Model Checking Reconfigurable Petri Nets with Maude

Julia Padberg, Alexander Schulz
Hamburg University of Applied Sciences
Germany

Abstract. Model checking is a widely used technique to prove properties such as liveness, deadlock or safety for a given model. Here we introduce model checking of reconfigurable Petri nets. These are Petri nets with a set of rules for changing the net dynamically. We obtain model checking by converting reconfigurable Petri nets to specific Maude modules and using then the LTLR model checker of Maude. The main result of this paper is the correctness of this conversion. We show that the corresponding labelled transition systems are bisimilar. In an ongoing example reconfigurable Petri nets are used to model and to verify partial dynamic reconfiguration of field programmable gate arrays.

Keywords: reconfigurable Petri nets, rewrite logic, Maude, model checking, field programmable gate array, dynamic partial reconfiguration

1 Introduction

Reconfigurable Petri nets – a family of formal modelling techniques – provide a powerful and intuitive formalism to model complex coordination and structural adaptation at run-time (e.g. mobile ad-hoc networks, communication spaces, ubiquitous computing). Their characteristic feature is the possibility to discriminate between different levels of change.

Model checking of reconfigurable Petri nets can be achieved by converting reconfigurable Petri nets into Maude specifications. Our main purpose is to guarantee that the reconfigurable Petri net and its conversion to a Maude specification are similar enough for the verification process to obtain valid results. The main theoretical contribution ensures the correctness of the conversion in terms of a bisimulation between the state space of a reconfigurable Petri net and the state space of corresponding Maude modules, see also [20]. Here, we give the main ideas and a sketch of the involved proofs. In general, a bisimulation relates state transition systems, that behave in the same way in the sense that one system simulates the other and vice versa. This is achieved by a bisimulation between the reachability graph for the reconfigurable Petri net and the search tree of the corresponding Maude modules. We have defined functions that convert syntactically all parts of the net such as places, transitions, the arcs or markings as well
as the set of net rules into the Maude modules. These functions are the basis for defining the relation between the corresponding labelled transition systems.

There are many proposals to use Petri nets to model the control logic for field programmable gate arrays (FPGAs). But they lack a formal foundation for the net’s reconfiguration which models the partial dynamic reconfiguration of the FPGA (see Sect. 6). This gap can be closed using Petri nets to model FPGAs and using net transformations to model the dynamic reconfiguration of FPGAs. So, the paper’s motivation and its ongoing example is to model dynamic partial reconfiguration of FPGA using reconfigurable Petri nets.

The paper is organized as follows: The next section deals with reconfigurable Petri nets and Sect. 3 illustrates our example. Sect. 4 introduces the rewriting logic Maude. Model checking reconfigurable Petri nets with Maude is given in Sect. 4. Correctness of the model checking approach is shown in Sect. 5. Finally we discuss related work in Sect. 6 and give some ideas concerning future work.

2 Reconfigurable Petri Nets

We use the algebraic approach to Petri nets, where the pre- and post-domain functions \( \text{pre}, \text{post} : T \rightarrow P^\oplus \) map the transitions \( T \) to a multiset of places \( P^\oplus \) given by the set of all linear sums over the set \( P \). A marking is given by \( m \in P^\oplus \) with \( m = \sum_{p \in P} k_p \cdot p \). The multiplicity of a single place \( p \) is given by \( (\sum_{p \in P} k_p \cdot p)_p = k_p \). The \( \leq \) operator can be extended to linear sums: For \( m_1, m_2 \in P^\oplus \) with \( m_1 = \sum_{p \in P} k_p \cdot p \) and \( m_2 = \sum_{p \in P} l_p \cdot p \) we have \( m_1 \leq m_2 \) if and only if \( k_p \leq l_p \) for all \( p \in P \). The operations \( \oplus \) and \( \ominus \) can be extended accordingly.

Definition 1 (Algebraic approach to Petri nets). A (marked) Petri net is given by \( N = (P, T, \text{pre}, \text{post}, \text{cap}, \text{name}, t, m) \) where \( P \) is a set of places, \( T \) is a set of transitions, \( \text{pre} : T \rightarrow P^\oplus \) maps a transition to its pre-domain and \( \text{post} : T \rightarrow P^\oplus \) maps it to its post-domain. Moreover \( \text{cap} : P \rightarrow \mathbb{N}_+ \), assigns to each place a capacity (either a natural number or infinity \( \omega \)), \( \text{name} : T \rightarrow A_P \) is a label function mapping places to a name space, \( \text{name} : T \rightarrow A_T \) is a label function mapping transitions to a name space and \( m \in P^\oplus \) is the marking denoted by a multiset of places.

A transition \( t \in T \) is \( m \)-enabled for a marking \( m \in P^\oplus \) if we have \( \text{pre}(t) \leq m \) and \( \forall p \in P : (m + \text{post}(t))_p \leq \text{cap}(p) \). The follower marking \( m' \) - computed by \( m' = m - \text{pre}(t) + \text{post}(t) \) - is the result of a firing step \( m[t]m' \).

If the marking is the focus we denote a net with its marking by \((N, m)\).

