Improved Semantics and Implementation through Property-Based Testing with QuickCheck

Huiqing Li
School of Computing
University of Kent, UK
H.Li@kent.ac.uk

Simon Thompson
School of Computing
University of Kent, UK
S.J.Thompson@kent.ac.uk

ABSTRACT

Testing is the primary method to validate that a software implementation meets its specification. In this paper, we demonstrate an approach to validating an executable semantics using property- and model-based random testing in QuickCheck to automate and unify the testing of the semantics and its implementation. Our approach shows the use of executable semantics to bridge the gap between formal mathematical specification and implementation, as well as emphasising the suitability of functional programming languages – in this case Erlang – for writing executable semantics.

The approach is illustrated through a concrete example, in which the implementation of a proposed extension to the Erlang programming language – scalable groups – is tested. This new component comes with a small-step operational semantics written in mathematical notation, and was initially tested using unit testing. Through our work, we were able to find new bugs in both the implementation and the specification.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Design

Keywords

Testing, Specification, Executable Semantics, Implementation, QuickCheck, Erlang, Property-Based Testing

1. INTRODUCTION

The goal of software testing is to validate that an implementation satisfies its specification. If a specification is written in a non-executable language, the implementation is generally checked against a set of examples for which the expected results have been manually computed, and the correctness of the specification is either manually determined or checked in a separate process. When a specification is written in an executable language, it is much easier to check the correctness of the specification itself, however the testing of implementation and specification may still have to be done separately without proper tool support.

As a matter of fact, most programmers are reluctant to formulate formal specifications of the program they write. This could be due to the effort involved in formulating formal specifications, or the minor short-term payoff that results when there is no easy way to check that the formal specification corresponds to the program implemented.

In this paper, we demonstrate an approach to writing executable semantic specification using property- and model-based random testing in QuickCheck [7] to automate the testing of the implementation and, indeed, the specification itself. Our work also highlights the benefit of using an executable approach to semantics from the very start, so that the specification, as well as the implementation, can be coherently evolved and tested during development. While the testing of a specific Erlang component is presented in this paper, the methodology should be applicable to other components of Erlang, or languages with QuickCheck support, as long as there is some way of observing the state of the implementation, and the semantics is deterministic.

Our approach is demonstrated by the testing of a proposed extension to the Erlang programming language. The proposed extension is a library for *scalable groups*, '`s_groups`', written in Erlang, which aims to provide better scalability support for massively distributed Erlang applications. Along with the implementation, a small-step transition semantics for the s_group operations in the library has been defined. Since most of these s_group operations affect the system state, the semantics first defines an abstract state representing the extended Erlang system, then describes each operation as a transition between states. The formal semantics was written in non-executable mathematical notation; prior to our testing automation, the correctness of the specification was principally ensured by manual inspection.

Property- and model-based testing is our chosen testing approach. Property-based testing (PBT) provides a high-level approach to testing, rather than focusing on individual test cases. In PBT the required behaviour is specified by properties, expressed in a logical form. For example, a function without side effects might be specified by means of the full input/output relation using a universal quantification

AST'14, May 31 – June 1, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2858-6/14/05...$15.00
http://dx.doi.org/10.1145/2593501.2593509

over all the inputs; a stateful system will be described by means of model, which is an extended finite state machine. The system is then tested by checking whether it has the required properties for randomly generated data, which may be inputs to functions, sequences of API calls to the stateful system, or other representations of test cases.

Property-based testing is gaining popularity, especially in the community of functional programming languages. For Haskell, there is the original QuickCheck tool [5] developed by Koen Claessen and John Hughes in 2000; for Erlang there are QuickCheck [7] commercialised by QuviQ, and PropEr (`http://proper.softlab.ntua.gr`).

QuviQ QuickCheck is the tool of choice for our testing, mainly because the system under test is written in Erlang. In particular, we utilise its support for abstract state machines through the *eqc_statem* library. With *eqc_statem*, the user defines an abstract state machine model of the system under test (SUT); this model includes an initial abstract state in which test cases begin, as well as describing how each command changes that state. For each command, a number of things can be described: *preconditions* to decide whether or not to include a candidate command in test cases; *postconditions* to check that the value returned by the command executed is correct; a description of the changes on the abstract state as a result of command execution; and how to generate an appropriate function call to appear next in a test case.

