Automated Test Generation for Industrial Erlang Applications

Johan Blom
Mobile Arts AB
Tjärhovsgatan 56
SE 102 67 Stockholm, Sweden
Johan.Blom@it.uu.se
Johan.Blom@mobilearts.se

Bengt Jonsson
Department of Computer Systems
Uppsala University, Sweden
Box 337, SE-751 05, Sweden
Bengt.Jonsson@it.uu.se

ABSTRACT

We present an implemented technique for generating test cases from state machine specifications. The work is motivated by a need for testing of protocols and services developed by the company Mobile Arts. We have developed a syntax for description of state machines extended with data variables. From such state machines, test cases are generated by symbolic execution. The test cases are symbolically represented; concrete test cases are generated by instantiation of data parameters.

1. INTRODUCTION

This paper describes methods and tools for specification and functional testing of telecommunication protocols and services, which are being developed in collaboration between the company Mobile Arts and the Department of Computer Systems at Uppsala University. Mobile Arts design, develop & market SS7 signaling gateway products that e.g., enables IP-based service applications to retrieve/send information, e.g., subscriber whereabouts, terminal/subscriber/service status and messaging from/to 2G, 2.5G and 3G mobile networks. To a large extent, Mobile Arts uses Erlang for specification, implementation and testing, which is belived to be a key factor for delivering telecom grade products in a timely manner with a limited number of designers.

The core modules in signaling gateway products typically perform protocol processing, using services of the SS7 network, in order to deliver an interesting service to an application residing in the control of a telecom operator. To support the implementation of such a module, Mobile Arts formally specify its functional behavior, for which sequence diagrams and state machines extended with data have turned out to be a suitable formalism. The actual implementation is performed manually, based on a detailed functional description. It is therefore necessary to validate that the specification conforms to the functional description and test that the implementation actually conforms to the specification. For a core module, testing should be thorough.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Erlang Workshop '03, 29/08/2003, Uppsala, Sweden
Copyright 2003 ACM 1-58113-772-9/03/08 ...$5.00.

In this paper, we present the specification formalism and our implemented technique for generating test suites from state machine specifications. For the state machine formalism, one desirable aspect is that specifications should resemble Erlang programs: Erlang developers will feel more comfortable when reading them, and it will be easier to build tools in Erlang to process them. Instead of using an existing syntax, we have therefore defined an "Erlang-like" syntax, which, e.g., can use matching and ordered clauses to process inputs. We have not yet produced tools to execute or compile state machine specifications. Instead we have concentrated on a tool to produce test suites for checking that an implementation conforms to a specification given as a state machine.

We present a technique for generating test suites from state machine specifications. We generate test cases that correspond to sequences of computation steps of the state machine. Each such sequence is determined by the initial configuration of the state machine and the actual inputs received in the computation steps. Our ambition is to generate a test suite which, if possible, checks the implementation for all possible combinations of initial and input values. This is of course impossible in general, but the product that Mobile Arts currently intends to test has the following properties which could make this tractable.

- Most state machine variables and parameters of inputs take values from domains with very small ranges (typically 2).
- Each sequence of computation steps is rather short (typically $2-4$ steps), stemming from the fact that it represents a transaction which is initiated by a request from a user, performs a few "queries" over the network, and concludes by a response back to the user.

In this paper, we present a technique for generating a representation for all possible test cases, consisting of a tree that spans the control paths of the state machine together with a function which symbolically represents the parameters of outputs in terms of initial state variables and parameter of

received inputs. This representation is convenient to use for automated test execution: one simply enumerates the possible values of input parameters, and uses the function to check that the output from the implementation is as expected. For the type of products considered by Mobile Arts, the test execution data must furthermore be distributed over modules that simulate users and modules that simulate the responses of other hosts in an SS7 network.

