Model checking parameterized by the semantics in Maude

Adrián Riesco

Universidad Complutense de Madrid, Madrid, Spain

FLOPS 2018, Nagoya
May 10th, 2018

Motivation

- Model checking is an automatic technique for checking whether a property, usually stated in modal logic, holds in a system.
- It starts from an initial state and exhaustively traverses all the reachable states.
- It is a useful verification tool for concurrent systems.
- It helps detecting complex interleaving failures might be overlooked during the implementation and testing phases.
- State-of-the-art model checkers usually check *models* of the systems.
- The translation from the real code to the model is an important issue.

Motivation

- Maude is a high-performance logical framework where the semantics of other programming languages can be specified and analyzed.
- Maude modules correspond to specifications in *rewriting logic*.
- Rewriting logic allows specifiers to represent many models of concurrent and distributed systems.
- This logic is an extension of *membership equational logic* (MEL).
- MEL supports, in addition to equations, the statement of *membership axioms* characterizing the elements of a sort.

Motivation

- Rewriting logic extends membership equational logic by adding *rewrite rules*.
- These rules represent transitions in a concurrent system and can be nondeterministic.
- An important feature of rewriting logic is that it is *reflective*
- It can be faithfully interpreted in terms of itself.
- This feature is efficiently implemented in Maude by means of the `META-LEVEL` module.
- This allows us to use Maude modules and terms as usual data.

Motivation

- Defining the semantics of a programming language in Maude presents advantages:
  - Maude specifications are executable, so the specification gives the specifier an interpreter of the semantics for free.
  - Maude provides several analysis tools, including an LTL model checker.
- Maude has been used to specify the semantics of many languages, such as CCS, Lotos, and Java.
- The $\mathbb{K}$-Maude compiler is able to translate $\mathbb{K}$ specifications into Maude.
- It has eased the methodology to describe programming language semantics in Maude, as shown for the C semantics.

Motivation

- In Maude, we can:
  - Define the semantics of programming languages.
  - Execute programs written in these languages.
  - Model check systems.
- If we put all together, we can model check programs whose semantics has been specified in Maude.
- However, we obtain a counterexample that refers to the semantics of the language but not directly to the actual program under analysis.
- This extra layer of complexity makes the counterexample even more difficult to understand in real applications.

Motivation

- We present two generic transformations for relating the counterexample generated by the Maude model checker with the semantics of the language.
- These transformations can be applied to concurrent programs following either a message-passing approach or a shared-memory approach.
- They:
  - Reduce the counterexample.
  - Focus on the main events depending on the semantics.
  - Return a JSON-like result.
- Maude specifiers get, in addition to an interpreter for their language, a model checker for the object language for free.
- They also get a model checker for real code for all those programming languages that are already specified in Maude.

Outline

- We first present how these semantics can be specified in Maude.
- We introduce some simple examples for each semantics.
- We present the counterexamples for each program.
- We present the transformation for improving these counterexamples.
- We focus on *what* we are solving, details about *how* to do it are discussed in the paper.

Syntax in Maude

- Given a grammar:

$$
\begin{array}{lcl}
\mathit{Ins} & ::= & \mathit{Var} := \mathit{Exp} \\
 & | & \texttt{If } \mathit{Cond} \texttt{ then } \mathit{Ins} \texttt{ fi} \\
 & | & \mathit{Ins} \ ; \ \mathit{Ins} \ | \ \ldots \\
\mathit{Exp} & ::= & \mathit{Exp} + \mathit{Exp} \\
 & | & \mathit{Exp} * \mathit{Exp} \ | \ \ldots \\
\mathit{Cond} & ::= & \texttt{true} \mid \texttt{false} \mid \ldots \\
\ldots & &
\end{array}
$$

- A parser is built by first defining sorts for each non-terminal symbol.
- Then, we must define an operator for each production rule.
- When a rule just maps symbols, subsorting is enough.
- Maude mixfix syntax allows us to define the real syntax of each particular programming language.

