Selene: A Generic Framework for Model Checking Concurrent Programs from Their Semantics in Maude*

Adrián Riesco Gorka Suárez-García
Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid, Madrid, Spain
ariesco@fdi.ucm.es gorka.suarez@ucm.es

Model checking is an automatic technique for verifying whether some properties hold in a concurrent system. Maude is a high-performance logical framework where other systems can be easily specified, executed, and analyzed. Moreover, Maude includes a model checker for checking properties expressed in Linear Temporal Logic. However, when a property on a program written in a programming language specified in Maude does not hold the counterexample generated by this system refers to the Maude semantics, which can be difficult to follow.

In this paper we present Selene, a generic framework for dealing with asynchronous concurrent systems that allows users to manipulate the counterexample generated by the Maude model checker to relate it to the program being analyzed. This is achieved by providing a kernel for dealing with messages and memory, which are later handled in the counterexample; the user can specify the details of his semantics on top of this kernel.

Keywords: Maude, model checking, semantics, Erlang.

1 Introduction

Model checking [3] is an automatic technique for checking whether some properties, expressed in modal logic, hold in a concurrent system. Although this is a very powerful technique, the implementation of an efficient and reliable model checker takes years and hence only a small number of model checkers are available. This is unfortunate, because the range of applications requiring model checking is huge and hence translations are usually required, which requires to formally prove their correctness and might make the obtained counterexamples difficult to understand.

On the other hand, we observe an increased interest in the context of rewriting systems in defining various languages to cover many programming language paradigms. This idea is presented in *the rewriting logic semantics project* [9], where the programming languages semantics are defined as rewriting systems using Maude, and it is followed by the work in the $\mathbb{K}$ framework [11]. Maude [4] is a high-level language and high-performance system supporting both equational and rewriting logic computation. Maude modules correspond to specifications in *rewriting logic* [8], a logic that allows the representation of many models of concurrent and distributed systems. This logic is an extension of *membership equational logic* [2], 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. In the context of [9], these rules correspond to the execution of the different instructions in our programming language, hence allowing a natural representation for any programming language semantics. As

*This research has been partially supported by MINECO Spanish projects *StrongSoft* (TIN2012-39391-C04-04), CAVI-ART (TIN2013-44742-C4-3-R), and *TRACES* (TIN2015-67522-C3-3-R), and by the Comunidad de Madrid project N-Greens Software-CM (S2013/ICE-2731).

Submitted to:
WFLP 2016

a semantical framework, Maude has been used to specify the semantics of several languages, such as LOTOS [13], CCS [13], and Java [5]. Moreover, the $\mathbb{K}$-Maude compiler [12], which is able to translate $\mathbb{K}$ specifications into Maude, has eased the methodology to describe programming language semantics in Maude.

Maude provides a model checker for properties defined Linear Temporal Logic. However, when checking properties on programs whose semantics has been defined in Maude the obtained counterexample does not refer to the the program itself but to the rewriting rules defining the semantics, hence intertwining rules relevant for the particular execution that failed to fulfill the property (e.g. memory updates) and rules only relevant from the semantics point of view (e.g. the transitive closure). This is quite unfortunate, since one of the main points of a semantic framework such as Maude is to specify and analyze new paradigms.

In this paper we present Selene, an experimental Maude framework that generalizes the definition of concurrent systems and simplifies the counterexamples obtained by performing model checking. This is achieved by providing the *world*, a set of elements that define our application such that the nodes that will run the program, the processes inside those nodes, the messages inside the network, and any valuable data we need in the execution, as well as a mechanism to manipulate them once a counterexample is obtained. Note that, since we talk about nodes and network, our goal is to apply these techniques to concurrent programming languages that uses message passing.

A practical example using model checking would be checking whether an error occurs. Taking our semantics, defined by our rules, we can define properties like `hasAnyFailed` to check if any process in our *world* will fail by an error or an uncaught exception. Then we will check—against the initial state of our *world*—the next logic formula: $\Box(\neg$ `hasAnyFailed`$)$. This formula means: always, in all the states, the current state does not have any failed process.

The rest of the paper is structured as follows: Section 2 introduces why a generic language is useful to our purpose. Section 3 explains the basic ideas underlying the tool, while Section 4 show us a model checking counterexample and how it would be like after being transformed into its processed version. Section 5 shows some related work to this project. Finally, Section 6 concludes and outlines some lines of future work to improve the tool. The Selene source code and examples are available at `https://github.com/gorkinovich/selene`.

2 One Machine to Rule Them All

In this section we motivate the use of a common kernel—based on the basics of actor-style languages, like mailboxes with infinite buffer space and asynchronous sending of messages—for our *world* in order to use different asynchronous concurrent programming languages. To do so, we need a generic language to be executed by a generic machine model. Using this generic "machine", we provide the implementation of shared components between languages, such as the memory, the messages, and the network.

This idea of a generic language is inspired in how most of the modern languages are transformed to be executed in machine language like C++ or FORTRAN. Moreover, we also have the example of other languages which are transformed into a bytecode language—to be interpreted or compiled when it is executed—like Java or Python. In both cases the final "language" is much simpler than the original used to write the source code in our projects.

