Applying Testability Transformations to Achieve Structural Coverage of Erlang Programs

Qiang Guo, John Derrick and Neil Walkinshaw

Department of Computer Science,
The University of Sheffield,
Regent Court, 211 Portobello, S1 4DP, UK
{Q.Guo, J.Derrick, N.Walkinshaw}@dcs.shef.ac.uk

Abstract. This paper studies the structural testing of Erlang applications. A program transformation is proposed that represents the program under test as a binary tree. The challenge of achieving structural coverage can thus be interpreted as a tree-search procedure. We have developed a testing-technique that takes advantage of this tree-structure, which we demonstrate with respect to a small case study of an Erlang telephony system.

Keywords: Erlang, Testing, Transformation, FBT, Structural Coverage.

1 Introduction

Erlang [1] was, along with its Open Telecoms Platform (OTP), originally developed by Ericsson for the rapid development of network applications. However, its usage has now spread beyond that domain to a number of sectors. Erlang has been designed to provide a paradigm for the development of distributed soft real-time systems, where multiple processes can be spread across many nodes in a network.

With its OTP libraries, complex Erlang applications can be rapidly developed and deployed across a large variety of hardware platforms, and this has caused it to become increasingly popular, not only within large telecoms companies such as Ericsson, but also with a variety of SMEs in different areas. It is increasingly used to develop applications that are business-critical, for example, its use in Ericsson's AXD-301 switch that provides British Telecom's internet backbone.

The ability to rigorously test Erlang applications is vital. In recognition of this, there has recently been a concerted drive to develop more automated testing tools, which complement the rapid Erlang development cycle. Tools such as QuickCheck [2] are being eagerly adopted within the community.

So far, the main thrust of the Erlang testing effort has been directed towards functional testing. Structural code-coverage has not been considered. This is however an important problem, and is often a requirement if the system under test is to be certified for a range of safety standards (e.g. the US Federal Aviation Authority software standards [13]). Achieving code coverage is a well-established challenge; the tester has to find a suitable set of test inputs that

will exercise every branch for every predicate in the program. Finding a suitable (and reasonably sized) set of inputs is often problematic, and a number of techniques such as symbolic execution [7], evolutionary search algorithms [9] and testability-transformations [6] have been proposed to address it.

This paper presents a technique that leverages certain features of the Erlang language to enable complete structural test coverage. The approach is inspired by Harman et al.'s notion of testability transformations [6]. We have developed a transformation that transforms an Erlang program into a binary tree. Each node in the tree corresponds to either a true or a false evaluation of a predicate in the original program. Predicates that may be hidden within functions in the original program are made explicit. Identifying a suitable test set to achieve structural coverage is thus reduced to a search over the binary tree.

To facilitate the debugging process, a simple debugging framework is presented. The framework takes advantage of the tree structure of the transformed program. This enables the developer to control and observe the execution of a program by directly manipulating the internal variable and parameter values.

The paper is structured as follows. Section 2 provides a background to Erlang and the structural testing problem. Section 3 introduces our program transformation. Section 4 shows how tests can be generated to cover the transformed program. Section 5 contains a case study with respect to a small telecommunications system, and section 6 contains conclusions and future work.

2 Background

This section provides a brief introduction to Erlang.

2.1 Erlang

Erlang is a concurrent functional language with specific support for the development of distributed,fault-tolerant systems with soft real-time requirements [1]. It was designed from the start to support a concurrency-oriented programming paradigm and large distributed implementations that this supports.

The Open Telecom Platform (OTP) is a set of Erlang libraries for building large fault-tolerant distributed applications. Extensive libraries for common network-applications and protocols are included, and there is a collection of source code and trace-analysis tools that provide a base for debugging and profiling tasks. It also provides a set of common and reusable *behaviours*, which encapsulate common behavioural patterns for processes, where the library module implements the generic behaviour, and the developer is left to add those aspects of behaviour that are specific to their system.

With the OTP, Erlang applications can be rapidly developed and deployed across a large variety of hardware platforms, and this has caused it to become increasingly popular, not only within large telecoms companies such as Ericsson, but also with a variety of SMEs in different areas such as Yahoo! Delicious, and the Facebook chat system.

However, verification and validation of Erlang systems is to-date a largely ad-hoc, manual process. Consequently there is an inherent danger that important functionality remains untested and undocumented. Thus along with its recent growth in popularity, there has been a concerted drive to develop more automated and systematic techniques.

2.2 Structural Testing of Erlang Implementations

In structural testing, the goal is to achieve some level of coverage of the source code. Branch coverage, where the aim is to cover every branch from every decision point in the source code, is one of the most popular measures. The challenge of generating a suitable test set is two-fold: (1) to identify a suitable set of test inputs that will reach every branch, and (2) to ensure that this set of tests is sufficiently small so that can be executed in a reasonable amount of time.

