest .
    eq M <=? P = false [owise] .
15
    *** Impl - lowerEqualThan ##########
17  *** place multiset is empty
    ceq emptyMarking leeqth p(Str | I | Cap1) with Counter
19      = true if Counter <= Cap1 .
    *** Cap-counter is too big
```

```
21  eq (p(Str | I | Cap2) ; MRest) leeqth p(Str | I | Cap2)
      with (Cap2 + 1)
      = false .
23  *** found same place
    ceq (p(Str | I | Cap2) ; MRest) leeqth p(Str | I | Cap2)
      with Counter
25    = true
      if (MRest leeqth p(Str | I | Cap2) with (Counter +
        1)) .
27  *** del another place
    ceq (p(Str | I | Cap1) ; MRest) leeqth p(Str2 | I2 |
      Cap2) with Counter
29    = true
      if (MRest leeqth p(Str2 | I2 | (Cap2)) with Counter)
        .
31  *** otherwise
    eq M leeqth P with I = false [owise] .
```

Listing 4: Capacity proof of each place in the post-set

3.2 LTL Properties

The aim of this work is to verify properties such 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 uses a Kripke structure, which is realised on the basis of the equation and rewriting logic, basically a finite transition system [14].

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 informations. 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 MaxID StepSize aidP aidT
    |= reachable(M) = true .

op t-enabled : -> Prop .

eq net(P , T ,
        pre{ (T1 --> PreValue) , MappingTuple } ,
        Post ,
        marking{ PreValue ; MRest } )
        Rules MaxID StepSize aidP aidT
    |= t-enabled = true .
  eq C |= t-enabled = false [owise] .

op enabled : -> Prop .

eq net(P , T ,
        pre{ (T1 --> PreValue) , MappingTuple } ,
        Post ,
        marking{ PreValue ; MRest } )
        Rules MaxID StepSize aidP aidT
    |= enabled = true .

eq net(places{ p("A" | Irule2017 | 2147483647) ,
               p("A" | Irule2020 | 2147483647) , P } ,
        transitions{ t("T" | Irule2024) : T } ,
```

```
32          pre{ (t("T" | Irule2024) --> p("A" | Irule2017 |
                2147483647)) ,
                    MTupleRest1 } ,
34          post{ (t("T" | Irule2024) --> p("A" | Irule2020 |
                2147483647)) ,
                    MTupleRest2 } ,
36          marking{ p("A" | Irule2017 | 2147483647) ; M } )
            Rules MaxID StepSize aidP aidT
38      |= enabled = true .

40  var C : Configuration .
    var Prop : Prop .
42  eq C |= Prop = false [owise] .
```

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 contain a net L for matching, a net R for the replacement and a net K that maps between the two nets.

The first project [1] is based on *ReConNet*. This tool provides the capability to find non-deterministic matches for a net and a set of rules [10]. 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 by a conversion to a Maude specification. In order to realise this process, an interface is designed for using *ReConNet* to find a match. Due to this constellation, only the initial state of a net and all rules can be verified with the LTL-process.

The new aim is to ensure that the Maude specification finds the matching 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-data configuration (such as current highest identifier) should adapt a net and a set of rules.

The definition of a rule and the meta configuration can be found in Listing 6. First, 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* consists of 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-sets. 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 will be added.

```
*** Rule R = (l_net, r_net)
sort Rule .
sorts LeftHandSide RightHandSide .
op emptyRule : -> Rule .
op _|_ : Rule Rule -> Rule [ctor assoc comm id: emptyRule]
    .
op l : Net -> LeftHandSide .
op r : Net -> RightHandSide .
op rule : LeftHandSide RightHandSide -> Rule .

*** Configuration
sort Configuration .
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 is 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. For example it contains the new elements (here a transition) and the related ID.
The whole rule is written as a condition replacement. If the conditions satisfied with the current net, the rule can be applied. Due to the replacement, the left-hand side of the rule is replaced with the right-hand side. Furthermore, a rule is working with a configuration object. It uses a net, a multi-set of rules as well as an ID-count.
The example in Listing 7 shows the left side of the rule in Figure 5. The net which should be found, consists 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 differs in the ID, which are not given as concrete numbers. Each *Irule<number>* is a variable that is used for finding a structural match. If the net has 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, each set contains a variable for possible residual elements. For example, in this net the set $places$ contains more than two places. The remaining place's are mapped into the variable $PRest$, so this rule is still activated. If a multiset has no more elements the identities (id) is used (for a definition see Listing 1). Each id itself is the empty-set constant operator (such as emptyPlace, emptyTransition, emptyMappingTuple or emptyMarking).