The semantic specification of *s_groups* is written in non-executable mathematical notation, so the first step of our testing process was to faithfully translate this notation to an abstract model as represented by a QuickCheck *eqc_statem* module. Erlang is a functional programming language; as a result the Erlang representation of semantics is naturally very close to the mathematical representation. The declarative nature of the language also tends to make Erlang code short and compact.

The rest of the paper is organised as follows. Section 2 introduces Erlang and scalable groups; Section 3 gives a brief introduction to PBT and the QuviQ QuickCheck tool. The translation from mathematical specification to executable semantics and the semantics-driven testing of s_groups are presented in Section 4. Related work is discussed in Section 6 and Section 7 concludes the paper.

2. ERLANG AND SCALABLE GROUPS

Erlang [3] is a functional programming language with built-in support for concurrency based on share-nothing processes and asynchronous message passing between them; on creation, processes are dynamically allocated a process identifier (or *pid*) which is used as an address for sending messages. Erlang processes are lightweight, and as a result an Erlang program can be made up of thousands or millions of processes that may run on a single processor, a multicore processor or a manycore system. In Erlang, the same function name can be used multiple times with different number of arguments; a general way to identify a function is to use the format `Module:Function/Arity` or `Function/Arity`.

Erlang compiles to code that runs on the BEAM virtual machine (VM), and each instance of a running VM is called an *Erlang node*. Erlang has built-in support for distribution, and a distributed Erlang system consists of a number of Erlang VMs communicating with each other. Message passing between processes on different nodes is indistinguishable from communication on the same node. Whilst different Erlang nodes in a distributed Erlang system can run on different computers (or 'hosts'), it is also possible for multiple Erlang nodes to run on the same host.

Besides addressing a process using its pid, it is also possible to register a process under a static name. The ability to globally register names in a system consisting of several Erlang nodes is central to distributed Erlang programming. Global names registered in a distributed Erlang system are stored in replicated global name tables on every node, so there is no central storage point. However, any action which results in a change to the global name table, requires that name tables on other nodes are (automatically) updated too.

By default, the connections between *normal* Erlang nodes, that is nodes started without the `-hidden` flag, is transitive. That is, if a node A connects to node B, and node B has a connection to node C, then node A will also try to connect to node C. Each of these connected nodes will hold a replication of the global name tables.

With the current model of distributed Erlang, the larger the number of nodes in a cluster, the more expensive it becomes for each node to periodically check the liveness of connections, and the longer it takes to get the replications of global names updated. This limits the scalability of Erlang when the number of nodes in an Erlang cluster goes into the hundreds. This problem is currently being addressed in the European FP7 RELEASE project [1, 2].

The RELEASE project aims to scale Erlang to build reliable general-purpose software on massively parallel machines. One of the outcomes of the project is the notion of *scalable groups*, short as *s_groups* [4]. A s_group is a cluster of transitively connected Erlang nodes with its own 'global' name space. A s_group could overlap with other s_groups, so that a single Erlang node could belong to multiple s_groups. For example, Fig 1 shows six Erlang nodes configured to form three s_groups, `G1`, `G2` and `G3`, with `G2` overlapping with both `G1` and `G3`. A node that belongs to at least one s_group is called a *s_group* node, and a node that does not belong to any s_groups is called a *free* node. A free node behaves as before except that its connection with an s_group node is not transitive. The support for s_groups is added to Erlang

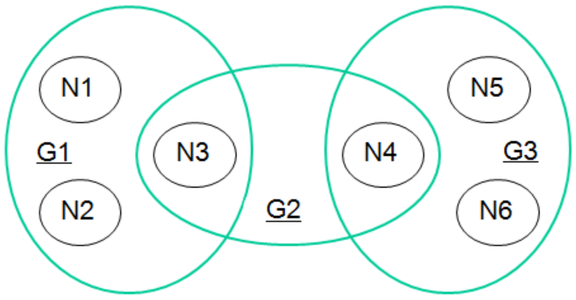

Figure 1: s_groups with overlapping

as a library consisting of 4,300 lines of code. Internally the s_group library maintains the full connections of s_groups, global name registration and global state synchronisation within s_groups. It provides operations for:

- creation and deletion of s_groups,
- adding and removing nodes to and from an s_group,
- (un_)registration of global names in a s_group,
- operations for inspecting s_groups and global names.