An Erlang program consists of a set of modules, each of which defines a number of functions. Functions that are accessible from other modules need to be explicitly declared by the *export* command. A function named *f_name* in the module *module* and with arity *N* is denoted as *module:f_name/N*. An Erlang function f_i is coded as a sequence of executional units $f_i = \langle cp_{i(1)}, ..., cp_{i(n)} \rangle$, each of which defines a set of statements. An example is demonstrated in Figure 1.

```
-module(client).                          :
-export([idle/2]).                        :
idle(AT,{MB,RS,CSs})→                     :
   P1 = gen_server:                       :
        call(hd(CSs),{request,AT,MB}),    :
   case P1 of                             :
     {error,invalid_mobile}→              :
        action:show(invalid_mobile),      :
        {next_state,idle,{MB,RS,CSs}};:
     {ok,connected,_CalledFS, RS}→        :
        action:show(mobile_connected),    :
        {next_state,connected,            :
             {MB,RS,CSs},?T};             :
     {error,already_connected}→
        action:show(already_connected),
        {next_state,connected,
                {MB,RS,CSs},?T};
     {error,busy}→
        action:show(sever_busy),
        idle(AT,{MB,RS,
           lists:append(tl(CSs),hd(CSs))});
     _Other→
        action:show(action_invalid),
        {next_state,idle,{MB,RS,CSs}}
end.
```

Fig. 1. An Erlang Example

From the structural-testing perspective, the aim is to identify a set of test-inputs that will collectively ensure that every branch in *client:idle* is executed at least once. Along with the input parameters, it is also necessary for the tester to control the values that are returned by *gen_server*. In this case, we would need to ensure that the variable *P1* is at some point assigned to one of the four predicate conditions (e.g. {*error,invalid_mobile*}), as well as an aribitrary value that corresponds to none of them.

The simplest strategy is to construct a separate test set for each branch. For each unit $cp_{i(k)}$ a test set is designed and applied to f_i. The I/O behavior observed from $\langle cp_{i(1)}, ..., cp_{i(k)} \rangle$ is used to evaluate $cp_{i(k)}$. When the test of $cp_{i(k)}$ is complete, the process continues to the next unit. The process continues until all units have been tested.

Such a test strategy could give rise to two disadvantages. First, this approach can be wasteful; testing $cp_{i(k)}$ requires the execution of all preceding units, $\{cp_{i(1)}, ..., cp_{i(k-1)}\}$, regardless of whether $cp_{i(l)}$, $l < l < k$, has already been tested. Secondly, identifying a suitable combination of inputs can be very challenging. The execution of $cp_{i(k)}$ is heavily dependent by the results of executing $\langle cp_{i(1)}, ..., cp_{i(k-1)} \rangle$ as the parameters for $cp_{i(k)}$ may be affected by previous results.

From this small example, it is possible to identify the input requirements by hand, simply from inspecting the predicate conditions. However, this approach becomes intractable when the software increases in scale and complexity. Given a conventional Erlang system with multiple modules, where certain branches can only be reached by complex combinations of conditions, an automated approach is required.

3 Transforming an Erlang Program into Binary Format

This paper proposes a program transformation that converts an Erlang function into a tree-structure. The transformed function retains the functional behavior, but makes the predicates that control its behavior explicit. This has two benefits that address the problem mentioned above. The tree-shaped structure makes it possible to identify an efficient test-set. Making the predicates explicit makes it possible to automatically identify the set of inputs that are required to reach every unit in a function. The rest of this section will introduce the program transformation.

3.1 Notations

To formalize the description, we introduce the following notations. Let $cp_{i(k)}|_{cond(k)}$ denote that the unit $cp_{i(k)} \in f_i$ is executed under the condition of $cond(k)$. If $cond(k) = true$, $cp_{i(k)}$ is activated for execution; otherwise a dummy operation[1] is performed. Let $cp_{i(l)} << cp_{i(m)}$ denote that $cp_{i(m)}$ is executed after $cp_{i(l)}$. The function f_i can be expressed as $f_i = cp_{i(1)}|_{cond(i(1))} << ... << cp_{i(n)}|_{cond(i(n))}$, $cp_{i(l)}|_{cond(i(l))} << cp_{i(m)}|_{false} << cp_{i(n)}|_{cond(i(n))} \equiv cp_{i(l)}|_{cond(i(l))} << cp_{i(n)}|_{cond(i(n))}$, $\forall(l < m < n)$. If $cp_{i(l)} \in f_i$ is an unconditional unit, $cond(i(l))$ is automatically set to $true$; otherwise, the value of $cond(i(l))$ is determined by the pattern evaluation. For example, the function *idle* shown in Figure 1 is expressed as:

[1] A dummy operation is a function that maps the inputs to the outputs without changing the values of the inputs.

$$
\begin{aligned}
&func = \\
&\quad P1 = gen\_server : call(hd(CSs), \{request, AT, MB\})|_{true} << \\
&\quad (action : show(invalid\_mobile)|_{true} << \\
&\quad \{next\_state, idle, \{MB, RS, CSs\}\})|_{cond(matches(P1,\{error,invalid\_mobile\}))} << \\
&\quad (action : show(mobile\_connected)|_{true} << \\
&\quad \{next\_state, connected, \{MB, RS, CSs\}, ?T\})|_{cond(not(matches(P1,} \\
&\qquad\qquad \{error,invalid\_mobile\})) \wedge matches(P1,\{ok,connected,\_CalledFS,RS\})) << \\
&\qquad << ... << \\
&\quad (action : show(action\_invalid)|_{true<<} \\
&\qquad \{next\_state, idle, \{MB, RS, CSs\}\})|_{cond(not(matches(P1,\{error,invalid\_mobile\}))} \\
&\qquad\qquad \wedge ... \wedge not(matches(P1,\{error,busy\})) \wedge matches(P1,\_Other))
\end{aligned}
$$

where the function *matches* evaluates whether *P1* matches *PV*, *PV*∈ {{*error, invalid_mobile*}, {*ok, connected, _CalledFS, RS*}, {*error, already_connected*}, {*error, busy*}, *_Other*}.

