A predicate abstraction tool for Maude

Miguel Palomino
Sistemas Informáticos y Programación, UCM

1 Introduction

Rewriting logic [7], through its executable Maude implementation [2], has proved to be a very flexible framework for the specification of concurrent systems. Designing such systems is hard and error-prone and thus studying whether the final design indeed satisfies the expected properties becomes an unavoidable task.

Model checking [1] is a verification technique that can be used efficiently to prove properties of finite systems in an automatic way, essentially by exhaustively enumerating all states. Unfortunately, many interesting systems are infinite and therefore not amenable to the convenience of model checking. In those cases, a possible way of dealing with the difficulty consists in computing a finite system that *simulates* the concrete system at hand, in the sense that if a certain property holds in the simpler one then it must be true also of the original; since the new system is finite, the validity of that property can be studied using model checking. As one would expect, finding such finite systems is not a straightforward task and some techniques have been proposed to construct them. These notes contain a brief description of the most popular such technique, predicate abstraction, an its realization in the Maude language.

2 What is predicate abstraction?

Predicate abstraction is a technique to automatically compute finite systems that simulate more complex one. More precisely, assume a system with states $a, b, c, \ldots$, belonging to a set S, and with transitions given by a relation $\rightarrow \subseteq S \times S$. Then, a predicate abstraction is defined by a set of predicates $\phi_1, \ldots, \phi_n$ over the set of states in the following manner:

- The set of states of the abstract system is the set of n-tuples of Boolean values, where n is the number of predicates.
- A concrete state a is mapped to the tuple $\alpha(a) = \langle \phi_1(a), \ldots, \phi_n(a) \rangle$.

The transition relation in the abstract system is then determined in a standard way:

- There is a transition from the abstract state $\langle b_1, \ldots, b_n \rangle$ to $\langle b'_1, \ldots, b'_n \rangle$ iff there are concrete states a and b such that $\alpha(a) = \langle b_1, \ldots, b_n \rangle$, $\alpha(b) = \langle b'_1, \ldots, b'_n \rangle$, and $a \rightarrow b$.

Computing the abstract state associated to a concrete a, though dependent on the complexity of the predicates, is usually straightforward. It is in the computation of the abstract transition relation where the difficulty really lies, since a single transition step potentially depends on an infinite number of concrete states. In practice there are two approaches to the construction of the abstract transition relation. In the first one, that we follow, each predicate or rule used to define the transition relation of the concrete system is directly

transformed into a different predicate or rule that is then used to define a transition relation in the abstract system. This relation does not usually coincide with the exact abstract transition relation as defined above but it is just a good enough approximation. In the second one, called *implicit* predicate abstraction in [5], the abstract transition system is not explicitly computed; instead, the initial state is abstracted and all states reachable from it are computed.

3 How are rules abstracted?

To illustrate the use of these functions, and of predicate abstraction in general, we will use the following simple example adapted from Das's thesis [5].

```
mod SD-EXAMPLE is
  protecting NAT .

  sort Config .
  subsort Nat < Config .

  op init : -> Config .
  eq init = 0 .

  var N : Nat .

  crl N => s(N) if (N < 10) = true .
  rl N => N * 2 .
  rl s(s(N)) => N .
endm
```

This module specifies a system with the natural numbers as the set of states, and with an infinite number of states reachable from the initial one. (The sort `Config` and the operator `init` are necessary because the tool described in these notes assumes that these are the names of the sort of the states and of the initial state, respectively.) Because of this infinite number, model checking cannot be directly used to prove that, say 151, is not reachable from 0. A possible alternative to verify the property, that we present, is to compute an abstraction of the system and apply model checking to it.

We use two predicates to create the abstract system:

- $\phi_1(n) \iff n \leq 10$.
- $\phi_2(n) \iff n$ is even.

ϕ_1 arises as a slight modification of the first rule's condition; this form is preferred because it produces a more precise abstract system. If a certain state satisfies ϕ_1, it will still satisfy it after applying the rule to it; however, had we directly used the condition this might not longer be the case, which would introduce additional undeterminism in the abstract system. The predicate ϕ_2 is motivated by the "condition" (as a pattern) of the third rule. Used in conjunction with ϕ_1, it would allow us to prove the desired property if we were able to prove that at any time at least one of them holds. Thus, the set of abstract states consists of the four different ordered pairs of Boolean values and the initial state `0` is mapped to $\langle \texttt{true}, \texttt{false} \rangle$.

To construct the transition relation, let us consider the first rule. We will describe its abstraction by means of a predicate $\lambda(b_1, b_2, b_1', b_2')$ that determines a transition step

$$\langle b_1, b_2 \rangle \rightarrow \langle b_1', b_2' \rangle$$

iff $\lambda(b_1, b_2, b_1', b_2')$ is `true`.

