Maude Summer School: Lecture 4

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign and
Leverhulme visiting professor at King's College, London

The Mathematical Model of a Rewrite Theory

Remember Lecture 1's statement about Maude:

> "The meaning of a program P is a mathematical model $\mathbb{C}_P$ in first-order logic, called its canonical model."

For a functional module `fmod` (Σ, E) `endfm` satisfying executability conditions (1)–(3) we saw in Lecture 3-I what that model is: the canonical term algebra $\mathbb{C}_{\Sigma/E}$. But what is the model $\mathbb{C}_P$ for P a system module `mod` (Σ, E, R) `endm`?

Intuitively, it should be a transition system. For its functional part (Σ, E), we should again require conditions (1)–(3). Therefore its states should the the elements of the canonical term algebra $\mathbb{C}_{\Sigma/E}$. What about its transition relation? They should be transitions defined by the rules R.

The Mathematical Model of a Rewrite Theory (II)

But there is a problem, called the coherence problem. Let (Σ, E, R) have Σ unsorted with just three constants a, b, c, $E = \{a = c\}$, and $R = \{a \to b\}$, with $\Omega = \{c, b\}$, so that $\mathbb{C}_{\Sigma/E} = \{c, b\}$ has just two states. The problem is that there is no meaningful way to apply the rule $a \to b$ to obtain the transition that should exist from state c to state b.

The mathematical model we want is called a Σ-transition system, where the states have a Σ-algebra structure —in our case $\mathbb{C}_{\Sigma/E}$— and there is a transition relation between states. We just need to have suitable executability conditions to properly define the transition relation.

Executability of Rewrite Theories: Coherence

When is a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$ executable? $(\Sigma, E \cup B)$ should be sort-decreasing, terminating, confluent, and sufficiently complete modulo B. But this is not enough. We also need that the rules R are coherent with E modulo the axioms B (in our example, we just add rule $c \to b$):

$$
\begin{tikzcd}
t \arrow[r, "R/B"'] \arrow[dd, "!_{E/B}"'] & t' \arrow[dr, dotted, "!_{E/B}"] & \\
 & & w \\
u \arrow[r, dotted, "R/B"'] & u' \arrow[ur, dotted, "!_{E/B}"'] &
\end{tikzcd}
$$

Maude's Coherence Checker tool checks this property.

The Canonical Σ-Transition System $\mathbb{C}_{\mathcal{R}}$

Given a system module `mod` $\mathcal{R}$ `endm`, with, say, $\mathcal{R} = (\Sigma, E \cup B, R)$, Maude assumes the following executability conditions: (i) the usual ones for $(\Sigma, E \cup B)$, and (ii) the coherence of R with respect to E modulo B.

Assuming (i)–(ii), we can define the canonical Σ-transition system $\mathcal{C}_{\mathcal{R}} = (\mathbb{C}_{\Sigma/E,B}, \rightarrow_{\mathcal{C}_{\mathcal{R}}})$, were $\mathbb{C}_{\Sigma/E,B}$ is the canonical term algebra modulo B, and given $[u], [v] \in C_{\Sigma/E,B,[s]}$, $[u] \rightarrow_{\mathcal{C}_{\mathcal{R}}} [v]$ holds iff there exists v' such that $u \rightarrow_{R/B} v'$ and $[v] = [v'!_{E/B}]$. I.e., states are elements of $\mathbb{C}_{\Sigma/E,B}$; transitions from $[u]$ are the normal forms $[v'!_{E/B}]$ of rewrites $u \rightarrow_{R/B} v'$.

Verification of Declarative Concurrent Programs in Maude

Remember again Lecture 1:

> "Saying that program P satisfies a formal property φ exactly means that $\mathbb{C}_P \models \varphi$ in the first-order logic sense."

What does this mean for a system module `mod` $\mathcal{R}$ `endm`, with $\mathcal{R} = (\Sigma, E \cup B, R)$?

It means exactly what it says. $\mathbb{C}_P$ is precisely the canonical Σ-transition system $\mathcal{C}_{\mathcal{R}}$. And φ can be a formula in the first-order language based on the signature Σ plus the binary transition relation symbol $\rightarrow$.

Then we say that program `mod` $\mathcal{R}$ `endm` satisfies property φ iff

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

Invariants

In fact, we may verify properties φ not expressible in the first-order language $\Sigma \cup \{\rightarrow\}$. For example, liveness properties about the infinite behavior of a system, expressible in temporal logic. Maude supports verification of linear temporal logic (LTL) properties in its LTL model checker (see Chapter 13 of All About Maude).

In these lectures we will just focus on verifying invariants, the most basic safety properties, which are expressible in the first-order language of $\Sigma \cup \{\rightarrow\}$.

