Framework for Model Checking Concurrent Programs in Maude

Gorka Suárez-García
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
gorka.suarez@ucm.es

Table of Contents

Introduction

Algorithms are not always error-free.

How to find those errors?

- Testing = Seeking errors randomly.
- Formal verification = Machine seeking some errors.

Model Checking

Model checking is an automatic technique for verifying whether some properties hold in a concurrent system.

$$M, s \models p$$

Where M is the model, s is the initial state, and p is the temporal logic formula to check.

The Maude System

- Maude is a high-performance logical framework where other systems can be easily specified, executed, and analyzed.

- Maude includes a model checker for checking properties expressed in Linear Temporal Logic.

The Maude Syntax

```
--- Functional module, used to
--- make equational theories.

fmod SIMPLE-NATURAL is
  sort Natural .
  op zero : -> Natural [ctor] .
  op s_ : Natural -> Natural [ctor] .
  op _+_ : Natural Natural -> Natural .
  vars N M : Natural .
  eq zero + N = N .
  eq s N + M = s (N + M) .
endfm
```

```
--- System module, used to
--- make rewriting theories.

mod SIMPLE-COUNTDOWN is
  pr SIMPLE-NATURAL .
  var N : Natural .
  rl [down] : s N => N .
endm
```

The Maude Syntax

```
Maude> red s s s zero + s s zero .
reduce in SIMPLE-NAT : s s s zero + s s zero .
rewrites: 4 in 6729318537ms cpu (0ms real) (0 rewrites/second)
result Nat: s s s s s zero
```

```
Maude> rew s s s s s zero .
rewrite in SIMPLE-COUNTDOWN : s s s s s zero .
rewrites: 5 in 1628036047000ms cpu (0ms real) (0 rewrites/second)
result Nat: zero
```

The Maude Syntax

```
--- Model checking property.

mod SIMPLE-PROPS is
  pr SATISFACTION .
  pr SIMPLE-COUNTDOWN .
  subsort Natural < State .
  var N : Natural .
  op cdfinished : -> Prop [ctor] .
  eq N |= cdfinished = (N == zero) .
endm
```

```
--- Model checking inital state.

mod SIMPLE-MCTEST is
  pr SIMPLE-PROPS .
  pr MODEL-CHECKER .
  pr LTL-SIMPLIFIER .
  op initial : -> Natural .
  eq initial = s s s s s zero .
endm
```

The Maude Syntax

```
Maude> red modelCheck(initial, [](<> cdfinished)) .
reduce in SIMPLE-MCTEST : modelCheck(initial, []<> cdfinished) .
rewrites: 39 in 13129332125ms cpu (24ms real) (0 rewrites/second)
result Bool: true
```

```
Maude> red modelCheck(initial, [](~ cdfinished)) .
reduce in SIMPLE-MCTEST : modelCheck(initial, []~ cdfinished) .
rewrites: 25 in 6264376255ms cpu (6ms real) (0 rewrites/second)
result ModelCheckResult: counterexample({s s s s s zero,'down}
    {s s s s zero,'down} {s s s zero,'down} {s s zero,'down}
    {s zero,'down}, {zero,deadlock})
```

The Echo Server Example in Erlang

```
-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".
```

The Echo Server Syntax Tree in Selene

```
@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")))))
```

The Selene Framework Core

- An abstract machine to run concurrent programs.
- Subsystem to handle memory and variables.
- Subsystem to handle function calls.
- Subsystem to handle message passing.
- Counterexample transformation from Maude counterexample to counterexample in JSON.

The Erlang Interpreter Over Selene

- Semantics built using the abstract machine of Selene.
- A set of transitional rules to define the semantics using small-step semantics with a FSM to evaluate composed expressions.
- Model-checking properties defined using the abstract machine of Selene.

The Maude Counterexample

```
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})
```

The Counterexample Transformed


```
[{"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":"null"}]},
 {"step":"statement.next","node":1,"process":1,"processes":[{"node":1,"process":1,"index":5,"variables":[],
  "messages":[],"result":"null"}]},
 {"step":"statement.init","node":1,"process":1,"processes":[{"node":1,"process":1,"index":5,"variables":[],
  "messages":[],"result":"null"}]},
 {"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"}]}]
```

The Counterexample Transformation

```
for i = 0 .. (N-1):
        c_state = counterexample.states_list[i]
        n_state = counterexample.states_list[i+1]
        process = get_changed_process(c_state, n_state)
        json_step = make_step(counterexample.rule[i], process, c_state)
        json_array.append(json_step)
```

Counterexample Interpretation

```
-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".
```

Colors: Process 1 & Process 2


```
{"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"}]}
```

Counterexample Interpretation

```
-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".
```

Colors: Process 1 & Process 2


```
{"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"}]}
```

Counterexample Interpretation

```
1.  -module(test).
2.
3.  server() ->
4.     register(server, self()),
5.     server_loop().
6.
7.  server_loop() ->
8.     receive V ->
9.        print(V, "\n"),
10.       server_loop(V)
11.    end.
12.
13. worker() ->
14.    server ! "EXTERMINATE",
15.    server ! "ANNIHILATE",
16.    server ! "DESTROY".
```

Colors: Process 1 & Process 2


```
{"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>"}]}
```

Future Work

- Improve the core parameterization with Maude theories and views.
- Complete the semantics of the Erlang syntax.
- Add more parameterization to the counterexample transformation algorithm.
- Make a visual representation of the transformed counterexample in HTML.

Conclusions

- The seeds of a generic abstract machine and framework to implement programming language semantics.
- A compact representation of the counterexample with meaningful information about the execution.
- Flexibility to write LTL formulae by the developer.

Questions