Model Checking Parameterized by the Semantics in Maude

Adrián Riesco(✉)

Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid, Madrid, Spain
ariesco@fdi.ucm.es

Abstract. Model checking is an automatic verification technique for analyzing whether some properties hold in a model. Maude is a high-performance logical framework and model checking tool where many different concurrent programming languages have been specified and analyzed. However, the counterexample generated by Maude when a property fails does not correspond to the language being specified but to the Maude rules, which makes it difficult to understand. In this paper we present two metalevel transformations for relating counterexamples and semantics when dealing with the semantics of concurrent languages, hence allowing users to model check real code while easing the interpretation of the counterexamples. These transformations can be applied to any semantics following a message-passing or a shared memory approach. These transformations have been implemented in a Maude prototype; we illustrate the tool with examples.

Keywords: Model checking · Semantics · Message passing
Shared memory · Maude

1 Introduction

Model checking [5] 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, which makes it a useful verification tool for concurrent systems, where complex interleaving failures might be overlooked during the implementation and testing phases. State-of-the-art model checkers, such as Spin [2] and NuSMV [4], allow users to analyze models of their algorithms, but they do not check the actual application code directly. For this reason, the relation between programs, models, and their corresponding translations are subjects of growing concern in the model-checking community, as shown for example by the Java PathFinder [13] community.

This research has been partially supported by the MINECO Spanish project *TRACES* (TIN2015-67522-C3-3-R) and by the Comunidad de Madrid project *N-Greens Software-CM* (S2013/ICE-2731).

J. P. Gallagher and M. Sulzmann (Eds.): FLOPS 2018, LNCS 10818, pp. 198–213, 2018.
https://doi.org/10.1007/978-3-319-90686-7_13

Maude [6] 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* [14], a logic that allows specifiers to represent many models of concurrent and distributed systems. This logic is an extension of *membership equational logic* [3], an equational logic that, in addition to equations, allows the statement of *membership axioms* characterizing the elements of a sort. Rewriting logic extends membership equational logic by adding rewrite rules that represent transitions in a concurrent system and can be nondeterministic. Moreover, an important feature of rewriting logic is that it is reflective, that is, it can be faithfully interpreted in terms of itself. This feature is efficiently implemented in Maude by means of the `META-LEVEL` module [6, Chap. 14], which allows us to use Maude modules and terms as usual data.

Defining the semantics of a programming language in Maude presents many advantages over other languages: first, Maude specifications are executable, so the specification gives the specifier an interpreter of the semantics for free. Moreover, Maude provides several analysis tools, including an LTL model checker. Hence, since the seminal proposal of using rewriting logic as a semantic framework in [15], Maude has been used to specify the semantics of many languages, such as LOTOS [19], CCS [19], and Java [10]. Moreover, the $\mathbb{K}$-Maude compiler [18], which is able to translate $\mathbb{K}$ [17] specifications into Maude, has eased the methodology to describe programming language semantics in Maude, as shown e.g. by the C semantics in [9]. However, when using the model checker for checking properties on programs whose semantics have been defined in Maude, we obtain a counterexample that refers to the semantics of the language but not directly to the actual program under analysis. Given the complex nature of concurrent systems, this extra layer of complexity makes the counterexample even more difficult to understand in real applications, preventing specifiers from understanding the error and being able to fix it.

We present in this paper two generic transformations, implemented using Maude metalevel, for relating the counterexample generated by the Maude model checker with the semantics of the language being executed. 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, and return a JSON-like[1] result that is easy to follow and manipulate later, in the sense that it can be parsed in an automatic way by other applications and programming languages like Python in order to perform other analyses. In this way Maude specifiers get, in addition to an interpreter for their language, a model checker for the object language for free. Moreover, we also get a model checker for real code for all those programming languages included that are already specified in Maude. To the best of our knowledge, this kind of generic, metalevel transformation is novel in model checking.