The above expressions show how each unit $cp_{i(k)} \in f_i$ can be represented by a generic pattern, $cp_{i(k)}|_{cond(i(k))}$. If $cond(i(k))$ is *true*, $cp_{i(k)}$ is executed; otherwise, $cp_{i(k+1)}|_{cond(i(k+1))}$ is evaluated.

3.2 Function Transformation

The transformation we introduce here decomposes a single large function f_i into a set of atomic functions $f'_{i(k)}$. Each function f_i is transformed into a set of ordered calls to atomic functions $f'_{i(1)} << ... << f'_{i(n)}$ where n is the number of executional units of f_i. The benefit of this is that the guards on the execution of these atomic functions are made explicit, and it becomes easier to control their execution, or even influence their execution for the purpose of debugging (this is demonstrated in the following sections).

The unit $cp_{i(k)} \in f_i$ is transformed into a called function defined as:

$f'_{i(k)}(true, Args) \rightarrow cp_{i(k)}$;

$f'_{i(k)}(false, Args) \rightarrow f'_{i(k+1)}(cond(i(k+1)), Args)$.

The function $f'_{i(k)}$ is guarded by a *switch* that can only be set to *true* or *false*. If *switch* is *true* ($cond(i(k)) = true$), the functional executions defined in $cp_{i(k)}$ are performed; otherwise, the function $f'_{i(k+1)}$ is invoked where $cp_{i(k+1)}|_{cond(i(k+1))}$ is evaluated.

When $f'_{i(k)}$ is invoked, if $f'_{i(k)}$ is derived from a unconditional unit $cp_{i(k)}$, *switch* is automatically set to *true*; otherwise, the pattern match evaluation function *patterns_match* is applied to decide the value of *switch*. Definition of the function *patterns_match* is discussed in the subsection 3.3. According to the pattern under evaluation, the function *patterns_match* returns *true* or *false*. If $cp_{ik} \in f_i$ defines a list of variables RV_k, when $cp_{im} \in f_i, m > k$, is transformed to $f'_{i(m)}$, RV_k is configured as an argument of $f'_{i(m)}$.

Thus, the defined rules transform the original function f_i into a set of calls to functions $\{f'_{i(1)}, ..., f'_{i(n)}\}$ where n is the number of units defined in f_i. Each

function $f^{'}_{i(k)}$ is coded by generic pattern with a *true* branch and a *false* branch being defined. The *true* branch performs the functional executions defined in $cp_{ik} \in f_i$, while, the *false* branch invokes the function $f^{'}_{i(k+1)}$ where $cond(i(k+1))$ is evaluated. Such a transformation is called Binary Transformation (BT). The *true* branch of a called function is called Functional Execution Branch (FEB) and the *false* branch is called Pattern Evaluation Branch (PEB). By applying the defined rules, the function *idle* shown in Figure 1 is transformed as shown in Figure 2.

```
idle(AT,{MB,RS,CSs})→
  idle_st_1(true,
        [AT,MB,RS,CSs,do_not_care]).

idle_st_1(true,[AT,MB,RS,CSs,_P1])→
  P1 = gen_server:call(hd(CSs),
                {request,AT,MB}),
  idle_case1(pattern_match([P1],[{error,
    invalid_mobile}]),[AT,MB,RS,CSs,P1]);
idle_st_1(false,[AT,MB,RS,CSs,P1])→
  idle_case1(pattern_match([P1],[{error,
    invalid_mobile}]),[AT,MB,RS,CSs,P1]).

idle_case1(true,[AT,MB,RS,CSs,P1])→
  action:show(invalid_mobile),
  {next_state,idle,{MB,RS,CSs}};
idle_case1(false,[AT,MB,RS,CSs,P1])→
  idle_case2(pattern_match([P1],
      [{ok,connected,do_not_care,sys_var}]),
            [AT,MB,RS,CSs,P1]).

idle_case2(true,[AT,MB,RS,CSs,P1])→
  action:show(mobile_connected),
    {next_state,connected,{MB,RS,CSs},?T};
idle_case2(false,[AT,MB,RS,CSs,P1])→
  idle_case3(pattern_match([P1],[{error,
    already_connected}]),
          [AT,MB,RS,CSs,P1]).

idle_case3(true,[AT,MB,RS,CSs,P1])→
  action:show(already_connected),
  {next_state,connected,{MB,RS,CSs},?T};
idle_case3(false,[AT,MB,RS,CSs,P1])→
  idle_case4(pattern_match([P1],
      [{error,busy}]),[AT,MB,RS,CSs,P1]).

idle_case4(true,[AT,MB,RS,CSs,P1])→
  action:show(sever_busy),
  {next_state,idle,{MB,RS,CSs}};
idle_case4(false,[AT,MB,RS,CSs,P1])→
  idle_case5(pattern_match([P1],
      [do_not_care]),[AT,MB,RS,CSs,P1]).

idle_case5(true,[AT,MB,RS,CSs,P1])→
  action:show(action_invalid),
  {next_state,idle,{MB,RS,CSs}}
```

Fig. 2. Transformed Erlang program.

The defined rules only transform the structure of P into $P^{'}$. The functionalities implemented in P are preserved in $P^{'}$. The functions in $P^{'}$ are of simpler structures. The predicates that guard the execution of those functions are explicitly defined. Compared to P, $P^{'}$ is well structured and easier to test.