```
crl [R1-PNML] :
  net(places{p("A" | Irule1017 | 2147483647) ,
             p("A" | Irule1020 | 2147483647) , PRest}
                 ,
    transitions{ t("T" | Irule1024) : TRest} ,
    pre{(t("T" | Irule1024) --> p("A" | Irule1017 |
       2147483647)) ,
        MTupleRest1} ,
    post{(t("T" | Irule1024) --> p("A" | Irule1020 |
        2147483647)) ,
         MTupleRest2} ,
    marking{p("A" | Irule1017 | 2147483647) ; MRest}
        )
  rule(l(net(places{p("A" | Irule2017 | 2147483647) ,
                    p("A" | Irule2020 | 2147483647)} ,
    transitions{t("T" | Irule2024)} ,
    pre{(t("T" | Irule2024) -->
          p("A" | Irule2017 | 2147483647))}
                 ,
    post{(t("T" | Irule2024) -->
           p("A" | Irule2020 | 2147483647))
              } ,
    marking{p("A" | Irule2017 | 2147483647)}
       ) ) ,
    r(net(places{p("A" | Irule3017 | 2147483647) ,
                 p("A" | Irule3020 | 2147483647)} ,
    transitions{t("T" | Irule3026)} ,
    pre{(t("T" | Irule3026) -->
```

```
22                          p("A" | Irule3020 | 2147483647))}
                                ,
                    post{(t("T" | Irule3026) -->
24                           p("A" | Irule3017 | 2147483647))
                                } ,
                    marking{p("A" | Irule3017 | 2147483647)}
                       ) ) )
26      | RRest MaxID StepSize
        aidPlace{AidPRest} aidTransition{AidTRest}
```

Listing 7: Example of rule r_1 written with Maude (left-hand side)

In the following example in Listing 8 the right-hand side differs from the left-hand side. It contains the net structure of the right-hand side. The example adds the new transition T. It calculates the new identifier which is detailed described in section 3.5.

```
1       =>
        net(places{p("A" | Irule1017 | 2147483647) ,
3                  p("A" | Irule1020 | 2147483647) , PRest}
                      ,
            transitions{t("T" | AidT1 ) : TRest} ,
5           pre{(t("T" | AidT1 ) --> p("A" | Irule1020 |
               2147483647)) ,
                  MTupleRest1} ,
7           post{(t("T" | AidT1 ) --> p("A" | Irule1017 |
               2147483647)) ,
                   MTupleRest2} ,
9           marking{p("A" | Irule1017 | 2147483647) ; MRest}
                )
        rule(l(net(places{p("A" | Irule2017 | 2147483647) ,
11                        p("A" | Irule2020 | 2147483647)} ,
                    transitions{t("T" | Irule2024)} ,
13                  pre{(t("T" | Irule2024) -->
                             p("A" | Irule2017 | 2147483647))
                                } ,
15                  post{(t("T" | Irule2024) -->
```

```
                                p("A" | Irule2020 | 2147483647))
                                    } ,
17                      marking{p("A" | Irule2017 | 2147483647)}
                            ) ) ,
                r(net(places{p("A" | Irule3017 | 2147483647) ,
19                              p("A" | Irule3020 | 2147483647)}
                                    ,
                        transitions{t("T" | Irule3026)} ,
21                      pre{(t("T" | Irule3026) -->
                                p("A" | Irule3020 | 2147483647))}
                                    ,
23                      post{(t("T" | Irule3026) -->
                                    p("A" | Irule3017 | 2147483647)
                                        )} ,
25                      marking{p("A" | Irule3017 | 2147483647)}
                            ) ) )
        | RRest NewMaxID StepSize
27      aidPlace{AidPRest} aidTransition{AidTRest2}
        if AidTRest1 := addOldID(AidTRest | Irule1024) /\
29         AidT1 := getAid(AidTRest1 | MaxID | StepSize) /\
           AidTRest2 := removeFirstElement(AidTRest1 | MaxID
               | StepSize) /\
31         NewMaxID := correctMaxID(MaxID | StepSize | 2) .
```

Listing 8: Example of the rule r_1, 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 specified to be simultaneously added and deleted. Further, the dangling condition defines that a place can only be deleted if there are only arcs to transitions, that are deleted as well. Transitions are not relevant for dangling condition.
The example net N_2 in Figure 7 shows a short 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 in an addition transition. This transition injured the dangling condition and the rule cannot be used at this point.

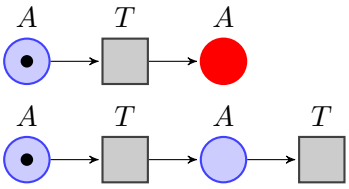

Figure 7: Example net N_2

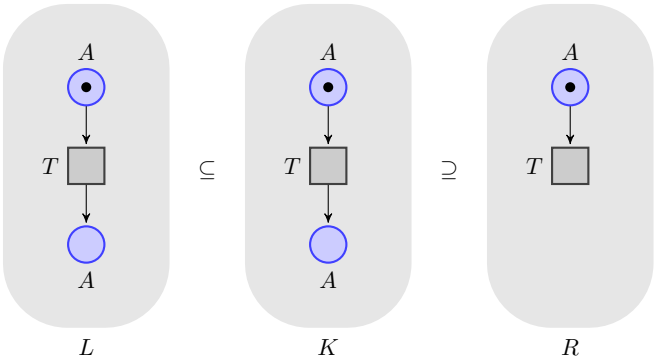

Figure 8: Example rule r_3 which deletes the place A

The associated Maude code can be found in Listing 9. It is seperated in two parts. First, the *equalMarking*-operator tests the current net marking. This condition ensures that the net marking and the rule are the same for all deleted places.
The marking of the net is given with:

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 contain all places that should be deleted. In this example the other place is called *A*. Hence, the following line contains all relevant information:

marking{ p(„A“| Irule2019 | 2147483647) }

The result is true, if the second set of markings is equal to the marking in the first set.
Furthermore, for each deleted place is a *emptyNeighbourForPlace*-operator defined. This operator tests the dangling condition with the remaining pre and post multi-sets. In Listing 9 this two sets are defined as *MTupleRest1* and *MTupleRest2*. All operator definitions can be found in Listing 10.