Invariants (II)

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 $\mathcal{R}$ and an equationally-defined Boolean predicate I, we say that I is an invariant for $\mathcal{C}_{\mathcal{R}}$ from an initial state $[t]$, written

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

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

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

Verifying Invariants in Maude

We can prove that an invariant I, i.e., a Boolean predicate on sates, holds for a Maude system module `mod` $\mathcal{R}$ `endm` from an initial state `init`, by searching for a violation of invariant I with the command:

```
search init =>* X:State s.t. I(X:State) =/= true .
```

If Maude, (i) replies `No solution`, then I has been proved. Instead, (ii) if I does not hold, we are guaranteed that Maude will find a counterexample. The only other possibility is (iii) I holds, but the set of states reachable from `init` is infinite; then we wait forever without getting an answer.

We can illustrate this model checking method by means of some examples.

The `QLOCK` Mutual Exclusion Protocol

`QLOCK` is a mutual exclusion protocol proposed by K. Futatsugi, where the number of processes is unbounded.

```
mod QLOCK is protecting NAT .
 sorts NatMSet NatList State .
 subsorts Nat < NatMSet NatList .
 op mt : -> NatMSet [ctor] .
 op _ _ : NatMSet NatMSet -> NatMSet [ctor assoc comm id: mt] .
 op nil : -> NatList [ctor] .
 op _;_ : NatList NatList -> NatList [ctor assoc id: nil] .
 op {_<_|_|_|_>} : NatMSet NatMSet NatMSet NatMSet NatList -> State [ctor] .
  op [_] : Nat -> NatMSet .  *** set of first n numbers
 op init : Nat -> State .   *** initial state, parametric on n

 vars n i j : Nat .  vars S U W C : NatMSet .  var Q : NatList .
 eq [0] = mt .
 eq [s(n)] = n [n] .
 eq init(n) = {[n] < mt | mt | mt | nil >} .
```

```
 rl [join] : {S i < U | W | C | Q >} => {S < U i | W | C | Q >} .
 rl [n2w] : {S < U i | W | C | Q >} => {S < U | W i | C | Q ; i >} .
 rl [w2c] : {S < U | W i | C | i ; Q >} => {S < U | W | C i | i ; Q >} .
 rl [c2n] : {S < U | W | C i | i ; Q >} => {S < U i | W | C | Q >} .
 rl [exit] : {S < U i | W | C | Q >} => {S i < U | W | C | Q >} .
endm
```

Processes are numbers. There is a left area for processes outside the protocol, and a protocol area (inside angle brackets). Processes outside can join the protocol (`[join]`). The protocol area has normal, waiting, and critical stages, plus a waiting queue, where a process can register its name to signal that it wants to enter the critical section (`[n2w]`). When its name appears at the front of the queue, it is allowed to enter the critical section (rule `[w2c]`). When it has finished, it can go back to normal (rule `[c2n]`). Finally, a normal process may leave the protocol (`[exit]`).

Verifying Invariants for `QLOCK`

We can verify two important invariants of `QLOCK`, namey,

- Mutual Exclusion, i.e., the critical section is either empty or has at most one process, and
- Deadlock Freedom, i.e., the protocol never stops.

for, e.g., the initial state `init(7)` with seven processes.

We can use two styles for proving this. Let us call them the cool style and the square style.

Cool Invariant Verification for `QLOCK`

In the cool style, we do not explicitly define an invariant predicates I. Instead we specify its negation or complement by a pattern (perhaps adding a s.t. constraint).

For example, we can characterize the violation of mutual exclusion in `QLOCK` by the pattern (by ACU, `C` could be `mt`):

```
{S < U | W | C i j | Q >}
```

and verify mutual exclusion with the `search` command:

```
Maude> search init(7) =>* {S < U | W | C i j | Q >} .

No solution.
```

Cool Invariant Verification for `QLOCK` (II)

Likewise, we do not need to define an explicit invariant predicate for deadlock freedom: we can instead take advantage of Maude's `=>!` search mode and give the `search` command to look for a terminating state:

```
Maude> search init(7) =>! X:State .

No solution.
```

Square Invariant Verification for `QLOCK`

We can explicitly define `mutex` and `enabled` predicates:

```
mod QLOCK-PREDS is protecting QLOCK .
 ops mutex enabled : State -> Bool .
 vars n i j : Nat .  vars S U W C : NatMSet .  var Q : NatList .

 eq mutex({S < U | W | mt | Q >}) = true .
 eq mutex({S < U | W | i | Q >}) = true .
 eq mutex({S < U | W | i j C | Q >}) = false .

 eq enabled({S i < U | W | C | Q >}) = true .
 eq enabled({S < U i | W | C | Q >}) = true .
 eq enabled({S < U | W i | C | i ; Q >}) = true .
 eq enabled({S < U | W | C i | i ; Q >}) = true .
 eq enabled({S < U i | W | C | Q >}) = true .
 eq enabled(X:State) = false [owise] .
endm
```

Square Invariant Verification for **QLOCK** (II)

Then we can verify both invariants the hard or square way by giving the **search** commands:

```
Maude> search init(7) =>* X:State s.t. mutex(X:State) =/= true .

No solution.

Maude> search init(7) =>* X:State s.t. enabled(X:State) =/= true .

No solution.
```

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 (the cool way) by giving a pattern characterizing their negation; or (ii) explicitly (the square way) 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)
```