Proposition 1 *Given an Erlang program P and its BT $P^{'}$, $P^{'}$ is functionally equivalent to P.*

Proof: Given an Erlang program P and its BT transformation $P^{'}$, it is sufficient to prove $P^{'}$ is functionally equivalent to P, if for $f_i \in P$, $i = 1, ..., m$, the following conditions hold: (1) each unit $cp_l \in f_i$ has a unique called function

$f'_{i(l)}$ in P'; (2) the functional behavior (input/output) of $f'_{i(l)}$ is identical to that of cp_l and (3) the called order of the FEB of f'_i is defined by the order of the executions of cp_i, namely, $(cp_{i(m)}|_{cond(i(m))=true} << cp_{i(n)}|_{cond(i(n))=true}) \rightarrow (f'_{i(m)}(true, Args) << f'_{i(m+1)}(false, Args) << ... << f'_{i(n-1)}(false, Args) << f'_{i(n)}(true, Args))$, $\forall(n > m)$.

The proof for the first condition is obvious as the transformation rules define that the relation between a unit and its called counterpart is a one-to-one mapping, that is, for each unit $cp_{i(l)} \in f_i$, there is only one called replacement $f'_{i(l)}$ in P'.

The transformation rules define that (1) the functional executions defined in $cp_{i(l)}$ are performed in the FEB of $f'_{i(l)}$; (2) if $cp_{i(g)} \in f_i$, $g < i$, returns a value to the variable $RV_{i(g)}$, $RV_{i(g)}$ is defined as an input of $f'_{i(l)}$, and (3) the name of $RV_{i(g)}$ is unique (module name plus function name plus variable name). These rules guarantee that the function $f'_{i(l)}$ and the unit $cp_{i(l)}$ receive the same input set and perform the same functional executions. Thus, $f'_{i(l)}$ and $cp_{i(l)}$ have the same functional behavior.

The third condition can be obtained by contradiction. Let $cp_{i(m)}|_{cond(i(m))=true} << cp_{i(n)}|_{cond(i(n))=true}$ be true. This implies that $cond(i(k)) = false$, $m < l < n$, as $cp_{i(m)}|_{cond(i(m))=true} << cp_{i(n)}|_{cond(i(n))=true} \equiv cp_{i(m)}|_{cond(i(m))=true} << cp_{i(l)}|_{cond(i(l))=false} << cp_{i(n)}|_{cond(i(n))=true}$, $\forall(m < k < n)$. Suppose in the transformation counterpart, there exists a called function $f'_{i(l)}$, $m < l < n$, such that $f'_{i(l)}(true, Args)$ is activated. According to the transformation rules, the PEB of the function $f'_{i(l-1)}$ must have invoked the function $f'_{i(l)(cond(i(l)),Args)}$ with $cond(i(l))$ being evaluated to *true*. This, however, contradicts to the fact that $cond(i(l))$ is *false*. Thus, $(cp_{i(m)}|_{cond(i(m))=true} << cp_{i(n)}|_{cond(i(n))=true}) \rightarrow (f'_{i(m)}(true, Args) << f'_{i(m+1)}(false, Args) << ... << f'_{i(n-1)}(false, Args) << f'_{i(n)}(true, Args))$, $\forall(n > m)$. □

3.3 Pattern Match Transformation

Erlang makes extensive use of pattern matching in its function definitions. This work transforms the pattern matching clauses into a binary format. To do so, the technique [4] proposed to eliminate overlapping between patterns are applied. Specifically, pattern match clauses in a function are replaced by a series of called functions, each of which is guarded by the *pattern_match* function.

A data type called the Structure Splitting Tree (SST) [4] is defined and applied for pattern evaluation, and its use guarantees the pattern match clauses being represented in binary formats.

3.4 Functional Binary Tree

After P is transformed into P', it is easily to see that all functions in P' are represented with a generic pattern as shown in Figure 3. The pattern is a binary

tree with one entry and two exits. The entry is identified with a function name *func_name* that takes a list of arguments *Args* as input. Both exits define a tuple {*Next_Func, Next_Switch, Next_Args*} where *Next_Func* states the function next to be called; *Next_Switch* defines the guard for *Next_Func*, and *Next_Args* consists of the arguments for *Next_Func*. *Switch* is applied to guard the selection

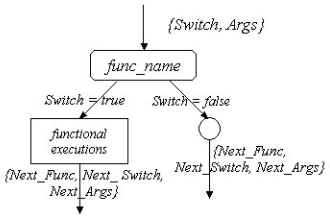

Fig. 3. Generic pattern of the called function.

of the *true* and the *false* branches. The *true* branch defines both the functional executions and the function to be called next, while the *false* branch only states the next called function. Such a binary tree is defined as a Functional Binary Tree (FBT).

If all called functions in $P^{'}$ are represented with their FBTs, a Complete Functional Binary Tree (CFBT) can be derived where the nodes that define no *Next_Func* constitute the terminals. For example, Figure 4 demonstrates the CFBT of the program shown in Figure 2.

4 Structural Testing from Transformation

This section discusses the test generation for structural coverage from the program transformation.

4.1 Test Generation

As discussed in the Section 3, a program P can be transformed into its BT $P^{'}$. Proposition 1 proves that $P^{'}$ is functionally equivalent to P. This suggests that, instead of testing P directly, one can test $P^{'}$ and use the test results to evaluate the correctness of P. Any errors detected in $P^{'}$ imply that the implementation of P is faulty. The structure of $P^{'}$ is a binary format and the predicates for controlling the execution of a node are explicitly specified. Deriving data to test an execution branch in $P^{'}$ should not be a difficult task. Compared to P, $P^{'}$ is easier to achieve structural coverage.