Due to its condition, if the first rule can be applied to a number n then it must be the case that $\phi_1(n)$ is `true`. This means that the abstract predicate λ has to be of the form

$$\lambda(b_1, b_2, b_1', b_2') = b_1 \wedge \lambda'(b_1, b_2, b_1', b_2')\,.$$

Likewise, the resulting number $n+1$ also satisfies ϕ_1, and with this piece of information λ can be further refined to

$$\lambda(b_1, b_2, b_1', b_2') = b_1 \wedge b_1' \wedge \lambda''(b_1, b_2, b_1', b_2')\,.$$

The same argument does not carry over to ϕ_2. However, note that if the rule is applied to a number n that satisfies ϕ_2, then the resulting number $n+1$ always satisfies the negation of ϕ_2; analogously, if n satisfies the negation of ϕ_2 then $n+1$ satisfies ϕ_2. Putting all together we end up with

$$\lambda(b_1, b_2, b_1', b_2') = b_1 \wedge b_1' \wedge (b_2 \rightarrow \neg b_2') \wedge (\neg b_2 \rightarrow b_2') \wedge \lambda'''(b_1, b_2, b_1', b_2')\,.$$

Other than these, and the ones that can be derived from them, like $\neg b_1 \rightarrow b_2'$, there are no more relations among the Boolean variables defining λ; its value is

$$\begin{array}{rcl} \lambda(b_1, b_2, b_1', b_2') & = & b_1 \wedge b_1' \wedge (b_2 \rightarrow \neg b_2') \wedge (\neg b_2 \rightarrow b_2') \wedge \\ & & (\neg b_1 \rightarrow b_2') \wedge (\neg b_1 \rightarrow \neg b_2') \wedge (b_2 \rightarrow b_1') \wedge (\neg b_2 \rightarrow b_1')\,. \end{array}$$

Of course, the last four conjuncts are unnecessary and could be removed to simplify λ.

The same procedure is also followed to construct the abstract predicates that correspond to the other rules.

Actually, the only Boolean expressions which are checked when computing an abstraction with n predicates are those in B, B', and $B \rightarrow B'$, where $B = \{b_i, \neg b_i\}_{1 \leq i \leq n}$, $B' = \{b_i', \neg b_i'\}_{1 \leq i \leq n}$, and $B \rightarrow B' = \{x \rightarrow x' \mid x \in B, x' \in B'\}$. The concrete algorithm as implemented by our tool is described in [4], where these expresions are called *test points*. Here we will not give the details of how it works because it is not needed in what follows; a detailed discusion, besides that in the previous reference, can also be found in Uribe's thesis [8].

4 Using the tool

The goal of this section is to introduce, both from a user and an implementator point of view, a tool for computing predicate abstractions in Maude. A high-level description of how to use it would be:

1. Write a module containing the specification of the systems one is interested in.
2. Extend this module wit the predicates that will be used for the abstraction, and load it into Maude's database.
3. Load the file `pa-prototype.maude` tool which contains the tool's code. This module imports the files `model-checher.maude` and `itp-tool.maude`, which must therefore be in the current directory.
4. Run one of the functions `computeAbsModule`, `abstractionGround`, or `abstractionGen`, to get one of the three different representations of the abstract module.

4.1 Predicate abstraction in Maude

Currently the tool assumes some requirements about the module for which the abstraction will be computed:

1. The sort of the states is `Config`.
2. There is an initial state named `init`.
3. The predicates defining the abstraction are included in the same module.
4. The module is `Config`-encapsulated (`Config` only appears in a single operator, and as its codomain) and the conditions of the rules only contain equations.

Except for the last one, these requirements are likely to be eventually removed.

In our running example, we then have to extend the module `SD-EXAMPLE` with the specification of the two predicates, which in turn need an auxiliary function that checks whether a number is even or not.

```
--- Predicates used for the abstraction and their auxiliary functions.

ops phi1 phi2 : Config -> Bool .
op isEven? : Nat -> Bool .

eq phi1(N) = (N <= 10) .
eq phi2(N) = isEven?(N) .

eq isEven?(0) = true .
eq isEven?(s(N)) = not(isEven?(N)) .
eq isEven?(N * 2) = true .   --- redundant, but necessary for ITP
```

Notice the last equation used in the specification of `isEven?`. In order to prove the implications needed to construct the abstract predicate, the tool calls the ITP theorem prover [3]. Like with all similar provers, it could happen that a certain formula is valid but nevertheless the ITP fails to prove it. Actually, this situation is not uncommon, and results in coarser abstract predicates because some of the relations among the Boolean variables are not discovered. But in general the ITP does a decent job, and sometimes when it fails it can be helped with additional equations like the one for `isEven?`.