Semantics in Maude

- Semantics are represented in Maude by means of conditional *rewrite rules*.
- Inference rules have the form:

$$\frac{P_1 \quad \ldots \quad P_n}{state_1 \Rightarrow state_2}\ id$$

  in the semantics, which indicates that $state_2$ is reached from $state_1$ if the premises $P_1 \ \ldots \ P_n$ hold
- The corresponding rewrite rules are written in Maude as:

```
crl [id] : state1 => state2 if P1 /\ ... /\ Pn .
```

  where the conditions `Pi` can be either equalities or rewrite conditions.
- In general we have small step semantics, because programs might do not terminate and the distributed nature of our process prevents the complete execution of any of them.

Shared-memory semantics

- We use a simple imperative language as running example.
- This language includes:
  - Assignments `X := E`.
  - Sequential composition `INS ; INS'`.
  - Conditional statements `if COND then INS fi`.
  - Loops `while COND do INS od` and `repeat INS forever`.

Shared-memory semantics

- Processes executing programs written with this syntax are wrapped into processes of the form `[ID, Prog]`.
- `PS` is a set of processes put together by using `P | P'`.
- `M` is the memory, which consists of a set of pairs `[V1, N1]...[Vn, Nn]`.
- The whole system is a pair of the form `[PS, M]`
- We assume all variables in the system are initialized beforehand.

Shared-memory semantics

- The semantics of this system is defined by using rewrite rules for each instruction.
- The rule `asg` executes an assignment in a process.
- The process `I` has `Q := N` as the first instruction.
- The memory contains the pair `[Q, X]`.
- Then the instruction is executed by updating the value of the variable from `X` to `N`:

```
rl [asg] : {[I, Q := N ; R] | S, [Q, X] M}
=>         {[I, R]           | S, [Q, N] M} .
```

- Note that this is a small-step semantics and hence `N` is completely evaluated.

Shared-memory semantics

- Similarly, a `repeat` puts the body of the loop before repeating the instruction:

```
rl [repeat] : {[I, repeat P forever ; R] | S, M}
=>            {[I, P ; repeat P forever ; R] | S, M} .
```

- Using this syntax, we describe the verification of the Dekker algorithm.
- It ensures mutual exclusion by making each process actively wait for its turn.
- This turn is indicated by a variable that is only changed by the process exiting the critical section.

Shared-memory semantics

- We present a simplification for the second process.
- A bug has been introduced, hence violating the mutual exclusion property.

```
repeat
   c1 := 1 ; *** It should be c2 := 1 ;
   while c1 = 1 do
     if turn = 1 then
       c2 := 0 ;
       while turn = 1 do skip od ;
       c2 := 1
     fi
   od ;
   cs2 := 1 ; *** start critical section for process 2
   cs2 := 0 ; *** end critical section for process 2
   turn := 1 ;
   c2 := 0
forever
```

Shared-memory semantics

- Hence, given:
  - The initial state {`[1,prog1] | [2,prog2], [c1,0][c2,0][cs1,0][cs2,0][turn,1]`}.
  - An atomic formula `enterCrit` that holds when the variable given as argument (either `cs1` or `cs2`) has value `1` (i.e., the corresponding process is in the critical section).
  - A formula `[]~ (enterCrit(cs1) /\ enterCrit(cs2))`.
- We check whether mutual exclusion holds.