The rest of the paper is organized as follows: Sect. 2 introduces the different types of semantics discussed throughout the rest of the paper. Section 3 presents

[1] See http://www.json.org/ for details.

the transformation for languages based on shared memory, while Sect. 4 presents the transformation for message-passing style. Finally, Sect. 5 concludes and outlines some lines of future work. The code of the tool, examples, and more information is available at https://github.com/ariesco/MCPS.

2 Preliminaries

We present in this section how the semantics for message-passing and shared-memory programming languages can be specified in Maude. We present one example for each semantics and show how it is model-checked; we will show in the next sections how the results obtained from these analyses are transformed. Note that the semantics here are just simple examples illustrating the power of the tool; it can be applied to any other semantics following the same principles.

2.1 Implementing Semantics in Maude

Semantics are represented in Maude by means of conditional *rewrite rules*, that stand for transitions between states. In this way, each inference rule of the form:

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

in the semantics, which indicates that $state_2$ is reached from $state_1$ if the premises $P_1 \ \dots \ P_n$ hold, is written in Maude as:

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

where the conditions `Pi` can be either equalities (possibly involving some auxiliary functions), that will be solved by applying equations, or rewrite conditions, that indicate that some extra transitions must hold.

In the following we will give the intuitive ideas underlying the syntax of our languages and limit Maude code to some rewrite rules defining the language semantics, so the ideas can be followed by non-experts. Type definitions, auxiliary functions, and much more information is available in the repository above.

2.2 Shared-Memory Semantics

We use a modification of the imperative language in [6, Chap. 13] as running example. This language includes assignments (`X := E`), sequential composition (`INS ; INS'`), conditional statements (`if COND then INS fi`), and loops (`while COND do INS od` and `repeat INS forever`), for `X` a variable, `E` an expression, `INS` and `INS'` sequences of instructions, and `COND` a condition. Processes executing programs written with this syntax are wrapped into processes of the form `[ID, P]`, with `ID` a natural number standing for the process identifier and `P` the program being executed. Finally, the whole system is a pair of the form `[PS, M]`, with `PS` a set of processes (put together by using `|`) and `M`

```
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
```

Fig. 1. Simplified Dekker algorithm

the memory, which consists of a set of pairs `[V, N]`, with `V` a variable and `N` a natural number. We assume all variables in the system are initialized beforehand.

The semantics of this system are defined by using rewrite rules for each instruction. For example, the rule `asg` indicates that, given a process `I` where the first instruction to be executed is the assignment `Q := N` (the variable `S` stands for the rest of processes) and the memory contains the pair `[Q, X]` (`M` stands for the rest of the pairs in the memory), the instruction is executed by updating the value of the variable from `X` to `N`:[2]

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

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, [6, Chap. 13] describes the verification of the Dekker algorithm, a well-known protocol for ensuring mutual exclusion where each process actively waits for its turn; this turn is indicated by a variable that is only changed by the process exiting the critical section. We present in Fig. 1 a simplification of the algorithm for the second process (the first is defined analogously by changing the variables `c1`/`c2`, `cs1`/`cs2`, and the value in `turn`) where a bug has been introduced, hence violating the mutual exclusion property. Note that the value of `csi` is used to determine if process `i` is in the critical section.

Hence, given (1) the initial state {`[1,p1] | [2,p2], [c1,0][c2,0] [cs1,0] [cs2,0] [turn,1]`}, with `p1` and `p2` the corresponding version of

[2] Note that this is a small-step semantics and hence `N` is completely evaluated. In general we may need to compute the expression on the righthand side in a rewrite condition.

