Iterative Refinement of Reverse-Engineered Models by Model-Based Testing

Neil Walkinshaw, John Derrick, and Qiang Guo

Department of Computer Science, The University of Sheffield, Sheffield, UK

Abstract. This paper presents an iterative technique to accurately reverse engineer models of the behaviour of software systems. A key novelty of the approach is the fact that it uses model-based testing to refine the hypothesised model. The process can in principle be entirely automated, and only requires a very small amount of manually generated information to begin with. We have implemented the technique for use in the development of Erlang systems and describe both the methodology as well as our implementation.

Keywords: Reverse engineering; model-based testing; Erlang.

1 Introduction

Several important software verification and validation techniques rely on the availability of models that describe the software behaviour, models which should be accurate, and capture every relevant requirement of the system. In practice this requirement is unrealistic. The manual process of developing a specification can be error-prone and expensive. Moreover, software is commonly developed under restrictive time constraints; developers tend to concentrate on developing the implementation and do not have time to generate and maintain accurate specifications in tandem. The notion of a fully accurate model being maintained (or even existing to begin with) is unfortunately at present largely a myth.

An alternative approach is to *generate* models, or specifications, of a system from the implementation itself, and this paper is concerned with the challenge of reverse-engineering *state machine models* from an implementation. Ideally, such a reverse-engineering technique can be run at any point during the development of an implementation to provide a snapshot of system behaviour. The use of reverse-engineered models is that they can be inspected by developers to give an understanding of how the system behaves in practice. If they are correct (with respect to the developer's requirements), they can be used for core software maintenance tasks such as documentation and regression-testing. If, upon inspection, they are found to be out of step with system requirements, they can be used to identify which parts of the system are faulty.

In reality, reverse-engineering techniques tend to be less straightforward. They tend to require an extensive sample of program execution traces, which can be difficult to identify and collect. Consequently, the models can be inaccurate, and are therefore less useful to the developer.

In this paper we introduce an iterative reverse-engineering method that will ensure an adequate sample of execution traces by incorporating a model-based testing technique. Unlike most other existing techniques, no initial traces or models are required at all. Instead, a model-based testing framework is used to iteratively populate the set of traces that are used to infer the model, the inference being performed with our StateChum model inference tool [1]. One nice aspect of the approach is that the inference technique uses heuristics that have been shown to be effective for sparse samples of traces, meaning that it does not rely on systematic, expensive testing techniques. In this paper we demonstrate its effectiveness with respect to the QuickCheck testing framework and the Erlang programming language.

Implementation of the technique for use in Erlang development. This work has been carried out in the context of the EU ProTest project[1], which aims to improve the model-based testing of concurrent and distributed telecoms systems.

Erlang [2] 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. Consequently, a lot of the development effort involved in implementing an Erlang system is concerned with how these processes interact with each other and their environment. The protocol an Erlang process follows when communicating with other processes or responds to internal events is often implemented in terms of finite state machines, which is why they play a particularly important role and our technique is particularly appropriate.

The rapid development that is facilitated by Erlang means that formal specifications are, however, often neglected. It is often perceived to be more expedient to verify and document the system on an ad-hoc basis, and it is unrealistic to expect a developer in a commercial environment to provide an accurate and complete formal specification that can be used for more rigorous verification techniques. Producing an accurate specification that captures all of the necessary functionality can be a challenging and time-consuming task, particularly for complex systems. Furthermore, as the requirements change and the system is modified, keeping complex specifications up to date can be overwhelming, even with the best of intentions.

It is this problem that the technique presented in this paper aims to solve. The technique we develop iteratively reverse-engineers a state-machine from the implementation by using program tests, with only a small amount of manual input required. The result is a model that closely conforms to the actual system behaviour. The intention is that this final model can be validated and, if necessary, refined by the developer, and then used as a reference model for

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

subsequent program development tasks such as regression testing, and as a basis for communication amongst developers.

The paper is structured as follows. Section 2 provides some background on Erlang and QuickCheck, and the challenge of reverse-engineering finite state machines from software implementations. Section 3 introduces our iterative process that adopts model-based testing techniques to drive the inference and provides details of our implementation. The process is illustrated with a case study in Section 4, and Section 5 concludes.

2 Background

2.1 Erlang and QuickCheck

Erlang is a concurrent functional language with specific support for the development of distributed, fault-tolerant systems with soft real-time requirements [2]. It was designed from the start to support a concurrency-oriented programming paradigm and large distributed implementations that this supports. It was developed initially by Ericsson as a platform for rapid development of network-applications, but its applications have now expanded to include computer telephony, banking, TCP/IP programming (HTTP, SSL, Email, Instant messaging, etc) and 3D-modelling. 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.

