Bounded Model Checking of Temporal Formulas with Alloy[⋆]

Alcino Cunha

HASLab – High Assurance Software Laboratory
INESC TEC & Universidade do Minho, Braga, Portugal
alcino@di.uminho.pt

Abstract. Alloy is a formal modeling language based on first-order relational logic, with no native support for specifying reactive systems. We propose an extension of Alloy to allow the specification of temporal formulas using LTL, and show how they can be verified by bounded model checking with the Alloy Analyzer.

1 Introduction

Alloy is a formal modeling language based on first-order relational logic [5]. Its Analyzer enables model validation and verification by translation to off-the-shelf SAT solvers. Alloy's logic is quite generic and does not commit to a particular specification style. In particular, there is no predefined way to specify and verify reactive systems, and several idioms and extensions have been proposed to address this issue. However, it is rather cumbersome and error-prone to specify and verify temporal properties with such idioms. In this paper we propose the usage of standard *Linear Temporal Logic* (LTL) to specify reactive systems in Alloy, and show how bounded model checking can be performed with its Analyzer, by resorting to the technique first proposed by Biere et al [1].

This paper is structured as follows. Section 2 shows how reactive systems and temporal properties can be specified and verified using Alloy and its Analyzer. Section 3 discusses how such properties can be specified more easily in LTL and then translated to Alloy, avoiding some of the potential problems pointed out in the previous section. Finally, we discuss some related work in Section 4.

2 Verifying reactive systems in Alloy

In Alloy, a *signature* represents a set of atoms. An atom is a unity with three fundamental properties: it is indivisible, immutable and uninterpreted. A signature declaration can introduce *fields*, sets of tuples of atoms capturing *relations*

[⋆] This work is financed by the ERDF – European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by national funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-037281.

```
open util/ordering[State]

sig State     {}
sig Message   { to, from : one Partition }
sig Channel   { messages : Message set -> State }
sig Partition { port : one Channel, pifp : set Partition }

fact NoSharedChannels { all c : Channel | lone port.c }

pred send     [m : Message, s,s’ : State] { ... }
pred receive  [m : Message, s,s’ : State] { ... }
pred transfer [m : Message, s,s’ : State] {
  m in m.from.port.messages.s and m.to in (m.from).pifp
  messages.s’ = messages.s - m.from.port->m + m.to.port->m
}

fact Init  { no Channel.messages.first }
fact Trans { all s : State, s’ : s.next |
  some m : Message | send[m,s,s’] or receive[m,s,s’] or transfer[m,s,s’]
}
```

Fig. 1. A PIFP specification in Alloy.

between the enclosing signature and others. Model constraints are defined by *facts*. *Assertions* express properties that are expected to hold as consequence of the stated facts. *Commands* are instructions to perform particular analysis. Alloy provides two commands: `run`, that instructs the Analyzer to search for an instance satisfying a given formula, and `check`, that attempts to contradict a formula by searching for a counterexample.

Since fields are immutable, to capture the dynamics of a state transition system a special signature whose atoms denote the possible states must be declared. Without loss of generality, we will denote this signature as `State`. Every field specifying a mutable relation must then include `State` as one of the signatures it relates. There are two typical Alloy *idioms* to declare such mutable fields: declare them all inside `State`, or add `State` as the last column in every mutable field declaration. The former idiom is known as *global state*, since all mutable fields are grouped together, while the latter is known as *local state*, since mutable fields are declared in the same signature as non-mutable ones of similar type.

Figure 1 presents an example of an Alloy model conforming to the local state idiom. It is a simplified model of the *Partition Information Flow Policy* (PIFP) of a *Secure Partitioning Kernel*. Essentially, the PIFP statically defines which information flows (triggered by message passing) are authorized between partitions. Signature `Message` declares the fields `to` and `fom`, capturing its destination and source `Partition`, respectively. Communication is done via channels, here simplified to contain sets of messages. Obviously, the messages contained in a channel vary over time. As such, signature `Channel` declares a mutable relation messages, that associates each channel with the set of its messages in each state. Signature Partition declares two binary relations: port, denoting the channel used by the partition to communicate, and pifp, that captures to which partitions it is authorized to send messages.

