Program Verification: Lecture 16

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

Verification of Declarative Concurrent Programs

We are now ready to discuss the subject of verification of declarative concurrent programs, and, more specifically, the verification of properties of Maude system modules, that is, of declarative concurrent programs that are rewrite theories.

There are two levels of specification involved: (1) a system specification level, provided by the rewrite theory $\mathcal{R}$ and yielding an initial model $\mathcal{T}_{\mathcal{R}}$ for our program; and (2) a property specification level, given by some property (or properties) φ that we want to prove about our program. To say that our program satisfies the property φ then means exactly to say that its initial model does.

Verification of Declarative Concurrent Programs (II)

Specifically, we have seen that $\mathcal{T}_{\mathcal{R}}$ is initial model among all Σ-transition systems satisfying the axioms of the given rewrite theory $\mathcal{R}$.

The question then becomes, which language shall we use to express the properties φ that we want to prove hold in the initial model $\mathcal{T}_{\mathcal{R}}$? That is, how should we express relevant properties φ such that,

$$\mathcal{T}_{\mathcal{R}} \models \varphi.$$

The first, most obvious possibility is to use a first-order language based on the signature Σ together with a family of binary transition relations $\{\rightarrow_k\}_{k \in K}$.

Verification of Declarative Concurrent Programs (IV)

We can consider a modal logic $\mathcal{M}(\Sigma)$ for Σ-transition systems expressing necessity, $\Box\varphi$, and possibility, $\Diamond\varphi$ properties, which can be regarded as a sublanguage of such a first-order language. We will focus on properties $\Box I$, with I a predicate on states, stating that I is an invariant.

Not all properties of interest are expressible in $\mathcal{M}(\Sigma)$. For example, properties involving fairness, and other properties related to the infinite behavior of a system typically are not expressible in $\mathcal{M}(\Sigma)$. They can be expressed in temporal logic. The simplest choice is linear temporal logic (LTL). Maude supports verification of LTL properties in its LTL model checker. LTL and LTL model checking verification in Maude are explained in Chapter 13 of *All About Maude*.

Invariants

Rather than developing in full detail the logic $\mathcal{M}(\Sigma)$, for the moment we will focus on invariants. Invariants specify safety properties, that is, properties guaranteeing that nothing "bad" can happen or, equivalently, that the system will always be in a "good" state. Given a rewrite theory $\mathcal{R}$, a chosen kind k of states, and an equationally-defined Boolean predicate I on states of kind k, we say that I is an invariant for $\mathcal{T}_{\mathcal{R}}$ from an initial state $[t]$, written

$$\mathcal{T}_{\mathcal{R}}, [t] \models \Box I$$

if and only if $\mathcal{T}_{\mathcal{R}}$ satisfies the following first-order formula:

$$(\forall x : k)\ (t \rightarrow^* x) \Rightarrow I(x) = true.$$

Invariants (II)

This exactly means that: (i) $I(t) = true$, and (ii) $I(x) = true$ for any state x reachable from t. Intuitively, the predicate I specifies some good property that our system must always satisfy. The fact that we have $\mathcal{T}_{\mathcal{R}}, [t] \models \Box I$ means that our system is I-safe, in the sense that the bad thing, namely $\neg I$, will never happen in any state reachable from our initial state $[t]$.

For any Σ-transition system $\mathcal{A} = (\mathbb{A}, \rightarrow_{\mathcal{A}})$ a kind k and an element $a \in A_k$, we define the set of states reachable from a by, $Reach_{\mathcal{A}}(a) = \{x \in A_k \mid a \rightarrow^*_{\mathcal{A}} x\}$. Similarly, given a Boolean predicate I with arguments of kind k, we define the invariant set defined by I in $\mathcal{A}$ as, $[\![I]\!]_{\mathcal{A}} = \{x \in A_k | I_{\mathbb{A}}(x) = true_{\mathbb{A}}\}$.

Invariants (III)

Therefore, we have the equivalence,

$$\mathcal{T}_{\mathcal{R}}, [t] \models \Box I \quad \Leftrightarrow \quad Reach_{\mathcal{T}_{\mathcal{R}}}([t]) \subseteq [\![I]\!]_{\mathcal{T}_{\mathcal{R}}}$$

More generally, given any Σ-transition system $\mathcal{A} = (\mathbb{A}, \rightarrow_{\mathcal{A}})$, the invariant satisfaction relation $\mathcal{A}, a \models \Box I$ can likewise be characterized by the equivalence:

$$\mathcal{A}, a \models \Box I \quad \Leftrightarrow \quad Reach_{\mathcal{A}}(a) \subseteq [\![I]\!]_{\mathcal{A}}.$$

In other words, the predicate I carves out a subset $[\![I]\!]_{\mathcal{A}}$ of "good" states. Satisfying the invariant I just means that the set $Reach_{\mathcal{A}}(a)$ of reachable states is always inside the I-safety envelope $[\![I]\!]_{\mathcal{A}}$. An interesting question is how to verify such invariants.