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.

QuickCheck. One of these techniques is QuickCheck [3], an automated model-based testing tool for Erlang. It has become one of the standard testing tools used by Erlang developers. The 'model' is conventionally provided by the developer, as a set of simple properties that must hold for the program to behave correctly, and these are expressed as temporal logic properties in Erlang itself. For example, the following property would check that the reverse function for lists behaves as expected:

```
prop\_reverse() ->
      ?FORALL(Xs,list(int()),
      lists:reverse(lists:reverse(Xs)) == Xs).
```

Given such a property, QuickCheck uses random data generators to produce inputs that will exercise the system, with the aim of producing counter-examples. Once a counter-example is found, QuickCheck attempts to use successive tests to home in on the precise reason for the failure, with the aim of producing the smallest possible counter-example.

QuickCheck has recently been extended to so that one can test an implementation against a model given as a finite state machine (rather than just a predicate). The use of a finite state machine allows one to specify the permitted sequences of program functions, along with their effect on the data-state of the system. As well as selecting random data-inputs for the functions, QuickCheck also selects random paths through the state machine, with the aim of verifying the existence of state transitions. The key fact for our reverse-engineering technique is that *for a given state-machine model, QuickCheck can produce the requisite sequences of inputs (with the necessary data parameters) to automatically test any path in the model against the actual software system.*

2.2 Reverse-Engineering State Machines

Reverse-engineering techniques aim to address this problem. Broadly speaking, these approaches can be separated into two categories: Those based on source-code analysis (c.f. [4]), and those based on analysis of execution traces. Here we focus on the latter (dynamic) approaches. They are based on the analysis of program traces [5,6] which are sequences of events (e.g. function calls, message-passing events etc.), that may optionally be annotated with variable values. The traces can be recorded by instrumenting the source code, or by using one of the trace tools, e.g. for Erlang those that are included in the OTP framework. Traces that lead to a program failure (i.e. an exception) are annotated as such, so that the last recorded trace event corresponds to the point of failure.

From a given set of traces, the challenge for reverse-engineering techniques is to produce a candidate state machine that conforms to the provided set of traces. This is akin to the challenge of inferring a regular grammar, which is conventionally represented as a state machine from a given set of strings (a problem originally posed in 1967 [7]). In fact, most reverse-engineering techniques are inspired by techniques that were initially devised as grammar-inference techniques [8,1,6].

It is unrealistic to expect an inference technique to be able to infer a machine that exactly represents the underlying software system from any arbitrary set of traces. An inference technique will only produce an accurate result if the provided set of traces is *characteristic* of the behaviour of the underlying software system [8,6]. In terms of state machines, this must include enough information about what the program can and cannot do to enable the inference technique to identify every state transition, and to distinguish between every pair of non-equivalent states. Thus the key challenges lie in (a) *identifying* the relevant subset of executions and (b) *collecting* them - a potentially expensive and time-consuming process.

Most reverse-engineering inference techniques are *passive* [9,10], in that they presume that the necessary traces have already been identified and collected prior to inferring the candidate model. However, given that the initial set of traces is unlikely to contain all of the necessary information, the resulting model is often only poor approximation of the real implementation.

In an effort to address this, a number of *active* techniques have been developed. Active techniques are augmented with the ability to pose questions about the target model, to help the developer to identify the set of required traces. Such techniques come in two flavours: those based on Angluin's L^* algorithm [11], and those based on state-merging techniques [8]. Both techniques are iterative; they construct a hypothesis model, and use it as a basis for posing questions to some oracle. The essential difference between the two techniques comes down to expense. Techniques based on Angluin's algorithm rely on asking a large number of questions in order to produce an accurate model - and such an approach is infeasible in the setting our work is placed. Thus here we use state-merging techniques which place a greater emphasis on heuristics. These are less demanding in terms of the number of questions asked, but have nonetheless been shown to produce models that are reasonably accurate [8,6].

In our previous work we have applied active state-merging to the challenge of reverse-engineering [1,12], however, this has relied on a substantial amount of human intervention, where each query posed by the technique either had to be answered directly by the human or had to be executed manually. Expecting a human to be able to directly answer every query is unrealistic; the amount of knowledge required would undermine the whole purpose of reverse engineering the model in the first place. Expecting a human to manually execute each query is a tedious and time-consuming process, requiring the generation of suitable data parameters for each execution as well.