Net morphisms are given as a pair of mappings for the places and the transitions preserving the structure, the decoration and the marking. Given two Petri nets \( N_i = (P_i, T_i, \text{pre}_i, \text{post}_i, \text{cap}_i, \text{name}_i, t_i, m_i) \) for \( i \in \{1, 2\} \) a net morphism \( f : N_1 \rightarrow N_2 \) is given by \( f = (f_P : P_1 \rightarrow P_2, f_T : T_1 \rightarrow T_2) \), so that \( \text{pre}_2 \circ f_T = f_P^\ominus \circ \text{pre}_1 \) and \( \text{post}_2 \circ f_T = f_P^\oplus \circ \text{post}_1 \) and \( m_1(p) \leq m_2(f_P(p)) \) for all \( p \in P_1 \). Moreover, the morphism \( f \) is called strict if both \( f_P \) and \( f_T \)
are injective and \( m_1(p) = m_2(f_P(p)) \) holds for all \( p \in P_1 \). A rule in the algebraic transformation approach is given by three nets called left-hand side \( L \), interface \( K \) and right-hand side \( R \), respectively, and a span of two strict net morphisms \( K \rightarrow L \) and \( K \rightarrow R \). Then an occurrence morphism \( o : L \rightarrow N \) is required that identifies the relevant parts of the left hand side in the given net \( N \).

A transformation step \( N \xrightarrow{(r, o)} M \) via rule \( r \) (see the commutative squares (1) and (2) in Fig. 1) can be constructed in two steps. Given a rule with an occurrence \( o : L \rightarrow N \) the gluing condition has to be satisfied in order to apply a rule at a given occurrence. Its satisfaction requires that the deletion of a place implies the deletion of the adjacent transitions, and that the deleted place’s marking does not contain more tokens than the corresponding place in \( L \).

Reconfigurable Petri nets exhibit dynamic behaviour using the token game of Petri nets and using net transformations by applying rules. So, a reconfigurable Petri net as in Def. 2 combines a net with a set of rules that modify the net [11,10].

**Definition 2 (Reconfigurable Petri nets).** A reconfigurable Petri net \( RN = (N, R) \) is given by a Petri net \( N \) and a set of rules \( R \).

The labelled transition system for a reconfigurable Petri net \( \text{LTS}_{\text{RPN}} \) is based on the isomorphism classes of nets, where all reachable states are considered up to isomorphisms of marked nets. Isomorphisms are given by bijective mappings of places and transitions. The corresponding isomorphism classes are compatible with firing and transformation steps.

**Definition 3 (Labelled transition system for reconfigurable Petri nets).** Given a reconfigurable Petri net \( (N_0, R) \) the labelled transition system \( \text{LTS}_{\text{RPN}} = (S_{\text{RPN}}, A_{\text{RPN}}, tr_{\text{RPN}}) \) is based on the isomorphism classes of nets:

1. **Initial states:** \( [(N_0, m_0)] \in S_{\text{RPN}} \) where \( m_0 \) is the marking of \( N_0 \) and \( [(N_0, m_0)] = \{(N, m) \mid (N, m) \cong (N_0, m_0)\} \) is the isomorphism class containing \( (N_0, m_0) \).
2. **Firing steps:** For \( m(t)m' \) in \( N \) with \( (N, m) \in [(N, m)] \in S_{\text{RPN}} \) we have:
   \( [(N, m')] \in S_{\text{RPN}} \), \( t_{\text{name}}(t) \in A_{\text{RPN}} \) and \( [(N, m)] \xrightarrow{t_{\text{name}}(t)} [(N, m')] \in tr_{\text{RPN}} \)
3. **Transformation steps:** For \( (N, m) \xrightarrow{(r, o)} (N', m') \) with some rule \( r = (r_{\text{name}}, L \leftarrow K \rightarrow R) \in R \) and some occurrence \( o : L \rightarrow N \) with \( (N, m) \in [(N, m)] \in S_{\text{RPN}} \) we have:
   \( [(N', m')] \in S_{\text{RPN}} \), \( r_{\text{name}} \in A_{\text{RPN}} \) and \( [(N, m)] \xrightarrow{r_{\text{name}}} [(N', m')] \in tr_{\text{RPN}} \)
4. **Finally:** \( S_{\text{RPN}}, A_{\text{RPN}}, tr_{\text{RPN}} \) are the smallest sets satisfying the above conditions.

### 3 Modelling Partial Dynamic Reconfiguration of FPGAs

In this section we give a small example for the use of reconfigurable Petri nets for modelling the control logic for FPGAs. The example is an extension of the
example in [6] where Petri nets also have been used for the modelling, but the reconfiguration has been modelled merely informally. In Fig. 2 an industrial mixer of two components and water is illustrated. The water is heated before being fed into the mixer. The sensors $x_1, \ldots, x_8$ measure the corresponding fill level, $x_9$ measures the water temperature and the actuators $y_1, \ldots, y_9$ control the valves, the heater and the mixer. The control logic has to determine in which order the components are fed into the mixer. Reconfiguration changes the control logic accordingly. In Fig. 3 the Petri net $N_{\text{mixer}}$ describes the control logic, where both components and the hot water are added at the same time. This control logic can be translated into the binary coding of an FPGA (see e.g. [6]).

