Bounded Model Checking with Alloy

Transforming elementary Petri nets into Alloy models

Iago Abal and Rui Gonçalo
{pg16305,pg18378}@alunos.uminho.pt

Formal Methods in Software Engineering
Department of Informatics
University of Minho

Abstract. As part of the *Analysis, Modeling and Testing* module of the MEI MSc programme at *University of Minho*, we developed a Haskell application that allows the verification of LTL formulas stated over elementary Petri nets through its transformation into Alloy models. This report describes the most important aspects of the tool's conception and implementation.

1 Overview

This report is organized as follows: First, section 2 describes the process by which Petri nets are read and transformed into an Alloy model; then, section 3 describes how LTL formulas are translated to Alloy and; finally, section 4 concludes with some observations and discuss how this application could be improved.

2 From Petri nets to Alloy models

Ideally, our application reads elementary Petri nets in PNML[1] format. In practice, since PNML standard is not free or charge, we have had to reverse engineer PNML from Petri nets files produced by PIPE[2]. Figure 1 shows the simplified PNML format understood by our application. Of course, more elements or attributes could be present, but they will be completely ignored.

```
<pnml>
  <net id="«net id»" type="P/T net">
    <place id="«place id»">
      <capacity>
        <value>1</value>
      </capacity>
      <initialMarking>
        <value>«initial marking»</value>
      </initialMarking>
    </place>
    ...
    <transition id="«transition id»" />
    ...
    <arc source="«source id»" target="«target id»" />
    ...
  </net>
</pnml>
```

Fig. 1. Model of PNML Petri net accepted by our application.

The application reads such PNML files and checks if the corresponding values for the net *type*, places *capacity* and *initial marking* are the expected ones for an elementary Petri net. If all went fine then the

[1] http://www.pnml.org
[2] http://pipe2.sourceforge.net

Petri net is stored internally using the typical representation of elementary Petri nets, thus: (P, T, F, M_0) where P is the set of places, T the set of transitions, F the flow relation and $M_0 \subseteq P$ the initial marking.

The representation of a Petri net in Alloy is based on the dynamics of Elementary Petri nets and its translation to Kripke structures. A model starts imposing a total ordering over states, so all states in the finite universe created by Alloy will represent a concrete execution path, as we will see below.

```
open util/ordering[State]
```

Places are translated to atoms, which are represented by an abstract signature. Each concrete place introduces a singleton signature declaring the corresponding atom, in this way we can make explicit reference to all of them.

```
abstract sig Atom {}

one sig «place id» extends Atom {}
```

Each state is described by the set of atoms that hold in it. The initial state is characterized by a predicate that compares an state atoms set with the initial one, given by the Petri net initial marking.

```
sig State { atoms : set Atom }

pred I[s:State] {
  s.atoms = «initial atoms»
}
```

To avoid the declaration of all possible states and transitions we follow a symbolic approach based on the dynamics of Elementary Petri nets: A transition is described by the set of tokens it requires to be fired and the set of tokens that it produces when fired. Tokens are translated to atoms and, as for places, each particular transition is declared as a singleton signature that specifies what are its pre/post atoms sets.

```
abstract sig Transition {
  pre  : set Atom,
  post : set Atom
}

one sig «transition id» extends Transition {}
{ pre  = «set of atoms required for execution»
  post = «set of atoms ensured after execution»
}
```

$M \xrightarrow{t}$ and $M \xrightarrow{t} M'$ are literally translated into two predicates describing transition allowance and firing. Since in a Kripke structure the transition relation must be total, we also introduce an additional predicate stating that if no Petri net transition is enabled in a given state, then always exists an implicit transition to itself.

```
pred enabled[t:Transition, s:State] {
  t.pre in s.atoms
  t.post not in (s.atoms - t.pre)
}

pred fired[t:Transition, s,s':State] {
  t.enabled[s]
  s'.atoms = (s.atoms - t.pre) + t.post
}

pred restLoop[s,s':State] {
  no t : Transition | t.enabled[s]
  s.places = s'.places
}
```

Finally, we declare a fact forcing Alloy to generate only valid paths. Note we have avoided to explicitly declare every possible state as we should do if we were strictly follow the rules to transform a Petri Net into a Kripke structure.

```
pred nextState[s,s':State] {
    some t:Transition | t.fired[s,s']
  or restLoop[s,s']
}

fact {
  I[first]
  all s:State, s':s.next | nextState[s,s']
}
```

However, Petri nets are not directly transformed into Alloy code as described above. Instead, there is an intermediate step transforming our net, represented as (P, T, F, M_0), into an abstract representation of the Alloy model: Atoms substitute places and the flow F is removed whereas T is redefined capturing F information by considering transitions as pairs $(Pre, Post)$ of consumed/produced atoms sets. In this way the generation of the Alloy code is done by simply pretty-printing this intermediate representation. Figure 2 shows the described transformation chain.