Once the module containing the specification of both the system and the predicates is loaded into Maude's database, we can run the tool and obtain the abstract module with the help of the function

```
op computeAbsModule : Qid QidList -> Module .
```

This function takes as its arguments the module's name preceded by a quote and a list of quoted identifiers with the names of the predicates used for the abstraction, and returns the metarepresentation of the abstract module containing the specification of the predicates that define the abstract transition relation.

```
Maude> red computeAbsModule('SD-EXAMPLE, 'phi1 'phi2) .

result FModule: fmod 'SD-EXAMPLE-ABS is
  including 'BOOL .
  sorts none .
  none
```

```
  op 'absInit : 'Bool 'Bool -> 'Bool [none] .
  op 'lambda1 : 'Bool 'Bool 'Bool 'Bool -> 'Bool [none] .
  op 'lambda2 : 'Bool 'Bool 'Bool 'Bool -> 'Bool [none] .
  op 'lambda3 : 'Bool 'Bool 'Bool 'Bool -> 'Bool [none] .
  none
  eq 'absInit['B1:Bool,'B2:Bool] = '_and_['B1:Bool,'B2:Bool] [none] .
  eq 'lambda1['B1:Bool,'B2:Bool,'B*1:Bool,'B*2:Bool] = '_and_['B*2:Bool,'_and_[
    '_implies_['B1:Bool,'B*2:Bool],'_and_['_implies_['B2:Bool,'B*2:Bool],
    '_and_['_implies_['not_['B1:Bool],'B*2:Bool],'_and_['_implies_['not_[
    'B1:Bool],'not_['B*1:Bool]],'_implies_['not_['B2:Bool],'B*2:Bool]]]]]] [
    none] .
  eq 'lambda2['B1:Bool,'B2:Bool,'B*1:Bool,'B*2:Bool] = '_and_['_implies_[
    'B1:Bool,'B*1:Bool],'_and_['_implies_['B2:Bool,'B*2:Bool],'_implies_['not_[
    'B2:Bool],'not_['B*2:Bool]]]] [none] .
  eq 'lambda3['B1:Bool,'B2:Bool,'B*1:Bool,'B*2:Bool] = '_and_['B1:Bool,'_and_[
    'B*1:Bool,'_and_['_implies_['B1:Bool,'B*1:Bool],'_and_['_implies_['B2:Bool,
    'B*1:Bool],'_and_['_implies_['B2:Bool,'not_['B*2:Bool]],'_and_['_implies_[
    'not_['B1:Bool],'B*1:Bool],'_and_['_implies_['not_['B1:Bool],'B*2:Bool],
    '_and_['_implies_['not_['B1:Bool],'not_['B*1:Bool]],'_and_['_implies_[
    'not_['B1:Bool],'not_['B*2:Bool]],'_and_['_implies_['not_['B2:Bool],
    'B*1:Bool],'_implies_['not_['B2:Bool],'B*2:Bool]]]]]]]]]] [none] .
endfm
```

To ease its understanding, we also include the object module that corresponds to the above metarepresentation.

```
fmod SD-EXAMPLE-ABS is
  including BOOL .
  op absInit : Bool Bool -> Bool .
  op lambda1 : Bool Bool Bool Bool -> Bool .
  op lambda2 : Bool Bool Bool Bool -> Bool .
  op lambda3 : Bool Bool Bool Bool -> Bool .
  vars B1 B2 B*1 B*2 : Bool .

  eq absInit(B1,B2) = B1 and B2 .
  eq lambda1(B1,B2,B*1,B*2) = B*2 and (B1 implies B*2) and (B2 implies B*2) and
                              (not(B1) implies B*2) and
                              (not(B1) implies not(B*1)) and
                              (not(B2) implies B*2) .
  eq lambda2(B1,B2,B*1,B*2) = (B1 implies B*1) and (B2 implies B*2) and
                              (not(B2) implies not(B*2:Bool)) .
  eq lambda3(B1,B2,B*1,B*2) = B1 and B*1 and (B1 implies B*1) and
                              (B2 implies B*1) and (B2 implies not(B*2)) and
                              (not(B1) implies B*1) and (not(B1) implies B*2) and
                              (not(B1) implies not(B*1)) and
                              (not(B1) implies not(B*2)) and
                              (not(B2) implies B*1) and
                              (not(B2) implies B*2) .
endfm
```

Note that the initial abstract state is also expressed as a predicate. Also, due to the way Maude internally handles the metarepresentation of modules, the order of the predicates in the abstract system does not follow that of the rules in the original one; here, note that it is `lambda3` that corresponds to the first rule.