Shared-memory semantics

```
red modelCheck(initial, []~ (enterCrit(cs1) /\ enterCrit(cs2))) .
result ModelCheckResult: counterexample({{[1,repeat c1 := 1 ;
while c2 = 1 do if turn = 2 then c1 := 0 ; while turn = 2 do skip
od ; c1 := 1 fi od ; cs1 := 1 ; cs1 := 0 ; turn := 2 ; c1 := 0 forever]
| [2,repeat c1 := 1 ; while c1 = 1 do if turn = 1 then c2 := 0 ;
while turn = 1 do skip od ; c2 := 1 fi od ; cs2 := 1 ; cs2 := 0 ;
turn := 1 ; c2 := 0 forever],[c1,0] [c2,0] [cs1,0] [cs2,0] [turn,1]},
'repeat}
{{[1,c1 := 1 ; while c2 = 1 do if turn = 2 then c1 := 0 ; while turn = 2
do skip od ; c1 := 1 fi od ; cs1 := 1 ; cs1 := 0 ; turn := 2 ; c1 := 0 ;
repeat c1 := 1 ; while c2 = 1 do if turn = 2 then c1 := 0 ; while turn = 2
do skip od ; c1 := 1 fi od ; cs1 := 1 ; cs1 := 0 ; turn := 2 ; c1 := 0
forever] | [2,repeat c1 := 1 ; while c1 = 1 do if turn = 1 then c2 := 0 ;
while turn = 1 do skip od ; c2 := 1 fi od ; cs2 := 1 ; cs2 := 0 ; turn := 1
; c2 := 0 forever],[c1,0] [c2,0] [cs1,0] [cs2,0] [turn,1]},'asg}, ...)
```

Shared-memory semantics

- This counterexample is difficult to follow:
  - As a minor issue, because of the presentation.
  - As a major issue, because it gives information that it is useful from the Maude point of view but not from the programming language point of view.
- For example, the user might not be interested in the steps involving the `repeat` rule, since it does not modify the memory.

Message-passing semantics

- We consider for our message-passing semantics a simple functional language that supports:
  - `let` expressions.
  - Conditional expressions
  - Basic arithmetic and Boolean operations.
  - `to ID : M` expressions for sending messages.
  - `receive` expressions for receiving them.
- Processes have the form `[ID | E | ML]`, with:
  - `ID` a natural number identifying the process.
  - `E` the expression being evaluated.
  - `ML` a list of natural numbers standing for the messages received thus far.
- The whole system is a term of the form `|| PS, D ||`.

Message-passing semantics

- We have rules of the form `D, ro |- e => e'` for simplifying expressions, given:
  - A set of declarations `D`.
  - An environment `ro`.
  - Expressions `e` and `e'`.
- Rule `Let1` shows how the expression `e` in a `let` expression is simplified.
- Rule `Let2` applies the appropriate substitution when a value has been obtained for the variable.

```
crl [Let1] : D,ro |- let x = e in e' => let x = e'' in e'
 if D,ro |- e => e'' .

rl [Let2] : D,ro |- let x = v in e' => e'[v / x] .
```

Message-passing semantics

- We use rules at the process level to model how messages are sent and received.
- Rule `send` shows how a message is introduced into the list of received messages of `id'`.
- Value `1` is used to indicate that the message was delivered correctly.
- Rule `receive` is in charge of consuming messages.
- It substitutes a `receive` expression by the first message in the list.

```
rl [send] :
   || [id | let x = (to id' : n) in e | nl] [id' | e' | nl'] ps , D ||
=> || [id | let x = 1 in e | nl] [id' | e' | nl' . n] ps , D || .

rl [receive] :
   [ id | let x = receive in e | n . nl]
=> [ id | let x = n in e | nl] .
```

Message-passing semantics

- We will use a simple synchronization protocol between a server and two clients to illustrate how the model checker behaves in this case.
- We have the following initial state, with the server identified by 0 and the clients by 1 and 2.
- The server receives the client identifiers as arguments.
- The clients receive the server identifier:

```
|| [0 | server(1, 2) | nilML]
   [1 | client(0) | nilML]
   [2 | client(0) | nilML], decs ||
```

Message-passing semantics

- The declarations `decs` indicate that
  1. The server sends a message (`0`) to the process identified by the first argument (client `1`).
  2. Another message (`1`) to the process identified by the second argument (`2`).
  3. Then waits for two messages and returns `1` if it receives `0` and `1` (in this order) and `0` otherwise.
  4. In turn, the client receives a message and just returns the same message to the server.

```
server(x, y) <= let a = to x : 0
                in let b = to y : 1
                   in let c = receive
                      in let d = receive
                         in If Equal(c, 0) And Equal(d, 1)
                            Then 1 Else 0 &
client(x)    <= let y = receive
                in let z = to x : y in z
```