```
equalMarking( ( marking{p("S" | Irule107 | 2147483647) ;
    MRest} )
                    =?=
                    marking{emptyMarking} ) /\
emptyNeighbourForPlace(p("S" | Irule106 | 2147483647) ,
                           pre{ MTupleRest1 } ,
                           post{ MTupleRest2 } )
```

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

All needed operators for the dangling condition are defined in Listing 10. The *equalMarking*-operator proves 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 only true, 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 differentiations. The first equation includes the situation where two identical marks are 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 returns true. The following equation differs only in case distinction. It returns true, including a recursion, when the markings are not in the multi-sets. Based on this equation, the following lines include the situation where the second set only contains one element. If the first remaining multi-set does not contain more than one of these markings, it returns true. Finally, if it is not possible to use another case, this is called the „otherwise case“ (Maude's keyword with brackets: *[owise]*). For all other conditions 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 other case is true, where no arc exists.

```
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
```

```
6  op equalMarking(_ =?= _) : Places Places -> Bool .
   eq equalMarking(p(Str | I1 | Cap) =?= p(Str | I2 | Cap)) =
       true .
8  ceq equalMarking(
       (p(Str | I1 | Cap) , MNet) =?= (p(Str | I2 | Cap) ,
          MRest))
10     = true
       if contains((p(Str | I1 | Cap)) | MNet) /\
12        contains((p(Str | I2 | Cap)) | MRest) /\
          (MRest =/= emptyMarking) /\
14        equalMarking(MNet =?= MRest) .
   ceq equalMarking(
16     (p(Str | I | Cap) , MNet) =?= (p(Str | I | Cap) ,
          MRest))
       = true
18     if not(contains((p(Str | I | Cap)) | MNet)) /\
          not(contains((p(Str | I | Cap)) | MRest)) /\
20        (MRest =/= emptyMarking) /\
          equalMarking(MNet =?= MRest) .
22 ceq equalMarking(
       (p(Str | I1 | Cap) , MNet) =?= (p(Str | I2 | Cap))) =
          true
24     if not(contains((p(Str | I2 | Cap)) | MNet)) .
   eq equalMarking(
26    (PNet) =?= (PRule))
      = false [owise] .
28
   *** READING: PLACE, PRE, POST
30 op emptyNeighbourForPlace(_, _, _) : Places Pre Post ->
      Bool .
   eq emptyNeighbourForPlace(P,
32       pre{ (T --> P , PRest) , MTupleRest },
         Post) = false .
34 eq emptyNeighbourForPlace(P,
         Pre,
```