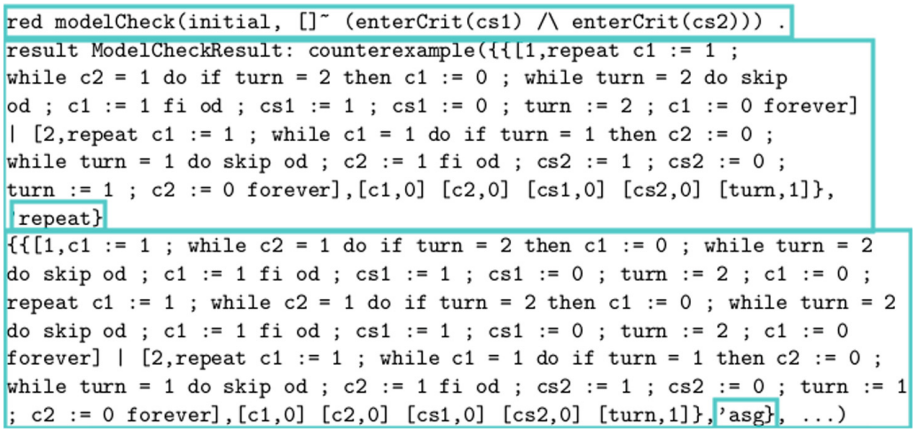

Fig. 2. Counterexample fragment for mutual exclusion in the (buggy) Dekker algorithm

the Dekker algorithm for the first and the second process, respectively, and (2) 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), we check whether mutual exclusion holds in Fig. 2, where we have highlighted the command (first rectangle) and the result (second and third rectangles). As shown in the figure, the property does not hold and hence a counterexample is returned; it consists of a list of pairs containing a state and the identifier of the rule used to reach the next state. In our case we just depict the first two pairs; the first one consists of the initial state and the rule label `repeat` (first inner rectangle), indicating that this rule is applied to reach the state in the second pair, where in turn `asg` will be used (second inner rectangle). This counterexample is difficult to follow not only because of the presentation, it also gives the user 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. We will show in Sect. 3 how this counterexample is transformed.

2.3 Message-Passing Semantics

We consider for our message-passing semantics the simple functional language in [19],[3] which supports `let` and conditional expressions, as well as basic arithmetic and Boolean operations. First, we expand it with expressions of the form `to ID : M` for sending messages, with both `ID` and `M` natural numbers standing for the identifier of the addressee and the message, respectively, and `receive` expressions for receiving them. Then, we define processes as terms of the form `[ID | E | ML]`, with `ID` a natural number identifying the process, `E` the expression being evaluated in the process, and `ML` a list of natural numbers standing for

[3] For the sake of conciseness we use syntactic sugar for numbers and variables.

the messages received thus far (we consider the head of the list is the leftmost element). Finally, the whole system is represented as a term of the form `|| PS, D ||`, for `PS` a set of processes and `D` a set of function declarations.

In this language we have rules of the form `D, ro |- e => e'` for simplifying expressions, given a set of declarations `D`, an environment `ro`, and expressions `e` and `e'`. For example, rule `Let1` below shows how the expression `e` in a `let` expression is simplified by applying a rewrite condition that computes `e''`. On the other hand, rule `Let2` just 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] .
```

Similarly, we need rules at the process level to model how messages are sent and received. Rule `send` below shows how a message being processed by `id` and addressed to `id'` is introduced into the list of received messages of `id'`, while value `1` is used in `id` 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] .
```

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

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

The declarations `decs`, shown below, indicate that the server sends a message (`0`) to the process identified by the first argument (client `1` in this case), another message (`1`) to the process identified by the second argument (`2`), then waits for two messages and returns `1` if it receives `0` and `1` (in this order) and `0` otherwise. In turn, the client receives a message and just returns the same message to the server, whose identifier received as parameter:

```
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}, ...)
```

Fig. 3. Counterexample for message-passing semantics

```
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
```

A naïve user might expect messages from clients to be received in the same order as they were sent from the server, and hence the final state to be always `1`. Figure 3 shows the command (first rectangle) and the first two states of the counterexample for this property (second and third rectangles, respectively), where `finalValue` is an atomic proposition that holds if the process identified by the first argument contains the expression given as second argument. The first step just substitutes the function call by the body of the function, while the second one is in charge of sending the first message in the server, as highlighted by the inner rectangle. We will see in Sect. 4 how to improve this trace.