Dynamic partial reconfiguration of FPGAs allows changing the control logic, so that a different order for the components and the water can be employed. In Fig. 4 we present the rule $\text{rule}_1: L \rightarrow R$ where the interface $K$ is indicated by node $\text{sin} L$ and $R$ with the same colour. It reconfigures the net by replacing the subnet that models the simultaneous feeding of both components and the water, by a subnet that models the sequential feeding, namely first the heated water, then both components. A second rule (omitted here) reconfigures the net by replacing the subnet that models the simultaneous feeding of both components and the water by a subnet that models the sequential feeding, namely first component 1 and the water, and at last component 2. There are more rules as well as their inverse rules modelling the dynamic partial reconfiguration of the FPGA that implement the control logic of the mixer.

The interaction of the system's control logic with the dynamic reconfiguration becomes much more complicated. Hence verification is of high importance. In this paper, we propose model checking of reconfigurable Petri nets by translating the net and its rules to a Maude specification that subsequently can be used for model checking. In this example the absence of deadlocks, a property like „the initial state can be reached again“ or a property like „the mixer never starts unless it is filled“ are sensible requirements. In Sect. 4 we verify the absence of deadlocks using Maude.
4 Model Checking Reconfigurable Petri Nets with Maude

Maude is a high-level language supporting both equational and rewriting logic computation. As a base, it uses a powerful algebraic language for models of a concurrent state system. Its internal representation is given in [15] as a labelled rewrite theory. Implementations in Maude are based on one or many modules. Each module has types that are declared with the keyword „sorts“. Subsequently we introduce Maude using a module describing a Petri net. Definitions of P/T nets, coloured Petri nets, and algebraic Petri nets are defined in [22] in a manner that makes Maude a suitable basis for the definition of a Maude net that models the net and rules of a reconfigurable Petri net. So, the types for a Petri net are given line 1 of Listing 1.1. Depending on a given set of sorts, the operators can be defined. The operators describe all functions needed to work with the defined types. For example, a multiset of markings can be expressed with a whitespace-functor. Place-holders, denoted by a underscore, are used for the types behind the colon, and the return type is given by the type to the right of the arrow, see line 3 of Listing 1.1. Equational attributes declare structural axioms. An operator being associative or commutative is denoted by with keywords such as „assoc“ and „comm“. These keywords are given at the end of line 3 for the multiset of places. The axioms are the equation logic of Maude that defines the operator’s behavior. For example, the initial marking of a Petri net can be exemplified with the initial operator. The operators in line 3-5 describe the markings of the net in Fig. 5 and the equation in line 6 states the initial marking. The rewrite rule in line 8 describes the firing of transition T of the Petri net of Fig. 5. The rewrite rules replaces one multiset by another one, namely the pre-domain of T with its post-domain. As usual in a functional language, all the terms are immutable so that a rule can replace the term A with the term B, see line 8 of Listing 1.1. The rewrite rule implements the token game of Petri nets, where the rewriting of the multiset A by B in rule T can be seen as the firing of transition T. This is just a basic example. The firing in our conversion has been formulated according to the algebraic definition of Petri nets using operations of the multisets over the pre- and post-domain of a transition (see the conditional rewrite rule crl [fire] in Listing 1.3).

Maude’s linear temporal logic for rewrite (LTLR) module can be used to test defined modules with LTL properties, such as deadlocks [3,12,13]. A first step to model checking of reconfigurable Petri nets using Maude has been given in [17]. The conversion of a net and a set of rules into a Maude modules used
for (LTL) model checking with the module LTLR can be found in detail in [19]. The LTLR model-checking module contains all the usual operators, such as true, false, conjunction, disjunction and negation, and complex operators with the next-operator being written with $O \phi$ or the until-operator notated with $\psi U \phi$. Further, it supports release-operator statements, such as $\psi R \phi$ that are internally converted into $\neg (\neg \phi U \neg \psi)$. The future-operator written as $\diamond \phi$ states that $\phi$ is possible in the future, and the global-operator written as $\square \phi$ claims that $\phi$ is true in all states. The correctness of the LTLR model checker has been proven in [2]. In this section we sketch this approach to model checking reconfigurable Petri nets. The ReConNet Model Checker (rMC) (see [19]) defines Maude modules Net and Rules for a given reconfigurable Petri net. The modules contain the net and a set of rules as well as all mechanisms to fire a transition

```plaintext
mod NET is
  including PROP .
  including MODEL-CHECKER .
  ops initial : -> Configuration .
  eq initial =
    net(
      places{ p("y9" | 1517 | w) , p("y5" | 1518 | w) , p("y9" | 1519 | w) ,
             p("y7" | 1520 | w) , p("y1" | 1513 | w) ,
             p("y8" | 1514 | w) , p("y3" | 1515 | w) , p("y4" | 1516 | w) ,
             p("y2" | 1512 | w) } ,
      transitions{ t("x2+x4+x6" | 1599) : t("x6+x9" | 1526) : t
                    ("x1" | 1525) : t("x7" | 1527) : t("x8" | 1522) : t("x5" | 1521) : t("x3" | 1524) } ,
      pre{ (t("x2+x4+x6" | 1599) --> p("y5" | 1518 |w) , p("y4" | 1516 | w) ,
          p("y7" | 1520 | w)) , (t("x6+x9" | 1526) --> p("y9" | 1519 | w)) ,
          (t("x1" | 1525) --> p("y1" | 1513 | w)) ,
          (t("x7" | 1527) --> p("y8" | 1514 | w)) ,
          (t("x8" | 1522) --> p("y9" | 1517 | w)) ,
          (t("x5" | 1521) --> p("y3" | 1515 | w)) ,
          (t("x3" | 1524) --> p("y2" | 1512 | w)) } ,
      post{ (t("x2+x4+x6" | 1599) --> p("y8" | 1514 | w)) ,
              (t("x6+x9" | 1526) --> p("y7" | 1520 | w)) ,
              (t("x1" | 1525) --> p("y4" | 1516 | w)) ,
              (t("x7" | 1527) --> p("y9" | 1517 | w)) ,
              (t("x8" | 1522) --> p("y3" | 1515 | w)) ,
              (t("y2" | 1512 | w)) ,
              (t("y1" | 1513 | w)) ,
              (t("y3" | 1515 | w)) ,
              (t("y2" | 1512 | w)) } ,
      marking{ p("y1" | 1513 | w) ; p("y3" | 1515 | w) ; p("y2" | 1512 | w) }
    )
endm
```