4.2 Two more functions

The function `computeAbsModule` solves the problem of computing the abstract system but presents the important drawback of not being directly executable: transitions are not expressed by means of rules. To remedy this situation the tool includes the following two functions:

```
op abstractionGround : Qid QidList -> Module .
op abstractionGen : Qid QidList -> Module .
```

Both receive the same arguments as the function `computeAbsModule`, which is invoked in a first step. Then, the function `abstractionGround` enumerates all possible tuples

$$\langle b_1, \ldots, b_n, b'_1, \ldots, b'_n \rangle$$

of Boolean values and checks, for each of them, whether they satisfy any of the predicates that define the abstract transition relation. It returns the metarepresentation of a new module in which every such satisfying tuple has been transformed into a ground rule

$$\langle b_1, \ldots, b_n \rangle \longrightarrow \langle b'_1, \ldots, b'_n \rangle$$

Besides, this module also includes a rule that defines the value of the constant `initial` representing the initial state.

For our running example, we get:

```
Maude> red abstractionGround('SD-EXAMPLE, 'phi1 'phi2) .

result Module: mod 'SD-EXAMPLE-ABS-RULES is
  including 'BOOL .
  sorts 'AbsState .
  none
  op 'initial : nil -> 'AbsState [none] .
  op 'st : 'Bool 'Bool -> 'AbsState [none] .
  none
  none
  rl 'initial.AbsState => 'st['true.Bool,'true.Bool] [none] .
  rl 'st['false.Bool,'false.Bool] => 'st['false.Bool,'false.Bool] [none] .
  rl 'st['false.Bool,'false.Bool] => 'st['false.Bool,'true.Bool] [none] .
  rl 'st['false.Bool,'false.Bool] => 'st['true.Bool,'false.Bool] [none] .
  rl 'st['false.Bool,'true.Bool] => 'st['false.Bool,'true.Bool] [none] .
  rl 'st['false.Bool,'true.Bool] => 'st['true.Bool,'true.Bool] [none] .
  rl 'st['true.Bool,'false.Bool] => 'st['false.Bool,'true.Bool] [none] .
  rl 'st['true.Bool,'false.Bool] => 'st['true.Bool,'false.Bool] [none] .
  rl 'st['true.Bool,'false.Bool] => 'st['true.Bool,'true.Bool] [none] .
  rl 'st['true.Bool,'true.Bool] => 'st['false.Bool,'true.Bool] [none] .
  rl 'st['true.Bool,'true.Bool] => 'st['true.Bool,'false.Bool] [none] .
  rl 'st['true.Bool,'true.Bool] => 'st['true.Bool,'true.Bool] [none] .
endm
```

Therefore, the result returned by `abstractionGround` is already an executable Maude (meta-)specification. Note, however, that the procedure is very expensive since 2^{2n} Boolean combinations are tried, where n is the number of predicates used for the abstraction.

The more economical alternative offered by `abstractionGen` consists in transforming the functional module returned by `computeAbsModule` into a system module and adding for each

predicate λ a single rule of the form

$$\langle b_1, \ldots, b_n \rangle \longrightarrow \langle b'_1, \ldots, b'_n \rangle \textbf{ if } cBool \longrightarrow b'_1 \wedge \ldots \wedge cBool \longrightarrow b'_n \wedge \lambda(b_1, \ldots, b_n, b'_1, \ldots, b'_n)$$

where *cBool* is a constant that can be rewritten to `true` and `false`. This way, the actual generation of the transitions is postponed until the phase of exploration of the state graph and the overall performance can be expected to improve because many of the rules produced previously only apply to unreachable states and are thus unnecessary.