At the current state, the applicability of our technique has the limitation that data variables of state machines must be expressible as booleans. Other data types must be manually abstracted and encoded into booleans. An exception is that a data type representing the identity of a transaction or user can be used for parameters of events. In the current test generation tool, it is assumed that such a parameter exists, and it is used as an identifier for test cases. A subset of possible test cases can be selected by giving contraints in cases where the test suite would otherwise become intractably large. We also consider only the functional testing of expected functionality of a single instantiation of the tested module. One should test also the response to unspecified inputs, potential interferences between concurrent instantiations of the tested modules, etc. We plan to consider these extensions in the future.

The paper is organized as follows. This introduction continues with a short overview of some related work. Section 2 introduces a product at Mobile Arts, which originally motivated this work. The state machine formalisms is described in Section 3, followed by a description of a "core version" in Section 3.1, and of the test case generation in Section 4. The current capabilities of our implemented tool, and results about applying it to the product described in Section 2 is described in Section 5. Finally, Section 7 contains conclusions and directions for future work.

Related Work

There are commercial tools for generating tests from state machine specifications, such as Autolink or TestComposer [7], and TGV [4]. These derive test sequences from the combination of a test purpose specification in MSC and a state machine model in SDL. A symbolic version of this approach is done by Rusu et al. [6]. The approach in this paper does not explicitly use a sequence diagram to select test cases. An approach handling testing of Erlang applications can be found in [8].

2. CASE STUDY - MOBILE ARTS NPC

Mobile Arts NPC (Network Presence Center) is a middleware product to allow Mobile Network Operators to provide various presence information from the GSM network. For example an instant messaging application may want to know if a certain mobile phone is turned on or off, to decide if a user is reachable. Applications using the NPC are typically hosted on the internet or at the mobile operators domain communicating with NPC via HTTP over IP and an XML based protocol. NPC in turn uses the SS7 protocol stack to communicate with the GSM network. This environment causes high requirements on availability and fault tolerance, thus a need for careful testing of the system.

The implementation of NPC was made mainly in Erlang, utilizing Erlang OTP, with approximately 130000 lines of Erlang and 5500 lines of C source code. Development was made in a typical fashion by first creating a requirement specification upon which a detailed functional specification consisting of a textual description and a set of MSC:s was created. Finally the implementation was based on the functional description. During this process there was large number of updates, creating problems to keep the three descriptions of the system consistent. In order to create good tests it was thus decided to formalize the functional specification and base the test creation on the formal specification.

Testing is handled by a *Test Client* issuing XML requests toward NPC, the XML response is then examined for correctness. While handling the XML request, NPC may issue a number of requests to a simulated network *Netsim*. The network then responses in a predefined way, decided when setting up the tests. By setting parameters in the XML request, decide configuration data, and providing *Netsim* with relevant data the response can be predicted. Furthermore, data delivered by *Netsim* is available in the XML response in such a way that correctness of the complete test trace can be examined.

3. ERLANG-EFSM

It is our assumption that the specification effort should be distinguished from the implementation effort, that may in addition include more details and specific solutions because of e.g., performance issues. Furthermore, a formal specification language should be easily understood by those implementing the desired product as well as by those thinking on a higher, more architectural level. With an implementation in Erlang it is thus a natural choice to use a syntax compatible with Erlang [2] as a specification language. Here we specifically have in mind a subset enough to handle Extended Finite State Machines (EFSM). In Erlang-EFSM we use function declarations to represent incoming events, and function applications to represent outgoing events. A state transition is made by assigning a dedicated variable, *NextState*, the name of the next state. State variables must be declared by including them in a *#state* record definition. Furthermore we take advantage of the pattern matching and evaluation order found in Erlang, keeping the specification small.