```
36      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[1]. This can result in an overflow if a rule inserts a node (place or transition), because each new node gets an identifier.
To solve this problem a multi-set of unique identifiers is used. It requires a modification of the *Configuration* definition which is introduced in Listing 6. Now it contains an integer for the *maxID* and for the defined *step_size*. Further, it has two sets for the place and transition identifiers. The implementation can be found in Listing 11.

```
1  *** READING: NET SET<RULE> MAXID STEP_SIZE PID TID
   op ______ :
3     Net Rule Int Int IDPoolPlace IDPoolTransition
      ->
5     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)* has 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 is reused in line 21, where the ID is added into the new identifier multi-set *AidTRest1*. The next step uses this multi-set to receive 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.

[1] `http://maude.cs.uiuc.edu/maude1/manual/maude-manual-html/maude-manual_42.html`, retrieval on 20/10/2014

```
crl [R1-PNML] :
    ...
    transitions{ t("X" | Irule1031) : TRest } ,
    ...
    MaxID
    StepSize
    aidPlace{ AidPRest }
    aidTransition{ AidTRest }
    =>
    ...
    transitions{ t("X" | AidT1) : TRest } ,
    pre{ (t("X" | AidT1) --> p("A" | Irule1013 |
        2147483647)) ,
            MTupleRest1 } ,
    post{ (t("X" | AidT1) --> 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) .
```

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

Every operator which is used in the Listing 12 is shown in 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 receive 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 the *getAid*-operator provides the function to receive the first element of a multi-set. Otherwise,

the *removeFirstElement*-operator can delete this first element of a given multi-set. Moreover, the *addOldID*-operator adds an element into a multi-set and the *correctMaxID*-operator defines a new *maxID* if it is necessary.

```
*** READING: IDSET MAXID COUNTER INTERNAL-VAR
op fill(_|_|_|_) : Int Int Int Int -> Int .
eq fill(I | MaxID | 0 | Count) = I .
ceq fill(IRest | MaxID | Count | I)
     = fill((MaxID + I, (IRest)) | MaxID | (Count - 1) | (
        I - 1))
     if I >= Count .
eq fill(I1 | MaxID | I2 | Count ) = I1 [owise] .

*** READING: CURRENT_SET MAXID STEP_SIZE
op getAid(_|_|_) : Int Int Int -> Int .
ceq getAid(I1 , (IRest) | MaxID | StepSize) = I1 if I1 =/=
    emptyIDSet .
eq getAid(SetOfInts | MaxID | StepSize)
    = getAid(fill(SetOfInts | MaxID | StepSize | StepSize)
               | MaxID + MaxID | StepSize) [owise] .

*** READING: CURRENT_SET MAXID STEP_SIZE
op removeFirstElement(_|_|_) : Int Int Int -> Int .
eq removeFirstElement(emptyIDSet | MaxID | StepSize) =
                           fill(emptyIDSet | MaxID |
                              StepSize | StepSize) .
ceq removeFirstElement(I1 , (IRest) | MaxID | StepSize) =
   IRest
                            if I1 =/= emptyIDSet [owise] .
*** READING: CURRENT_SET OLD_ID
op addOldID(_|_) : Int Int -> Int .
eq addOldID(SetOfInts | I) = I, (SetOfInts) .