We illustrate these concepts by means of a simple example. Since Maude has been designed to write system specifications, what would be needed to bring a program written in Erlang[1]—like the next one at

[1] The `print` function is not a BIF in Erlang, but is used in the examples instead `io:format` to simplify the case study.

Figure 1—to the rewriting engine?

```
-module(test).

server() ->
    register(server, self()),
    server_loop().

server_loop() ->
    receive V ->
        print(V, "\n"),
        server_loop(V)
    end.

worker() ->
    server ! "EXTERMINATE",
    server ! "ANNIHILATE",
    server ! "DESTROY".
```

Figure 1: The echo server test example in the Erlang language.

Before answering this question, let us see the purpose of the code shown. This is an echo server example. We have 3 functions to work. The `server` function is the main entry of the echo server; it registers the process with a name and goes to the server loop. The `server_loop` function is the server loop and it waits to receive anything and prints the received data in the output buffer. The `receive` expression in Erlang can contain several clauses, each one with a different pattern, and optionally a guard condition to check over the received value previously matched. The `worker` function sends 3 strings to the server.

What would be needed to bring that written program to the rewriting engine? We would make an equational theory inside a Maude functional module to express the Erlang syntax. After that, we would implement a system module to represent the inference rules of the semantics as rewrite rules. Then we would have our system inside Maude, and since Maude theories are executable this semantics become an interpreter of our new language *for free*, and hence it is possible to execute Erlang programs by rewriting with the `rew` command.

Note, however, that any modification in either the syntax or the semantics would force us to modify these modules. So once we have our Erlang specification, let us say we want to use the same idea with Scala. We would need to make a whole new implementation, start from the ground, because the syntax and some of the semantics are not the same. But taking a closer look, we can realize—as we said before in this section—there are some shared concepts and ideas in the way many languages works.

So the idea we work with, is to make a generic language to be executed by Maude, handling most of the shared concepts in the execution of a program, and using a different semantics implementation to add the differences between semantics. To achieve this, we have to make a translator from the original language to the generic one. Then we will add the transitional rules that define certain unique aspects of the language we are working on.

We chose Erlang as first language to work with because of its simplicity and the concurrent programming model, which follows the actor model. Many concepts in the modeled machine used by Selene are inspired by the Erlang platform, without losing the scope of being able to use other languages. Let us see an example to illustrate this concept:

```
% Erlang inc function
inc(X) when is_number(X) -> X + 1.
```

```
/* C inc function */
int inc(int X) { return X + 1; }
```

Both functions have only one clause, with only one parameter, which has to satisfy one condition (to be a number in Erlang and to be an integer in C). While its semantics may differ—because you are able to call `inc` in Erlang with a wrong value, provoking an exception in runtime, but in C will not compile if you try to pass an structure as parameter in the call—the concepts expressed in the syntax are very similar and can be expressed the same in a generic way, in most cases.

3 The Selene Framework

The main idea underlying Selene is to make an abstract machine to run concurrent programs, distributed or not. Once the code is translated into the abstract language of the machine in Selene and taking rules with the semantics, we are able to define the properties to be used in the Maude model checker. Before continuing, let us check at Figure 2 how it would be the previous Erlang example coded in the Selene generic language.

```
@ns(1, 'test,
    @fn(3, 'server,
        @cs(3, nil, nil,
            @op(4, @call, @lt(4, 'register), @sq(4, @lt(4, 'server)
                @op(4, @call, @lt(4, 'self), @sq(4, nil))))
            @op(5, @call, @lt(5, 'server_loop), @sq(5, nil))))
    @fn(7, 'server_loop,
        @cs(7, nil, nil,
            @rc(8, @cs(8, @lt(8, 'V), nil,
                @op(9, @call, @lt(9, 'print), @sq(9, @lt(9, 'V) @lt(9, "\n")))
                @op(10, @call, @lt(10, 'server_loop), @sq(10, nil))
            ), nil)))
    @fn(13, 'worker,
        @cs(13, nil, nil,
            @op(14, @snd,  @lt(14, 'server), @lt(14, "EXTERMINATE"))
            @op(15, @snd,  @lt(15, 'server), @lt(15, "ANNIHILATE"))
            @op(16, @snd,  @lt(16, 'server), @lt(16, "DESTROY")))))
```

Figure 2: The echo server test example in the Selene generic language.

We have a namespace (`@ns`) test with three functions (`@fn`): `server`, `server_loop`, and `worker`. Each function has one clause (`@cs`); none of them has parameters or conditions to check. Inside the clauses, we have a list of statements, like unary or binary operations (`@op`), the `if` expression (`@if`), the `case-of` expression (`@cf`), the `receive` expression (`@rc`), or the `try-catch-finally` expression. Each statement has one or more expressions to be executed. As you can see, all the data structures have an integer number as first field; this number is used as an index to find the line[2] inside the source code project. Another important expression in the language is literal (`@lt`), where we can store Boolean values, numbers, strings, atoms,[3] etc...

[2]Only source lines and not source columns are used in the framework, since source columns have not been used in the current state of the project. If they are needed in the future, the project would be refactored to allow their use.

[3]We are using the Maude sort `Qid`, which stands for any string of characters starting by a quote (`'`), to work with the atom concept.