A specification in Erlang-EFSM has many similarities with the gen_fsm behavior found in Erlang OTP [3], specifically

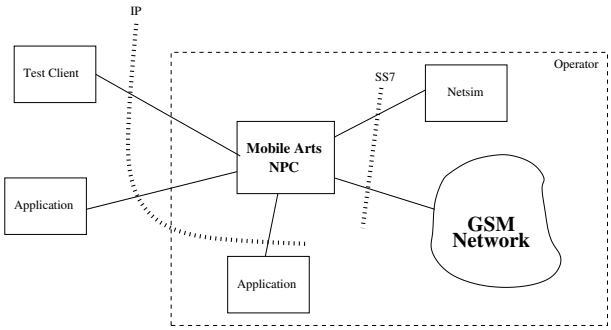


Figure 1: Mobile Arts NPC

designed to implement a generic behavior for handling of a concurrent EFSM process. As we are specifically interested in specifying state machines and generate tests on these a few simplifications have been made, in particular:

- Concurrency is not supported, thus handling of start, initializing and terminating of an EFSM process, as found in gen_fsm, is not covered.
- Instead of having a State variable passed between the states, as in the Erlang OTP gen_fsm behavior, state variables are declared in a state record and assumed to be global.
- Erlang-EFSM does not have single-assignment semantics. Thus variable names are reused such that a global variable will always be referred to with the same name.
- The name of each control state is declared by assigning it to the @State variable. This is a special variable recognised by the Erlang-EFSM parser.
- Certain values have an associated implicit type, e.g., values of the boolean type are represented with the atoms 'true' and 'false'.

The given simplifications makes it easier to implement test case generation and validation tools. To be able to generate finite test sequences we also assume a deterministic behavior and assumes all possible traces are finite.

EXAMPLE 1.

```
%%% Global State variables used
-record(state,{'Bit','Buf','Cnt'}).

%% @State=idle
start(Uid) ->
    Bit=false,
    Cnt=false,
    NextState=send_data.

%% @State=send_data
send(Data) ->
    data(Uid,Data,Bit),
    Buf=Data,
    NextState=wait_ack.

%% @State=wait_ack
ack(AckBit) ->
    case AckBit of
       Bit when Bit==true ->
          stop(Uid,ok),
          NextState=done;
       Bit when Bit==false ->
          Bit=true,
          send_ok(Uid),
          NextState=send_data;
       _ ->
            if
                Cnt==false ->
                    Cnt=true,
                    data(Uid,Buf,Bit),
                    NextState=wait_ack;
                Cnt==true ->
                    stop(Uid,fail),
                    NextState=done
            end
    end.
```

An Erlang-EFSM specification of a sender utilizing the Alternating Bit Protocol can be found in Example 1. The sender contains three state variables; Bit is the control bit, Buf a buffer used to store sent data, and Cnt a retransmission counter. The sender is started with a $start()$ event in control state $idle$, initializing variables which cause the sender to go to the $send\_data$ state. On a $send(Data)$ event the sender goes to $wait\_ack$ and waits for the acknowledge, $ack(AckBit)$, with $AckBit$ set to a correct value. The sender only allows 2 messages to be sent, and max 1 resend, then it needs to be restarted.

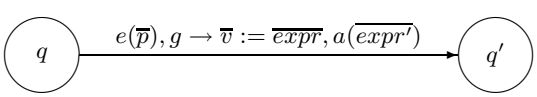
3.1 Core Semantic Model for Extended Finite State Machines

For the presentation of our test generation technique, in Section 4, we will not use the full Erlang-EFSM notation, but a intermediate "core" EFSM notation, which contains only the elements necessary for representing extended state machines.

A deterministic *Extended Finite State Machine, EFSM* is a tuple $\langle Q, q_0, \overline{v}, confs, E, \delta \rangle$ where

- Q is a finite set of *control states* (sometimes called locations).
- $q_0 \in Q$ is the initial control state.
- $\overline{v} = \{v_1, \ldots, v_n\}$ is a finite set of *state variables*. Each variable v_i ranges over a domain D_i. The initial values of all state variables are undefined.
- $\overline{c}$ is a finite set of named *configuration constants*, each of which has a value that is defined initially and cannot be modified during normal protocol operation.
- E is a finite set of *event types*. With each event type e is associated a fixed tuple of formal parameters, (called the *formal parameters* of e) each of which ranges over a given domain. An expression of form $e(\overline{expr})$ consisting of an event type e and expressions, whose ranges are in the domains of its formal parameters is called an *event expression*; if $\overline{expr}$ are the formal parameters of e it is called a *parameterized event*; if $\overline{expr}$ are values $\overline{d_p}$ in the domains of the formal parameters, it is called an *event*.