Model Checking Invariants through Search

Suppose that we have specified a rewrite theory $\mathcal{R}$ in Maude as a system module, and that, for k a chosen kind of states with, say, `init` the chosen initial state, $\mathcal{R}$ contains also a Boolean predicate `I` that we want to check it is an invariant, that is, that

$$\mathcal{T}_{\mathcal{R}}, \texttt{init} \models \Box \texttt{I}$$

How can we do this in an automatic way? The key observaton is that I holds if and only if the search command

```
search init =>* x:k such that I(x:k) =/= true .
```

has no solutions. Indeed, having no solutions exactly means that on `init`, and on all states reachable from it, the predicate `I` evaluates to `true`, that is, that `I` is an invariant.

Model Checking Invariants through Search (II)

Consider a simple clock that marks the hours of the day. Such a clock can be specified by the system module

```
mod CLOCK is
  protecting INT .
  sort Clock .
  op clock : Int -> Clock [ctor] .
  var T : Int .
  rl [tick] : clock(T) => clock((T + 1) rem 24) .
endm
```

Let `clock(0)` be the initial state. Note that, in principle, the clock could be in an infinite number of states, such as `clock(7633157)` or `clock(-33457129)`. The point, however, is that if the initial state is `clock(0)`, then only states `clock(T)` with times `T` such that `0 <= T < 24` can be reached.

Model Checking Invariants through Search (III)

This suggests making the predicate `0 <= T < 24` an invariant of our clock system.

Since using simple linear arithmetic reasoning we can express the negation of such an invariant as the predicate `(T < 0) or (T >= 24)`, we can automatically verify that our simple clock satisfies the invariant by giving the command:

```
Maude> search in CLOCK : clock(0) =>* clock(T)
         such that T < 0 or T >= 24 = true .

No solution.
states: 24  rewrites: 216 in 0ms cpu (2ms real) (~ rews/sec)
```

Model Checking Invariants through Search (IV)

We call this process of automatically checking an invariant through search model checking, since we are cheching if our model, namely the initial model $\mathcal{T}_{\mathcal{R}}$ together with a chosen initial state satisfies a given invariant `I`.

If, as in the clock example, the number of states reachable from the initial state is finite, then search provides a decision procedure for the satisfaction of invariants: in finite time Maude will either find no solutions to a search for states violating the invariant, or will find an invariant-violating state together with a sequence of rewrites from the initial state to it, that is, a counterexample.

Model Checking Invariants through Search (V)

But what if the number of states reachable from the initial state is infinite? In such a case, if the invariant `I` is violated, the `search` command will terminate in finite time yielding a counterexample. Assuming that the rules in $\mathcal{R}$ have no rewrites in their conditions, termination is guaranteed by the breadth-first nature of the search.

A state violating the invariant is a reachable state: there is a finite sequence of rewrites from the initial state to it. Since we assume that there is a finite number of rules R, and therefore a finite number of ways that each state can be rewritten, even though the number of reachable states is infinite, the number of states reachable from the initial state by a sequence of rewites of length less than a given bound is finite.

Model Checking Invariants through Search (VI)

This bounded subset is always explored in finite time by the `search` command. This means that, for systems where the set of reachable states is infinite, search becomes a semi-decision procedure for detecting the violation of an invariant. That is, if the invariant is violated, we are guaranteed to get a counterexample; but, if it is not violated, we will search forever, never finding it.

We can illustrate the semi-decision procedure nature of search for the verification of invariant failures with a simple infinite-state example of processes and resources. Processes and resources have no identities or topology; also, the number of processes and resources can grow dynamically in an unbounded manner.

Model Checking Invariants through Search (VII)

```
mod PROCS-RESOURCES is
  sorts State Resources Process .
  subsort Process < State .
  subsort Resources < State .
  ops res null : -> Resources [ctor] .
  op p : Resources -> Process [ctor] .
  op __ : Resources Resources -> Resources
       [ctor assoc comm id: null] .
  op __ : State State -> State [ctor ditto] .

  rl [acq1] : p(null) res => p(res) .
  rl [acq2] : p(res) res => p(res res) .
  rl [rel] : p(res res) => p(null) res res .
  rl [dupl] : p(null) res => p(null) res p(null) res .
endm
```

Model Checking Invariants through Search (VIII)

The state is a soup (multiset) of processes and resources. Each process needs to acquire two resources. Originally, each process `p` contains the `null` state. But if a resource `res` is available, it can acquire it (rule `[acq1]`). If a second resource becomes available, it can also acquire it (rule `[acq2]`).