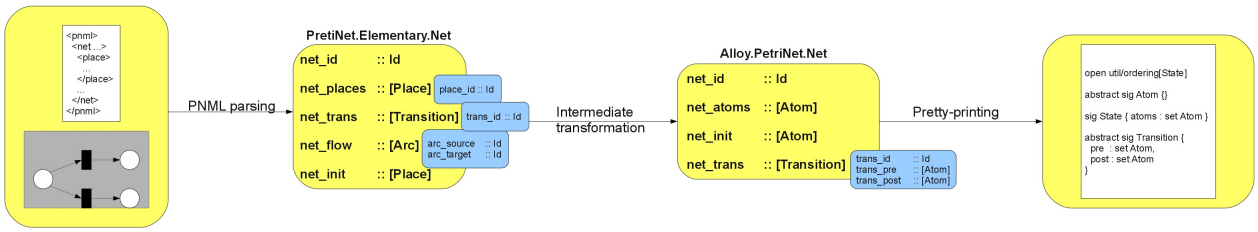

Fig. 2. From Petri nets to Alloy models

As part of the code generation process there is a transformation of the Petri net identifiers in order to avoid some possible conflicts. First, all non alpha-numeric characters are removed; then identifiers are capitalized and; finally, we prefix each identifier by a letter indicating its corresponding namespace: net (n), atom (a) or transition (t).

3 From LTL to Alloy formulas

Table 1 shows the syntax of LTL formulas as well as the ASCII symbols accepted by our parser as representing literals, atoms and LTL operators. Internally, this syntax is captured by an straightforward algebraic data type which is omitted here[3], for a question of clarity. In the following, we will make use of the well-known Unicode symbols used to refer LTL operators instead of the ASCII ones.

Analogously to what we have done for Preti nets, LTL formulas are translated to Alloy through an intermediate representation from which Alloy code is produced by a simple pretty-printing step. This intermediate language is an abstracted subset of Alloy designed to capture the semantics of LTL formulas. The first component of this representation are state expressions, described in Table 2, allowing the navigation through the states of a path. In contrast with the classical presentation of LTL semantics path is not a first-class concept but, instead, a path is referred by its initial state (st_0) and consequently, concrete states or subpaths of a path are not referred by the use of indices (π_i or π^i) but as an expression

[3] See `LTL.Syntax` module for further details.

Table 1. Syntax of LTL formulas.

LTL formula *f, g*	Description	Accepted ASCII symbols
$bool \in \{true, false\}$	Boolean literal	`true`, `false`
$p \in Atoms$	Atoms	$alpha(alpha\|digit\|\_)^*$
$\neg p$	Negation	`not`, $\sim$, `!`
$f \vee g$	Disjunction	`or`, `||`, `\/`
$f \wedge g$	Conjunction	`and`, `&&`, `/\`
$f \Rightarrow g$	Implication	`=>`, `implies`
$f \Leftrightarrow g$	Bi-Implication	`<=>`, `iff`
$\circ f$	Next	`X`
$\Box f$	Globally	`G`
$\Diamond f$	Finally	`F`
$f \mathcal{U} g$	Until	`U`
$f \mathcal{R} g$	Release	`R`

relative to that initial state. This approach is more convenient to work with in Alloy because the concept of path is not being modeled as a sequence of states, and also makes the transformation of state expressions into Alloy code straightforward as showed in the third column of Table 2. Note that `from` and `fromBelow` are very simple utility functions defined in terms of other `util/ordering` functions, used to improve readability and to keep the generation of Alloy code as simple as possible.

```
fun from[s:State] : set State {
  s + s.nexts
}

fun fromBelow[s,t:State] : set State {
  s.from & t.prevs
}
```

Table 2. State expressions.

State expr. *s, t*	Description	Alloy translation
st_0	Path's first state	`first`
$v_k \in \mathcal{V}_{st}$	State variable	`st`k
$next(s)$	Next state of s	s`.next`
$[s..]$	All states from s	t`.from`
$[s..t)$	All states from s but below t	`fromBelow[`s`,`t`]`

State expressions are used by the formula language described in Table 3 to specify concrete states in which some atoms must hold, or the domain of quantifications over a set of states denoting subpaths. As you could appreciate the translation of one of these formulas to Alloy is also simple and clear. Again, we make of use of some simple utilities to provide a definition for `true` and `false` in Alloy; they are unnecessary but their use improve the readability of the generated code. Note that both state expressions and this abstract Alloy subset have corresponding algebraic data types that were omitted, as for LTL syntax, for a question of clarity[4].

```
pred true { no none }

pred false { some none }
```

The transformation of LTL formulas to the intermediate representation described above is shown in Figure 3. The algorithm shown is a literal implementation of LTL semantics, only with two minor tweaks. First, as we said previously, we use state expressions instead of indices to range over the states of the

[4] See `Alloy.LTL` module for further details.

Table 3. Abstract Alloy to describe LTL semantics.