3. PROPERTY-BASED TESTING

QuickCheck [7] supports random testing of Erlang, and also of C programs through a foreign function interface. Properties of the programs are stated in a subset of first-order logic. A property is verified by QuickCheck generating multiple test cases for which the property is checked. When a counterexample is found, QuickCheck tries to generate a simpler – and thus more comprehensible – counterexample by discarding commands which do not contribute to the failure; this process is called *shrinking*. A failing case indicates bugs in either the implementation under test or the written properties. For example, testing the following property

```
prop_list_delete()->
  ?FORALL(I, int(),
    ?FORALL(List, List(int()),
       not (lists:member(I, lists:delete(I, List)))))
```

of the Erlang function lists:delete/2 might report

```
Failed! After 37 tests.
-8
[5, -8, 12, -8, 9]
Shrinking......(6 times)
-8
[-8,-8]
```

this happens because lists:delete(I, List) only removes the *first occurrence* of I in List, not all occurrences.

The general way to test state-based systems is to build an abstract model of the system in the form of either an abstract state machine or a finite state machine, and to use this model to drive the testing of real system. An abstract state machine can be implemented as a client module of the pre-defined QuickCheck *eqc_statem* behaviour; whereas a finite state machine with a collection of named states and transitions between them can be implemented as a client module of the pre-defined *eqc_fsm* behaviour. These machines have a finite number of (control) states, but also they have a *data* state which is modified by performing commands (or transitions); strictly they are extended finite state machines (or EFSMs). The machines are presented *symbolically* – with variables for states etc. – so that commands can be generated and executed by the system under test.

To implement an abstract state machine, the user needs to define a number of callback functions:

- initial_state() returns the initial model state.
- precondition(S, C) returns true if the symbolic command C can be performed in state S. This is used to decide whether or not to include a candidate command in test cases. A symbolic command binds a symbolic variable to the result of a symbolic function call, for example

  ```
  {set, {var, 1}, {call, lists, sort, [[2,1]]}}
  ```

  sets variable {var, 1} to the result of the symbolic call: lists:sort([1,2]). When the symbolic call is executed during test execution, the symbolic variable will be replaced by the actual value it was set to.
- postcondition(S, C, R) checks the postcondition of symbolic call C, executed in dynamic state S, with result R.
- next_state(S, R, C) is the state transition function. During test generation, it computes the symbolic state after symbolic call C, performed in symbolic state S, with result R; during test execution, the same function is used to compute the next *dynamic* state.
- command(S) generates a candidate symbolic call to appear next in a test case, if the current symbolic state is S; test sequences are generated by calling this repeatedly.
- invariant(S) is an optional call-back which can be used to check an invariant during test execution.

Implementing a finite state machine using eqc_fsm follows a similar structure, except that a state is separated into a state name and state data.

4. TESTING SCALABLE GROUPS

In this section we describe the use of QuickCheck to derive the executable semantics of s_groups from its formal specification as well as the simultaneous testing of semantics and implementation.

4.1 The approach

The architecture of the testing framework is shown in Figure 2. First an abstract state machine embedded an *eqc_statem* client module is derived from the semantic specification. The state machine defines the abstract state representation and the transition from one state to another when an operation is applied. Test case and data generators are then defined to control the test case generation; this includes the automatic generation of eligible s_group operations and the input data to those operations. Test oracles are encoded as the postcondition for s_group operations.

During testing, each test command is applied to both the abstract model and the actual s_group implementation. The application of the test command to the abstract model takes the abstract model from its current state to a new state as described by the transition functions; whereas the application of the test command to the real system leads the system to a new actual state. The actual state information is collected from each node in the distributed system, then merged and normalised to the same format as the abstract state representation. For a successful testing, after the execution of a test command, the test oracles specified for this command should be satisfied. Various test oracles can be defined for s_group operations; for instance one of the generic constraints that applies to all the s_group operations is that after each s_group operation, the normalised system state should be equivalent to the abstract state.