After acquiring both resources and using them, the process can release them (rule `[rel]`).

Furthermore, the number of processes and resources can grow in an unbounded manner by the duplication of each process-resource pair (rule `[dupl]`).

Model Checking Invariants through Search (IX)

One invariant we might like to verify about this system is deadlock freedom from an initial state `res p(null)`. There are two ways to model check this property: one completely straightforward, and another requiring some extra work. The straightforward manner is to give the search command

```
Maude> search in PROCS-RESOURCES : res p(null) =>! X:State .

Solution 1 (state 1)
states: 3  rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
X:State --> p(res)

Solution 2 (state 5)
states: 9  rewrites: 9 in 0ms cpu (1ms real) (~ rews/sec)
X:State --> p(res) p(res)
```

```
Solution 3 (state 13)
states: 19  rewrites: 26 in 0ms cpu (3ms real) (~ rews/sec)
X:State --> p(res) p(res) p(res)

Solution 4 (state 25)
states: 34  rewrites: 56 in 0ms cpu (4ms real) (~ rews/sec)
X:State --> p(res) p(res) p(res) p(res)

Solution 5 (state 43)
states: 55  rewrites: 104 in 0ms cpu (23ms real) (~ rews/sec)
X:State --> p(res) p(res) p(res) p(res) p(res)

......

Solution 20 (state 1649)
states: 1770  rewrites: 5640 in 20ms cpu (67ms real)
  (282000 rews/sec)
X:State --> p(res) p(res) p(res) p(res) p(res) p(res) p(res) p(res)
            p(res) p(res) p(res) p(res) p(res) p(res) p(res) p(res)
            p(res) p(res) p(res) p(res)
......
```

Model Checking Invariants through Search (X)

Maude will indeed continue printing all the solutions it finds. But since there is an infinite number of deadlock states, it may be preferable to specify in advance a bound on the number of solutions, giving, for example, a command like the following, that looks for at most 5 solutions.

```
Maude> search [5] in PROCS-RESOURCES : res p(null) =>! X:State .
```

The nice thing about model checking deadlock freedom this way is that there is no need to explicitly specify the invariant as a Boolean predicate. This is because the negation of the invariant is by definition the set of deadlock states, which is what the `search` command with the `=>!` qualification precisely looks for.

Model Checking Invariants through Search (XI)

But, if one wishes, one can, with a little more work, perform an equivalent model checking of the same property by using an explicit enabledness predicate, telling us that a state can make a transition. Of course, this cannot be done in the original module, because such a predicate has not been defined, but this is easy enough to do:

```
mod PROCS-RESOURCES-ENABLED is
  protecting PROCS-RESOURCES .
  op enabled : State -> Bool .
  eq enabled(p(null) res X:State) = true .
  eq enabled(p(res) res X:State) = true .
  eq enabled(p(res res) X:State) = true .
  eq enabled(X:State) = false [owise] .
endm
```

Model Checking Invariants through Search (XII)

One can then give the command

```
Maude> search [5] in PROCS-RESOURCES-ENABLED : res p(null)
         =>* X:State such that enabled(X:State) =/= true .
```

getting the following 5 solutions:

```
Solution 1 (state 1)
states: 2  rewrites: 4 in 0ms cpu (0ms real) (~ rews/sec)
X:State --> p(res)

Solution 2 (state 5)
states: 6  rewrites: 15 in 0ms cpu (0ms real) (~ rews/sec)
X:State --> p(res) p(res)

Solution 3 (state 13)
```

```
states: 14  rewrites: 41 in 0ms cpu (0ms real) (~ rews/sec)
X:State --> p(res) p(res) p(res)

Solution 4 (state 25)
states: 26  rewrites: 87 in 0ms cpu (1ms real) (~ rews/sec)
X:State --> p(res) p(res) p(res) p(res)

Solution 5 (state 43)
states: 44  rewrites: 160 in 0ms cpu (1ms real) (~ rews/sec)
X:State --> p(res) p(res) p(res) p(res) p(res)
```

Bounded Model Checking of Invariants

Although search can be a quite effective model checking technique for invariants, it has some limitations:

- if the set of reachable states is infinite and the invariant is satisfied, the search process never terminates;
- even if the number of reachable states is finite, it may be too large to be explored in reasonable time and space, due to time and memory limitations.

In such cases we have several alternatives. The most obvious is to give up on completeness and settle for searching states only up to a bound on the depth of paths reaching them. Another alternative is to use an equational abstraction (see §12.4 of *All About Maude*) to make the set of reachable states finite.