*** READING: MAXID STEP_SIZE NEW_ID_COUNT
op correctMaxID(_|_|_) : Int Int Int -> Int .
```

```
28 ceq correctMaxID(MaxID | StepSize | Count)
                      = correctMaxID(MaxID + StepSize |
                          StepSize |
30                    Count - StepSize)
                      if Count > StepSize .
32 eq correctMaxID(MaxID | StepSize | Count) = MaxID .
```

Listing 13: Identifier multiset implementation

4 Transformation

This section includes the architecture of the conversion process as well as the results of some model checking formula.

4.1 Architecture

The output base of *ReConNet* consists an extension of PNML[1]. PNML is a XML-based standard for the Petri net export. The graphical editor *ReConNet* uses this standard for the persistence of developed nets. In addition to the pure PNML-standard, a rule is stored with its three net in a PNML file.
Based on PNML, this work uses XSL to realise the conversion. The result uses the Maude modules which are defined before the conversion.The sorts for *Places, Transitions* and the *net* itself are previously defined. And further it contains 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*. Further, it has the specific sub xsl-templates for the conversions as in the *net, rules, prop* or *rpn*. The structure is summarized in Figure 9. The global types are defined above the specific modules that are grouped together in separate packages.

[1] `http://www.pnml.org/`, retrieval on 23/09/2014

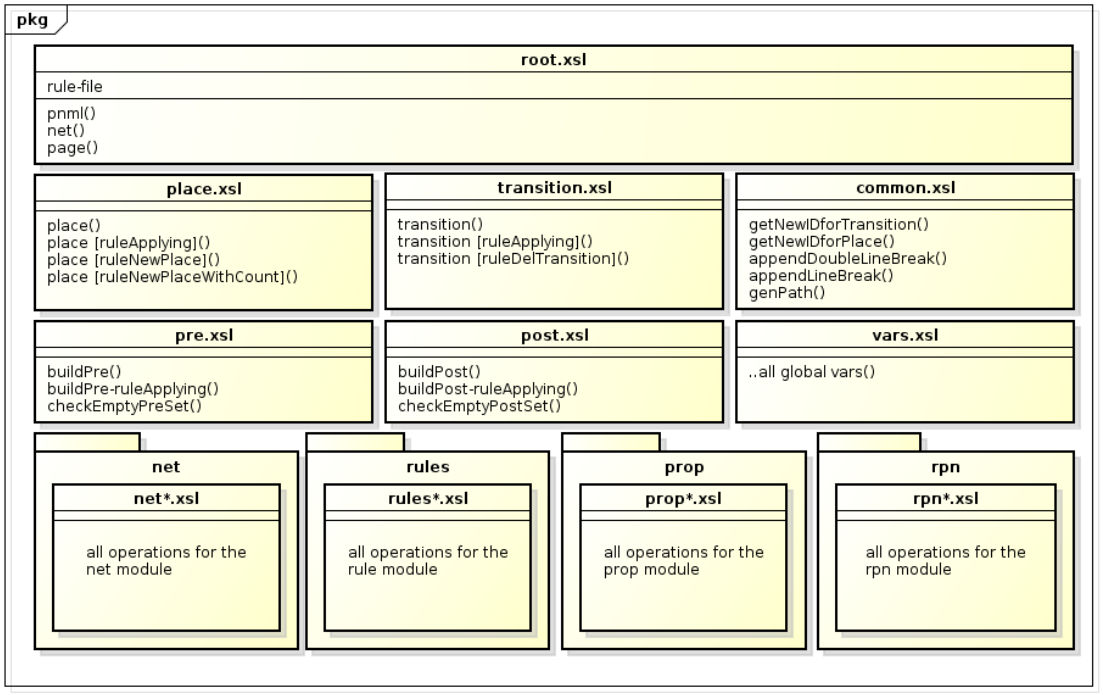

Figure 9: Structure of the stylesheets for the conversion

The difference to the first project is that this approach is independent of *ReConNets* implementation. The first project uses the *persistence*-module of *ReConNets* to load PNML-files (net or rule) [1]. This approach is superior because it is built up directly on the PNML data. The conversion process is written with XSL, which provides a well known language. The interface between this approach and *ReConNet* is realised through an export to the PNML files that can be used with a net or few rules.

4.2 Results

Based on the result of the conversion process it is possible to use Maude's LTL implementation. The *prop.maude*-module includes all the 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 .
2       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 is based on the box- and diamond-operator. In total it describes the semantic of liveness. It means that a property is globally (box) repeatedly (diamond) true. To write the liveness property for the reconfigurable Petri net modules, the following line can be used:

```
rew modelCheck(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 <> as well as the *enabled*-operator (for the definition see Listing 5).
In terms of this 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) (270604 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)},
```

```
20                post{t("T" | 24) --> p("A" | 20 | 2147483647)
                      },
                  marking{p("A" | 17 | 2147483647)})),
22          r(net(places{p("A" | 17 | 2147483647),p("A" | 20 |
                2147483647)},
                  transitions{t("T" | 26)},
24                pre{t("T" | 26) --> p("A" | 20 | 2147483647)},
                  post{t("T" | 26) --> p("A" | 17 | 2147483647)
                      },
26                marking{p("A" | 17 | 2147483647)})))
    26
28  10
    aidPlace{26,(27,(28,(29,(30,(31,(32,(33,(34,(35,(
30  36)))))))))}
    aidTransition{26,(27,(28,(29,(30,(31,(32,(33,(34,(35,(36)
       )))))))))}
32  ,deadlock})
```

Listing 15: Counterexample of a deadlock

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

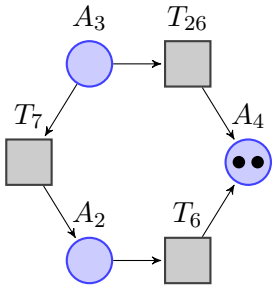

Figure 10: State of N_1 with a deadlock (r_1 can not be applied)

In assumption that the markings were changed on the places within the rule (see Figure 11) the result is varied, shown 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. First, the rule is not

enabled when a marking is at a place, where all arcs are starting. 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 *enabled*-operator is defined for the firing and transformation step.

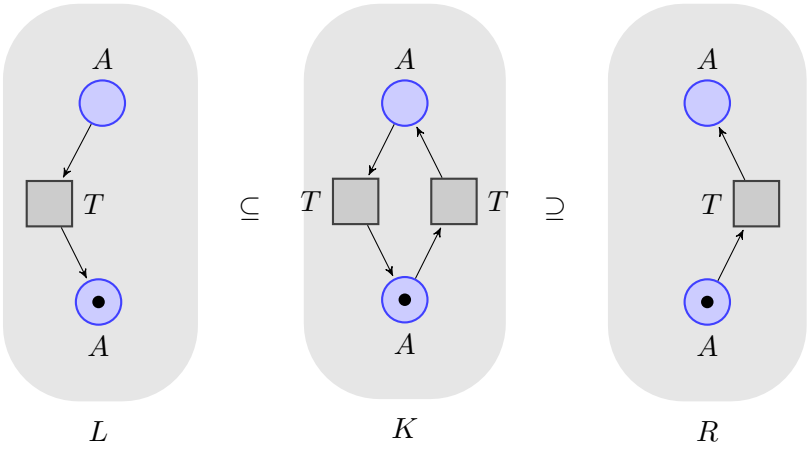

Figure 11: Example rule $r2$ which changes the direction of the arc (different marking in contrast to r_1 in Figure 5)

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

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

Listing 16: Counterexample for the deadlock freeness

In addition, the enable-test can only be realised for the transitions with the *t-enabled*-operator. It only consists of the transitions and no rules (see Listing 17).