The Selene framework is experimental; this means that the generic language we are talking about still lack some features. The current state of the tool is presented in Figure 3. Languages similar to Erlang—functional languages with messages passing—can be represented by the current state of the framework. Things we cannot express—at this moment—are loops, object-oriented programming and some advanced features of Erlang such as list comprehensions. Certainly, those features should be lines of future work.

People familiar with the `erl_syntax` module in Erlang, should notice the strong inspiration—of the way Erlang represents its programs in syntax trees—over the current generic language inside Selene. To include something like a while loop in the generic language, we would extend the current framework with a structure similar to: `@wl(Index, ConditionExpression, BodyExpressions)`. One of the cons of this style—to express source code—is the danger to have lots of similar syntactical structures to fit different languages, adding complexity to the implementation or even losing the objective of making a the kernel as generic as we would like to.

Symbol	Type	Fields
`@lt`	Literal	Index, Value
`@op`	Operation	Index, Symbol, Right Expression
		Index, Symbol, Left Expression, Right Expression
`@sq`	Sequence	Index, List of Expressions
`@ns`	Namespace	Index, Name, Declarations
`@fn`	Function	Index, Name, Clauses
`@bk`	Block	Index, Body
`@cs`	Clause	Index, Pattern, Guard, Body
`@if`	If–Else	Index, Clauses, Else Body
`@cf`	Case–Switch	Index, Test Expression, Clauses
`@tc`	Try–Catch	Index, Test Expression, Clauses/Body, Catch Clauses, Finally Body
`@rc`	Receive–After	Index, Clauses, After Clause

Figure 3: The current structures of the generic language.

The Selene machine executes—in the *world* of the application—the code written in the generic language shown in Figure 2. The structure of the *world* in the Selene framework consists of a set of nodes. Each node can execute any number of processes. The processes can send messages to any other process in the *world* network. We also have an object with the source code project and another object with general information about the current status[4] of the *world*.

When we instantiate a process we call a function as main entry of the process. Any function has a context with the data and the code to be executed. Since functions can call another function, the process has a stack of context. Inside the context there is a stack of ambits, each one contains the variables inside the scope,[5] the code to be executed, and a flag with the execution state of the current statement.

Any function contains a list of statements. Functions can be called by another function or by a new

[4]Actually, the status object only contains the program coded in the generic language and a natural number to assign the next index number to be used in the system (e.g. when the tool needs an id number to identify a node or a process to be created by an operation).

[5]Some languages like Java has several scope levels inside its functions, others like JavaScript do not. So the semantics of the current language to execute must decide—when the program exits an ambit—if the created variables inside the ambit must be merged in the previous one or just disappear.

process to be spawned. A statement is an expression, but it can be a complex expression that may need to be executed in sub-expressions. This lead us to have a state machine in Figure 4 to evaluate statements.

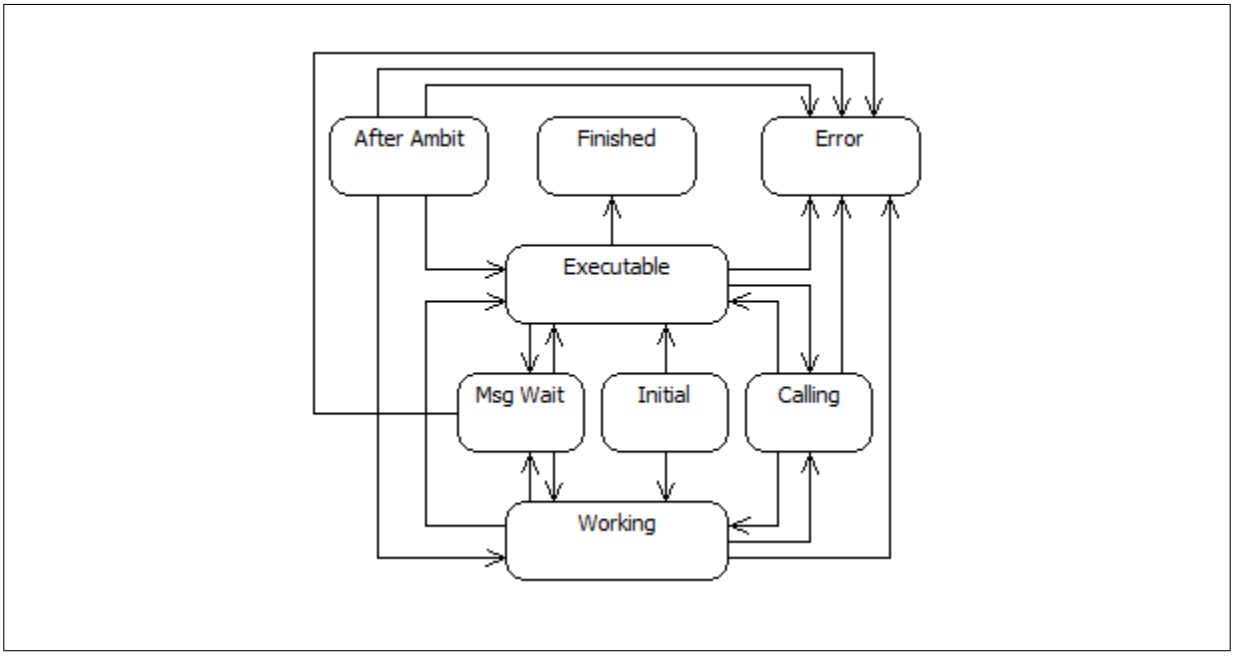

Figure 4: The statement states flowchart.