2.4 Maude Metalevel and Loop Mode

The transformations presented in this paper have been implemented in an interactive Maude tool extending Full Maude [6, Part II] and using Maude metalevel capabilities [6, Chap. 14].

Full Maude is an extension of Maude written in Maude itself. It provides an input/output loop, an explicit state, and facilities to define, parse, and execute new commands, making it the most appropriate option to develop interactive Maude applications. It is worth noting that commands in Full Maude must be enclosed in parentheses, as required by the Loop Maude [6, Chap. 17], the built-in Maude module in charge of dealing with input/output information. For this reason, all commands in Sects. 3 and 4 will follow this convention.

On the other hand, Maude metalevel allows users to use Maude modules and terms as usual data. This feature allows us to:

- Traverse modules and identify those rules modifying terms of a given sort (e.g. the memory) or creating/consuming terms built with particular operators (e.g. messages).
- Identify the subterms involved in each step. This analysis is twofold: (i) given the whole state, we are interested in identifying the particular subterm being rewritten (e.g. identify the process executing the code among the set of processes), and (ii) recognize particular parts of the subterm found in the previous step to isolate elements of interest (e.g. messages).
- Manipulate the counterexample obtained when model checking a system. In particular, we can use the information obtained when traversing the module to prune the counterexample and the information about subterms to distinguish among the different parts of each state (e.g. memory, processes, and messages).

3 Model Checking Shared-Memory Languages

In this section we present the transformation used for programming languages following a shared-memory approach. In this transformation we rely in the following assumption: *properties refer to memory states*. Hence, we only need to keep those transitions in the original counterexample performed by rules that modify the memory. For example, for the program in Sect. 2.1 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.

Once we have decided the transitions that we want to keep, we must decide how to display each step. We decided to follow a JSON-like format and 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. Since the memory will be modified by the application of the rule, we present the state *before* applying the rule (field `memory-before`) and *after* applying it (field `memory-after`). In this way the user can inspect the effects of the rule. Note that this field is a list, since in general different types of memory can be used.
- Since we can work at the metalevel, we decided to display the value of all atomic formulas before and after applying the rule, so the user can understand the values taken by the LTL formula (filed `props`). For each atomic proposition in the formula we display its name, arguments, and how its value changed when the current rule is applied.

Algorithm 1 presents the transformation for shared memory, where all functions but `head` and `tail` are implemented at the metalevel, since they manipulate

```
Data: Counterexample c, semantics S, sorts ms for memory terms, sort p for
      processes, atomic propositions aps, and (optionally) id argument.
Result: Transformed counterexample.
rule_labels = memoryRules(S);
c = close(S, c, rule_labels);
while not empty(c) do
    (term, label) = head(c);
    c = tail(c);
    if label ∈ rule_labels then
        (term', label') = head(c);
        sub = match(S, term, label, term');
        lhs = apply(sub, getLefthandSide(S, label));
        m_info = getMemoryInfo(term, term', ms);
        s_info = getStateInfo(term, ms);
        p_info = getProcessInfo(lhs, p, [id]);
        props_info = getPropsInfo(term, term', aps);
        display(m_info, s_info, p_info, props_info);
    end
end
```

Algorithm 1. Transformation for shared-memory semantics

modules, rules, and terms. We first extract from the semantics those rules that modify the memory by using `memoryRules`. Then, we make sure the last transition in the counterexample does not use a rule in this set; if this is the case, the function `close` explicitly adds the next state[4] and uses a special label not in *rule_labels* to make sure the condition in the `while` loop skips it. Then the loop traverses all the states in the counterexample; when we find a step whose label is in *rule_labels* then we take the next state to find the matching (function `match`) that was used in the rule. This is required because, given a term and a rule, many different matchings are possible, so we need to ensure that we use the correct one. Then, we apply this matching to instantiate the lefthand side of the rule being used, hence obtaining *lhs* (function `apply`). This subterm is the one containing the information about the process being executed, while the term in the counterexample (*term*) contains the information about the whole system. We use appropriate pretty-printing functions to display the corresponding information.