  Each event type is either an input or output event, and is either exchanged with the user or the network. We will use the terms *network input event type*, *user output event type*, etc. with obvious meanings.
- δ is a set of *transitions*. Each transition is of form



  where
  - q is the source location, and q' is the target location,
  - $e(\overline{p})$ is a parameterized input event (either from the user or from the network),

- g is a guard,
- $\overline{v} := \overline{expr}$ is an assignment of new values to the state variables,
- $a(\overline{expr'})$ is an output event expression,
- g, $expr$, and $expr'$ may depend on the formal parameters $\overline{p}$ of the input event, the state variables $\overline{v}$, and the configuration constants $\overline{c}$.

Additionally, we can specify some control states as *final states*, or alternatively view a control state without outgoing transitions as a final state.

A *system state* is a tuple $\langle q, \overline{d_v}, \overline{d_c}\rangle$ where q is a control state, $\overline{d_v}$ are values (in the appropriate domains) of the state variables, and $\overline{d_c}$ are values of the configuration constants. A *computation step* is a four-tuple

$$\langle q, \overline{d_v}, \overline{d_c}\rangle \quad \xrightarrow{e(\overline{d_p})/a(\overline{d_p}')} \quad \langle q', \overline{d_v}', \overline{d_c}\rangle$$

consisting of system states $\langle q, \overline{d_v}, \overline{d_c}\rangle$ and $\langle q', \overline{d_v}', \overline{d_c}\rangle$, an input event $e(\overline{d_p})$, and an output event $a(\overline{d_p}')$. Such a computation step is an *instance* of a transition of the form described above (i.e., from control state q to q' labeled by $e(\overline{p}), g \rightarrow \overline{v} := \overline{expr}, a(\overline{expr'})$), if

- $g[\overline{d_p}, \overline{d_v}, \overline{d_c}/\overline{p}, \overline{v}, \overline{c}]$ is true, and
- $\overline{d_v}' = \overline{expr[\overline{d_p}, \overline{d_v}, \overline{d_c}/\overline{p}, \overline{v}, \overline{c}]}$, and if
- $\overline{d_p}' = \overline{expr'[\overline{d_p}, \overline{d_v}, \overline{d_c}/\overline{p}, \overline{v}, \overline{c}]}$.

In the following, we will omit the configuration constants and their values when describing computation steps, etc., to save space. One could think of this as including configuration constants among the state variables.

Assumptions

We assume that all EFSM that we consider are deterministic, in the sense that any two outgoing transitions from a state, labeled by the same parameterized input event, have non-overlapping guards. For the test case generation, we further assume that the EFSM allows no infinite sequences of computation steps.

4. TEST CASE GENERATION

We intend to generate test cases that correspond to complete sequences of computation steps of an EFSM specification. Such a sequence has a form

$$\langle q_0, \overline{d}_0\rangle \quad \xrightarrow{e_1(\overline{d}_1')/a_1(\overline{d}_1'')} \quad \cdots \quad \xrightarrow{e_n(\overline{d}_n')/a_n(\overline{d}_n'')} \quad \langle q_n, \overline{d}_n\rangle$$

where q_0 is an initial control state, and q_n is a final control state. Typically, the first input event type e_1 is received from the user, all other events, except the last one (i.e., $e_2, \ldots, e_n$ and $a_1, \ldots, a_{n-1}$ are exchanged with the network, and a_n is returned to the user. Such a sequence of computation steps naturally represents a transaction from the point of the of the user, and it is natural to regard each such transaction as a test case.