By default, QuickCheck generates 100 test cases for each run, with each test case consisting of a sequence of test commands. The number of test cases to test can be changed however. The testing claims to be successful if all the test cases have been passed, otherwise it fails and a minimised counter example is given after the first failing test case. The number of test cases can be

The function defining the top-level property for testing s_groups is shown in Figure 3. This function generates a command sequence Cmds, i.e. a test case, using the command generator defined in the test module (the name of test module is specified by the macro ?MODULE), then runs these commands one by one. Before each command is run, its precondition is checked by the precondition function associated with it, and after the command is executed, its postcondition is checked by the postcondition function associated it. The result of run_commands(?MODULE,Cmds) is a three-element tuple containing the history H of execution, the state S after the last command that was executed successfully and the reason Res that the execution stopped. For

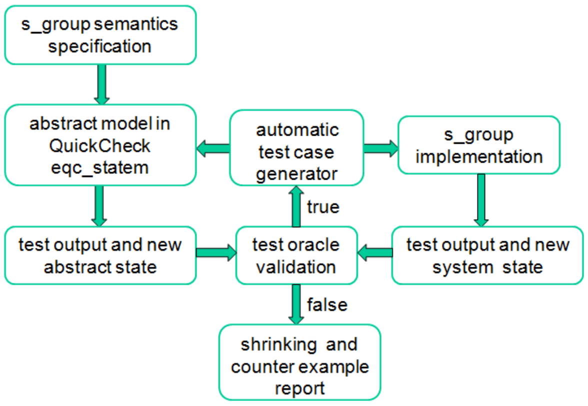
s_group semantics specification
abstract model in QuickCheck eqc_statem
automatic test case generator
s_group implementation
true
test output and new abstract state
test oracle validation
test output and new system state
false
shrinking and counter example report

Figure 2: Testing s_groups Using QuickCheck

```
1. prop_s_group() ->
2.  ?SETUP(
3.    fun setup/0,
4.    ?FORALL(Cmds,commands(?MODULE),
5.      begin
6.        {H,S,Res} = run_commands(?MODULE,Cmds),
7.        re_setup(),
8.        pretty_commands(?MODULE,Cmds,{H,S,Res},Res==ok)
9.     end)).
```

Figure 3: Testing s_groups: the top-level property

a successful execution, we specify that `Res` should be the literal '`ok`'. In the case that the property `Res==ok` fails to hold, the function `pretty_command` pretty-prints the execution history of the failing test, showing the calls made with actual arguments, the results and the model state.

The use of macro `?FORALL` specifies a property that holds if for test cases that can be generated the property `Res==ok` holds. The `?SETUP` macro is defined in QuickCheck to allow setting up and tearing down of the environment for one QuickCheck run. In line 3, `setup/0` is used to start a set of Erlang nodes in distributed mode with s_group support enabled before the QuickCheck run; this function returns a teardown function which is called after the QuickCheck run for shutting the SUT down. In line 7, `re_setup()` restarts and initialises the SUT after each test case.

4.2 Formal semantic specification of s_groups

In a distributed Erlang system with s_groups, three kinds of nodes are distinguished, and they are:

- s_group nodes: a s_group node is a node that belongs to at least one s_group.
- Free hidden nodes: a free hidden node is a non s_group node started with the command line flag `-hidden`. Connections between hidden nodes and other nodes are not transitive, and a hidden node does not share globally registered names with other nodes.
- Free normal nodes: a free normal node is a non s_group node started without the command line flag `-hidden`. Connections between free normal nodes are transitive, and transitively connected free normal nodes share the same global name space.

Both free hidden nodes and free normal nodes can join a s_group, hence become s_group nodes. When leaving an s_group, a node becomes free normal, or hidden, depending how the node was initially started. The semantic specification of s_group operations gives more detailed accounts about how the state of a node is affected after each operation, but that is not necessary for understanding the testing methodology presented, hence not covered in this paper. At the top level, the state of a distributed Erlang system with s_groups is formally modelled as a four-element tuple:

$$state \equiv (grps, fgs, fhs, nds), \text{ where}$$

$$\begin{aligned}
&grps \equiv \{s\_group\} \equiv \{(s\_group\_name, \{node\_id\}, namespace)\}\\
&fgs \equiv \{free\_group\} \equiv \{(\{node\_id\}, namespace)\}\\
&fhs \equiv \{free\_hidden\_group\} \equiv \{(node\_id, namespace)\}\\
&nds \equiv \{node\} \equiv \{(node\_id, node\_type, connections, gr\_names)\}\\
&gr\_names \equiv NoGroupName \mid \{s\_group\_name\}\\
&namespace \equiv \{(name, pid)\}\\
&connections \equiv \{node\_id\}\\
&node\_type \equiv Normal \mid Hidden
\end{aligned}$$

The first element of the tuple, *grps*, represents the set of *s_groups* in the system. Each *s_group* is represented by its name specified by *s_group_name*, the identifiers of nodes that belong to the *s_group*, and the global namespace (i.e. globally registered names within this s_group), that is replicated among the nodes in that *s_group*. A namespace is a set of two-element tuples each of which is a mapping from a globally registered name to a pid.