A statement starts at the "Initial" state. A statement is executable when all the sub-expressions are literal values or obtained values.[6] With complex statements, Selene evaluates the leftmost executable sub-expression and replaces it in the statement with the obtained value. After executing all the statements, the whole expression will end in the "Finished" state; this will lead the process to go to the next statement in the current ambit. There are situations where it is needed to start a new context (when calling a function) or start a new ambit inside the current context (e.g. when we compute one branch of an `if` statement). This will stop the evaluation of the statement and continue working in the first statement of the new context or ambit. When the program exists the context or the ambit, resumes the evaluation of the previous statement, taking the obtained value to replace the evaluated sub-expression.

A special state is "Message Wait". This one occurs when we enter a `receive` statement. The `receive` statement is executable, except in the case we have a non-value expression in the `after` clause.[7] Executing the `receive` statement will set the expression in a wait state, until a new message is received or if the messages mailbox contains a previous message received before entering the receive statement.

To execute our program—with the rewriting engine—we have a configuration[8] in Maude with an object that contains the initial data to configure the Selene engine. The engine translates the code into the generic language and Maude rewrites the *world* with the defined semantics rules. The result obtained by executing the code in Figure 2 is depicted in Figure 5.

[6] An obtained value is a value received by the evaluation operation in the framework. Since Maude does not support to have different types of values—like integer and floats—inside the same sort without an encapsulation, we need a temporal data structure to be use as an expression—inside the code we execute—to store the calculated final values of any evaluation.

[7] In the current version of the framework, the `after` clause is not yet implemented. The `after` is used by Erlang as a time-out, because the `receive` statement blocks the execution. If the time-out—set in milliseconds—is reached, the execution exits the current `receive` statement.

[8] The sort `Configuration` in Maude is a multiset of objects and messages.

```
[Rule: world.init]                                [Rule: statement.exec]
[Rule: statement.init]                            [Rule: process.msg.get]
[Rule: statement.work]                            [Rule: statement.work]
[Rule: statement.exec]                            [Rule: statement.exec]
[Rule: statement.init]                            [Node: @id(1)] => "ANNIHILATE\n"
[Rule: statement.next]                            [Rule: statement.next]
[Rule: statement.exec]                            [Rule: statement.exec]
[Rule: statement.init]                            [Rule: statement.init]
[Rule: process.msg.add]                           [Rule: process.msg.add]
[Rule: statement.next]                            [Rule: statement.next]
[Rule: statement.exec]                            "Final result:" "DESTROY"
[Rule: statement.init]                            [Rule: process.finish]
[Rule: statement.exec]                            [Rule: statement.exec]
[Rule: process.msg.get]                           [Rule: statement.exec]
[Rule: statement.work]                            [Rule: process.msg.get]
[Rule: statement.exec]                            [Rule: statement.work]
[Node: @id(1)] => "EXTERMINATE\n"                 [Rule: statement.exec]
[Rule: statement.next]                            [Node: @id(1)] => "DESTROY\n"
[Rule: statement.exec]                            [Rule: statement.next]
[Rule: statement.init]                            [Rule: statement.init]
[Rule: process.msg.add]                           [Rule: statement.exec]
[Rule: statement.next]                            [Rule: statement.exec]
[Rule: statement.exec]                            [Rule: process.msg.get]
[Rule: statement.init]                            [Rule: statement.exec]

rewrite in TESTS :
  testworld
result Configuration :
  < 'project : Project | files : @sf("test.erl","-module(test).\n\nserver() ->\n    register(ser
ver, self()),\n    server_loop().\n\nserver_loop() ->\n    receive V ->\n        print(V, \"\\n\
"),\n        server_loop(V)\n    end.\n\nworker() ->\n    server ! \"EXTERMINATE\",\n    server
! \"ANNIHILATE\",\n    server ! \"DESTROY\".",16)> < 'status : Status | nextIndex : 6,program :
    @ns(1,'test,@fn(3,'server,@cs(3,nil,nil,@op(4,@call,@lt(4,'register),@sq(4,@lt(4,
    'server)@op(4,@call,@lt(4,'self),@sq(4,nil))))@op(5,@call,@lt(5,'server_loop),@sq(5,
    nil))))@fn(7,'server_loop,@cs(7,nil,nil,@rc(8,@cs(8,@lt(8,'V),nil,@op(9,@call,@lt(9,
    'print),@sq(9,@lt(9,'V)@lt(9,"\n")))@op(10,@call,@lt(10,'server_loop),@sq(10,nil))),
    nil)))@fn(13,'worker,@cs(13,nil,nil,@op(14,@snd,@lt(14,'server),@lt(14,"EXTERMINATE"))@op(
    15,@snd,@lt(15,'server),@lt(15,"ANNIHILATE"))@op(16,@snd,@lt(16,'server),@lt(16,
    "DESTROY"))))))> < @id(1): Node | cin : "",cout : "EXTERMINATE\nANNIHILATE\nDESTROY\n",heap
    : @ms(nil),info : < 'server : ProcessAlias | node : 1,process : 1 > > < @id(1): Process |
    context :(@cx('test 'server,@am(@op(5,@call,@lt(5,'server_loop),@sq(5,nil)),@CallingState,
    nil),@ms(nil),@vl(true))@cx('test 'server_loop,@am(@rc(8,@cs(8,@lt(8,'V),nil,@op(9,@call,
    @lt(9,'print),@sq(9,@lt(9,'V)@lt(9,"\n")))@op(10,@call,@lt(10,'server_loop),@sq(10,nil))),
    nil),@MsgWaitState,nil)@am(@op(10,@call,@lt(10,'server_loop),@sq(10,nil)),@CallingState,
    @vr('V,@sr(1,1,1))),@ms(@mc(1,@vl("EXTERMINATE"))),@vl('ok))@cx('test 'server_loop,@am(@rc(
    8,@cs(8,@lt(8,'V),nil,@op(9,@call,@lt(9,'print),@sq(9,@lt(9,'V)@lt(9,"\n")))@op(10,@call,
    @lt(10,'server_loop),@sq(10,nil))),nil),@MsgWaitState,nil)@am(@op(10,@call,@lt(10,
    'server_loop),@sq(10,nil)),@CallingState,@vr('V,@sr(2,1,1))),@ms(@mc(2,@vl("ANNIHILATE"))),
    @vl('ok))@cx('test 'server_loop,@am(@rc(8,@cs(8,@lt(8,'V),nil,@op(9,@call,@lt(9,'print),
    @sq(9,@lt(9,'V)@lt(9,"\n")))@op(10,@call,@lt(10,'server_loop),@sq(10,nil))),nil),
    @MsgWaitState,nil)@am(@op(10,@call,@lt(10,'server_loop),@sq(10,nil)),@CallingState,@vr('V,
    @sr(3,1,1))),@ms(@mc(3,@vl("DESTROY"))),@vl('ok))@cx('test 'server_loop,@am(@rc(8,@cs(8,
    @lt(8,'V),nil,@op(9,@call,@lt(9,'print),@sq(9,@lt(9,'V)@lt(9,"\n")))@op(10,@call,@lt(10,
    'server_loop),@sq(10,nil))),nil),@MsgWaitState,nil),@ms(nil),@vl(nothing))),messages : nil,
    newMsgsFlag : false,owner : @id(1)>
```