We intend to generate a representation of all test cases that are defined by an EFSM specification, i.e., sequences of computation steps that start in the initial control state and end with an event returned to the user. By assumption, there are not infinite sequences of computation steps.

Since the EFSM is deterministic, a test case is determined by the initial values of state variables and configuration constants, and by the values of parameters received in input events. In most cases, it would be too space consuming to explicitly enumerate all possible input values. Another alternative is to regard the EFSM "as is" as an implicit representation of test cases. During test execution, the specification then acts as a reference which is simulated in order to calculate the expected output events of each test case. This solution may work well if the test execution is performed entirely within one computer. However, if the test execution is performed over a physical network, as in Figure 2, it is desirable to statically precompute and separate the behavior of the test driver and of the network elements.

This motivates an intermediate approach, in which we compute a set of *symbolic test cases*. Each symbolic test case consists of

- a control path from the initial state of the EFSM to a final state,
- an output function, which is a symbolic expression which expresses the values of parameters in output events in terms of parameters received in earlier input events and initial values of configuration constants,
- a guard, which gives the conditions on parameters received in earlier input events and initial values of configuration constants under which this control path will be taken.

The set of symbolic test cases can be computed by unfolding the EFSM into a tree, and by computing the output functions and guards by symbolic execution. More precisely, our algorithm generates a tree of nodes. Each node in the tree is labeled by a triple of form

$$\langle q \quad , \quad g \quad , \quad \overline{v} = \overline{expr}\rangle \quad ,$$

where q is a control state of the EFSM, g is a boolean expression, and $\overline{v} = \overline{expr}$ expresses the values of state variables in terms of an expression $\overline{expr}$. Both g and $\overline{expr}$ may depend only on parameters $\overline{p}_1, \ldots, \overline{p}_i$ of earlier input events, and on initial values of configuration constants $\overline{c}$. The intended meaning of such a triple is as follows. By construction, the path from the root node to this node is of form

$$(q_0) \quad \xrightarrow{e_1(\overline{p}_1)/a_1(\overline{expr_1})} \quad \cdots \quad \xrightarrow{e_i(\overline{p}_i)/a_i(\overline{expr_i})} \quad (q_i)$$

In this context

- q is the last control state q_i,
- the guard g is the condition on parameters of input events $e_1(\overline{p}_1), \ldots e_i(\overline{p}_i)$ and configuration data under which this path will be executed.feasible
- The equality $\overline{v} = \overline{expr}$ expresses the values of state variables $\overline{v}$ after execution of this symbolic path.

Our algorithm for constructing a completed tree of symbolic test cases is then the following.

1. The initial node is labeled by a triple $\langle q_0 \ , \ true \ , \ \rangle$, where q_0 is the initial control state, and where there are still no values of state variables (this is no problem, since they are not live).
2. If a node is labeled by a final control state, then no leaves are created from the node.
3. Otherwise, if a node is labeled by $\langle q \ , \ g \ , \ \overline{v} = \overline{expr} \rangle$, then for each transition in the EFSM from control state q to another control state q', labeled by $e(\overline{p}), g' \rightarrow \overline{v} := \overline{expr'}, a(\overline{expr''})$, we create a child of the node labeled by the triple

   $$\left\langle q' \ , \ g \wedge g'[\overline{expr}/\overline{v}] \ , \ \overline{v} = \overline{expr'[\overline{expr}/\overline{v}]} \right\rangle \quad .$$

   Intuitively, this represents the postcondition of the action $g' \rightarrow \overline{v} := \overline{expr'}$. The arc to that node is labeled by

   $$e(\overline{p})/a(\overline{expr''[\overline{expr}/\overline{v}]}) \quad .$$

   Here $\overline{expr''[\overline{expr}/\overline{v}]}$ expresses the values of the parameters of the output event in terms of parameters of previous input events and configuration constants. This expression can be used to check the output from the module under test during test execution.