Listing 1.2: Maude conversion of $N_{mixer}$ from Fig. 3
or to transform the net with a rule [19,20]. Together with the Maude modules RPN defining the firing behaviour, the Maude module PROP stating the properties to be verified and the Maude model checker LTLR-MODEL-CHECKER, these modules yield a rewrite theory that allows the verification of the linear temporal logic formulas over the properties implemented in module PROP. Listing 1.2 shows the net in Fig. 3 converted into the Maude module NET. Each net is modelled by the multisets of places and transitions. A place is defined as \( p(<\text{label}|<\text{identifier}|<\text{capacity}>). \) Transitions only consist of \( t(<\text{label}|<\text{identifier}>). \) The pre- and post-domains are wrapped to a set by the \( \text{pre- or post-} \) operator. Finally, the initial marking is modelled as the multiset in Listing 1.2, line 13. The conditional rewriting rule for firing \( \text{crl [fire]} \) in Listing 1.3, line 1 uses the transition's pre-domain to determine if a transition is enabled and considers the capacity of each place in its post-domain.

```
1  \text{crl [fire]} :
2    \text{net (P, Transitions } \{ T : TRest \},
3    \text{pre} \{ (T --> PreValue) , MTupleRest1 \},
4    \text{post} \{ (T --> PostValue) , MTupleRest2 \},
5    \text{marking} \{ \text{PreValue ; M} \})
6     \Rightarrow
7     \text{net (P, Transitions } \{ T : TRest \},
8    \text{pre} \{ (T --> PreValue) , MTupleRest1 \},
9    \text{post} \{ (T --> PostValue) , MTupleRest2 \},
10   \text{calc} ( (\text{PreValue ; M} \text{ minus PreValue})
11     \text{plus PostValue}))
12     \ldots
13     \text{if calc} ( (\text{PreValue ; M} \text{ plus PostValue}) <=? \text{PostValue})
```

Listing 1.3: Firing as a rewrite rule in Maude module RPN

Listing 1.3 shows the firing of a transition, where each pre-domain condition is implemented in line 6. The subterm \( \text{PreValue ; M} \) ensures that at least the pre-domain of the transition is part of the current marking. The if condition in line 16 in Listing 1.2 ensures the capacities using an operator \( \leq ? \) (called smallerAsCap) defined in the Maude module RPN.

```
1  \text{crl [rule1-PNML] :}
2    \text{net (}
3    \text{places} \{ p(“y5” | Irule1044 | w) , p(“y8” | Irule101298 | w) , p(“y4” | Irule1047 | w) , p(“y7” | Irule1041 | w) , PRest } ,
4    \text{transitions} \{ t(“x2+x4+x6” | Irule1060) : TRest } ,
5    \text{pref} \{ (t(“x2+x4+x6” | Irule1060) --> p(“y7” | Irule1041 | w) , p(“y4” | Irule1047 | w) , p(“y5” | Irule1044 | w)) , MTupleRest1 } ,
```
Listing 1.4: Rule rule1 in Maude module RULES

The implementation of net rules as given in Fact 2 is illustrated in Listing 1.4. The rule application coincides with the pattern-matching algorithm of Maude, which ensures that the left-hand side is a subset of the current net state. If the conditions are successfully proven, the term describing the net is rewritten. Fact. 1 states that the conditions freeOfMarking and emptyNeighbourForPlace ensure the satisfaction of the gluing condition (see page 3) in the current net. For more details see [20].
Fact 1 (Gluing condition for rewrite rules). Given a rule application with \( r = (r_{\text{name}}, L \leftarrow K \rightarrow R) \) in a reconfigurable Petri net that satisfies the gluing condition. Then we have in the module `RULES`:

- `emptyNeighbourForPlace` ensures that \( p \) is not used in `pre(MTupleRest1)` or `post(MTupleRest2)`. So, a place \( p \) may be deleted only if there are no adjacent transitions that are not deleted.
- `freeOfMarking` ensures that for each deleted place \( p \not\in M_{\text{rest}} \) holds.