```
Maude> rewrite [1] in NET : modelCheck(initial, []<> t-
    enabled) .
```

```
2 rewrites: 1521 in 4ms cpu (6ms real) (380250 rewrites/
    second)
result Bool: true
```

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:

op reachable : Markings → Prop.

Therefore, it is possible to test one or many tokens. In the example bellow (see Listing 18) two tokens (A_3 and A_3) are searched in N_1. Maude's result describes a situation where it found a deadlock with two tokens on A_2.

```
1 Maude> rew modelCheck(initial, <> reachable(p("A" | 3 |
    2147483647) ; p("A" | 3 | 2147483647))) .
  rewrite in NET : modelCheck(initial, <> reachable(p("A" |
    3 | 2147483647) ; p("A" | 3 | 2147483647))) .
3 rewrites: 141 in 0ms cpu (0ms real) (262081 rewrites/
    second)
  result ModelCheckResult: counterexample(
5 ...
  {net(places{p("A" | 2 | 2147483647), p("A" | 3 |
    2147483647),
7              p("A" | 4 | 2147483647)},
    transitions{t("T" | 5) : t("T" | 7) : t("T" | 26)},
9   pre{(t("T" | 5) --> p("A" | 4 | 2147483647)),
        (t("T" | 7) --> p("A" | 3 | 2147483647)),
11       t("T" | 26) --> p("A" | 4 | 2147483647)},
    post{(t("T" | 5) --> p("A" | 3 | 2147483647)),
13       (t("T" | 7) --> p("A" | 2 | 2147483647)),
         t("T" | 26) --> p("A" | 2 | 2147483647)},
15  marking{p("A" | 2 | 2147483647) ;
           p("A" | 2 | 2147483647)})
```

```
17  rule(l(net(places{p("A" | 17 | 2147483647), p("A" | 20 |
        2147483647)},
                transitions{t("T" | 24)},
19              pre{t("T" | 24) --> p("A" | 17 | 2147483647)
                    },
                post{t("T" | 24) --> p("A" | 20 |
                    2147483647)},
21              marking{p("A" | 17 | 2147483647)})),
      r(net(places{p("A" | 17 | 2147483647),p("A" | 20 |
          2147483647)},
23              transitions{t("T" | 26)},
                pre{t("T" | 26) --> p("A" | 20 | 2147483647)
                    },
25              post{t("T" | 26) --> p("A" | 17 |
                    2147483647)},
                marking{p("A" | 17 | 2147483647)})))
27  26
    10
29  aidPlace{26,(27,(28,(29,(30,(31,(32,(33,(34,(35,(36))))))
        )))))}
    aidTransition{26,(27,(28,(29,(30,(31,(32,(33,(34,(35,(36)
        ))))))))))}
31  ,deadlock}
```

Listing 18: Counterexample for a reachable test

An example where the LTL-formal is true can be found in Listing 21. This is a basic test where the formula verifies the reachability of the initial marking. Maude reached true after six steps because the marking already contains the search parameter.