Figure 5: The Maude output after rewrite the echo test example.

The first part of this output refers to the `print` attributes—inside the Maude rules and equations—in the current implementation of the Erlang semantics, which provide a trace of the execution. The second part is the Maude output of the command `(rew testworld .)` for rewriting the constant `testworld` containing the example above, which might be difficult to follow.

Inside the `Configuration` of the result are 4 objects, with syntax `< OI : CI | AtS >`, with `OI` the object identifier, `CI` the class identifier, and `AtS` a set of attributes, each of them with syntax `AI : V`, with `AI` the attribute identifier and `V` its current value. The first object, with the id `'project`, is a project class; it contains the source code files of the project in the field `files`. The second object, with the id `'status`, is a status class with the program code in the generic language used by Selene in the field `program`. The third object, with the id `@id(1)`,[9] is a node class; it contains the current output buffer of that "computer" in the field `cout`, and inside the current general info the `'server` object is a process alias class to identify the server atom as a process id. Finally, the fourth object, with the id `@id(1)`, is a process class; it contains the current state of the echo server process, where the `context` field contains the context stack with the `server` and `server_loop` calls, the `messages` field contains the queue of received messages, the `newMsgsFlag` indicates if a new message has arrived, and the `owner` contains the object identifier of the node object where the process has been instantiated.

4 Model Checking in Selene

With the Selene framework and the semantics of a language specified in transitional rules, we can execute code inside Maude. We can also apply model checking over the different states of the Selene machine. A useful and simple example is check whether our execution can be free of execution errors, using one of the formulas shown in Section 1. The Maude command used is `(red modelCheck(testworld, [] (~ ?hasAnyFailed)) .)` and a minor fragment of the obtained output is at the Figure 6. The `?hasAnyFailed` inside the formula is an atomic proposition, which checks whether a process has an error or exception at the top of its context stack. The `testworld` is the initial `Configuration` used in Figure 5.

In this example, we only show the second transition of the counterexample. Before and after this transition we use ellipses to indicate where the rest of the transitions should be. The counterexample ends with the final state, labelled with the `deadlock`. The size of the counterexample is 243 lines.

The transition shown in this counterexample contains several Maude objects inside a configuration: the project with the code, the status with the translated code, a node, the server process, and the worker process. As you may see, following an execution through all that text could be quite harsh. But all the execution data is right there.

The idea used in Selene is to transform the counterexample into a compact data file containing only relevant info. It is possible to filter which rules the user wants to obtain in the transformed data, hence providing different granularity level.[10] From the beginning to the deadlock, Selene takes the state and the rule label of the current transition, and the state of the next transition. Finding the differences between both states, Selene can find which process is the last evaluated in the transition rules, and then select the desired data by the engine.