This section discusses the derivation of tests for $P^{'}$. As $P^{'}$ can be represented by a CFBT, it is easy to see that a test set whose elements traverse and test all nodes in the CFBT will test $P^{'}$, and the test achieves complete structural

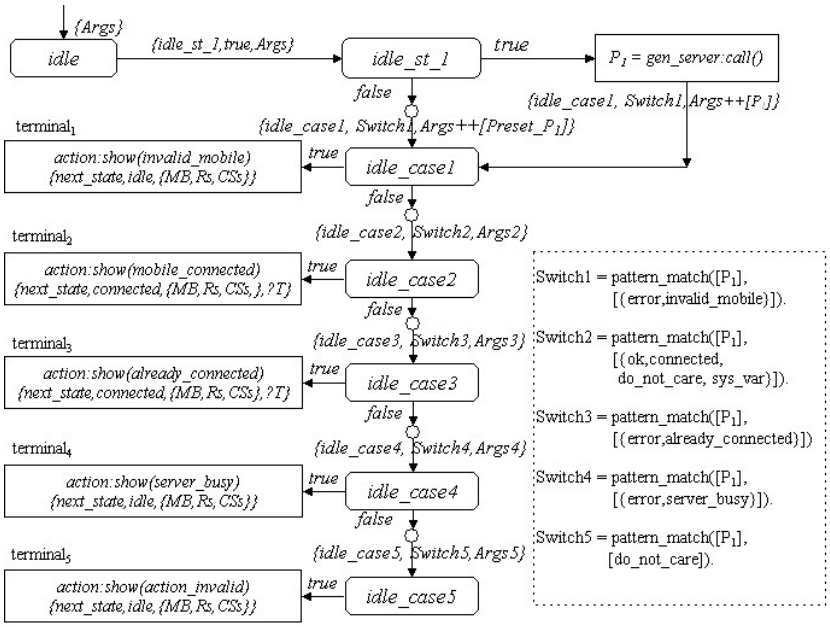

Fig. 4. Complete Functional Binary Tree derived from the program shown in Figure 2.

coverage. Thus, the generation of a test set for $P^{'}$ is achieved by the construction of data that traverse every node in the $CFBT_{P'}$ and test the corresponding function at least once. As a CFBT is a binary tree, standard techniques for the search of nodes in a tree such as a breadth-first search algorithm [3] or a depth-first search algorithm [3] can be applied.

As an example, a test for the CFBT shown in Figure 4 is defined by constructing a set of data TS whose elements traverse and test each node in the graph at least once. To do so, each terminal $terminal_i \in \{terminal_1, ..., terminal_5\}$ needs to be tested by $ts_i \subseteq TS$ at least once. Let ts_3 test $terminal_3$. The test ts_3 is constructed by examining the path between the root node (the entry of the function *idle*) and the terminal $terminal_3$. The path between $terminal_3$ and *idle* is $\langle terminal_3,\ idle\_case3(true),\ idle\_case2(false),\ idle\_case1(false),\ idle\_st\_1,\ idle\rangle$. The function $idle\_st\_1$ can pass P_1 to its successor. The value of P_1 can be either computed from the execution $gen\_server : call()$ or preset. The subsection 4.2 discusses the technique on deriving partial testing by presetting the inputs for the functions under evaluation.

Thus, in the test, P_1 should receive {*error, already_connected*} (either received from the gen_server:call() or preset). When the test is applied, the IUT must produce the message *already_connected*.

Proposition 2 *Given a program P and its binary transformed $P^{'}$ represented by the complete functional binary tree $CFBT_{P'}$, if a test set TS achieves complete structural coverage of $P^{'}$, the elements in TS must traverse all nodes in $CFBT_{P'}$ and test the corresponding functions at least once and vice versa.*

Proof: Let n be the number of functions in $P^{'}$ and N_i be the node that stands for the function $f_i^{'} \in P^{'}$ in the $CFBT_{P'}$. Suppose a test set TS has achieved complete structural coverage of $P^{'}$, there must exist a subset $ts_i \subseteq TS$ such that ts_i tests the function $f_i^{'} \in P^{'}$, which implies that the function $f_i^{'}$ is executed by ts_i at least once. This is equivalent to that ts_i traverses the node N_i in the $CFBT_{P'}$ at least once. Similarly, if a set of data ts_i traverses the node N_i in $CFBT_{P'}$, it should execute and test the function defined in N_i once. The union of all such sets of data constitutes a test set TS, $TS = ts_1 \cup ... \cup ts_n$, that achieves complete structural coverage for $P^{'}$. □

4.2 Improving Test Controllability and Observability

Once faults are detected, tests need to be constructed to isolate these faults. The process of identifying the locations of faults is called *debugging*. The debugging process defines a number of tests, each of which executes a set of designated functions whose outputs should exhibit the properties defined by the design. A test in the debugging process is called a *partial test*. By observing the results of all partial tests, the faults are expected to be isolated.

The effectiveness of the debugging process is primitively determined by two factors - the *test controllability* and the *test observability*. The *test controllability* determines whether it is always possible to derive a partial test, while the *test observability* determines whether it is always possible to observe the outputs when the test is executed.

This work shows that, by applying the program transformation, the test controllability and the observability are improved. As discussed in the section 3, in the transformation program $P^{'}$, all functions are represented by the BFTs. If $f_i^{'} \in P^{'}$ is further modified into the format as shown in Figure 5, the function under test is then assigned with two running modes, the *normal* mode and the *debugging* mode. In the debugging mode, a debugging framework is associated. The debugging framework is used to track the values of the variables under evaluation, or bypass the functional execution of $f_i^{'}$ by presetting its *switch* to *false*.