For our particular example, `abstractionGen` extends the specification on page 4 with the rules:

```
rl 'cBool.Bool => 'false.Bool [none] .
rl 'cBool.Bool => 'true.Bool [none] .
crl 'initial.AbsState => 'st['B*1:Bool,'B*2:Bool] if 'cBool.Bool => 'B*1:Bool
  /\ 'cBool.Bool => 'B*2:Bool /\ 'absInit['B*1:Bool,'B*2:Bool] = 'true.Bool [
  none] .
crl 'st['B1:Bool,'B2:Bool] => 'st['B*1:Bool,'B*2:Bool] if 'cBool.Bool =>
  'B*1:Bool /\ 'cBool.Bool => 'B*2:Bool /\ 'lambda1['B1:Bool,'B2:Bool,
  'B*1:Bool,'B*2:Bool] = 'true.Bool [none] .
crl 'st['B1:Bool,'B2:Bool] => 'st['B*1:Bool,'B*2:Bool] if 'cBool.Bool =>
  'B*1:Bool /\ 'cBool.Bool => 'B*2:Bool /\ 'lambda2['B1:Bool,'B2:Bool,
  'B*1:Bool,'B*2:Bool] = 'true.Bool [none] .
crl 'st['B1:Bool,'B2:Bool] => 'st['B*1:Bool,'B*2:Bool] if 'cBool.Bool =>
  'B*1:Bool /\ 'cBool.Bool => 'B*2:Bool /\ 'lambda3['B1:Bool,'B2:Bool,
  'B*1:Bool,'B*2:Bool] = 'true.Bool [none] .
```

4.3 Proving properties

Now it is due time to think again of the question that motivated all the discussion in the previous sections, namely, whether the state `151` is reachable from the initial `0`. To prove that this is not the case it is enough to show that for all reachable states $\langle b_1, b_2 \rangle$ in the abstract system, either b_1 or b_2 is equal to `true`. This can be done by applying the `metaSearch` command to the result returned by either `abstractionGround` or `abstractionGen`, to search for a state $\langle b_1, b_2 \rangle$ such that $b_1 \vee b_2 =$ `false`. By the way the abstract transition relation has been defined (see page 1) and the fact that our approximation contains it, this would prove that all reachable numbers in the original system are either less than or equal to `10`, or even; in particular, `151` would not be reachable.

```
Maude> red metaSearch(abstractionGround('SD-EXAMPLE, 'phi1 'phi2),
                      'initial.AbsState,
                      'st['B1:Bool,'B2:Bool],
                      '_or_['B1:Bool,'B2:Bool] = 'false.Bool,'*,unbounded,0) .

result ResultTriple?: (failure).ResultTriple?
```

What the `metaSearch` operation does is to search, in the abstract system returned by `abstractionGround`, for a state `st[B1, B2]` reachable from `initial` in which the condition `B1 or B2` is false. The result obtained, `failure`, implies the negation of the condition for all reachable states: this establishes that it is always the case that either b_1 or b_2 holds.

4.4 Some examples

Readers and writers. To illustrate the use of the tool we first consider the following specification of a readers/writers system:

```
mod R&W is
  protecting NAT .
  sort Config .
  op <_,_> : Nat Nat -> Config .  --- 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
```

States are represented by pairs `< R, W >` that indicate the number `R` of readers and the number `W` of writers accesing the critical resource. Readers and writers can leave the resource at any time, but writers can only access it if none else is using it, and readers only whenever there are no writers.

We wish to prove that there are never both readers and writers simultaneously using the resource, and never more than a writer. Using this as a guide, we come up with the following predicates:

```
  ops phi1 phi2 phi3 : Config -> Bool .

  eq phi1(< R, W >) = (R <= 0) .
  eq phi2(< R, W >) = (W <= 0) .
  eq phi3(< R, W >) = (W < 2) .
```

The purpose of the first two predicates is to recognize if the natural numbers `R` and `W` are equal to `0`, and the operator `_<=_` (which can be handled by the ITP's decision procedures) is used for that instead of the double equal `_==_`. This is necessary because this last operator is not a logical one but predefined in Maude and returns unsound results when applied to non-ground terms. Therefore, its use in conjunction with the ITP to prove arbitrary formulas can lead to false results.

Once the predicates for the abstraction are specified together with the system, the property can be checked by means of the following command:

```
Maude> red metaSearch(abstractionGen('R&W, 'phi1 'phi2 'phi3),
                      'initial.AbsState,
                      'st['B1:Bool,'B2:Bool, 'B3:Bool],
                      '_and_['_or_['B1:Bool,'B2:Bool],
                             'B3:Bool] = 'false.Bool, '+, unbounded, 0) .
```

The initial state in the abstract system returned by `abstractionGen` is also called `initial`, and `metaSearch` checks whether it is possible to reach a state `st[B1, B2, B3]` in which the condition `(B1 or B2) and B3` is false. That is, `metaSearch` tries to find a state in which neither of the predicates hold (so that there would be both readers and writers in the system) or in which the third one is false (so that there would be at least two writers). The result

```
result ResultTriple?: (failure).ResultTriple?
```

shows that it is not possible.