Bounded Model Checking of Invariants (II)

Bounded model checking is an appealing and widely used formal analysis method. It cannot guarantee that an invariant holds everywhere, but it can either: (i) find very useful and subtle counterexamples; or (ii) guarantee that up to a certain depth the invariant holds.

Bounded model checking of invariants is supported in Maude by means of the bounded search command.

Consider the following specification of a readers-writers system.

Bounded Model Checking of Invariants (III)

```
mod R&W is
  protecting NAT .
  sort Config .
  op <_,_> : Nat Nat -> Config [ctor] .  --- readers/writers

  vars R W : Nat .

  rl < 0, 0 > => < 0, s(0) > .
  rl < R, s(W) > => < R, W > .
  rl < R, 0 > => < s(R), 0 > .
  rl < s(R), W > => < R, W > .
endm
```

A state is represented by a tuple `< R, W >` indicating the number `R` of readers and the number `W` of writers accessing a critical resource. Readers and writers can leave the resource at any time, but writers can only gain access to it if nobody else is using it, and readers only if there are no writers.

Bounded Model Checking of Invariants (IV)

With initial state `< 0, 0 >` want to verify three invariants:

- mutual exclusion: readers and writers never access the resource simultaneously: only readers or only writers can do so at any given time.
- one writer: at most one writer will be able to access the resource at any given time.
- deadlock freedom: there are no deadlocks.

We can try to model check these three invariants. In this example the invariants themselves can be expressed in two different ways: (i) implicitly, by giving a pattern characterizing their negation; or (ii) explicitly by defining appropriate state predicates.

Bounded Model Checking of Invariants (V)

The implicit method is the easiest:

```
Maude> search < 0,0 > =>* < s(N:Nat), s(M:Nat) > .

Maude> search < 0,0 > =>* < N:Nat, s(s(M:Nat)) > .

Maude> search < 0,0 > =>! C:Config .
```

The negations of each of the first two invariants do not need to be given explicitly: they can be described by the patterns we search for. The negation of the first invariant corresponds to the simultaneous presence of readers and writers, which is exactly captured by the pattern `< s(N:Nat), s(M:Nat) >`; whereas the negation of the fact that at most one writer should be present at any given time is exactly captured by the pattern `< N:Nat, s(s(M:Nat)) >`. For deadlock-freedom the pattern is trivial: `C:Config`.

Bounded Model Checking of Invariants (V)

Since the number or readers is unbounded, the set of reachable states is infinite and the search commands never terminate. We can perform bounded model checking of these three invariants by giving a 10^6 depth bound:

```
Maude> search [1, 1000000] < 0,0 > =>* < s(N:Nat), s(M:Nat) > .
No solution.
states: 1000002  rewrites: 2000001 in 36480ms cpu (50317ms real)

Maude> search [1, 1000000] < 0,0 > =>* < N:Nat, s(s(M:Nat)) > .
No solution.
states: 1000002  rewrites: 2000001 in 38910ms cpu (41650ms real)

Maude> search [1, 1000000] < 0,0 > =>! C:Config .
No solution.
states: 1000003  rewrites: 2000002 in 5752ms cpu (5821ms real)
```

Bounded Model Checking of Invariants (VI)

The second method is to explicitly define our invariants by means of state predicates. This is also easy to do:

```
mod R&W-PREDS is
   protecting R&W .
   ops mutex one-writer enabled : Config -> Bool .
   eq mutex(< s(N:Nat),s(M:Nat) >) = false .
   eq mutex(< 0,N:Nat >) = true .
   eq mutex(< N:Nat,0 >) = true .
   eq one-writer(< N:Nat,s(s(M:Nat)) >) = false .
   eq one-writer(< N:Nat,0 >) = true .
   eq one-writer(< N:Nat,s(0) >) = true .
   eq enabled(< 0, 0 >) = true .
   eq enabled(< R:Nat, s(W:Nat) >) = true .
   eq enabled(< R:Nat, 0 >) = true .
   eq enabled(< s(R:Nat), W:Nat >) = true .
   eq enabled(< N:Nat, M:Nat >) = false [owise] .
endm
```

Bounded Model Checking of Invariants (VII)

```
search [1, 1000000] < 0,0 > =>* C:Config s.t. mutex(C:Config) = false .

No solution.
states: 1000002  rewrites: 3000003 in 7935ms cpu (8027ms real)

search [1, 1000000] < 0,0 > =>* C:Config s.t. one-writer(C:Config) =
    false .

No solution.
states: 1000002  rewrites: 3000003 in 7662ms cpu (7720ms real)

search [1, 1000000] < 0,0 > =>* C:Config s.t. enabled(C:Config) =
    false .

No solution.
states: 1000002  rewrites: 3000003 in 11516ms cpu (13303ms real)
```