Message-passing semantics

- A naïve user might expect messages from clients to be received in the same order as they were sent from the server.
- The final state would always be 1.
- We check it with the formula `<> [] finalValue(0, 1)`.

```
reduce in TEST : modelCheck(init, <> [] finalValue(0, 1)) .
result ModelCheckResult: counterexample({| [0 | server(1,2) | nilML]
[1 | client(0) | nilML] [2 | client(0) | nilML], client(x) <= let y =
receive in let z = to x : y in z & server(x,y) <= let a = to x : 0 in
let b = to y : 1 in let c = receive in let d = receive in If Equal(c, 0)
And Equal(d, 1) Then 1 Else 0 |,'advance-process}
{| [0 | let a = to 1 : 0 in let b = to 2 : 1 in let c = receive in
let d = receive in If Equal(c, 0) And Equal(d, 1) Then 1 Else 0 | nilML]
[1 | client(0) | nilML] [2 | client(0) | nilML], client(x)<= let y =
receive in let z = to x : y in z & server(x,y)<= let a = to x : 0 in
let b = to y : 1 in let c = receive in let d = receive in If Equal(c, 0)
And Equal(d, 1) Then 1 Else 0 |,'send}, ...)
```

Transformation for shared-memory semantics

- We rely in the following assumption: properties refer to memory states.
- We only need to keep those transitions in the original counterexample performed by rules that modify the memory.
- For example, for the previous program we will only keep those steps involving the `asg` rule.
- We consider this is a safe assumption, since in these systems the access to the shared resources is critical.

Transformation for shared-memory semantics

- We need some information from the user to transform the counterexample.
- We need the name of the sort used for processes.
- If processes have an id it can be indicated to make the transformed counterexample easier to read.
- We also require the sorts for the memory.
- In this way we can identify the rules modifying it and keep them in the result.

Transformation for shared-memory semantics

We display the following information:

- The process executed (field `unit`) when the rule is applied.
- If the process has an identifier it will be displayed in the `id` field.
- The whole system (field `system`) before the rewrite rule is applied.
- The state of the memory:
  - The state *before* applying the rule (field `memory-before`).
  - The state *after* applying it (field `memory-after`).
- The value of all atomic formulas before and after applying the rule (field `props`).
- For each atomic proposition in the formula we display its name, arguments, and how its value changed with the current rule.

Transformation for shared-memory semantics

- In our example, we would start by introducing `Memory` as the sort used for the memory.
- In turn, `Process` is the sort used for processes.
- We also indicate that the first argument for `Process` stands for the identifier.

```
Maude> (memory sorts Memory .)
Memory sorts introduced: Memory

Maude> (unit Process id 1 .)
Unit sort introduced: Process
It is identified by the 1 argument.
```

Transformation for shared-memory semantics

```
Maude> (shared memory analysis modelCheck(initial,
                          []~ (enterCrit(cs1) /\ enterCrit(cs2))) .)
...
{ id =  1,
  unit = ...,
  system = ...,
  memory-before = {[c1,1][c2,0][cs1,0][cs2,1][turn,1]},
  memory-after  = {[c1,1][c2,0][cs1,1][cs2,1][turn,1]},
  props  = [{name = enterCrit,
             args =[cs1],
             prop = false -> true},
            {name = enterCrit,
             args =[cs2],
             prop = true -> true}]
} ...
```

Transformation for message-passing semantics

- We designed two different ways to transform counterexamples for message-passing semantics.
- The first one summarizes the actions performed by the processes during the computation.
- The second one presents trace-like information with the main actions that took place.