In Alloy *everything is a relation*. For example, sets are unary relations and variables are just unary singleton relations. As such, relational operators (in particular the *dot join* composition) can be used for various purposes. For example, in the fact NoSharedChannels the relational expression port.c denotes the set of partitions connected to port c. Multiplicities are also used in several contexts to constrain or check the cardinality of a relation. For example, in the same fact, multiplicity lone ensures that each channel is the port of at most one partition.

An operation can be specified using a predicate pred op[...,s,s':State] specifying when does op hold between the given pre- and post-states: to access the value of a mutable field in the pre-state (or post-state) it suffices to compose it with s (respectively s'). Our model declares three operations: send and receive message (whose specifications are omitted due to space limitations), and transfer message between partitions. The send operation just deposits the message in the sending partition port – the transfer operation is executed by the kernel and is the one responsible to enforce the PIFP. Inside an operation, a formula that does not refer s' can be seen as a pre-condition. Otherwise it is a post-condition. For example, m in m.from.port.messages.s is a pre-condition to transfer, requiring m to be in the port of the source partition prior to its execution. Notice that frame conditions, specifying which mutable relations remain unchanged, should be stated explicitly in each operation.

To specify temporal properties we need to model execution traces. A typical Alloy idiom for representing finite prefixes of traces is to impose a total ordering on signature State (by including the parameterized module util/ordering) and force every pair of consecutive states to be related by one of the operations (see fact Trans). Inside module util/ordering, the total order is defined by the binary relation next, together with its first and last states. The initial state of our example, where all channels are empty, is constrained by fact Init. A desirable safety property states that the port of every partition only contains messages sent from authorized partitions:

```
assert Safety { all p : Partition, m : Message | all s : State |
  m.to = p and m in p.port.messages.s implies p in m.from.pifp
}
```

Model checking this assertion with the Alloy Analyzer yields a counter-example: every partition can send a message to itself, even if not allowed by the PIFP. To correct this problem we can, for example, add to our model the fact all p:Partition | p in p.pifp, stating that all partitions should be allowed to send messages to themselves. Non-invariant temporal assertions can also be expressed with this idiom, but the complexity of the formulas and expertise required by the modeler increases substantially. Consider, for example the liveness property stating that all authorized messages are eventually transferred to the destination. At first glance, it could be specified as follows (using transitive closure to access the successors of a given state):

```
assert Liveness { all p : Partition, m : Message |
  all s : State | m in p.port.messages.s and m.to in p.pifp implies
    some s' : s.*next | m in m.to.port.messages.s'
}
```

A simple (but artificial) way to ensure `Liveness` in this model is to disallow message sending while there are still pending messages to transfer. This could be done by adding the pre-condition `no Channel.messages.s` to operation `send`. However, even with such pre-condition, model-checking assertion `Liveness` with the Alloy Analyzer yields a false counter-example, where a message is sent in the last state of the trace prefix. In fact, if we consider only finite prefixes of execution traces, it is almost always possible to produce a false counter-example to a liveness property. This problem is well-known in the bounded model-checking community, and the solution, first proposed in [1], is to only consider as (true) counter-examples to such properties prefixes of traces containing a *back loop* in the last state, i.e., those that actually model infinite execution traces. It is easy to define a parameterized `trace` module[1], that adapts `util/ordering` to specify potential infinite traces instead of total orders, by allowing such back loop in the last state. Module `trace` also defines a predicate `infinite` that checks if the loop is present in a trace instance: if so, `next` always assigns a successor to every state, thus modeling an infinite trace. For convenience a dual predicate `finite` is also defined. By replacing `open util/ordering[State]` with `open trace[State]`, the above liveness property can now be correctly specified (and verified) as follows:

```
assert Liveness { all p : Partition, m : Message |
  all s : State | m in p.port.messages.s and m.to in p.pifp implies
    finite or some s' : s.*next | m in m.to.port.messages.s'
}
```

3 Embedding LTL formulas in Alloy

As seen in the previous section, although we can specify and verify (by bounded model checking) temporal properties in standard Alloy, it is a rather tricky and error-prone task, in particular since the user must be careful about where to check for finitude of trace prefixes. As such, we propose that, instead of using explicit quantifiers over the states in a trace, such properties be expressed using the standard LTL operators: `X` for next, `G` for always, `F` for eventually, `U` for until, and `R` for release. For example, the above temporal properties could be specified as follows:

```
assert Safety { all p : Partition, m : Message |
  G (m.to = p and m in p.port.messages implies p in m.from.pifp)
}
```