Fact 2 (Transformation step in the Maude module `RULES`). For each transformation step \( m \overset{(r,o)}{\Rightarrow} m' \) with \( r = (r_{\text{name}}, L \leftarrow K \rightarrow R) \) in a reconfigurable Petri net, there exists a rewriting rule in `RULES` so that

- there is a pattern match of the left-hand side ensuring that the left-hand side is a subset of the current net state
- the match satisfies the gluing condition of Fact 1.

The implementation of this conversion is given by the ReConNet Model Checker (rMC) [19]. rMC is a Java-based tool that enables a user to convert a given reconfigurable Petri net\(^1\) to the Maude modules introduced above. These Maude modules can be executed and analysed by the Maude interpreter. For the example in Sect. 3 the absence of deadlocks is expressed in Maude’s notation using the LTL operators „finally“ \(<>\) and „globally“ \([\square]\). The property `enabled` given in the Maude module `PROP` and states that at least the preconditions of one of the transitions or of one of the rules are met. So, the formula \([\square]<> \text{enabled} \) asserts that the property enabled is globally finally true. The state `initial` corresponds to the reconfigurable Petri net \((N,R)\). Hence, `modelCheck(initial, [\square]<> \text{enabled})` is the operation that checks the formula \([\square]<> \text{enabled} \) for the state `initial`. So, in Listing 1.5 we check that the reconfigurable Petri net \((N_{\text{mixer}}, R)\) is always enabled, i.e. there are no deadlocks.

```
1 rewrite in NET : modelCheck(initial, [\square]<> \text{enabled}) .
2 rewrites: 17601 in 25ms cpu (48ms real) (704040 rewrites/second)
3 result Bool: true
```

Listing 1.5: Absence of deadlocks for \((N_{\text{mixer}}, R)\); proven by Maude

A labelled transition system for the Maude module `NET` is defined by \( LTS_{MNC} = (S_{MNC}, A_{MNC}, tr_{MNC}) \), where \( S_{MNC} \) is a non-empty set that contains all states of a Maude breadth-first search tree. Maude’s deduction rules are used to execute all rewrite rules, such as firing or transformation steps in the `RULES` module. A

\(^1\) ReCONNet(see [16]) is the tool for modelling and simulating reconfigurable Petri nets saving them as an extension of PNML.
state $s \in S_{MNC}$ consists of a Net term as current state. $A_{MNC}$ is defined by $A_{MNC} = A_T \cup A_R$ and contains the labels of rewrite rules, such as the firing or the transformation. $tr_{MNC}$ is defined as a set of transition relations that is based on $tr_{MNC} \subseteq S_{MNC} \times A_{MNC} \times S_{MNC}$. Therefore, two terms of $S_{MNC}$ are related by a transition labelled with the name of the corresponding rewrite rule in $A_{MNC}$.

Definition 4 (Labelled transition system for the Maude module NET). Given the Maude module NET, a labelled transition system $LTS_{MNC} = (S_{MNC}, A_{MNC}, tr_{MNC})$ is defined with respect to the term sets over the equation conditions of the Maude modules by:

1. **Initial**: initial $\in S_{MNC}$
2. **Firing steps**: If $s \in S_{MNC}$ and $s \to s'$ is a replacement for a rewrite rule $[\text{fire}]$ of Listing 1.3 so that

   \[
   s = \text{net}(P, \\
   \text{transitions}\{t(\text{label}|\text{identifier}) : TRest\}, \\
   \text{pre}\{t(\text{label}|\text{identifier}) - > \text{PreValue}, MTupleRest1\}, \\
   \text{post}\{t(\text{label}|\text{identifier}) - > \text{PostValue}, MTupleRest2\}, \\
   \text{marking}\{\text{PreValue}; M\}
   \]

   is used as left-hand side of Listing 1.3, then $s' \in S_{MNC}$, $t(\text{label}) \in A_{MNC}$ and $s \xrightarrow{t(\text{label})} s' \in tr_{MNC}$
3. **Transformation steps**: If $s \in S_{MNC}$ and $s \to s'$ is a replacement for a rewrite rule $[r_{name}]$ in the Maude module RULE and

   \[
   s = \text{net}(\text{places} \{ P_L, PRest \}, \\
   \text{transitions}\{T_L : TRest\}, \\
   \text{pre}\{Pre_L, MTupleRest1\}, \\
   \text{post}\{Post_L, MTupleRest2\}, \\
   \text{marking}\{M_L; M\}
   \]

   then:

   \[
   s' \in S_{MNC}, r \in A_{RPN} \text{ and } s \xrightarrow{r_{name}} s' \in tr_{MNC}
   \]
4. **Finally**: $S_{MNC}, A_{MNC}, tr_{MNC}$ are the smallest sets satisfying the above conditions.