The tuple element fgs represents the set of $free\_groups$ in the system. A free group is a maximal set of transitively connected free normal nodes. A free group is represented by the identifiers of nodes that belong to this group and the global namespace that is shared by the group nodes.

The tuple element fhs is the set of $free\_hidden\_groups$ in the system. A $free\_hidden\_group$ consist of only one hidden node with its own namespace.

The tuple element *nds* is the set of all the nodes contained in the distributed Erlang system. Each node itself is also represented by a tuple consisting of the identifier of the node, *node_id*, the type of the node (normal or hidden), the identifiers of nodes that this node has connection with, as well as the names of s_groups this node belongs to. A free node has the group name of *NoGroupName*.

With the abstract state being defined, a s_group operation is then defined as a transition with the following form:

$$(state, \texttt{command}, ni) \rightarrow (state', value)$$

Among the operations supported by the s_group library, nine of them change the system state if performed successfully. The nine functions are the focus of our testing, and they are: `new_s_group/2`, `delete_s_group/1`, `add_nodes/2`, `remove_nodes/2`, `register_name/3`, `re_register_name/3`, `unregister_name/2`, `whereis_name/2` and `send/2`.

4.3 From specification to `eqc_statem` model

The translation from formal specification to `eqc_statem` abstract state machine was done manually. However the declarative nature of the Erlang programming language and the coding structure of `eqc_statem` allows a natural mapping from the formal specification to the abstract model. For instance, Figure 4 shows the state representation in Erlang, which is used by the `eqc_statem` model to hold the system state information during testing. The Erlang representation of state is defined as a record with four fields, each

```
-- 'state' is defined as a record with four fields.
--  For each field, its default value are specified.
-record(state,
         {s_groups          =[] ::[s_group()],
          free_groups       =[] ::[free_group()],
          free_hidden_groups=[] ::[free_hidden_group()],
          nodes             =[] ::[a_node()]}).

-- '-type' is used to define type synonyms.
-type s_group()::{s_group_name(),
                 [node_id()], namespace()}.
-type free_group()::{[node_id()], namespace()}.
-type free_hidden_group()::{node_id(), namespace()}.
-type a_node()::{node_id(), node_type(),
                 connections(), gr_names()}.
-type gr_names()::no_group_name|[s_group_name()].
-type namespace()::[{atom(), pid()}].
-type connections()::[node_id()].
-type node_type()::visible|hidden.
-type s_group_name()::string().
-type node_id()::node().
```

Figure 4: Abstract State Representation in Erlang

$$
\begin{array}{ll}
((\mathit{grs},\mathit{fgs},\mathit{fhs},\mathit{nds}), \text{new\_s\_group}(s,\mathit{nis}), \mathit{ni}) & \\
\quad \longrightarrow ((\mathit{grs}',\mathit{fgs}',\mathit{fhs}',\mathit{nds}''),(s,\mathit{nis})) & \text{If } \mathit{ni} \in \mathit{nis} \\
\quad \longrightarrow ((\mathit{grs},\mathit{fgs},\mathit{fhs},\mathit{nds}), \text{Error}) & \text{Otherwise} \\
\text{where} & \\
\qquad \mathit{nds}' \equiv \text{InterConnectNodes}(\mathit{nis},\mathit{nds}) & \\
\qquad \mathit{nds}'' \equiv \text{AddSGroup}(s,\mathit{nis},\mathit{nds}') & \\
\qquad \mathit{grs}' \equiv \mathit{grs} \oplus \{(s,\mathit{nis},\{\})\} & \\
\qquad (\mathit{fgs}',\mathit{fhs}') \equiv \text{RemoveNodes}(\mathit{nis},\mathit{fgs},\mathit{fhs}) &
\end{array}
$$

Figure 5: Semantic specification of a new_s_group

of which is of the type of `list`. As the field name indicates, each tuple element of the formal state tuple representation is represented as a record field.

The state transitions of s_group operations are implemented by the `next_state` callback function, which is the state transition function of the abstract state machine. This function takes the current abstract state, the symbolic function call to a test command, and returns the new abstract state. This function is also used internally by Quickcheck during test case generation to symbolically perform state transitions.