When the algorithm finishes, it has generated a tree of symbolic test cases. It now remains to use this representation to generate appropriate information to the test execution environment.

The test environment is distributed over several nodes, see Figure 2.

- There is a *Test Client*, which supplies the initial user input and receives the final output to the user, and checks its correctness.
- There is a module, called *Netsim*, which generates input over the network.

In order to coordinate the different modules in the test environment, we associate with each test case a unique *key*. This key should uniquely determine the initial values of configuration constants, and the sequence of input events (including their parameters) that will be transmitted in a test case.

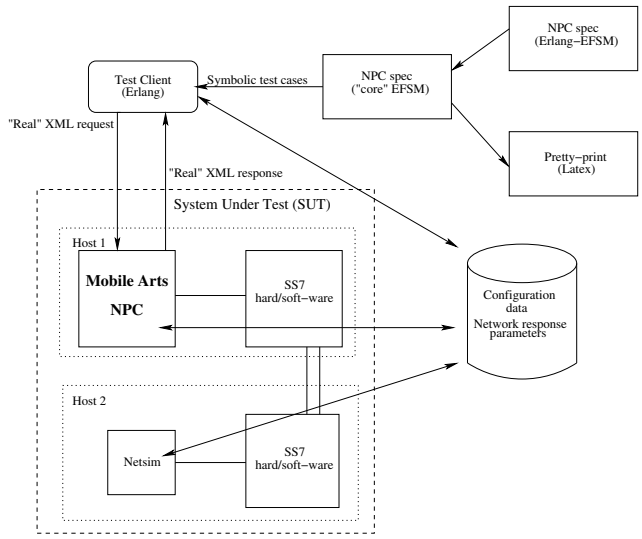


Figure 2: Overview of testing environment for the NPC product.

In order to coordinate the inputs of the Test Client and Netsim module in each test case, we use a particular feature of the class of products that we consider, namely that all events have a particular parameter, a *user id* (hereafter called *uid*), which is an identifier of the user that initiates the transaction represented by the test case. This parameter is forwarded in all events of a transaction. The *uid* is used by the Test Client as follows. The Test Client transmits the first input event $e(\overline{d})$ to the SUT. The *uid* (which is one parameter among $\overline{d}$) and the the other parameters of $\overline{d}$ then uniquely determine the remaining test case, in particular the parameters of subsequent input events from the network. This puts a burden on the *Netsim* module: when the module under test transmits an event $e(\overline{d})$, then the Netsim module must be able to determine the next input event from e and $\overline{d}$. This is done as follows.

- For each network output event type e, we define a function, which determines the reponse to network output events of form $e(\overline{d})$
- From the tree of symbolic test cases, we determine for each network output event type e the set of input events that may follow an event of form $e(\overline{d})$ in a test case. We thereafter "reserve" a certain set of bits in the binary representation of *uid* to represent all possible combinations of event types and parameter values in input events that may follow an output event of form $e(\overline{d})$.
- If an output event type may occur several times in a test case, then a separate set of bits must be reserved for the first occurrence, for the second occurrence, etc.
- In the worst case, we may need to reserve a fixed number of bits in the binary representation of *uid*s for each output event type. However, if two output event types never occur in the same test case, the same set of bits can be used to represent choices of subsequent input events for both event types.

We can now define the behavior of the Test Client and Netsim modules as follows:

- The *Test Client* generates keys of all possible test cases, and sends input requests with parameters computed from the key. It then waits to receive the final output from the module under test, and compares this output with the output computed by the function in the symbolic test case.
- The *Netsim* module waits to receive output from the module under test. Whenever an event of form $e(\overline{d})$ arrives, Netsim uses the key (which is one of the parameters in $\overline{d}$) to determine the appropriate next network input event. Netsim should also compare the event $e(\overline{d})$ with the output computed by the appropriate function in the symbolic test case (we have not yet implemented this last function in Netsim).