## 5 Correctness of Model Checking Approach

Model checking reconfigurable Petri nets using Maude is shown to be correct by proving a bisimulation between the corresponding labelled transition systems (LTS). Theorem 1 states a conversion of a reconfigurable Petri net $(N, R)$ into the corresponding Maude module NET. Then the LTS are calculated for both and in Theorem 2 these LTS are shown to be bisimilar. The $LTS_{RPN}$ is the reachability graph (up to isomorphism) of the reconfigurable Petri nets and is
given by all reachable states using both firing and transformation steps (see Def. 3). The states are the isomorphism classes of marked Petri nets. \( L_{TSMNC} \) of a Maude module \( \text{NET} \) (see Def. 4) includes all rewriting rule applications of firing and transforming steps. Theorem 1 specifies the syntactical conversion for a given reconfigurable Petri net to the Maude modules. For the theorem itself, the following injective functions are used to convert all parts of a reconfigurable Petri net into a \( \text{NET} \)- and a \( \text{RULES} \)-module: \( \text{buildPlace} \) defines the conversion for places (see Lemma 1). The functions \( \text{buildTransition} \) (defining the conversion for transitions similar to \( \text{buildPlace} \)), \( \text{buildPre} \) (defining the conversion for each \( \text{pre}(t) \) with \( t \in T^\oplus \)), \( \text{buildPost} \) (defining the conversion for each \( \text{post}(t) \) with \( t \in T^\oplus \) similar to \( \text{buildPre} \)), \( \text{buildNet} \) (defining the conversion of a net) and \( \text{buildRule} \) (defining the conversion of rules in \( \mathcal{R} \)) are constructed inductively as well, and can be found in [20]. Lemma 1 contains functions for the mapping of identifiers and capacities leading to conversion of places and transitions. The identifiers are defined as unique keys for nodes such as places or transitions and are used by the pre- and post-domain operations. The conversion of places is defined in Lemma 1. Each new place \( p' \) is converted to the Maude term \( p(<\text{label}>, <\text{identifier}>, <\text{capacity}>) \) using the identifier function and the place operator \( p \).

**Lemma 1 (buildPlace).** Let \( N = (P,T,\text{pre},\text{post},\text{cap},\text{name},\text{tname},m) \) be a Petri net together with an injective identity function \( id_P : P \rightarrow \mathbb{N}^+ \), then there is an injective function \( \text{buildPlace} : P^\oplus \rightarrow \mathcal{T}_{\text{Places}} \).

**Proof sketch:** \( \text{buildPlace} \) is defined inductively over \( |P| \) by:

- for \( P = \emptyset \), \( P^\oplus = \{0\} \) and \( \text{buildPlace}(0) = \text{emptyPlace} \)
- for \( P' = P \cup \{p'\} \) there is a \( \text{buildPlace}' : P' \rightarrow \mathcal{T}_{\text{Places}} \) defined by

\[
\text{buildPlace}'(s) = \text{buildPlace}(s) \text{ if } s \in P^\oplus \text{ and } \text{buildPlace}'(s) = \text{buildPlace}(s'), \text{p}(\text{name}(p'_1)|\text{id}_P(p'_1)|\text{cap}(p'_1)),...,,\text{p}(\text{name}(p'_k)|\text{id}_P(p'_k)|\text{cap}(p'_k))
\]

with \( p'_i = p' \) for \( 1 \leq i \leq k \) and \( s = s' + k \cdot p' \), \( k \geq 1 \) and \( s' \in P^\oplus \)

\( \text{buildPlace} \) is injective since \( id_P \) is injective and its inverse function \( \text{buildPlace}^{-1} \) is defined accordingly [20].

Next, Theorem 1 states the conversion of one given reconfigurable Petri net into the Maude modules \( \text{NET} \) and \( \text{RULES} \). The first part of the proof states that the functions \( \text{buildNet} \) and \( \text{buildRule} \) correspond to the initial state in Maude’s term algebra. The Maude module \( \text{NET} \) comprises the \( \text{net} \)-operator, a set of rules defined by the \( \text{rule} \)-operator and some metadata. This metadata is used for example to ensure efficient use of identifiers. Moreover, the theorem states that the module \( \text{RULES} \) comprises rewrite rules for each rule in \( \mathcal{R} \).

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

**Proof sketch:** Using \( \text{buildPlace} \) (see Lemma 1) \( \text{buildTransition}, \text{buildPre}, \text{buildPost} \) and \( \text{buildRule} \) (see [20]) each reconfigurable Petri net \( (N,\mathcal{R}) \) with \( N = \)}
\((P, T, \text{pre}, \text{post}, \text{cap}, \text{name}, t_{\text{name}}, m)\) and \(R = \{(r_{\text{name}}, L_i \leftarrow K_i \rightarrow R_i) \mid 1 \leq i \leq n\}\) yields the well-formed Maude module \(\text{NET}\):

\[
\begin{align*}
\text{eq initial} &= \text{buildNet}(N) \\
&\quad \text{buildRule}(R) \\
&\quad \text{metadata}
\end{align*}
\]