Which info—thinks Selene—is relevant to the user? The selected info is the current used rule in the step, the process where the rule was executed, and a list of the current process in the *world* (each one of

[9] The `@id` container is used to assign a natural number as an object identifier.

[10] In our example we only excluded the `world.init` rule, because at the beginning we have no processes in the *world*, so we cannot compare how the processes may have changed, since there was no one.

```
reduce in TESTS :
  modelCheck(testworld, [] (~ ?hasAnyFailed))
result ModelCheckResult :
  counterexample(...{< 'project : Project | files : @sf("test.erl","-module(test).\n
\nserver() ->\n    register(server, self()),\n    server_loop().\n\nserver_loop() ->\n    receiv
e V ->\n        print(V, \"\\n\"),\n        server_loop(V)\n    end.\n\nworker() ->\n    server
! \"EXTERMINATE\",\n    server ! \"ANNIHILATE\",\n    server ! \"DESTROY\".",16)> < 'status :
    Status | nextIndex : 3,program : @ns(1,'test,@fn(3,'server,@cs(3,nil,nil,@op(4,@call,@lt(4,
    'register),@sq(4,@lt(4,'server)@op(4,@call,@lt(4,'self),@sq(4,nil))))@op(5,@call,@lt(5,
    'server_loop),@sq(5,nil))))@fn(7,'server_loop,@cs(7,nil,nil,@rc(8,@cs(8,@lt(8,'V),nil,@op(
    9,@call,@lt(9,'print),@sq(9,@lt(9,'V)@lt(9,"\n")))@op(10,@call,@lt(10,'server_loop),@sq(10,
    nil))),nil)))@fn(13,'worker,@cs(13,nil,nil,@op(14,@snd,@lt(14,'server),@lt(14,
    "EXTERMINATE"))@op(15,@snd,@lt(15,'server),@lt(15,"ANNIHILATE"))@op(16,@snd,@lt(16,
    'server),@lt(16,"DESTROY")))))> < @id(1): Node | cin : "",cout : "",heap : @ms(nil),info :
    none > < @id(1): Process | context : @cx('test 'server,@am(@op(4,@call,@lt(4,'register),
    @sq(4,@lt(4,'server)@op(4,@call,@lt(4,'self),@sq(4,nil))))@op(5,@call,@lt(5,'server_loop),
    @sq(5,nil)),@InitialState,nil),@ms(nil),@vl(nothing)),messages : nil,newMsgsFlag : false,
    owner : @id(1)> < @id(2): Process | context : @cx('test 'worker,@am(@op(14,@snd,@lt(14,
    'server),@lt(14,"EXTERMINATE"))@op(15,@snd,@lt(15,'server),@lt(15,"ANNIHILATE"))@op(16,
    @snd,@lt(16,'server),@lt(16,"DESTROY")),@InitialState,nil),@ms(nil),@vl(nothing)),messages
    : nil,newMsgsFlag : false,owner : @id(1)>,'statement.init}...,{...,deadlock})
```

Figure 6: The Maude model checker output after reduce a formula with the echo test example.

the contains where in the code is the execution—with the index id—, the variables in the top context, the queue of messages, and the final result returned by the top context).

The result of the transformation is shown in Figure 7. Selene gives a String that represents a JSON object to represent the transformed info. Using the `JSON.parse` method in JavaScript we obtain the final object shown in Figure 8. Using this standard format, we are able to make a visual representation of the execution, like a virtual debug session with the generated report. Since JSON is quite popular, we can use the object in an application or in a web page inside a web browser.

Before finishing this section, let me explain some details about how the Figure 8 is related to the Figure 1. When the user translated the Erlang source code into the generic language (shown at Figure 2), the source lines are stored inside the structures used by the generic language as a global index value. The program transformed into the generic language can be executed by the rewriting engine in Maude, so the model checker rewrites the *world*—where our program is—to build a counterexample. A little peek of the counterexample of the echo test is shown at Figure 6. Inside the chain of states of the counterexample, we have each state our *world* goes through. Inside the *world* we can find the created processes of our application, inside each of them we can obtain the current expression ready to execute, and inside each expression is stored the index related to the current source code line. Taking the important info—between each change of the *world*, inside the chain of states of the counterexample—to build each object in the JSON array report, we must not forget to include between that info the source code line index (which can be found inside the index field of the objects inside the array stored in the processes field, shown at Figure 8).

