Formal Methods: Lecture 21

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

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,

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

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 with the system module

```
mod CLOCK is
  protecting INT .
  sort Clock .
  op clock : Int -> Clock [ctor] .
  var T : Int .
  rl 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 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. 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)
```

A Cryptographic Protocol Example

We can illustrate the power of this model checking technique for invariants of infinite state systems by showing how it can be used to find subtle attacks for cryptographic protocols, including some that have been used extensively and have been considered secure for a long time.

One such protocol is the 1978 Needham-Schroeder authentication protocol (NSPK) for which a subtle "man-in-the-middle" attack was found by G. Lowe in 1996 using model checking.

The goal of the NSPK Protocol is to provide authentication of two agents who want to be assured of each other's identity before they exchange safety-critical data.

A Cryptographic Protocol Example (II)

That is, an intruder should not be allowed to impersonate another agent. For this purpose, initiator and responder of a communication mutually authenticate each other.

NSPK uses public key cryptography, i.e., each agent has a public key which can be accessed by all agents, and a secret key which is the inverse of the public key.

Moreover, nonces are used in the protocol. Nonces are freshly generated, unguessable random numbers to be used in a single run of the protocol.

A Cryptographic Protocol Example (III)

Here is a textbook-style simplified description of NSPK:

Message 1	$A \rightarrow B : A.B.\{N_a, A\}_{PK(B)}$
Message 2	$B \rightarrow A : B.A.\{N_a.N_b\}_{PK(A)}$
Message 3	$A \rightarrow B : A.B.\{N_b\}_{PK(B)}$

This level of description is ambiguous, in that a fair amount of implicit assumptions are left unspecified. An object-oriented rewriting logic specification of the protocol (developed in joint work with G. Denker and C. Talcott) makes these assumptions explicit, and allows model checking.

Maude Specification of NSPK

We first specify key algebraic properties of the cryptographic infrastructure in a functional module.

```
fmod DATATYPES is
  sorts Key Field Nonce Principal Run Role EstabComm .
  subsort Nonce Principal Key < Field .
  op keypair : Key Key -> Bool [comm] .
  op ped : Key Field -> Field . *** encryption function
  op n : Principal Nat -> Nonce .
  ceq ped(sk,ped(pk,f)) = f  if keypair(sk,pk) .
  ...
endfm
```

The protocol itself, as well as the actions of an attacker, are specified as follows (fragment):

Maude Specification of NSPK (II)

```
class Agent | e_com: EstabCom, sec_key: Key, role_i: Run, role_r: Run,
              d_com: FieldSet cnt: Nat .

msg from_to_send_ : Principal Principal Field  -> Message .

vars A B P : Principal . vars RI RR : Run . vars NI : Nonce . ...
rl [BeginRun] :
  < A : Agent | role_i: RI, d_com: B U S, cnt: J >
  => < A : Agent | role_i: RI U (n(A,J),B,mtfield), d_com: S, cnt: J + 1 >
     from(A)to(B)send(ped(pk(B),n(A,J),A)) .

crl [Message1Rec] :
  < B : Agent | sec_key: SKB, role_i: RI, role_r: RR, cnt: J >
  from(A)to(B)send(ped(PKB,F,A))
  => < B : Agent | role_r: RR U (n(B,J),A,F), cnt: J + 1 >
     from(B)to(A)send(ped(pk(A),F.n(B,J)))
  if keypair(SKB,PKB) and not(F in RR) .
```

Maude Specification of NSPK (III)

```
crl [Message2RecCorrect] :
  < A : Agent | sec_key: SKA, role_i: RI U (NI,P,mtfield), e_com: C >
  from(B)to(A)send(ped(PKA,F))
  => < A : Agent | role_i: RI, ECom: C U (i,NI,B,rest(F)) >
     from(A)to(B)send(ped(pk(B),rest(F)))
  if keypair(SKA,PKA) and (B == P) and (NI == first(F)) .

crl [Message2RecIncorrect] :
  < A : Agent | sec_key: SKA, role_i: RI U (NI,P,mtfield) >
  from(B)to(A)send(ped(PKA,F))
  => < A : Agent | role_i: RI >
  if keypair(SKA, PKA) and (NI == first(F)) and (B =/= P) .
```

Maude Specification of the Intruder

```
class Intruder | e_com: EstabCom sec_key: Key ncs: FieldSet,
                 msgs: FieldSet agents: FieldSet role_i: Run,
                 role_r: Run d_com: Field cnt: Nat.
```

```
crl [IntruderFakeMessage] :
  < I : Intruder | ncs: N U F, agents: S U A U B >
  => < I : Intruder | ncs: N U F > from(A)to(B)send(ped(pk(B),F))
  if B =/= I .
```

similar: `IntruderInterceptMessage`, `IntruderOverhearMessage`, `IntruderReplayMessage`