For example, in Figure 5, if $f_i^{'}$ is previously set to the debugging mode, the results of all internal computations RS_1, ..., RS_i will be posted to the debugging framework for comparison. Before $f_{i+1}^{'}$ is called, the values of the inputs *Switch*, Arg_1, ..., Arg_k will be sent to the debugging framework. The debugging framework returns a tuple {*NextMode*, *DSwitch*, [$DArg_1$, ..., $DArg_k$]} where $NextMode$ specifies the running mode of $f_{i+1}^{'}$; $DArg_1$,...,$DArg_k$] consists of the inputs for $f_{i+1}^{'}$ with $DArg_l$, $1 \leq l \leq k$, being preset; $DSwith$ saves the value of *switch*. If $DSwith$ is set to *false*, the evaluation of $f_{i+1}^{'}$ is bypassed and the testing moves on to $f_{i+2}^{'}$.

```
f'_i(Switch,[Arg_1,...,Arg_k],Mode)→
  RS_1 = functional_execution_1([Arg_1,...,Arg_k]),
       ...
  RS_n = functional_execution_n([Arg_1,...,Arg_k]),
  case Mode of
     normal→      nothing;
     debugging→   RS={func_i,[{'RS_1',RS_1},...,{'RS_n',RS_n}]},
                  debug_panel:setting(RS)
  end,
  Switch = pattern_match([Arg_1,...,Arg_k],[pattern defined for f'_{i+1}]),
  {NextMode,DSwitch,[DArg_1,DArg_k]} =
    debug_panel:setting(func_{i+1},[{'Switch',Switch},{'Arg_1',Arg_1},...,{'Arg_k',Arg_k}])
  f'_{i+1}(DSwitch,[DArg_1,...,DArg_k],NextMode).
```

Fig. 5. The generic debugging pattern of a function.

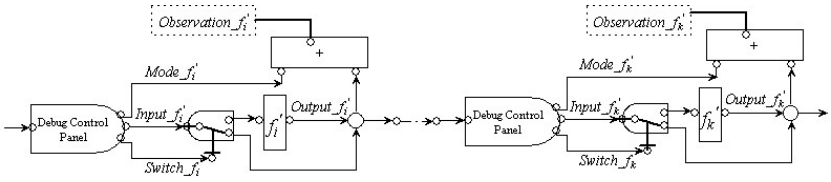

Fig. 6. Debugging framework.

Thus, the configuration of the debugging process can be modelled as shown in Figure 6. Each function $f_i^{'}$ either takes input values computed from $f_{i-1}^{'}$ or preset by the debugging framework. The functional executions of $f_i^{'}$ can be controlled by presetting the value for *switch*. If *switch* is preset to *false*, the functional execution of $f_i^{'}$ is bypassed; otherwise, $f_i^{'}$ will be functionally evaluated. When $f_i^{'}$ is running on the debugging mode, all computational results performed within the function can be externally observed.

By using the debugging framework to control the values of the inputs for a function under test, the test controllability is improved and, by opening the observation windows in the debugging framework, tracking the changes of the variables under evaluation is possible, which helps to improve the test observability.

5 A Case Study

A case study is used to evaluate the proposed model. The case study simulates a telecommunication system.

5.1 System Infrastructure

The telecoms uses a client-server structure, and comprises of a database server (DBS) that maintains all client's data and a number of functional servers to process clients' requests. An FS has a capacity that defines the maximum number of mobiles to be connected.

A client can communicate with any FSs and perform some functional operations such as *calling* and *top-up*. Each client has an account maintained in the DBS, and in order to make a phone call, a client needs to top-up enough money in its account. Before performing any functional operations, a client needs to connect to an FS. A client can only be connected to one FS, and if a client has connected to an FS and tries to connect to another FS, the request will be denied.

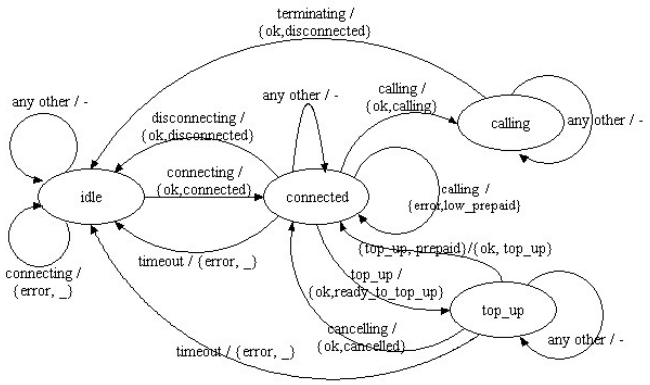

Fig. 7. Client behaviour modelled as an FSM.

The behavior of a client (mobile) is modelled as a finite state machine (FSM), and the initial design is shown in Figure 7. There are four states: *idle*, *connected*, *calling* and *top_up*, where initially, the system is set to the *idle* state.

The FSM defines the behavior of a number of operations: *connecting*, *disconnecting*, *calling*, *terminating*, *top_up* and *cancelling*. Before performing any operations, a client FSM needs to connect to an FS through sending the *connecting* request.

A client FSM has a timing restriction applicable when in states *connected* or *top_up*. Specifically, when the FSM is directed to the state *connected* or *top_up*, a timer will be instantiated which enables the timing process. If, within the predefined time period, no action is performed by the client, a *timeout* event will be generated and sent to the FS. By receiving *timeout* event, the FS cuts off the connection and releases the resource from its user list. The FSM is then reset to the state *idle*.