```
1  Maude> rew modelCheck(initial, <> reachable(p("A" | 3 |
       2147483647) ; p("A" | 4 | 2147483647))) .
   rewrite in NET : modelCheck(initial, <> reachable(p("A" |
       3 | 2147483647) ; p("A" | 4 | 2147483647))) .
3  rewrites: 6 in 0ms cpu (0ms real) (50000 rewrites/second)
   result Bool: true
```

Listing 19: Counterexample for a reachable of the initial marking

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 differs 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> rew [1] modelCheck(initial, ~( <> reachable(p("A" |
    4 | 2147483647) ; p("A" | 4 | 2147483647)))) .
rewrite [1] in NET : modelCheck(initial, ~ <> reachable(p
   ("A" | 4 | 2147483647) ; p("A" | 4 | 2147483647))) .
rewrites: 174 in 0ms cpu (0ms real) (~ 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" | 7)},
  pre{(t("T" | 5) --> p("A" | 4 | 2147483647)),
       (t("T" | 6) --> p("A" | 2 | 2147483647)),
        t("T" | 7) --> p("A" | 3 | 2147483647)},
  post{(t("T" | 5) --> p("A" | 3 | 2147483647)),
        (t("T" | 6) --> p("A" | 4 | 2147483647)),
         t("T" | 7) --> p("A" | 2 | 2147483647)},
  markingp("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)},
26      transitions{t("T" | 26)},
        pre{t("T" | 26) --> p("A" | 20 | 2147483647)},
28      post{t("T" | 26) --> p("A" | 17 | 2147483647)},
        marking{p(
30    "A" | 17 | 2147483647)}))))
    26,{'fire} }, nil)
```

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 the test will return a situation where the net arrests 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).

```
1  Maude> rew modelCheck(initial, []<> reachable(p("A" | 3 |
       2147483647))) .
   rewrite in NET : modelCheck(initial, []<> reachable(p("A"
       | 3 | 2147483647))) .
3  rewrites: 197 in 0ms cpu (0ms real) (285094 rewrites/
       second)
   result ModelCheckResult: counterexample(
5  ...
   {net(places{p("A" | 2 | 2147483647), p("A" | 3 |
       2147483647),
7                p("A" | 4 | 2147483647)},
     transitions{t("T" | 6) : t("T" | 7) : t("T" | 26)},
9    pre{(t("T" | 6) --> p("A" | 2 | 2147483647)),
          (t("T" | 7) --> p("A" | 3 | 2147483647)),
11         t("T" | 26) --> p("A" | 3 | 2147483647)},
     post{(t("T" | 6) --> p("A" | 4 | 2147483647)),
13         (t("T" | 7) --> p("A" | 2 | 2147483647)),
           t("T" | 26) --> p("A" | 4 | 2147483647)},
```

```
15      marking{p("A" | 4 | 2147483647) ; p("A" | 4 |
            2147483647)})
    rule(l(net(places{p("A" | 17 | 2147483647),p("A" | 20 |
        2147483647)},
17              transitions{t("T" | 24)},
                pre{t("T" | 24) --> p("A" | 17 | 2147483647)
                    },
19              post{t("T" | 24) --> p("A" | 20 |
                    2147483647)},
                marking{p("A" | 17 | 2147483647)})),
21      r(net(places{p("A" | 17 | 2147483647),p("A" | 20 |
            2147483647)},
                transitions{t("T" | 26)},
23              pre{t("T" | 26) --> p("A" | 20 | 2147483647)
                    },
                post{t("T" | 26) --> p("A" | 17 | 2147483647)
                    },
25              marking{p("A" | 17 | 2147483647)}))),deadlock
                    })
```

Listing 21: Counterexample for the liveness-condition for A_3

4.3 Tests

Test cases are separated into two steps for the conversion to Maude and a verification with Maude's LTL model checker. XSLT provides a test mechanism with XSLTunit[2], which is implemented for the used XSL processors SAXON[3]. Further, the Maude test cases are realised with JUnit which implements a Java process. It is executing a shell script including a set of commands such as verifications for a deadlock. The *Maude Development Tools*[4] can not be used, since there is no support for model checking commands.

[2] `http://xsltunit.org/`, retrieval on 20/10/2014
[3] `http://saxon.sourceforge.net/`, retrieval on 20/10/2014
[4] `http://moment.dsic.upv.es/mdt/`, retrieval on 20/10/2014

5 Evaluation

This section presents a first step of the performance evaluation of this approach. The evaluation is based on two steps. At first, a net is converted into the Maude modules. And after that step it is tested with the liveness formula (see the first formula in section **??**).
The conversion 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 four net sizes are used, which allow to make a meaningful statement. Each net has the same semantic and should return true. Hence, only the runtime meta-data such as rewrite count and time are different. The conversion 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). The resulting state-space explosion was expectable as a well known issue of LTL [22].
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.