The State Predicate of an Attack

In a topmost version of the specification, the situation where authentication information has been compromised is specified by the following state predicate:

```
op attack? : Configuration -> Prop .

ceq attack?(boundary(
     < INTR : Intruder | ecom : EC, rolei : RI, roler : RR,
                         ncs : fset+(fset+(FSET1, N1), N2) >
     < A : NSPKAgent | ecom : ecom+(EC2, ecom(ROLE,N1,B,N2)) >
     Conf)) = true
     if (not(inEstabCom(ecom(r,N2,A,N1),EC))
         and not(inEstabCom(ecom(i,N2,A,N1), EC))
         and not(in(N1,RI)) and not(in(N1,RR))
         and not(in(N2,RI)) and not(in(N2,RR))
         and B =/= INTR) == true .
```

Finding an Attack

The relevant invariant is that no such attack is possible under reasonable initial conditions. For example, we can consider a simple scenario with two agents and an attacker given by an initial state `cf2Agents1Intruder` equationally defined in the obvious manner. Then the desired invariant safety property for $\mathcal{R}$ the rewrite theory specifying the protocol is:

$$\mathcal{T}_{\mathcal{R}}, \texttt{cf2Agents1Intruder} \models \Box I$$

where, by definition, $I(S) = \neg(\texttt{attack?}(S) = \texttt{true})$.

Maude's `search` command finds Lowe's countarexample to such a property:

Finding an Attack (II)

```
Maude> search [1] cf2Agents1Intruder =>+ C:Configuration s.t.
                                        attack?(C:Configuration) = true .
search [1] in NSPK : cf2Agents1Intruder =>+ C:Configuration such that
                                        attack?(C:Configuration) = true .
Solution 1 (state 37826)
states: 37827 in 25350ms cpu (44300ms real)
C:Configuration -->
 boundary(< alice : NSPKAgent | cnt : 2,dcom : mtfset,roler
 : mtrun,rolei : mtrun,seckey : skalice,ecom : ecom(i, n(alice, 1), mrx, n(
 bob, 1)) > < bob : NSPKAgent | cnt : 2,dcom : mtfield,roler : mtrun,rolei :
 mtrun,seckey : skbob,ecom : ecom(r, n(bob, 1), alice, n(alice, 1)) > < mrx
 : Intruder | cnt : 1,dcom : mtfield,roler : mtrun,rolei : mtrun,seckey :
 skmrx,ecom : mtecom,agents : fset+(alice, bob, mrx),ncs : fset+(mtfset, n(
 alice, 1), n(bob, 1)),msgs : fset+(mtfset, ped(pkalice, cat(n(alice, 1), n(
 bob, 1)))) >)
```

Finding an Attack (III)

We can find the actual sequence of rewrites leading to the "man-in-the-middle" attack by giving to Maude the command `show path 37826 .` A detailed trace is then shown, corresponding to the sequence of rewrite rule applications:

```
[BeginRun];[IntruderAcceptEveryMessage1];[IntruderFakeMessage1];

[Message1Rec];[IntruderInterceptMessage2];[IntruderReplayMessage];

[Message2Rec];[IntruderAcceptEveryMessage3];

[IntruderFakeMessage3];[Message3Rec]
```

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. A second alternative is to use an abstraction (this will be discussed in Lecture 23).

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 READERS-WRITERS 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)

Taking `< 0, 0 >` as the initial state, we would like to verify two important invariants, namely:

- 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.

We can try to model check these two 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)) > .
```

In this case the state predicates corresponding to the negation of each of the two invariants do not need to be given explicitly as Boolean conditions: they can instead be described more simply by the patterns we are searching 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 zero or at most one writer should be present at any given time is exactly captured by the pattern `< N:Nat, s(s(M:Nat)) >`.

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 instead perform bounded model checking of these two invariants by giving a depth bound, for example 10^6, with the commands:

```
Maude> search [1, 1000000] in READERS-WRITERS :
         < 0,0 > =>* < s(N:Nat), s(M:Nat) > .
No solution.
states: 1000002  rewrites: 2000001 in 36480ms cpu (50317ms real)
  (54824 rews/sec)

Maude> search [1, 1000000] in READERS-WRITERS :
         < 0,0 > =>* < N:Nat, s(s(M:Nat)) > .
No solution.
states: 1000002  rewrites: 2000001 in 38910ms cpu (41650ms real)
  (51400 rews/sec)
```

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 : 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 .
endm
```

Bounded Model Checking of Invariants (VII)

```
==========================================
search [1, 1000000] in R&W-PREDS : < 0,0 > =>* C:Config such that
                                        mutex(C:Config) = false .

No solution.
states: 1000002  rewrites: 3000003 in 46700ms cpu (101775ms real)
(64239 rewrites/second)
==========================================
search [1, 1000000] in R&W-PREDS : < 0,0 > =>* C:Config such that
                                     one-writer(C:Config) = false .

No solution.
states: 1000002  rewrites: 3000003 in 49190ms cpu (109273ms real)
(60988 rewrites/second)
```