Transformation for message-passing semantics

- In both cases we require the user to introduce:
  - The sort for the processes.
  - The argument standing for its identifier, if it exists.
  - The constructors for sending and consuming messages.
- The tool will use this information to identify those rules in charge of dealing with messages and to locate the processes and their identifiers, as well as the messages sent and consumed.

Transformation for message-passing semantics

- We denote as `summary` mode our first approach.
- This transformation presents the following information for each process:
  - Its identifier (`id` field).
  - Its final value (`value` field). Note that in some cases this value will not be a normal form, since some functions (e.g. servers) might be non-terminating.
  - The list of messages it has sent (`sent` field).
  - The list of messages it has consumed (`consumed` field).

Transformation for message-passing semantics

- We would indicate that processes are terms of sort `Process` and their identifier is its first argument.
- We also state `to_:_` as the instruction for sending messages and `receive` for the one consuming them.

```
Maude> (unit Process id 1 .)
Unit sort introduced: Process
It is identified by the 1 argument.

Maude> (msg creation to_:_ .)
Message creation operators introduced: to_:_
Maude> (msg consumption receive .)
Message consumption operators introduced: receive
```

Transformation for message-passing semantics

```
Maude> (msg passing analysis modelCheck(init, <> [] finalValue(0, 1)) .)

{processes =[
  {   id = 0,
      value = [0 | 0 | nilML],
      sent = [to 1 : 0, to 2 : 1],
      consumed = [1, 0]},
  {   id = 1,
      value = [1 | 1 | nilML],
      sent = [to 0 : 0],
      consumed = [0]},
  {   id = 2,
      value = [2 | 1 | nilML],
      sent = [to 0 : 0],
      consumed = [1]}
]}
```

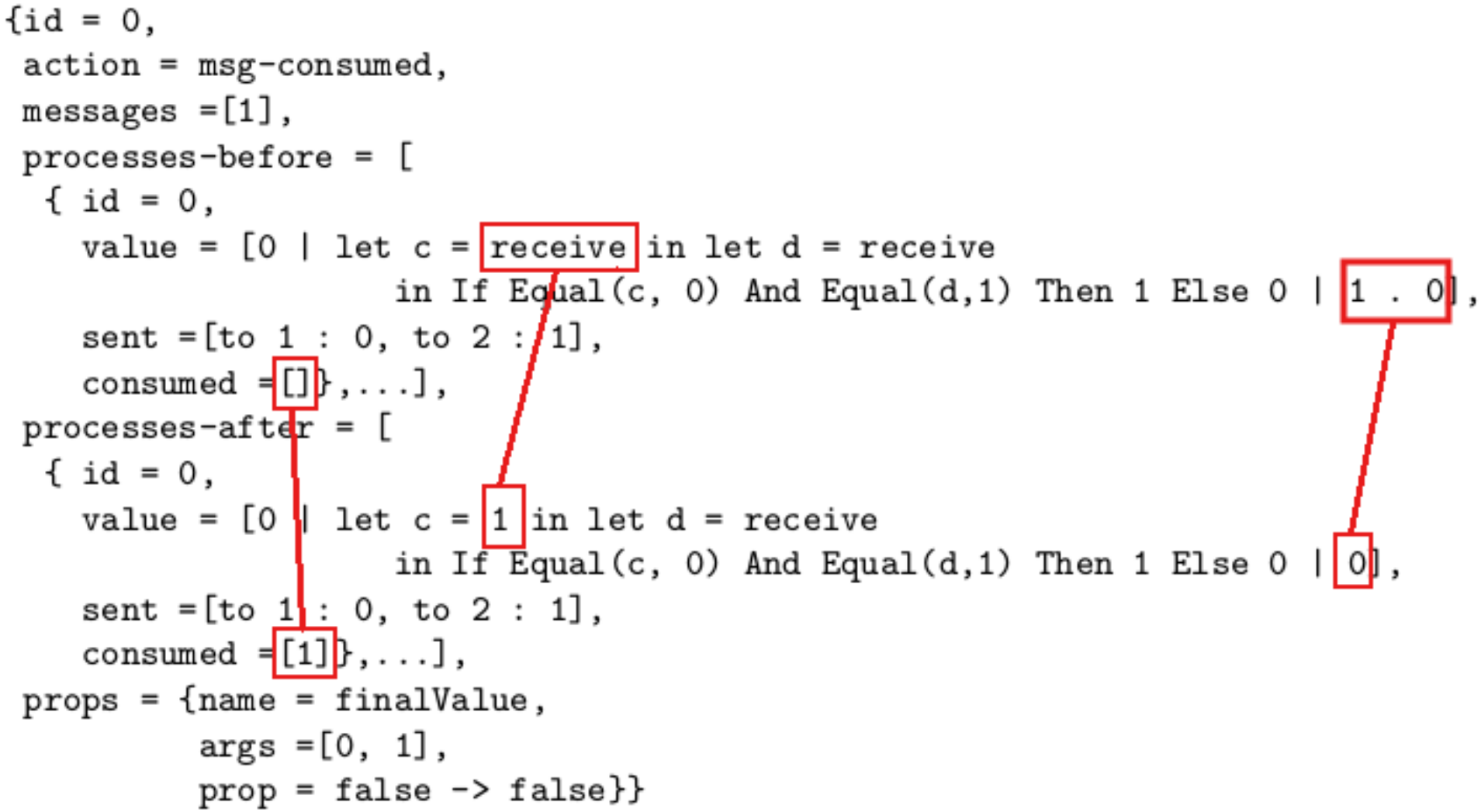
Transformation for message-passing semantics