The current version of the system cannot infer the sort for the memory or the sort for the processes (that we call *units*). Hence, we require the user to introduce these sorts. In our example, we would start by introducing `Memory` as the sort used for the memory and `Process` as the sort used for processes. Moreover, we indicate that the first argument for `Process` stands for the identifier:[5]

[4] Since a cycle is required to evaluate an LTL formula, this new state has appeared before in the counterexample and there is no need to explore it again.

[5] If the constructor does not include an identifier we would use `(unit Process .)`.

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

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

```
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}]
} ...
```

Fig. 4. Transformed counterexample for shared-memory semantics

Note that the first command allows the user to introduce several sorts for the memory, since different representations can be used for registers, main memory, etc. Once this information has been introduced into the system, it infers that the single rule modifying the memory is `asg`, so only the steps using this rule in the counterexample will be displayed. Now, we can execute the `shared memory analysis` command with the same model-checking command that we used in Sect. 2.2. We present a fragment of the transformed trace in Fig. 4, where `unit` and `system` are not shown for the sake of readability. The step shown in the figure corresponds to the rewrite step that violates mutual exclusion: the process identified by `1` is executed and it goes into the critical section (variable `cs1` changes its value from `0` to `1`, as we have highlighted in the figure), satisfying the corresponding property (`enterCrit(cs1)`); since the second process was into the critical section as well (as we can see by checking `cs2` or the corresponding property), the formula fails. However, we also notice that the process `1` behaved appropriately, since it was its turn (see variable `turn`), so we would inspect the trace for the second process to find the error.

Hence, the trace now can be read more easily, it contains less states (while the original counterexample had 89 states, the transformed one has 58), and it is displayed in a format that can be parsed and analyzed later if required.

4 Model Checking Message-Passing Languages

We present in this section two different ways to transform counterexamples like the one shown in Sect. 2.3, so the Maude semantics become transparent to the user. While 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.

We denote as `summary` mode our first approach, which presents the expression reached in each process, as well as the sent and consumed messages. In order to do so, we need the user to introduce the sort for the processes (and the argument standing for its identifier, if it exists) and 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, so they can be displayed. Hence, 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).

However, in some cases it is also useful to understand the interleaving between different messages and processes. For this reason, we decided to present a trace-like counterexample, that we call `trace` mode. However, in this semantics is not clear the notion of "step," so we first decided to focus on messages and display information when a message is sent or consumed. Then, we noticed that some properties might change some steps after a message was sent or received, and hence we decided to include in the trace those steps where at least one atomic property changes its truth value. As explained in the previous section, this information is used by executing at the metalevel all the atomic properties in the state reached in the corresponding state. 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), which can be either `msg-consumed`, `msg-sent`, and `prop-changed`, which stand for messages consumed, messages sent, and truth value of atomic propositions changed, respectively.
- The messages involved in the action (`messages` field). This field is omitted when the action is not referred to message creation or consumption.
- The state of all processes before and after applying the rule (`processes-before` and `processes-after` fields, respectively).
- How the properties changed with the rewrite rule (`props` field), which are displayed as explained in the previous section for shared memory.