p x t		time [ms]
10x10	in	1701
20x20	in	1737
22x22	in	1704
23x23	in	1737

Table 1: Conversion from PNML to Maude

p x t	rew	time [ms]
10x10	109 in	13
20x20	209 in	23994
22x22	229 in	189939
23x23	239 in	834358

Table 2: Verification of liveness

6 Future Work

An outlook of the following work is shown in this section. The formal founding represents the essential part of the conversion from a reconfigurable Petri net to Maude. The aim is the verification of the soundness and correctness for the conversion and target representation of the net in Maude.
Furthermore, an integration of parts from a reconfigurable Petri net such as negative application conditions (NACs) (see [23]) or decorations (see [24]) should be realised. This enables the verification of the nets and rules from the *Living Place Hamburg* [24].
Finally, a benchmarking is necessary between this approach and a tool such as Charlie[1] to obtain a meaningful statement. This implies a way which converts a reconfigurable Petri net into a net that can be used from other verification tools. The main challenge is to realise a conversion which contains the net and all possible rule conditions.

[1] `http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Charlie`, retrieval on 19/10/2014

7 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.

8 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.

Bibliography

[1] Alexander Schulz. ReConNet-Modul zur Erzeugung von Maude-Modulen. 2014.

[2] Carl Adam Petri. Kommunikation mit automaten. 1962.

[3] José Meseguer and Ugo Montanari. Petri nets are monoids. *Information and computation*, 88(2):105–155, 1990.

[4] G. Juhas, F. Lehocki, and R. Lorenz. Semantics of petri nets: A comparison. In *Simulation Conference, 2007 Winter*, pages 617–628, 2007.

[5] Hartmut Ehrig, Kathrin Hoffmann, Julia Padberg, Ulrike Prange, and Claudia Ermel. Independence of net transformations and token firing in reconfigurable place/transition systems. In *Petri Nets and Other Models of Concurrency–ICATPN 2007*, pages 104–123. Springer, 2007.

[6] Ulrike Prange, Hartmut Ehrig, Kathrin Hoffmann, and Julia Padberg. Transformations in reconfigurable place/transition systems. In *Concurrency, Graphs and Models*, pages 96–113. Springer, 2008.

[7] Laïd Kahloul, Allaoua Chaoui, and Karim Djouani. Modeling and analysis of reconfigurable systems using flexible petri nets. In *Theoretical Aspects of Software Engineering (TASE), 2010 4th IEEE International Symposium on*, pages 107–116. IEEE, 2010.

[8] Julia Padberg. Abstract interleaving semantics for reconfigurable petri nets. *Electronic Communications of the EASST*, 51, 2012.

[9] Hartmut Ehrig, Frank Hermann, and Ulrike Prange. Cospan dpo approach: An alternative for dpo graph transformations. *Bulletin of the EATCS*, 2009.

[10] Mathias Blumreiter. Algorithmus zum nichtdeterministischen matching in rekonfigurierbaren petrinetzen, 2013.

[11] Santiago Escobar, José Meseguer, and Ralf Sasse. Variant narrowing and equational unification. *Electronic Notes in Theoretical Computer Science*, 238(3):103–119, 2009.

[12] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and José F Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2), 2002.

[13] Joost-Pieter Katoen and C Baier. Principles of model checking, 2008.

[14] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The maude ltl model checker. *Electronic Notes in Theoretical Computer Science*, 71:162–187, 2004.

[15] 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.

[16] 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, 2001.

[17] W Chama, Raida Elmansouri, and Allaoua 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. IEEE, 2013.

[18] Paulo Barbosa, Joao Paulo Barros, Franklin Ramalho, Luis Gomes, Jorge Figueiredo, Filipe Moutinho, Aniko Costa, and Andre Aranha. Sysveritas: A framework for verifying iopt nets and execution semantics within embedded systems design. pages 256–265, 2011.

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

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

[21] 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.

[22] Antti Valmari. The state explosion problem. In *Lectures on Petri nets I: Basic models*, pages 429–528. Springer, 1998.

[23] Alexander Rein, Ulrike Prange, Leen Lambers, Kathrin Hoffmann, and Julia Padberg. Negative application conditions for reconfigurable place/transition systems. *Electronic Communications of the EASST*, 10, 2008.

[24] Marvin Ede, Kathrin Hoffmann, Gerhard Oelker, and Julia Padberg. Reconnet: A tool for modeling and simulating with reconfigurable place/transition nets. In *Pre-Proceedings of the 7th International Workshop on Graph Based Tools (GraBaTs 2012)*, volume 10, page 79, 2012.