[1] Available at `http://www.di.uminho.pt/~mac/Publications/trace.als`.

$$
\begin{aligned}
\llbracket \texttt{X}\ \phi \rrbracket_s &\equiv \texttt{some}\ s\texttt{.next and}\ \llbracket \phi \rrbracket_{s\texttt{.next}} \\
\llbracket \texttt{G}\ \phi \rrbracket_s &\equiv \texttt{infinite and all}\ s' \texttt{:}\ s\texttt{.*next} \mid \llbracket \phi \rrbracket_{s'} \\
\llbracket \texttt{F}\ \phi \rrbracket_s &\equiv \texttt{some}\ s' \texttt{:}\ s\texttt{.*next} \mid \llbracket \phi \rrbracket_{s'} \\
\llbracket \phi\ \texttt{U}\ \psi \rrbracket_s &\equiv \texttt{some}\ s' \texttt{:}\ s\texttt{.*next} \mid \llbracket \psi \rrbracket_{s'}\ \texttt{and all}\ s'' \texttt{:upto[}s\texttt{,}s'\texttt{]} \mid \llbracket \phi \rrbracket_{s''} \\
\llbracket \phi\ \texttt{R}\ \psi \rrbracket_s &\equiv \llbracket \texttt{G}\ \psi \rrbracket_s\ \texttt{or some}\ s' \texttt{:}\ s\texttt{.*next} \mid \llbracket \phi \rrbracket_{s'}\ \texttt{and all}\ s'' \texttt{:upto[}s\texttt{,}s'\texttt{]+}s' \mid \llbracket \psi \rrbracket_{s''}
\end{aligned}
$$

$$
\begin{aligned}
\llbracket \texttt{not}\ \phi \rrbracket_s &\equiv \texttt{not}\ \llbracket \phi \rrbracket_s & \llbracket \Phi\ \texttt{.}\ \Psi \rrbracket_s &\equiv \llbracket \Phi \rrbracket_s\ \texttt{.}\ \llbracket \Psi \rrbracket_s \\
\llbracket \phi\ \texttt{and}\ \psi \rrbracket_s &\equiv \llbracket \phi \rrbracket_s\ \texttt{and}\ \llbracket \psi \rrbracket_s & \llbracket \Phi\ \texttt{\&}\ \Psi \rrbracket_s &\equiv \llbracket \Phi \rrbracket_s\ \texttt{\&}\ \llbracket \Psi \rrbracket_s \\
\llbracket \phi\ \texttt{or}\ \psi \rrbracket_s &\equiv \llbracket \phi \rrbracket_s\ \texttt{or}\ \llbracket \psi \rrbracket_s & \llbracket \Phi\ \texttt{+}\ \Psi \rrbracket_s &\equiv \llbracket \Phi \rrbracket_s\ \texttt{+}\ \llbracket \Psi \rrbracket_s \\
\llbracket \texttt{all}\ x : \Phi \mid \phi \rrbracket_s &\equiv \texttt{all}\ x : \llbracket \Phi \rrbracket_s \mid \llbracket \phi \rrbracket_s & \llbracket \Phi\ \texttt{->}\ \Psi \rrbracket_s &\equiv \llbracket \Phi \rrbracket_s\ \texttt{->}\ \llbracket \Psi \rrbracket_s \\
\llbracket \texttt{some}\ x : \Phi \mid \phi \rrbracket_s &\equiv \texttt{some}\ x : \llbracket \Phi \rrbracket_s \mid \llbracket \phi \rrbracket_s & \llbracket \texttt{*}\Phi \rrbracket_s &\equiv \texttt{*}\llbracket \Phi \rrbracket_s \\
\llbracket \Phi\ \texttt{in}\ \Psi \rrbracket_s &\equiv \llbracket \Phi \rrbracket_s\ \texttt{in}\ \llbracket \Psi \rrbracket_s & \llbracket \texttt{none} \rrbracket_s &\equiv \texttt{none}
\end{aligned}
$$

$$
\llbracket x \rrbracket_s \equiv \begin{cases} x.s & \text{if } x \text{ is the id of a mutable field declared with the local state idiom} \\ s.x & \text{if } x \text{ is the id of a mutable field declared with the global state idiom} \\ x & \text{otherwise (i.e., a variable or the id of an immutable field)} \end{cases}
$$