Algorithm 2 presents this transformation, where again all functions but `head` and `tail` must be implemented at the metalevel. It first analyzes the semantics to

extract those rules in charge of sending and consuming messages, respectively. This step requires the function to traverse all rules and choose those whose righthand side either creates terms built with operators in *os* or removes terms built with *oc* (both actions with respect to the lefthand side). As explained in the previous section, we use the function `close` to add an extra final state if needed (i.e., if a message is involved or the properties change), since we need pairs of states to infer the matching. Then, we initialize the list of processes and start the loop: if the current label involves messages then we compute the substitution and the lefthand side of the rule as we did in the previous section and distinguish whether the event consisted of sending or consuming messages. We use the appropriate operators (either *os* or *oc*) to obtain the current process, the messages, and the event that took place. Note that this inference is more complex than the one for shared memory, since in the case of synchronous communication many processes might appear in the rule and we must select the one being executed, that is, containing terms built with the operators *os* or *oc*. Finally, we update the appropriate process in the list (if it did not exist a new process with that identifier is created) with the `update` function and the information is displayed depending on the selected mode.

In our example, we would indicate that processes are terms of sort `Process` and their identifier is its first argument. Similarly, we would 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
```

Similarly, if we want a summary of the execution we would set the mode to `summary` (which is the default one) as follows:

```
Maude> (set mode summary .)
Mode summary selected.
```

Figure 5 shows the result when using this transformation to the example in Sect. 2.3. We see that the server, identified by `0`, finished with value `0` after consuming the messages in the order `1`, `0`, while the clients finished as expected. With this information the user realizes that a different interleaving is possible and can fix its program. On the other hand, to use the `trace` mode in our example we would use the following command:

```
Maude> (set mode trace .)
Mode trace selected.
```

Figure 6 shows the instant when process `0` (the server) consumes the message from client `1`. We have omitted processes `1` and `2` for the sake of readability,

```
Data: Counterexample c, semantics S, operators os for sending messages,
      operators oc for consuming messages, sort p for processes, atomic
      propositions aps, and (optionally) id argument.
Result: Transformed counterexample.
send_labels = sendRules(S, os);
cons_labels = consumeRules(S, oc);
c = close(S, c, rule_labels);
proc_list = [];
while (|c| > 1) do
    (term, label) = head(c);
    c = tail(c);
    (term', label') = head(c);
    p_info = getPropsInfo(term, term', aps);
    if label ∈ (send_labels ∪ cons_labels) then
        sub = match(S, term, label, term');
        lhs = apply(sub, getLefthandSide(S, label));
        if label ∈ send_labels then
            p_info = getProcessInfo(lhs, p, [id], os);
            msgs = getMsgProcessed(lhs, os);
            event = send;
        else
            p_info = getProcessInfo(lhs, p, [id], cs);
            msgs = getMsgProcessed(lhs, cs);
            event = consume;
        end
        proc_list[p_info] = update(proc_list, p_info, event, msgs);
        display(proc_list, p_info, event, msgs, p_info) ; // only in trace mode
    end
    else if changed(p_info) then
        display(term, term', p_info) ;                    // only in trace mode
    end
end
display(proc_list) ;                                  // only in summary mode
```

Algorithm 2. Transformation for message-passing semantics

while changes have been highlighted. Note that this message is the first one consumed by the server (in the state before the rule the list of consumed messages) because they have arrived in this order (the third argument of the `value` field, the ordered list of messages, has value `1 . 0`); after applying the rule the message has disappeared from the list, message `1` appears in the list of consumed messages, and the first `receive` in the state has been replaced by `1`. Once the user realizes this behavior was expected, he/she should change the program to take this interleaving into account.

Finally, it is worth noting that, in addition to improving the readability and providing a friendly representation, the number of steps in `trace` mode is reduced from 24 in the original counterexample to 8.