5.2 Erlang Implementation

Erlang is used to implement the telecoms system, making use of the OTP design patterns as is common practice. The FS is implemented using the Erlang/OTP *gen_server* module. A generic server is implemented by providing a *callback module* where (*callback*) functions are defined specifying the concrete actions of the server such as server state handling and response to messages.

The client behavior is realized using the OTP *gen_fsm* module. In accordance with the design, four state functions are defined: *idle*, *connected*, *calling* and *top_up*. The state function *idle* shown in Figure 1 initiates a *connecting* request to an FS.

The state function *connected* evaluates the requests for the consequent actions. For example, if *calling* is requested, the function will call the FS to check if the client has saved enough money to make a call. The reply {*ok,calling*} enables the client's calling request, which moves the FSM to the state *calling*.

```
connected(timeout,{M,SVR,SVRList})→
  gen_server:call(SVR,{request,timeout,M}),
  display(M,timeout),
  {next_state,idle,{M,nil,SVRList}};
connected([Act,_SVR],{M,SVR,SVRList})→
  case Act==terminating of
    true →
      display(action,invalid),
      {next_state,connected,
            {M,SVR,SVRList},20000};
    false →
      F=gen_server:call(SVR,{request,Act,M}):
      case F of
        {ok,disconnected}→
          display(disconnected),
          {next_state,idle,
                {M,SVR,SVRList}};
        {ok,calling}→
          display(client,calling),
        {next_state,calling,
          {M,SVR,SVRList}};
        {error,low_prepaid}→
          display(low_prepaid),
          {next_state,connected,
              {M,SVR,SVRList},20000};
        {ok,ready_to_top_up}→
          display(ready_to_top_up),
          {next_state,top_up,
              {M,SVR,SVRList},20000};
        _Other →
          display(action,invalid),
          {next_state,connected,
              {M,SVR,SVRList},20000}
      end
end.
```

The state function *calling* enables the calling process and when the FSM is in the state *calling*, only the *terminating* action can stop the process. This prevents the calling from being disrupted by any unintended actions.

```
calling([Act,_SVR],{M,SVR,SVRList})→
  case Act of
    terminating →
      gen_server:call(SVR,{request,Act,M}),
      display(call,terminating),
      {next_state,idle,
                {M,nil,SVRList}};
    false →
      display(server,invalid),
      {next_state,calling,
            {M,SVR,SVRList}}
end.
```

When being in the state *connected*, the client can ask to top up its account by sending the *top_up* request to the FS. If {*ok,ready_to_top_up*} is replied, the top up process is enabled, and the FSM moves to the state *top_up*. An action will trigger the state function *top_up* to either start the transaction by {*top_up, Prepaid*} operation

(*Prepaid* is the amount of money the client is about to transfer), or cancel the process by sending the *cancelling* request.

```
top_up(timeout,{MB,RS,CSs})→                :        {MB,RS,CSs},20000};
  gen_server:call(RS,{request,timeout,MB}),: {ok,cancelled} →
  action:show(timeout),                     :     action:show(top_up_cancelled),
  {next_state,idle,{MB,nil,CSs}};           :     {next_state,connected,
top_up(AT,{MB,RS,CSs})→                     :           {MB,RS,CSs},20000};
  P3=gen_server:call(RS,{request,AT,MB}) :  _Other →
  case P3 of                                :     action:show(action_invalid),
    {ok,top_up} →                           :     {next_state,top_up,
      action:show(top_up_completes),        :         {MB,RS,CSs},20000}
      {next_state,connected,                : end.
```

When the FSM moves to the state *connected* and *top_up*, a timer is initiated. The timer is set to 20,000ms. If within the time period, no action is performed, a *timeout* event will be generated and sent to the FS. The FSM is reset to the state *idle*. A function *command* is defined to simulate the receiving of external actions. It calls *gen_server:send_event* to triggers the state functions.

5.3 Test Design

The implementation is tested by applying the proposed testing scheme. The programs of the FS and the client are transformed into the BT formats. The corresponding CF-BTs are constructed for test generation. The BT programs are further modified into the debugging mode as discussed in the subsection 4.2. For example, the function *idle_st_1* shown in Figure 2 is modified to the debugging mode:

```
idle_st_1(true,[AT,MB,RS,CSs,_P1],Mode)→
  P1 = gen_server:call(hd(CSs),{request,AT,MB}),
  case Mode of
     debugging→ debug_panel:posting({{idle_st_1,true},[{'P1',P1}]});
     running→ nothing
  end,
  {NextMode,DSwitch,DAT,DMB,DRS,DCSs,DP1} =
       debug_panel:setting({idle_case1, [{'Mode',debugging},
            {'Switch',pattern_match([P1],[{error,invalid_mobile}])},
            {'AT',AT},{'MB',MB},{'RS',RS},{'CSs',CSs},{'P1',P1}]}),
  idle_case1(DSwitch,[DAT,DMB,DRS,DCSs,DP1],NextMode);
idle_st_1(false,[AT,MB,RS,CSs,_P1],Mode)→
  {NextMode,DSwitch,DAT,DMB,DRS,DCSs,DP1} =
       debug_panel:setting({idle_case1, [{'Mode',debugging},
            {'Switch',pattern_match([P1],[{error,invalid_mobile}])},
            {'AT',AT},{'MB',MB},{'RS',RS},{'CSs',CSs},{'P1',P1}]}),
  idle_case1(DSwitch,[DAT,DMB,DRS,DCSs,DP1],NextMode).
```