Let's take the *create a new s_group* operation as an example. The function `new_s_group(SGroupName, NodeIds)` creates a new s_group `SGroupName` consisting of the nodes specified by `NodeIds`. This function can only be performed from a node that belongs to `NodeIds`. Figure 5 shows the top-level specification of this operation. The skeleton of Erlang representation of the state transition in `eqc_statem` is as follows:

```
next_state(State, _V, {call, ?MODULE, new_s_group,
                 [SGroupName, NodeIds, CurNode]}) ->
  case lists:member(CurNode, NodeIds) of
     false -> State;
     true -> ... derive the new abstract state ...
  end.
```

In the code fragment above, the tuple
`{call,?MODULE,new_s_group,[SGroupName,NodeIds,CurNode]}` is a symbolic call to the `new_s_group` operation defined in the test module, where `SGroupName`, `NodeIds` and `CurNode` map respectively to *s*,*nis*, and *ni* in the formal specification. During test execution, `next_state` returns the new abstract state which is computed by the body of the `next_state` function; the actual value returned by the call to `new_s_group` is passed on to the *postcondition* checking instead.

The function `new_s_group` defined in the test module is a wrapper function in which the actual implementation of `new_s_group` is invoked. This function issues a remote procedure call (rpc) to the Erlang node `CurNode`, and asks it to perform the operation `s_group:new_s_group(SGroupName, NodeIds)`. After this rpc operation, the wrapper function issues a new rpc call to each node in the distributed system collecting their state information. The state information collected is then analysed and compared to the new abstract state in the *postcondition* checking phase of the testing, more details about this are covered in Section 4.5.

4.4 Test case generation

The abstract state machine model cannot be run without test case generation. With our test, automatic test case generation includes two levels of data generation: test command generation and command input data generation.

QuickCheck comes with a library of data generators. It supports not only the random generation of values of an Erlang built-in type, such as `integer`, `float`, `char`, `boolean`, etc, but also the generation of values of compound data types. It allows tuples and lists containing generators to be used as generators for values of the same form. For example, `{int(), bool()}` is a generator that generates random pairs of integers and booleans. In QuickCheck, constants can be used as generators for their own value.

While random data generation is used, QuickCheck provides a collection of macros for generating data that satisfy certain constraints. For example the macro `?SUCHTHAT(X, G, P)` generates values `X` from `G` such that the condition `P` is true; the macro `?SIZED(Size, G)` is used to control the size of the generated data, etc.

In the context of our testing, test data generators take the abstract state representation as input, and generate data that satisfies certain constraints. For instance, the following code fragment shows the random generation of a fresh non-empty s_group name: that is a string of lower case letters.

```
gen_s_group_name(_S=#state{groups=Grps}) ->
    %% get existing s_group names.
    GrpNames= [GrpName||{GrpName, _, _}<-Grps],
    %% generate a new unused s_group name.
    ?SUCHTHAT(Name, gen_reg_name(),
              not lists:member(Name, GrpNames)).

%% generate a non empty list of lower case letters.
gen_reg_name()->
    eqc_gen:non_empty(eqc_gen:list(
              eqc_gen:choose(97, 122)))).
```

Command sequence generation is controlled by the callback function *command(S)*, as shown in Figure 6. In the definition of `command`, each symbolic call is a command generator; the function `frequency` makes a weighted choice between the generators in its argument, proportional to the weight paired with it. In our case, the probability of choosing an s_group operation is half the probability of choosing a global name operation.

```
command(S) -> frequency(
    [{5, {call, ?MODULE, new_s_group,
                gen_new_s_group_pars(S)}}
    ,{5, {call, ?MODULE, add_nodes,
                gen_add_nodes_pars(S)}}
    ,{5, {call, ?MODULE, remove_nodes,
                gen_remove_nodes_pars(S)}}
    ,{5, {call, ?MODULE, delete_s_group,
                gen_delete_s_group_pars(S)}}
    ,{10,{call, ?MODULE, register_name,
                gen_register_name_pars(S)}}
    ,{10,{call, ?MODULE, whereis_name,
                gen_whereis_name_pars(S)}}
    ,{10,{call, ?MODULE, re_register_name,
                gen_re_register_name_pars(S)}}
    ,{10,{call, ?MODULE, unregister_name,
                gen_unregister_name_pars(S)}}
    ,{10,{call, ?MODULE, send, gen_send_pars(S)}}]).
```

Figure 6: Command Generator

Test sequences are generated by using *command(S)* repeatedly. However, generated calls are only included in test sequences if their precondition is also true. For instance, the `precondition` function defined below is used to make sure that the set of node identifiers fed to the `new_s_group` operation cannot be empty.

```
precondition(_S, {call, ?MODULE, new_s_group,
            [_SGroupName, NodeIds, _CurNode]}) ->
        NodeIds/=[];
```

4.5 Test Oracles

With the state transition functions and test case generators defined, we now have a nearly executable abstract state machine. However, with postconditions returning `True` by default, the execution will always be successful unless an exception occurs. For more rigorous testing, *postconditions* need to be defined for each operation being tested.