From the final JSON can be implemented an HTML page where the source code at Figure 1 can be display. Since the object at Figure 1 is an array of steps, the page can paint the line—where the execution is located at the current step and in the process which has been made the change—with a highlight colour. It also can display the variables of the current context of any process, to give a peek to the user about the current data. So in the standard execution of the echo server program, no error is reached, everything is

```
reduce in TESTS :
  modelCheckTransformer(testworld, [] (~ ?hasAnyFailed))
result String :
  "[{\"step\":\"statement.init\",\"node\":1,\"process\":1,\"processes\":[{\"node\":1,\"process\"
  :1,\"index\":4,\"variables\":[],\"messages\":[],\"result\":\"null\"},{\"node\":1,\"process\":2
  ,\"index\":14,\"variables\":[],\"messages\":[],\"result\":\"null\"}]},{\"step\":\"statement.in
  it\",\"node\":1,\"process\":2,\"processes\":[{\"node\":1,\"process\":1,\"index\":4,\"variables
  \":[],\"messages\":[],\"result\":\"null\"},{\"node\":1,\"process\":2,\"index\":14,\"variables\
  ":[],\"messages\":[],\"result\":\"null\"}]},{\"step\":\"statement.exec\",\"node\":1,\"process\
  ":2,\"processes\":[{\"node\":1,\"process\":1,\"index\":4,\"variables\":[],\"messages\":[],\"re
  sult\":\"null\"},{\"node\":1,\"process\":2,\"index\":14,\"variables\":[],\"messages\":[],\"res
  ult\":\"<error>\"}]},{\"step\":\"statement.error\",\"node\":1,\"process\":2,\"processes\":[{\"
  node\":1,\"process\":1,\"index\":4,\"variables\":[],\"messages\":[],\"result\":\"null\"},{\"no
  de\":1,\"process\":2,\"index\":0,\"variables\":[],\"messages\":[],\"result\":\"<error>\"}]},{\
  "step\":\"statement.work\",\"node\":1,\"process\":1,\"processes\":[{\"node\":1,\"process\":1,\
  "index\":4,\"variables\":[],\"messages\":[],\"result\":\"null\"}]},{\"step\":\"statement.exec\
  ",\"node\":1,\"process\":1,\"processes\":[{\"node\":1,\"process\":1,\"index\":4,\"variables\":
  [],\"messages\":[],\"result\":\"true\"}]},{\"step\":\"statement.next\",\"node\":1,\"process\":
  1,\"processes\":[{\"node\":1,\"process\":1,\"index\":5,\"variables\":[],\"messages\":[],\"resu
  lt\":\"true\"}]},{\"step\":\"statement.init\",\"node\":1,\"process\":1,\"processes\":[{\"node\
  ":1,\"process\":1,\"index\":5,\"variables\":[],\"messages\":[],\"result\":\"true\"}]},{\"step\
  ":\"statement.exec\",\"node\":1,\"process\":1,\"processes\":[{\"node\":1,\"process\":1,\"index
  \":8,\"variables\":[],\"messages\":[],\"result\":\"null\"}]},{\"step\":\"statement.exec\",\"no
  de\":1,\"process\":1,\"processes\":[{\"node\":1,\"process\":1,\"index\":8,\"variables\":[],\"m
  essages\":[],\"result\":\"null\"}]}]"
```

Figure 7: The transformed Maude model checker output after reduce the previous example.


```
[{"step":"statement.init","node":1,"process":1,"processes":[{"node":1,"process":1,"index":4,
"variables":[],"messages":[],"result":"null"},{"node":1,"process":2,"index":14,"variables":[],
"messages":[],"result":"null"}]},
{"step":"statement.init","node":1,"process":2,"processes":[{"node":1,"process":1,"index":4,
"variables":[],"messages":[],"result":"null"},{"node":1,"process":2,"index":14,"variables":[],
"messages":[],"result":"null"}]},
{"step":"statement.exec","node":1,"process":2,"processes":[{"node":1,"process":1,"index":4,
"variables":[],"messages":[],"result":"null"},{"node":1,"process":2,"index":14,"variables":[],
"messages":[],"result":"<error>"}]},
{"step":"statement.error","node":1,"process":2,"processes":[{"node":1,"process":1,"index":4,
"variables":[],"messages":[],"result":"null"},{"node":1,"process":2,"index":0,"variables":[],
"messages":[],"result":"<error>"}]},
{"step":"statement.work","node":1,"process":1,"processes":[{"node":1,"process":1,"index":4,
"variables":[],"messages":[],"result":"null"}]},
{"step":"statement.exec","node":1,"process":1,"processes":[{"node":1,"process":1,"index":4,
"variables":[],"messages":[],"result":"true"}]},
{"step":"statement.next","node":1,"process":1,"processes":[{"node":1,"process":1,"index":5,
"variables":[],"messages":[],"result":"true"}]},
{"step":"statement.init","node":1,"process":1,"processes":[{"node":1,"process":1,"index":5,
"variables":[],"messages":[],"result":"true"}]},
{"step":"statement.exec","node":1,"process":1,"processes":[{"node":1,"process":1,"index":8,
"variables":[],"messages":[],"result":"null"}]},
{"step":"statement.exec","node":1,"process":1,"processes":[{"node":1,"process":1,"index":8,
"variables":[],"messages":[],"result":"null"}]}]
```


Figure 8: The transformed Maude model checker output JSON object.

ok. But using model checking we find that—in this test sample—the order of execution of the processes matters.

It might not be clear enough the potential behind this methodology, since the developer can just simply open the Erlang console and type `debugger:start()` to start a classical debugging session. But this approach uses model checking to verify programs, which allows finding potential errors inside an algorithm. In concurrent programs, this allows the developer to find the context and state where the program breaks, even when in debugging sessions was not found any error.