To derive a partial test to check a particular system property, one needs to identify the set of designated functions from the CFBT and preset the inputs for each function by using the debugging framework. For example, to test "*A client is the state idle and sends the connecting request to the FS svr_1. When {ok,connected,_CalledFS,svr_1} is*

replied, the connection is set up". The set of functions is identified from the CFBT (Figure 4) as [*idle_st_1, idle_case1, idle_case2*]. The settings for the the corresponding functions are: [{*idle_st_1*, [ ]}, {*idle_case1*, [{*'Switch',false*}, {*idle_case2*, [{*'P_1', {ok, connected, _CalledFS, svr_1*}}]}]}]. The debugging framework presets P_1 to {*ok, connected, _CalledFS, svr_1*} and bypasses the functional executions of *idle_case1*. For each function, the default running mode is *debugging*. The function *idle_st_1* will post the result of P_1 to the debugging framework for comparison before the executions are completed.

After the inputs being applied, the program should produce the message *mobile_connected*; otherwise, the implementation on such a property is faulty.

6 Conclusions and Future Work

Erlang is becoming an increasingly popular language, because it provides a sophisticated platform for the rapid development of concurrent and distributed applications. These often play a business-critical role, which makes testing a crucial component of the Erlang development process. So far, the majority of testing techniques and tools that have been developed for Erlang have focussed on functional testing. Tools such as QuickCheck [2] have become popular because they can rapidely test the system during development.

So far there has been no established technique for the structural testing of Erlang programs. Achieving targets such as branch-coverage has always depended on the ability of a developer to manually identify the necessary sets of test inputs. This paper has presented a technique to automate this process by applying a testability transformation [6].

The basic technique transforms an Erlang IUT into a functionally equivalent counterpart where each atomic function is represented by a binary format called the Functional Binary Tree. These functions are then aggregated into a complete tree (the Complete Functional Binary Tree). An important attribute of the tree is that every predicate that is required to reach any part of the tree is made explicit. The set of tests required to exercise the entire program simply correspond to those tests that are required to explore the entire tree.

A debugging framework is presented to facilitate the observation and manipulation of internal program variables as it is executed. As a function is executed, it provides the ability to track variable values. The execution of specific functions can be bypassed, and the input parameters to functions can be altered via the framework. This is particularly valuable for homing in on faulty areas in the source code.

A small telecoms case study has been presented to illustrate and evaluate the proposed model. By applying the proposed testing scheme, the components in the implementation were transformed into binary formats and then modified into the debugging mode. The CFBT was then constructed to derive the tests.

There still remains much to be done. Future work will apply the techniques here to larger industrial examples. When deriving a partial test to check a particular system property, the proposed model manually checks the CFBT to identify the set of functions. This on occasion can be very time consuming, and it is of interests if one can express the system properties with a formal language such as temporal logic [8], and use the formal expression as a guide to automatically identify the set of functions. This, however, remains a topic for the future work.

Acknowledgements

This work was funded by the FP7 project *ProTest*, number 215868: www.protest-project.eu. We're grateful to its academic and industrial members for input to this work and suggestions to improve the process.

References

1. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.
2. T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing Telecoms Software with Quviq Quickcheck. In Marc Feeley and Philip W. Trinder, editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang (Erlang'06)*, pages 02–10. ACM Press, 2006.
3. J. Bang-Jensen and G. Gutin. *Digraphs: Theory Algorithms and Applications*. Springer-Verlag, London, 2001.
4. Q. Guo and J. Derrick. Eliminating overlapping of pattern matching when verifying Erlang programs in μCRL. In *12th International Erlang User Conference (EUC'06), Stockholm, Sweden*, 2006.
5. Q. Guo, J. Derrick, and C. Hoch. Verifying Erlang Telecommunication Systems with the Process Algebra μCRL. In K. Suzuki, T. Higashino, K. Yasumoto, and K. El-Fakih, editors, *the 28th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems (FORTE'08)*, volume 5048 of *LNCS*, pages 201–217. Springer-Verlag, June 2008.
6. M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, and A. Baresel. Testability Transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.
7. J. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7), 1976.
8. F. Kröger. *Temporal logic of programs*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.
9. P. McMinn. Search-based software test data generation: a survey. *Softw. Test, Verif. Reliab*, 14(2):105–156, 2004.
10. J-M. Mottu, B. Baudry, and Y. Traon. Model Transformation Testing: Oracle Issue. In L. Frantzen, M. Merayo, and M. Múñez, editors, *28th International Conference on Software Testing, Verificaiton and Validation Workshop (ICSTW'08)*, pages 105–112, Washington, DC, USA, 2008. IEEE Computer Society.
11. L. Naslavsky, H. Ziv, and D. J. Richardson. Using Model Transformation to Support Model-based Test Coverage Measurement. In *AST '08: Proceedings of the 3rd international workshop on Automation of software test*, pages 1–6, New York, NY, USA, 2008. ACM.
12. C-A Sun. A Transformation-Based Approach to Generating Scenario-Oriented Test Cases from UML Activity Diagrams for Concurrent Applications. In *32nd Annual IEEE International Computer Software and Applications Conference*, pages 160–167, Los Alamitos, CA, USA, 2008. IEEE Computer Society.
13. US Department of Transportation, Federal Aviation Administration. *Software Approval Guidelines*, 2003. 8110.49.
14. J. Voas and K. Miller. Software Testability: The New Verification. *IEEE Software*, 12(3):17–28, 1995.
15. H. Zhu, P.A.V. Hall, and J.H.R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.