5. TOOL SUPPORT

A tool has been developed to transform Erlang-EFSM specifications to the "core" EFSM notation and to automatically generate symbolic test cases from this. The resulting test cases are represented by an Erlang function with parameters for all input parameters and configuration constants accessible. Each function clause has a guard equal to the calculated symbolic precondition, and a body with a specification dependent response record, containing the parameters of the stop event. Running a test thus corresponds to choosing an instance of input parameters and configuration constants, and evaluate an Erlang function. The tool can also generate a pretty printed Latex version of the "core" EFSM notation.

While scanning through the Erlang-EFSM specification, some sanity checks of the specification are made. In particular:

- "Dead code" is discovered by evaluating preconditions, i.e. by a tautology test on all guards to function clauses for a function, case clauses for a case expression and if clauses for an if expression.
- Detection of suspected loops in control paths, by restricting the maximum allowed length.
- A semantic check is made on state variables such that they maintain the same type (e.g., boolean) during the complete control path, and always have a defined value when used in a precondition.

Also note that as the evaluation order of Erlang is assumed, all implicit preconditions are calculated and added to the corresponding precondition in the "core" EFSM model.

```
start(Uid,I) when element(1,I#info.ackbit)==true and
                  element(2,I#info.ackbit)==true ->
    #stop{result=fail}.
```

Figure 3: Generated symbolic test case, in Erlang.

A symbolic test case generated from the Erlang-EFSM specification in Example 1 can be found in Figure 5. Here the function start is the incoming start event from the user accepting two parameters; `Uid` is the User identifier, and `I` a record that contains all values expected from client responses for this particular `Uid`. The `I#info.ackbit` tuple thus contains all responses, in the right order, expected in the $ack(AckBit)$ events for the test case.

5.1 Limiting the Number of Test Cases

If our test sequence generation algorithm is used naively, it may generate all possible combinations of input parameters in all symbolic test cases. This can cause considerable redundancy. We consider the following techniques for limiting the explosion in test cases.

- In a symbolic test case, there may be several parameters whose values are not used for computing output events. These are parameters that do not occur in guards or output functions of the symbolic test case. When generating concrete test cases from such a symbolic test case, it appears sensible to use a default setting for unused parameters and vary only the used parameters.
- If the number of concrete test cases is still to large to be executed, one may apply filters on the set of possible combinations of parameter values. An example of a possible such filter is to generate sufficiently many combinations of input parameters so that there is one test case for each set of values of any subset of n parameters, where n may be chosen suitable small (smaller than the total number of parameters).
- For large specifications the number of symbolic test cases can be too large to handle. In our tool, we can therefore constrain the set of symbolic test cases by imposing arbitrary constraints on the parameters. Such invariants are used to minimize the specification by projecting it to a "smaller"one before any tests are generated. An invariant can also be passed to the test case generation algorithm, all computations invalidating the invariant are then immediately dropped. By varying these constraints, we can generate all symbolic test cases divided into a number of "chunks". We may also use these constraints to tailor test suites for a specific set of functionality.

The conclusion is that in our tool, we should have the following facilities.

- Constraining the values of parameters when generating symbolic test cases,
- In a symbolic test case, it should be possible to specify a notion of "coverage", stating e.g., that all combinations of certain subsets of parameters be exercised.

6. TESTING OF MOBILE ARTS NPC

The formal specification of Mobile Arts NPC is an Erlang-EFSM specification of about 2400 lines. It consists of 12 control states, a single input event type from the user, a single output event type to the user, and utilizes 14 state variables, $\overline{v}$.

Before any tests are generated, all parameters of the input events are carefully examined and grouped in equivalence classes to minimize the number of tests. For example, we may get many possible error codes from the network, but will according to the specification only react in two different ways in all those cases. We can then conclude that this error code can be represented with a boolean covering both of these cases. Thus creating input data so that a minimum number of test cases needs to be generated. We should now have full source code coverage of the EFSM in the SUT (see the Cover tool in [3]).