5 Related Work

There are some model checkers in several languages, like McErlang [6] in Erlang scene. McErlang main focus is use model checking in concurrent software, not in sequential software, looking for errors in the execution of a source code.[11] McErlang parameters select the *algorithm* to be used while model checking, which evaluates some properties defined in the *algorithm* itself, so the user cannot write a LTL formula to be checked by the tool.

Another model checker—this one focused on Java scene—is Java Pathfinder [7], developed at the NASA Ames Research Center. Focused on apply model checking to concurrent programs, tries to find defects such as data races, deadlocks, and uncaught exceptions (e.g., division by zero exception). Since JPF model checks Java bytecode, should be possible to model check Scala programs as well. Some of the current JPF extensions allows to use the tool as user interface model checker, test case generator by means of symbolic execution, etc.

A last example of a programming language model checker is CPAchecker [1], which is focused on C language programs. There are other model checking tools to verify properties over Petri nets, Algebra of Communicating Processes, etc. So there are not many model checking tools focused in source code and the few ones focused on source code analysis—like McErlang—are developed to analyze only one language.

Meanwhile in the Maude community there are projects that implements operational semantics in languages like the Calculus of Communicating Systems [13], giving an opportunity to the developers to use the Maude model checker to verify concurrent systems modeled in the language. In the Erlang scene, a Core Erlang [10] implementation in Maude has been made to demonstrate the use of the Maude model checker with this language.

There are hardly any standard APIs or frameworks built on the idea of porting model checker tools across different languages in a generic way. Nonetheless, the ideas behind Selene have not been fully developed yet on this subject, to prove whether the proposed concepts are going in the right direction.

6 Conclusions and Future Work

We have presented in this paper a generic tool to evaluate concurrent programs with the Maude rewriting engine. The core of the tool is a kernel to manage shared concepts like the memory or the messages. This core uses a generic language to represent the source code of the program. The user defines the semantics of the language, using the execution framework of the core inside the transitional rules—defined by the user—used by the Maude's rewriting engine. It also allows us to use model checking with

[11] You can find an example of how to use McErlang in a tutorial presentation at: `https://babel.ls.fi.upm.es/trac/McErlang/attachment/wiki/midTermWorkshop/paper.pdf`

the written programs and transforms the output into a JSON object which contains the final report when a counterexample is found. That final report can be represented in HTML to the final user.

The immediate line of future work would be to implement more features of the Erlang language—syntactical structures, pattern matching, BIFs, etc.—to improve the framework. The second line of future work would be the implementation of the HTML representation of the counterexample transformed report. It would be also interesting to find a way to use semantics as a parameter in the system, making Selene more flexible. Moreover, we also plan to study how to apply slicing to the counterexample, so we can track the source of an unexpected value.

References

[1] Dirk Beyer & M. Erkan Keremoglu (2011): *CPAchecker: A Tool for Configurable Software Verification*. In: *Proceedings of the 23rd International Conference on Computer Aided Verification, CAV 2011*, pp. 184–190.

[2] Adel Bouhoula, Jean-Pierre Jouannaud & José Meseguer (2000): *Specification and Proof in Membership Equational Logic*. *Theoretical Computer Science* 236, pp. 35–132.

[3] Edmund M. Clarke, Orna Grumberg & Doron A. Peled (1999): *Model Checking*. MIT Press.

[4] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer & Carolyn Talcott (2007): *All About Maude: A High-Performance Logical Framework*. *Lecture Notes in Computer Science* 4350, Springer.

[5] Azadeh Farzan, Feng Chen, José Meseguer & Grigore Rosu (2004): *Formal Analysis of Java Programs in JavaFAN*. In: *Proceedings of the 16th International Conference on Computer Aided Verification, CAV 2004*, pp. 501–505.

[6] Lars-Åke Fredlund & Hans Svensson (2007): *McErlang: a model checker for a distributed functional programming language*. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, pp. 125–136.

[7] Klaus Havelund & Thomas Pressburger (2000): *Model Checking JAVA Programs using JAVA PathFinder*. *International Journal on Software Tools for Technology Transfer* 2(4), pp. 366–381.

[8] José Meseguer (1992): *Conditional Rewriting Logic as a Unified Model of Concurrency*. *Theoretical Computer Science* 96(1), pp. 73–155.

[9] José Meseguer & Grigore Roşu (2007): *The rewriting logic semantics project*. *Theoretical Computer Science* 373(3), pp. 213–237.

[10] Martin R. Neuhäußer & Thomas Noll (2007): *Abstraction and Model Checking of Core Erlang Programs in Maude*. *Electronic Notes on Theoretical Computer Science* 176(4), pp. 147–163.

[11] Grigore Roşu & Traian Florin Şerbănuţă (2010): *An Overview of the K Semantic Framework*. *Journal of Logic and Algebraic Programming* 79(6), pp. 397–434.

[12] Vlad Rusu, Dorel Lucanu, Traian-Florin Serbanuta, Andrei Arusoaie, Andrei Stefanescu & Grigore Rosu (2016): *Language definitions as rewrite theories*. *Journal of Logical and Algebraic Methods in Programming* 85(1), pp. 98–120.

[13] Alberto Verdejo & Narciso Martí-Oliet (2006): *Executable Structural Operational Semantics in Maude*. *Journal of Logic and Algebraic Programming* 67, pp. 226–293.