- It might be useful to understand the interleaving between different messages and processes.
- We also present a trace-like counterexample, that we call `trace` mode.
- In this semantics is not clear the notion of *step*.
- We decided to focus on messages and display information when a message is sent or consumed.
- However, we noticed that some properties might change some steps after a message was sent or received.
- We include in the trace those steps where at least one atomic property changes its truth value.

Transformation for message-passing semantics

- In this approach each step contains the following information:
  - The identifier of the process that performed the action (`id` field).
  - The action that took place (`action` field):
    - `msg-consumed`,
    - `msg-sent`, or
    - `prop-changed`.
  - The messages involved in the action (`messages` field).
  - The state of all processes before and after applying the rule (`processes-before` and `processes-after` fields).
  - How the properties changed with the rewrite rule (`props` field).

Transformation for message-passing semantics



Conclusions

- We have presented two transformations that allow specifiers to model check real code and interpret the counterexamples obtained.
- These transformations are restricted to languages following either a shared-memory or a message passing approach.
- They have been implemented using Maude metalevel.
- Semantics extracted from Maude literature (references in the paper).
- This tool sets the basis for further development in this direction.

Ongoing work

- On the theoretical side, it is interesting to study how this approach relates to similar approaches, like partial evaluation transformations.
- On the tool side, it would be interesting to define transformations for other approaches.
- We are also interested in performing a pre-analysis of the semantics to infer information about the language and hence save time and work to the user.

Ongoing work

- It would be interesting to extend the tool with *slicing*.
- This technique keeps only those instructions related to the values reached by a set of variable of interest, to reduce the size of Maude traces.
- Regarding efficiency, it is possible to reduce the number of states when model checking Maude specifications.
- The transformation transforms rules into equations given some properties hold.

Ongoing work

- Some of these properties are the executability requirements, which can be proved in some cases using the Maude Formal Environment.
- The last property *invisibility*, which requires that the transformed rules do not change the truth value of the predicates.
- In our shared-memory model we would transform all those rules that do not modify the memory.
- Further assumptions on the message-passing approach would be required to ensure soundness.
- Overall, our long-term goal is to obtain a parameterized transformation for real languages, in the same way as Java PathFinder works for Java.
- In this sense we will probably need to generalize other aspects of the tool, so it deals with structures such as objects.