Additionally, we have the Maude module \(\text{RULES}\) with the rewrite rules (see Fact 2), so that for each rule \(r \in R\) with \(r = (r_{\text{name}}, L_i \leftarrow K_i \rightarrow R_i)\) there is:

\[
\begin{align*}
\text{crl [r_{\text{name}}]} : \text{net( places}\{\text{buildPlaces}(P_{Li}), \text{PRest}\}, \\
&\quad \text{transitions}\{\text{buildTransition}(T_{Li}) : \text{TRest}\}, \\
&\quad \text{pre}\{\text{buildPre}(T_{Li}), \text{MTupleRest1}\}, \\
&\quad \text{post}\{\text{buildPost}(T_{Li}), \text{MTupleRest2}\}, \\
&\quad \text{marking}\{\text{buildPlaces}(M_{Li}); \text{MRest}\}) \\
&\quad \text{buildRule}(R) \\
&\quad \text{metadata} \\
&\quad = \to \\
&\quad \text{net( places}\{\text{buildPlaces}(P_{Ri}), \text{PRest}\}, \\
&\quad \text{transitions}\{\text{buildTransition}(T_{Ri}) : \text{TRest}\}, \\
&\quad \text{pre}\{\text{buildPre}(T_{Ri}), \text{MTupleRest1}\}, \\
&\quad \text{post}\{\text{buildPost}(T_{Ri}), \text{MTupleRest2}\}, \\
&\quad \text{marking}\{\text{buildPlaces}(M_{Ri}); \text{MRest}\}) \\
&\quad \text{buildRule}(R) \\
&\quad \text{new metadata}
\end{align*}
\]

\begin{align*}
&\text{if *** for deleted places} \\
&\quad \text{freeOfMarking}(\forall p \in P_{Li} \mid \text{MRest}) \land \\
&\text{*** for places of deleted transitions} \\
&\quad \text{emptyNeighbourForPlace}(\forall p \in P_{Li} \setminus P_{Ri} \mid \\
&\quad \text{pre}\{\text{MTupleRest1}\} \mid \text{post}\{\text{MTupleRest2}\}) \land \\
&\quad \text{calculate new metadata}.
\end{align*}

Listing 1.2 and Listing 1.4 provide examples for both modules. Labelled transition systems are defined for reconfigurable Petri nets by \(\text{LTS}_{\text{RPN}}\) and for the corresponding Maude module \(\text{NET}\) by \(\text{LTS}_{\text{MNC}}\). Both labelled transition systems are related by a surjective function \(\text{map}\) defined in Lemma 2. To distinguish the states of the respective labelled transition systems, the variables \(s\) for state in \(\text{LTS}_{\text{MNC}}\) and \(r\) for state in \(\text{LTS}_{\text{RPN}}\) are used. \(\text{map}\) relates a state \(s \in \text{LTS}_{\text{MNC}}\) to a state \(r \in \text{LTS}_{\text{RPN}}\). Note, the function \(\text{map}\) in Lemma 2 is not injective due to the isomorphism classes in Def. 3.
Lemma 2 (Surjective mapping of $LTS_{MNC}$ to $LTS_{RPN}$). Given a reconfigurable Petri net $(N_0, R)$ with $N_0 = (P_0, T_0, \text{pre}_0, \text{post}_0, \text{pname}_0, \text{tname}_0, \text{cap}_0, m_0)$ and the set of rules $R$ together with the corresponding Maude modules $\text{NET}$ and $\text{RULE}$ as in Theorem 1. Then there is the surjective mapping $\text{map} : S_{MNC} \rightarrow S_{RPN}$ from the labelled transition system $LTS_{MNC}$ in Def. 4 to the labelled transition system $LTS_{RPN}$ in Def. 3 with $s = \text{net} (\text{Places}, \text{Transitions}, \text{Pre}, \text{Post}, \text{Markings}) \mid \text{Rule Int Int}$

$IDPool$ by $\text{map}$ and $\text{NET}$ and $R$ and the set of rules $\text{urable Petri net}$ $(\text{LTS})$

Lemma 2 (Surjective mapping of $LTS_{MNC}$ to $LTS_{RPN}$). Given a reconfigurable Petri net $(N_0, R)$ with $N_0 = (P_0, T_0, \text{pre}_0, \text{post}_0, \text{pname}_0, \text{tname}_0, \text{cap}_0, m_0)$ and the set of rules $R$ together with the corresponding Maude modules $\text{NET}$ and $\text{RULE}$ as in Theorem 1. Then there is the surjective mapping $\text{map} : S_{MNC} \rightarrow S_{RPN}$ from the labelled transition system $LTS_{MNC}$ in Def. 4 to the labelled transition system $LTS_{RPN}$ in Def. 3 with $s = \text{net} (\text{Places}, \text{Transitions}, \text{Pre}, \text{Post}, \text{Markings}) \mid \text{Rule Int Int}$

$IDPool$ by $\text{map}$ and $\text{NET}$ and $R$ and the set of rules $\text{urable Petri net}$ $(\text{LTS})$

Lemma 2 (Surjective mapping of $LTS_{MNC}$ to $LTS_{RPN}$). Given a reconfigurable Petri net $(N_0, R)$ with $N_0 = (P_0, T_0, \text{pre}_0, \text{post}_0, \text{pname}_0, \text{tname}_0, \text{cap}_0, m_0)$ and the set of rules $R$ together with the corresponding Maude modules $\text{NET}$ and $\text{RULE}$ as in Theorem 1. Then there is the surjective mapping $\text{map} : S_{MNC} \rightarrow S_{RPN}$ from the labelled transition system $LTS_{MNC}$ in Def. 4 to the labelled transition system $LTS_{RPN}$ in Def. 3 with $s = \text{net} (\text{Places}, \text{Transitions}, \text{Pre}, \text{Post}, \text{Markings}) \mid \text{Rule Int Int}$