The call to `postcondition(S, C, R)` in an `eqc_statem` client module checks the postcondition of symbolic call `C`, executed in dynamic abstract state `S`, with result `R`. Postconditions are validated after execution of every test command. If a validation fails the failing test case is reported. In the context of our s_group testing, the actual result `R` is the value returned by applying a s_group operation to the distributed Erlang system under test, hence is independent of the abstract state `S`.

For the conformance testing of semantics and implementation, a natural test oracle is the consistent evolution of abstract state and actual state. In other words, the new abstract state derived by the application of the test command to the abstract state should be semantically equivalent to the actual state collected from the real system after the execution of the test command on the real system. This is the major test oracle used by our testing.

The postcondition of the `new_s_group` is as shown in Figure 7, in which the application `new_s_group_next_state` is used to derive the value returned and the new abstract state when applying the command to the abstract state `S`, and the function `is_the_same` compares the abstract state and the actual state. As one may have noticed, the calculation of the new abstract state is done twice during the testing, once in the `new_state` function and once is the `postcondition`. This could be avoided if QuickCheck allowed `new_state` to pass the new abstract state to `postcondition`.

```
%%Res is the actual value returned by the command;
%%ActualState is the state collected from nodes.
postcondition(S,{call, ?MODULE, new_s_group,
                 [SGroupName, NodeIds, CurNode]}),
              {Res, ActualState}) ->
  {AbsRes, NewS} = new_s_group_next_state(
                   S, [SGroupName, NodeIds, CurNode]),
  (AbsRes == Res) and is_the_same(ActualState, NewS).
```

Figure 7: Postcondition

Other invariants on either the abstract state or the real system state could also be specified. For instance, the three classes of nodes, i.e. s_group nodes, free hidden nodes and free normal nodes, should always form a partition of the set of nodes in a distributed Erlang system; a normal free node should not have any connections with free nodes outside its group and so on.

4.6 The Initial State

Before running the test, we need to define the callback function *initial_state* which returns the initial abstract state to start the testing with. When the semantics and implementation are to be simultaneously tested, one should make sure that the abstract state is initialised in the same way as the real system so that we have a consistent abstract and real state when test starts. With the testing of s_groups, the number of Erlang nodes included in the system as well as the type of each node (hidden or normal) are customisable, so the scale of the system under tested can be adjusted without extra effort.

5. RESULTS

The test code covering the nine s_groups operations contains 1,100 lines of code. So far, thousands of tests have been run using this test model. In this section, we summarise the kinds of errors encountered during testing.

- Errors in the test code. Test code is code, hence not immune from errors. As a result, some of the errors encountered, especially in the early stage of the testing, were errors in the test code itself.
- Errors in the semantic specification. In this case, the actual state is different from the abstract state after some test execution, and human examination identifies that the actual state represents the expected result. We found two semantic errors during testing. One error was that a free normal node was not properly removed from its original free group when the node joins a s_group; the other error was due to erroneous manipulation of the *gr_names* of a node resulting that *gr_names* contains both *NoGroupName* and a s_group name.
- Errors in the implementation. An error in the implementation also leads to a disagreement between the actual state and the abstract state, but in this case the abstract state represents the expected result. Our testing revealed two errors in the implementation. One error was due to the synchronisation between nodes where one node was expecting a `'nodeup'` message from another node but failed to receive it after a timeout although the

other node was actually up; the other error was related to the *remove_nodes* operation, where a mismatch between the expected result and actual value returned by a list search operation happened and crashed the Erlang node.

- Inconsistency between semantics and implementation. In this case, although the actual system state and the abstract state are equivalent, the value returned by the implementation and the abstract state machine are not always the same. In one case the formal semantics specified that the *send* operation should return `'undefined'` as the result if the message receiving process does not exist, however the actual implementation returned a tuple with the first element as `'badarg'` and the second element being the arguments supplied; in another case the semantics specified that the `'unregister_name'` operation always returns `'True'`, whereas the implementation could also return `'{no, cannot_unregister_from_remote_group}'`.