For NPC the input parameters can be summarized as:

- 11 configuration parameters, $\overline{c}$
- 9 user request event parameters, $\overline{p}$, where $e(\overline{p})$ and $e \in E_I^{user}$
- 8 network response parameters, $\overline{p}$, where $e(\overline{p})$ and $e \in E_I^{net}$

Summing up to a total of 2^{30} possible combinations of input data. Once all test cases has been generated according to Section 4 and a network database has been filled with response data, tests can be created. Clearly it is not possible to generate tests for all input combinations, but examining the symbolic test cases more carefully it turns out many input combinations are redundant. We are also working on reducing necessary tests further, see Section 7. It should be noted that having a complete coverage of all symbolic test cases can be considered good enough for testing the EFSM alone, but generating tests for more input combinations ensures further testing of interfaces etc.

Although the system is rigorously tested with respect to the generated tests, it must be pointed out that only a subset of all the possible tests are generated. In particular the following aspects are not covered by the generated tests:

- All operation and maintenance related functionality, including log handling, alarms, counters, GUI and a command line interface.
- Invalid input data, such as bad XML or unexpected SS7 traffic.
- All High Availability (HA) aspects such as node crash handling, fail over etc.
- Testing with high load of the system. The implementation uses a number of queues and timers, not visible in the specification that may generate errors.

7. FUTURE WORK

This is still a novel project, with a number of possible future directions. Of specific interest are:

- Validation of an Erlang-EFSM specification against requirement specifications and MSC:s. Transformation to the "core" EFSM notation makes it simpler to use existing tools such as the symbolic model checker SMV for validating properties. Validation of traces found in MSC:s is another impossibility
- Further limit the number of uninteresting test cases. This could for example be done by:
  - Use some coverage criteria when generating the symbolic test cases, see e.g., [5].
- Transformation between Erlang and Erlang-EFSM. The syntax is deliberately chosen close to that of Erlang, to be able to easily transform between theses languages. There are two possibilities:

  Erlang → Erlang-EFSM Involves doing an abstraction to create test traces only based on interesting properties. A related work can be found in [1], where Erlang programs are automatically translated to the μCRL formalism.

  Erlang-EFSM → Erlang This can be handled by creating an Erlang stub from the Erlang-EFSM, with call-backs to implementation specific details. A problem is how to handle cases where we want to hide a more efficient implementation solution from the specification.

  A goal is to try to automate the transformation.
- Handling of concurrent configurations. This involves handling of communication between state machines, spawning of new processes and message passing.

8. REFERENCES

[1] Thomas Arts, Clara Benac Earle, and John Derrick. Verifying Erlang code: a resource locker case-study. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *Formal Methods Europe: Getting IT Right, Copenhagen, Denmark*, volume 2391 of *Lecture Notes in Computer Science*, pages 184–203. Springer Verlag, July 2002.

[2] Jonas Barklund and Robert Virding. Erlang 4,7.3, reference manual. Draft 0.7, June 1999.

[3] Ericsson. Erlang OTP R9-B1, 2003.

[4] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protoc ols with verification technology. *Science of Computer Programming*, 29, 1997.

[5] Anders Hessel, Kim G. Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-optimal real-time test case generation using UPPAAL. Submitted for publication 2003.

[6] V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *Int. Conf. on Integrating Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer Verlag, 2000.

[7] M. Schmitt, M. Ebner, and J. Grabowski. Test generation with autolink and testcomposer. In *Proc. 2nd Workshop of the SDL Forum Society on SDL and MSC - SAM'2000*, June 2000.

[8] Ulf Wiger, Gösta Ask, and Kent Boortz. World class product certification using Erlang. In *PLI 2002, ACM SIGPLAN Erlang Workshop*, October 2002.