The bakery protocol. The bakery protocol [6] is an infinite state protocol that achieves mutual exclusion between processes competing for a critical resource; processes are assigned a number and are attended sequentially starting with that with the least number. A Maude specification for the case of two processes is as follows:

```
mod BAKERY is
  protecting NAT .
  sorts Mode State .

  ops sleep wait crit : -> Mode .
  op <_,_,_,_> : Mode Nat Mode Nat -> State .
  op initial : -> State .

  vars P Q : Mode .
  vars X Y : Nat .

  eq initial = < sleep, 0, sleep, 0 > .

  rl [p1_sleep] : < sleep, X, Q, Y > => < wait, s Y, Q, Y > .
  rl [p1_wait] : < wait, X, Q, 0 > => < crit, X, Q, 0 > .
  crl [p1_wait] : < wait, X, Q, Y > => < crit, X, Q, Y > if not (Y < X) .
  rl [p1_crit] : < crit, X, Q, Y > => < sleep, 0, Q, Y > .
  rl [p2_sleep] : < P, X, sleep, Y > => < P, X, wait, s X > .
  rl [p2_wait] : < P, 0, wait, Y > => < P, 0, crit, Y > .
  crl [p2_wait] : < P, X, wait, Y > => < P, X, crit, Y > if Y < X .
  rl [p2_crit] : < P, X, crit, Y > => < P, X, sleep, 0 > .
endm
```

States are tuples `<_,_,_,_>`; the first two components describe the condition of the first process (its current mode and its priority, as given by its assigned number) and the remaining two the condition of the second. The rewrite rules (four for each process) describe how a process evolves from `sleep` to `wait`, from waiting to the critical resource (`crit`) and back to sleep.

We use the following predicates for the abstraction:

```
  eq phi1(< P, X, Q, Y >) = equal(P, wait) .
  eq phi2(< P, X, Q, Y >) = equal(P, crit) .
  eq phi3(< P, X, Q, Y >) = equal(Q, wait) .
  eq phi4(< P, X, Q, Y >) = equal(Q, crit) .
  eq phi5(< P, X, Q, Y >) = (X <= 0) and (0 <= X) .
  eq phi6(< P, X, Q, Y >) = (Y <= 0) and (0 <= Y) .
  eq phi7(< P, X, Q, Y >) = (Y < X) .
```

The fact that two processes cannot be at the same time in the critical section is proved by checking that no state in which the second and fourth components are `true` is reachable.

```
Maude> red metaSearch(
               abstractionGen('BAKERY,'phi1 'phi2 'phi3 'phi4 'phi5 'phi6 'phi7),
               'initial.AbsState,
               'st['B1:Bool,'true.Bool,'B3:Bool,'true.Bool,'B5:Bool,'B6:Bool,
                   'B7:Bool],
               nil, '+, unbounded, 0) .
result ResultTriple?: (failure).ResultTriple?
```

So far, all properties illustrated have been *safety* properties that state that a certain undesirable state cannot be reached. Sometimes, however, we are interested in checking more general properties like *liveness*, that guarantee that if a certain state can be reached then eventually some other state will also be reached. In particular, this is the case for another property that the bakery protocol satisfies: a process in waiting mode will eventually access the critical resource.

The abstract system returned by the prototype can also be used to check if these kinds of properties hold in the original one, but that requires the use of the model checker instead of simply the function `metaSearch`. This gives rise to a problem, since the model checker works with modules at the object level whereas the prototype returns metarepresentations. Currently, the prototype does not support this situation; the simplest solution consists in manually editing the abstract module and modifying the representation (essentially, removing the backquotes and replacing brackets with parenthesis) until a module at the object level is obtained.

In our running example, the representation at the object level of the module returned by `abstractGround` is

```
mod BAKERY-ABS-RULES is
  including BOOL .
  sorts AbsState .

  op initial : -> AbsState  .
  op st : Bool Bool Bool Bool Bool Bool Bool -> AbsState  .

  rl initial => st(false,false,false,false,true,true,false)  .
  rl st(false,false,false,false,false,false,false) =>
     st(false,false,true,false,false,false,false)  .
  rl st(false,false,false,false,false,false,false) =>
     st(true,false,false,false,false,false,true)  .
  ...
endm
```

It can then be extended with the definitions for the state predicates needed to express the liveness property:

```
mod BAKERY-ABS-RULES-CHECK is
  inc BAKERY-ABS-RULES .
  inc MODEL-CHECKER .
  subsort AbsState < State .

  ops 1wait 1crit 2wait 2crit : -> Prop .

  vars B1 B2 B3 B4 B5 B6 B7 : Bool .

  eq st(B1,B2,B3,B4,B5,B6,B7) |= 1wait = B1 .
  eq st(B1,B2,B3,B4,B5,B6,B7) |= 1crit = B2 .
  eq st(B1,B2,B3,B4,B5,B6,B7) |= 2wait = B3 .
  eq st(B1,B2,B3,B4,B5,B6,B7) |= 2crit = B4 .
endm
```

Finally, the property is proved by using the following command:

```
Maude> red modelCheck(initial, (1wait |-> 1crit) /\ (2wait |-> 2crit)) .
result Bool: true
```

5 Implementation and technicals observations

Though we are not going to describe the implementation in detail since most of it is uninteresting, we do believe it appropriate to give a brief overview of the tool's design and main functions. To begin with, the prototype mainly deals with with metaobjects and makes extensive use of reflection through the `META-LEVEL` module. Also, as explained below, it relies on the ITP theorem prover and the Maude's model checker.