QuickCheck's support for shrinking is extremely helpful when testing fails. Minimising the sequence of commands leading to the failure makes it possible for manual execution and inspection of the failing case.

While our example jointly tests both semantics and implementation, the same model can also be used to test the semantics itself before the implementation is available. Of course, to do so we need to redefine the test oracles so that only properties about the abstract state are specified.

Overall, the automation of testing boosted our confidence in the correctness of the implementation and the semantic specification. As a link between the formal mathematical specification and the implementation, the abstract model also makes it more feasible for the co-evolution of specification and implementation. An executable abstract model provides us with a means to explore the new features to be added to the library.

6. RELATED WORK

There are many tools and methodologies for random test generation and for model based testing, and their usefulness has been demonstrated in many cases. For instance, In [10], Yang, Chen, et.al. reported their use of randomized differential testing technique to find hundreds of bugs in production C compilers. Instead of attempting a formal specification of the C-language semantics, they used a black-box approach by generating random C programs and comparing the output of several compilers.

Model-based testing has attracted extensive research interest because it offers substantial advantages over traditional software testing methods. Model-based testing requires the definition of a suitable model, the generation of the actual test suite and the actual execution of the generated cases. Various techniques have been proposed for each aspect of the testing process. A detailed taxonomy of model-based testing is given in [8] by Utting, et. al.

QuickCheck supports property- and model-based testing in a rather lightweight. Similar to our work, in [9], Vasconcelos reports the use of Haskell QuickCheck to express the correctness of a toy compiler against a denotational semantic specification; however their testing was not state-based. In Haskell, writing and testing against model-based specifications can be achieved using a monadic property language [6].

7. CONCLUSIONS AND FUTURE WORK

A specification is of less value if there is no check at all that it corresponds to the implemented program. The gap between specification and implementation somehow discourages programmers to formulate formal specification. In this paper, we demonstrated how tools like QuickCheck can serve as the link between formal specification and implementation. Through the testing of a proposed extension to the Erlang programming language, this paper reports a property-based random testing approach to automating and unifying the testing of the semantics and its implementation, so that testing of both the implementation and specification, as well as the conformance between them, can be done simultaneously.

Property-based testing provides a high level approach to testing in the form of abstract invariants that functions should satisfy universally. Together with automatic test data generation, property-based testing could easily generate and apply thousands of tests that would be infeasible to write by hand, and covering subtle corner cases that would not be found otherwise.

As to future work, we would like to further explore this approach to test other critical language components, or libraries, written either in Erlang or other programming languages, such as C, supported by QuickCheck. We foresee that the same methodology can be used to test other components of Erlang/C as long as there is some way of observing the state of the implementation, and the semantics is deterministic. We would also like to investigate ways to automate the generation of formal mathematical specification from a QuickCheck model.

This research is supported by EU FP7 project RELEASE, no. 287510. We would like to thank Natalia Chechina and Phil Trinder for providing the s_group specification.

8. REFERENCES

[1] RELEASE. `http://www.release-project.eu/`.

[2] O. Boudeville, F. Cesarini, et al. RELEASE: A High-level Paradigm for Reliable Large-scale Server Software. In *TFP'12*, St Andrews, UK, 2012.

[3] F. Cesarini and S. Thompson. *Erlang Programming.* O'Reilly Media, Inc., 2009.

[4] N. Chechina, P. Trinder, A. Ghaffari, et al. The Design of Scalable Distributed Erlang. In *Draft Proceedings of IFL'12*, Oxford, UK, 2012.

[5] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ACM SIGPLAN Notices*, 2000.

[6] K. Claessen and J. Hughes. Testing Monadic Code with QuickCheck. In *ACM workshop on Haskell*, 2002.

[7] J. Hughes. QuickCheck Testing for Fun and Profit. In *PADL'07*, Berlin, 2007.

[8] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-based Testing Approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312, 2012.

[9] P. Vasconcelos. Experience Report: Verifying a Simple Compiler Using Property-based Random Testing. `http://www.dcc.fc.up.pt/~pbv/compcheck/`.

[10] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In M. W. Hall and D. A. Padua, editors, *PLDI*, 2011.