Fig. 2. Embedding of temporal formulas.

```
assert Liveness { all p : Partition, m : Message |
  G (m in p.port.messages and m.to in p.pifp implies
    F (m in m.to.port.messages))
}
```

Assuming traces are specified with module `trace`, the embedding of LTL into Alloy can be done via an (almost) direct encoding of the translation proposed by Biere et al. [1] (for bounded model checking of LTL with a SAT solver). Formally, a formula ϕ occurring in a fact or `run` command should be replaced by $\llbracket \mathit{NNF}(\phi) \rrbracket_{\texttt{first}}$, where $\llbracket \phi \rrbracket_s$ is the embedding function defined in Figure 2, and $\mathit{NNF}(\phi)$ is the well-known transformation that converts formula ϕ to *Negation Normal Form* (where all negations appear only in front of atomic formulas). When finding a model for `G` ϕ, only prefixes capturing infinite traces should be considered, thus assuring that ϕ is not violated further down the trace. As clarified in [1], conversion to NNF is necessary since in the bounded semantics of LTL the duality of `G` and `F` no longer hold. In the encoding of `U` and `R` we use the function `upto`, defined in module `trace`, that, given s and s' computes all states from s up to s' (not including the latter). The embedding of logic and relational operators is trivial, and thus only a representative subset of Alloy's logic is presented. A formula ϕ occurring in an assertion or `check` command should be replaced by `not` $\llbracket \mathit{NNF}(\texttt{not}\ \phi) \rrbracket_{\texttt{first}}$. Since assertions in check commands are negated in order to find counter-examples, the outermost negation ensures they still remain in NNF.

Note that, to improve efficiency (and likewise to `util/ordering`), when the `trace` module is imported the scope of the parameter signature is interpreted as an exact scope. This means that trace prefixes are bounded to be of size equal to the scope of the `State` signature. Thus, to perform bounded model checking of an assertion, the user should manually increase the scope of `State` one unit at a time up to the desired bound.

4 Related work

Several extensions of Alloy to deal with dynamic behavior have been proposed. DynAlloy [3] proposes an Alloy variant that allows the specification of properties over execution traces using a formalism inspired by dynamic logic. Imperative Alloy [6] proposes a more minimal extension to the language, with a simple semantics by means of an embedding to standard Alloy. Unfortunately, in both these works the verification of liveness properties may yield spurious counterexamples, similar to the one presented in Section 2.

One of the advantages of our approach is that reactive systems can be specified declaratively using Alloy's relational logic, as opposed to traditional model checkers where transitions must be specified imperatively. Chang and Jackson [2] proposed a BDD-based model checker for declarative models specified with relational logic enhanced with CTL temporal formulas. The current proposal shows how the Alloy Analyzer can directly be used to perform bounded model checking of temporal formulas without the need for a new tool.

Recently, Vakili and Day [7] showed how CTL formulas with fairness constraints can be model checked in Alloy, by using the encoding to first order logic with transitive closure first proposed by Immerman and Vardi [4]. Their technique performs full model checking on state transition systems specified declaratively, but bounded to have at most the number of states specified in the scope. This non-standard form of bounded model checking can yield non-intuitive results in many application scenarios, or even prevent verification at all if the the specification cannot be satisfied by a transition system that fits in the (necessarily small) scope of `State`. Moreover, instead of proposing an Alloy extension, CTL formulas are expressed using library functions that compute the set of states where the formula holds. This leads to unintuitive specifications, since the user is then forced to use relational operators to combine formulas instead of the standard logical connectives.

References

1. Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.
2. Felix Chang and Daniel Jackson. Symbolic model checking of declarative relational models. In *ICSE*, pages 312–320. ACM, 2006.
3. Marcelo Frias, Juan Galeotti, Carlos Pombo, and Nazareno Aguirre. DynAlloy: upgrading Alloy with actions. In *ICSE*, pages 442–451. ACM, 2005.
4. Neil Immerman and Moshe Vardi. Model checking and transitive-closure logic. In *CAV*, volume 1254 of *LNCS*, pages 291–302. Springer, 1997.
5. Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, revised edition, 2012.

6. Joseph Near and Daniel Jackson. An imperative extension to Alloy. In *ABZ*, volume 5977 of *LNCS*, pages 118–131. Springer, 2010.
7. Amirhossein Vakili and Nancy Day. Temporal logic model checking in Alloy. In *ABZ*, volume 7316 of *LNCS*, pages 150–163. Springer, 2012.