As mentioned before, to prove the validity of formulas the prototype uses the ITP theorem prover. Logical inference in the ITP is specified by means of rules, so it is not immediately available to be used in equational definitions; a hack out of this problem is to call Maude's model checker function `modelCheck`, that allows the integration of rule-based functions into equational definitions. This is the reason why the prototype loads the file `model-checker.maude` in addition to `itp-tool.maude`; this dependency may be removed in the future.

The main function behind the computation of the abstraction is `computeAbsModule`, which is then used by `abstractionGround` and `abstractionGen`.

```
op computeAbsModule : Qid QidList -> Module .
eq computeAbsModule(QMod, QIL) =
         (fmod qid(string(QMod) + "-ABS") is
            including 'BOOL .
            sorts none .
            none
            computeOps(cardRS(getRls(upModule(QMod, false))),
                       lengthQidList(QIL))
            none
            computeInit(QMod,QIL)
            computeRules(QMod,QIL)
          endfm) .
```

Given the names of the module to be abstracted and the predicates to be used, this function builds the metarepresentation of the abstract module with the help of three auxiliary functions in charge, respectively, of declaring the module's operators, computing the initial abstract state, and abstracting the rules. This last task is performed by `computeRules`; after a series of transformations, it calls the function `abstractRule` for each rule `Rl` in the module `QMod` being abstracted.

```
op abstractRule : Qid QidList Rule AbsSpaceList Nat -> Equation .
ceq abstractRule(QMod, QIL, Rl, ASL, N) =
    (eq qid("lambda" + string(N,10))
           [generateBoolList("B",M), generateBoolList("B*",M)] =
        abs2MetaAbsVar(abstractRuleAux(QMod, QIL, computePreFormula(Rl),
                                       computePreVariables(Rl),ASL)) [none] .)
    if M := lengthQidList(QIL) .
```

Again, the specific details are irrelevant. Note that this function builds the skeleton of the abstract rule, naming it with the corresponding `lambda` label (see the output on page 4.1). It also generates the list of its arguments before calling the function `abstractRuleAux` which, together with `tryRulePredicate`, is really responsible of the bulk of the job.

```
op abstractRuleAux : Qid QidList Formula VarList AbsSpaceList -> AbsSpace .
eq abstractRuleAux(QMod, QIL, F, VL, AS) =
   tryRulePredicate(QMod, QIL, F, VL, AS) .
eq abstractRuleAux(QMod, QIL, F, VL, (AS, ASL)) =
```

```
    aAnd(tryRulePredicate(QMod, QIL, F, VL, AS),
         abstractRuleAux(QMod, QIL, F, VL, ASL)) .
```

Finally, the function `abstractRuleAux` studies the possible relations among the Boolean variables in the predicate that defines the abstract rule. For each such relation, say $b_1 \rightarrow b_2'$, stored as part of the last argument, it calls `tryRulePredicate`, which in its turn invokes the ITP to check if the relation holds in the original module: if so, it is added to the predicate being built; otherwise, it returns `true`, so that no new information is added.

Logical inference is specified in the ITP by means of rewrite rules. Hence, it is not immediately available for use in equational definitions (the condition of an equation cannot include rewrites). A way out of the problem without the overhead that would imply the use of the metalevel and `metaSearch` consists in calling the function `modelCheck` of Maude's model checker. This is a built-in function that uses the module's rules for its computation and that can be employed in the equational specification of operations, thus allowing the integration of definitions based on rules. This is the reason why the prototype loads the file `model-checker.maude` besides `itp-tool.maude`; this dependency could be removed in the future.