Formula f,g	Description	Alloy translation
$bool \in \{true, false\}$	Boolean literal	`true`, `false`
$all(s, A)$	All atoms in A hold in state s	A `in` s`.atoms`
$some(s, A)$	Exist a in A such it holds in state s	`some (`A `&` s`.atoms)`
$\neg f$	Negation	`not` f
$f \vee g$	Disjunction	f `or` g
$f \wedge g$	Conjunction	f `and` g
$f \Rightarrow g$	Implication	f `implies` g
$f \Leftrightarrow g$	Bi-Implication	f `iff` g
$exists(s)$	Exists state s	`some` s
$\forall v_k : s, f$	All states s verify f	`all st`k `:` s `|` f
$\exists v_k : s, f$	Exist state in s verifying f	`some st`k `:` s `|` f

(s denotes an state expression, A a set of atoms and v_k a state variable.)

path. Second, the translation of the $\circ$ (*next*) operator not simply states that f must hold in the next state of s, but only in the case such next state exists, which is not ensured in bounded model checking since we work with finite paths. Remember that we make reference to a path through its first state so the *State* parameter of $\mathcal{A}$ provides the path in which the given formula has to hold; therefore, the translation of a LTL formula f is obtained by $\mathcal{A}(st_0, f)$.

$$
\begin{aligned}
\mathcal{A} : State \times LTL &\rightarrow Alloy \\
(s, bool) &\rightarrow bool \\
(s, a \in Atoms) &\rightarrow all(s, \{a\}) \\
(s, \neg f) &\rightarrow \neg\mathcal{A}(s, f) \\
(s, f \vee g) &\rightarrow \mathcal{A}(s, f) \vee \mathcal{A}(s, g) \\
(s, f \wedge g) &\rightarrow \mathcal{A}(s, f) \wedge \mathcal{A}(s, g) \\
(s, f \Rightarrow g) &\rightarrow \mathcal{A}(s, f) \Rightarrow \mathcal{A}(s, g) \\
(s, f \Leftrightarrow g) &\rightarrow \mathcal{A}(s, f) \Leftrightarrow \mathcal{A}(s, g) \\
(s, \circ f) &\rightarrow exists(s.next) \Rightarrow \mathcal{A}(s.next, f) \\
(s, \Box f) &\rightarrow \forall t : [s..], \mathcal{A}(t, f) \\
(s, \Diamond f) &\rightarrow \exists t : [s..], \mathcal{A}(t, f) \\
(s, f \mathcal{U} g) &\rightarrow \exists t : [s..], \mathcal{A}(t, g) \wedge \langle \forall u : [s..t), \mathcal{A}(u, f) \rangle \\
(s, g \mathcal{R} f) &\rightarrow \forall t : [s..], \mathcal{A}(t, f) \vee \langle \exists u : [s..t), \mathcal{A}(u, g) \rangle
\end{aligned}
$$

(bool denotes a boolean literal; s, t and u are state expressions; a a concrete atom and v_k a state variable.)

Fig. 3. Transformation of LTL formulas to formulas in an abstract subset of Alloy.

Note that an alternative translation for $\circ$ (*next*) could be $exists(s.next) \wedge \mathcal{A}(s.next, f)$, but we have considered this way too restrictive because it makes impossible to verify several interesting properties like $\Box \circ f$ that always fail in the last state. By other hand, the actual encoding of $\circ$ (*next*) makes senseless to check formulas like $\Diamond \circ f$ because they always will succeed in the last state. We have considered the conditional approach more convenient and natural; we assume that any use of $\Diamond$ (*finally*) will be problematic when you work with finite paths so the behavior of formulas like $\Diamond \circ f$ looks reasonable.

Finally, the Alloy model is complete by filling the following template with the translated LTL formula and the desired path length.

```
assert formula {
  «LTL formula translation»
} check formula for «path length»
```

4 Conclusions and Future work

Alloy was demonstrated to be a good back-end to apply bounded model checking for the verification of LTL formulas, allowing a quite simple encoding of both Petri nets and LTL. By using Alloy we have access to all its facilities and, in particular, one we have found very useful is its ability to provide counterexamples for failed assertions. With respect to explicit model checking, our strategy has the advantage of be symbolic avoiding explicit enumeration of states and transitions, and hence it should be more scalable; this is not an advantage with respect to symbolic model checkers though. Our intuition is that the use of SAT solvers, which are rather optimized these days, should make our way faster. But the cost we pay for this is high since results provided by this approach must be taken very carefully: a formula may be considered true or false depending on the path length, and both false positives and false negatives may occur. Though easy for simple formulas, may be complex to determine if a formula will be considered true or false by using our method, even knowing its real truth value. Unfortunately, due to our lack of experience in the field of model checking, we are not able to make a deeper comparison of this strategy with explicit or symbolic model checkers.

The presented encoding should be easily extended to cover a wider range of P/T nets, such those with finite capacity, which will allow the verification of more interesting problems. We may start changing the definition of state to consider a set of places and the number of tokens they contain, i.e. converting `sig State { atoms : set Atom }` into something like `sig State { places : Place -> Int }` and then reformulate everything else consequently. Another interesting direction for future work would be to investigate how some notion of fairness could be encoded into Alloy, which is far from obvious. Finally, from the UI perspective the integration of this development as part of the PIPE tool could be very helpful too.