$IDPool$ by $\text{map}$ and $\text{NET}$ and $R$ and the set of rules $\text{urable Petri net}$ $(\text{LTS})$
– Analogously, for each follower state \( r_{n+1} \in S_{RPN} \) with \( r_n \xrightarrow{l} r_{n+1} \in tr_{RPN} \) there is a \( s_{n+1} \in S_{MNC} \) with \( s_n \xrightarrow{l} s_{n+1} \in tr_{MNC} \) and \( \text{map}(s_{n+1}) = r_{n+1} \).

The bisimulation between \( LTS_{RPN} \) and \( LTS_{MNC} \) is defined by the function \( \text{map} \).

Theorem 2 states the behavioural equivalence of both transition systems. For each pair \( (s_n, r_n) \in \text{map} \) with \( n \geq 0 \) all outgoing actions are the same. The proof ensures that the reachable states \( s_{n+1} \) and \( r_{n+1} \) are again related by \( \text{map} \).

**Theorem 2 (Bisimulation).** \( LTS_{RPN} \) and \( LTS_{MNC} \) are bisimilar.

**Proof sketch:** Given \( s \in S_{MNC} \) and \( r \in S_{RPN} \) with \( \text{map}(s) = r = [N, m] \), we have:

– \( s \rightarrow s' \): There is \( \text{map}(s) = r \) and for each \( a \in A_{MNC} \) we have \( r \xrightarrow{a} r' \in tr_{RPN} \) because \( s \xrightarrow{a} s' \in tr_{MNC} \) so that \( \text{map}(s') = r' \), since \( \text{map} \) is well-defined (see Lemma 2).

– \( r \rightarrow r' \): There is \( \text{map}(s) = r \) and for each \( a \in A_{MNC} \) we have \( s \xrightarrow{a} s' \in tr_{MNC} \), due to \( r \xrightarrow{a} r' \in tr_{RPN} \) so that \( \text{map}(s') = r' \), since \( \text{map} \) is surjective (see Lemma 2).

**Corollary 1 (LTS properties are preserved).** For any LTL property \( \phi \) we have:

\[ LTS_{RPN} \models \phi \iff LTS_{MNC} \models \phi \]

Due to Theorem 3.1.5 and Theorem 7.6 in [3].

6 Conclusion

Related work concerns the translation of some modelling techniques into Maude. In [22] high-level Petri nets are modelled using Maude and the focus is on the soundness and correctness of the Maude structure. [7] shows a mapping for UML models to a Maude specification, where \( AtoM \) is used to convert the model into a Python-code representation that solves constraints inside the UML model. Closely related to our approach is [4], where Petri nets are converted into several Maude modules. [5] presents a graphical editor for CPNs, which uses Maude in the background to verify LTL properties. Specified Maude modules (similar to [22]) are defined, which contain one-step commands for the simulation. In [18] reference nets (a variant of the net-in-a-net approach) are used to model and decompose embedded systems.

Concerning the application to dynamic reconfigurable field programmable gate arrays (FPGAs) all the following approaches use Petri nets to model FPGAs, but merely have some informal mechanism to model its dynamic reconfiguration. [21] discusses how the FPGA architectures affect the implementation of Petri net specifications and shows how to obtain VHDL descriptions amenable to synthesis. [1] deals with the automatic translation of interpreted generalized Petri Nets with time into VHDL. [8] is concerned with an FPGA-based controller
design to achieve simpler and affordable verification and validation. To model the interactions among processes both of state diagrams and Petri nets are used to model the concurrent processes. In [9] a Petri net variant called hierarchical configurable Petri nets modelling reconfigurable logic controllers are translated into Verilog language to be implemented in FPGAs. [14] proposes an approach analysis and testing of communication tasks of distributed control systems that uses timed colored Petri Nets for the simulation and performance estimation.

Summarizing, we have presented a correct model-checking technique for reconfigurable Petri nets. A first step towards the underlying conversion has been presented in [17]. In [20] these basis concepts have been extended, e.g. including capacities, gluing conditions, garbage collection etc. Here, we have sketched the improved conversion and have shown that it leads to a valid verification technique as the the LTL properties are preserved by this conversion.

In [20] a preliminary evaluation our the approach to model checking reconfigurable Petri nets has been given based on another example. The performance of our approach based on Maude in version 2.7, including LTLR in version 1.0 has been compared to the established tool Charlie version 2.0. A Petri net with the same semantics (the same state space), as the reconfigurable Petri net, has been used as an example. This net is used to perform a comparative analysis, including a transfer into a flat Petri net, where all transformations steps are modelled as part of the net.

Ongoing work is the introduction of control structures, as transformation units, negative application conditions and others into the tool ReConNet. These new features cause the need to adopt the conversion to Maude accordingly.

References


\[ \text{http://maude.cs.illinois.edu/tools/tlr/}, \text{ 11 March 2015} \]
\[ \text{http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Charlie}, \]
\[ \text{11 March 2015} \]