The interaction between the ITP and the model checker, and actually the only place in the code where any of them is used (besides the function `tryInitPredicate` that similarly computes the initial abstract state) is reflected in the first of the two equations that define `tryRulePredicate`.

```
op tryRulePredicate : Qid QidList Formula VarList AbsSpace -> AbsSpace .
ceq tryRulePredicate(QMod, QIL, PreF, VL, AS) = AS
    if F := AQuantification('C@0:Config : 'C@1:Config : VL,
                implication(PreF,
                            abs2XitpPredicate(AS,QIL)))
       /\
       modelCheck(state(attrs(
               db : createNewGoalModule(QMod, 'abs$0),
               input : ('auto*‘..Input),
               output : nil,
               proofState : < prove("abs$0", 0, 0, 'abs$0, F, nilTermList) ;
                              nil ; lemma('abs, F, QMod, "abs$0") >,
               defaultGoal : "abs$0")),
           <> isProved?) .
eq tryRulePredicate(QMod, QIL, F, VL, AS) = aTrue [owise] .
```

The ITP works with states that store, in addition to other attributes, a database of modules (`db`), the command being executed (`input`), and information about the current goal (`proofState`) containing, in particular, the formula `F` being proved. Rewrite rules that implement the logic's proof calculus and the decision procedures are applied to these states; a formula is proved when a state in which the field `input` is `nilTermList` is reached. This is captures by the state predicate `isProved?`:

```
eq state(attrs(db : DB, input : nilTermList, output  : QIL,
               proofState : < emptyGoalSet ; PT ; L >,
               defaultGoal : ST, Atts))
   |= isProved? = true .
```

What the prototype does is to initialize the ITP's internal state with the name `QMod` of the module to be reasoned about, with the formula `F` we wish to prove (in an appropriate format), and with the strategy `auto` to be used for the proof. Next, it calls the model checker

to verify if a state in which `input` is `nilTermList` is reachable, that is, a state in which the formula has been proved.

Finally, let us take a look to the formulas the prototype sends to the ITP. As explained in Section 3, for each rule we try to uncover relations of the form $b_i \to b'_j$, meaning that if ϕ_i holds in the current state then ϕ_j also holds in the state reached by using such rule. Let us assume that the rule is of the form $(\forall X)\, t_1 \longrightarrow t_2$ **if** C; the formula sent to the ITP to check if $b_i \to b'_j$ holds is the universal quantification of the implication

$$x = t_1 \wedge y = t_2 \wedge C \to (\phi_i(x) \to \phi_j(y))\,.$$

More precisely, the conjunction $x = t_1 \wedge y = t_2 \wedge C$ is precomputed and sent to `tryRulePredicate` through the variable `PreF`, whereas the substitution of the predicates for the Boolean variables in the expression $b_i \to b'_j$ is performed by `abs2XitpPredicate`.

6 By way of conclusion

The tool makes heavy use of the ITP, and both its accuracy and efficiency depend critically on it. Obviously, the more valid formulas the ITP is able to prove, the better the abstract system will be. Similarly, the ITP is invoked $4mn(n+1)$ times, where m is the number of rules in the original module and n the number of predicates used in the abstraction. Hence, a high speed on the part of the ITP is crucial for the usefulness of the tool. Currently, computing the abstraction for the bakery protocol, a system with 8 rules and 7 predicates, took the tool almost 6 minutes in a 1.25Ghz G4; though manageable, this duration is already too high. The cause does not lie in the inherent complexity of the problem but is related to some difficulties with the ITP's implementation that trigger many more rewritings than necessary when proving a goal.

When compared to similar tools (for example, those described in [4, 5]), our prototype comes out as significantly modest. Despite this, we believe it is important to stress that its reflective design has allowed us to develop it very quickly and that, in any case and as the word "prototype" indicates, the goal was not to achieve the highest performance possible but to experiment with predicate abstraction in Maude. We hope to be able to use the experience gained to improve the ITP and develop a more modern version of the present tool comparable to other advanced research tools.

Acknowledgments. I warmly thank Manuel Clavel for all his help with the ITP.

References

[1] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[2] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. Maude manual (version 2.1.1). `http://maude.cs.uiuc.edu/manual/`, 2005.

[3] Manuel Clavel and Miguel Palomino. A quick ITP tutorial. In Francisco López-Fraguas, editor, *PROLE 2005*, 2005. `http:/maude.sip.ucm.es/~miguelpt/bibliography.html`.

[4] Michael A. Colón and Tomás E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification. 10th International Conference, CAV'98, Vancouver, BC, Canada,*

June 28-July 2, 1998, Proceedings, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer-Verlag, 1998.

[5] Satyaki Das. *Predicate Abstraction*. PhD thesis, Department of Electrical Engineering, Stanford University, December 2003.

[6] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[7] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[8] Tomás E. Uribe Restrepo. *Abstraction-Based Deductive-Algorithmic Verification of Reactive Systems*. PhD thesis, Department of Computer Science, Stanford University, December 1998.