```
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]}
```

Fig. 5. Final state for message-passing semantics

```
{id = 0,
 action = msg-consumed,
 messages =[1],
 processes-before = [
  { id = 0,
    value = [0 | let c = receive in let d = receive
                  in If Equal(c, 0) And Equal(d,1) Then 1 Else 0 | 1 . 0],
    sent =[to 1 : 0, to 2 : 1],
    consumed =[]},...],
 processes-after = [
  { id = 0,
    value = [0 | let c = 1 in let d = receive
                  in If Equal(c, 0) And Equal(d,1) Then 1 Else 0 | 0],
    sent =[to 1 : 0, to 2 : 1],
    consumed =[1]},...],
 props = {name = finalValue,
          args =[0, 1],
          prop = false -> false}}
```

Fig. 6. Trace-like representation: message consumption

5 Concluding Remarks and Ongoing Work

In this paper 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 and are available online. To the best of our knowledge this is the first generic transformation that allows users to model check real code based on its semantics. Hence, this tool sets the basis for further development in this direction.

On the theoretical side, it is interesting to study how this approach relates to similar approaches, like the partial evaluation transformations in [7,12].

On the tool side, it would be interesting to define transformations for other approaches, in particular for hybrid ones implementing both shared memory and message passing. 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. Notably, following the analyses proposed in [16] it would be possible to identify the sorts for the memory.

Then, it would be interesting to see whether the current transformations can be improved. In [1] the authors use *slicing*, a technique to keep only those instructions related to the values reached by a set of variable of interest, to reduce the size of Maude traces. When applying this technique we face again the problems outlined in the introduction, since it works on *Maude variables* but we need it to work on *program variables*, which depend on the semantics. We would need to follow the ideas in [16] to obtain the desired result.

Regarding efficiency, following the ideas in [11] it is possible to reduce the number of states when model checking Maude specifications, hence avoiding the state-space explosion problem, by transforming rules (that generate transitions when model checking a system) into equations (that do not generate transitions) if some properties hold. These properties are the executability requirements (termination, confluence, and coherence), which can be proved in some cases using the Maude Formal Environment [8], and *invisibility*, which requires that the transformed rules do not change the truth value of the predicates. Hence, 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 [13] 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.

References

1. Alpuente, M., Ballis, D., Espert, J., Romero, D.: Backward trace slicing for rewriting logic theories. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 34–48. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22438-6_5
2. Ben-Ari, M.: Principles of the Spin Model Checker. Springer, London (2008). https://doi.org/10.1007/978-1-84628-770-1
3. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. Theor. Comput. Sci. **236**, 35–132 (2000)
4. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_29
5. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)

6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
7. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Semantics-based generation of verification conditions via program specialization. Sci. Comput. Program. **147**, 78–108 (2017)
8. Durán, F., Rocha, C., Álvarez, J.M.: Towards a Maude formal environment. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 329–351. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24933-4_17
9. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: Proceedings of the 39th Symposium on Principles of Programming Languages, POPL 2012, pp. 533–544. ACM (2012)
10. Farzan, A., Chen, F., Meseguer, J., Roşu, G.: Formal analysis of Java programs in JavaFAN. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 501–505. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_46
11. Farzan, A., Meseguer, J.: State space reduction of rewrite theories using invisible transitions. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 142–157. Springer, Heidelberg (2006). https://doi.org/10.1007/11784180_13
12. Gómez-Zamalloa, M., Albert, E., Puebla, G.: Test case generation for object-oriented imperative languages in CLP. Theor. Pract. Log. Program. **10**(4–6), 659–674 (2010)
13. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. Int. J. Softw. Tools Technol. Transf. **2**(4), 366–381 (2000)
14. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992)
15. Meseguer, J., Roşu, G.: The rewriting logic semantics project. Theor. Comput. Sci. **373**(3), 213–237 (2007)
16. Riesco, A., Asăvoae, I.M., Asăvoae, M.: Slicing from formal semantics: Chisel. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 374–378. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_21
17. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. J. Log. Algebr. Program. **79**(6), 397–434 (2010)
18. Rusu, V., Lucanu, D., Serbanuta, T., Arusoaie, A., Stefanescu, A., Roşu, G.: Language definitions as rewrite theories. J. Log. Algebraic Methods Program. **85**(1), 98–120 (2016)
19. Verdejo, A., Martí-Oliet, N.: Executable structural operational semantics in Maude. J. Log. Algebr. Program. **67**, 226–293 (2006)