Here we describe an extension of this approach, that leverages the strengths of model-based testing techniques with the powerful heuristic inference abilities of state-merging techniques. The resulting process removes the human bottle-neck. Instead of being driven by a built-in question-generator, which can be very expensive, it will be up to the model-based tester to select the tests and to execute them. This means that the developer can choose the testing technique, and determine the expense (and resulting accuracy) of the technique. In our implementation of the technique we use the QuickCheck framework, but this can be substituted according to circumstance.

3 Iteratively Testing Reverse-Engineered Models

This paper introduces a technique to remove the human bottle-neck that arises with conventional dynamic model inference techniques. Instead of requiring a human to identify relevant program executions and collect the ensuing traces, model-based testing techniques are used to automate the process. The amount of a-priori knowledge of the program under analysis is minimal, although there are a number of optional mechanisms that can be used to add this information if it is available.

To explain the technique we use a simple example of a text editor, the LTS for which is shown in Figure 1(a), where as usual we take an LTS is a quadruple $A = (Q, \Sigma, \delta, q_0)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is a partial function and $q_0 \in Q$.

Our technique builds up a sequence of candidate models, each one being a bit more accurate than the last, these models can be viewed as partial LTS's, defined as follows [13], which allow us to distinguish between model transitions that are known to be invalid, and transitions that are simply not known to exist at all.

Definition 1 (Partial LTS (PLTS)). *A PLTS is a tuple* $A = (Q, \Sigma, \delta, q_0, \Psi)$*. This is defined as a LTS, but it is assumed to be only partial. To make the explicit distinction between unknown and invalid behaviour,* Ψ *makes the set of invalid labels from a given state explicit –* $\Psi \subseteq Q \times \Sigma$ *where* $(q, \sigma) \in \Psi$ *implies that* $\delta(q, \sigma) \notin Q$*.*

To define the language of a PLTS, we draw on the inductive definition for an extended transition function $\hat{\delta}$ used by Hopcroft *et al.* [14] to define two notions of language: prescribed and proscribed which are used below.

Definition 2 (Prescribed and Proscribed Languages of a PLTS). *For a state* p *and a string* w*, the extended transition function* $\hat{\delta}$ *returns the state* p *that is reached when starting in state* p *and processing sequence* w*. For the base case* $\hat{\delta}(q, \epsilon) = q$*. For the inductive case, let* w *be of the form* xa*, where* a *is the last element, and* x *is the prefix. Then* $\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a)$*.*

Given the extended transition function, the prescribed *language of a PLTS A can be defined as follows:* $PreL(A) = \{w | \hat{\delta}(q_0, w) \in Q\}$*.*

The proscribed *language of a PLTS can be defined as:*
$ProL(A)\{xa | (\hat{\delta}(q_0, x), a) \in \psi\}$*. By construction* $PreL(A) \cap ProL(A) = \emptyset$*.*

3.1 The Basic Process

The basic technique is straightforward. A human user provides the program of interest, along with a small initial set of traces, these are required to identify the set of functions of the program that are of interest (i.e. the alphabet of the target machine). From this an initial hypothesis model is constructed - a single state, with transitions for each element of the alphabet that loop back to that state. This is provided as input to a state-machine testing framework, which generates tests from the model. These tests are executed in the program, and a tracing mechanism is used to record the executions. As soon as a test is found that contradicts the expected behaviour as described by the model, the process is restarted, but this time the model is inferred from the test traces. This process iterates until no further discrepancies can be found by testing.

The basic process is captured by algorithm 1. It takes as input the program under analysis *Prog*, along with a valid trace (or several if necessary) that contains every element in the alphabet of the target machine. It uses four external functions: $inferPLTS$, *sub*, *generateTests* and *runTest*. $inferPLTS$ will be described in more detail below. *sub* simply returns a substring of a string *String* up to some index *i*. *generateTests* and *runTest* represent the functionality of the model-based testing framework. *generateTests* is responsible for generating tests from a PLTS, which may be achieved by a number of standard state machine testing algorithms. *runTest* executes a test *test* on a program *Prog*, and returns a zero if the test passes, or a number pointing to the index of *test* where the failure happened.

```
   **Input**: $Prog$, $Pos$
   **Data**: $Neg, test, fail, failedPLTS$
   **Uses**: $inferPLTS(T^+, T^-), generateTests(PLTS), runTest(t, Prog), sub(String, i)$
   **Result**: $PLTS$
1  $Neg \leftarrow \emptyset$;
2  $PLTS \leftarrow inferPLTS(Pos, Neg)$;
3  while $test \leftarrow generateTests(PLTS)$ do
4      $fail \leftarrow runTest(test, Prog)$;
5      if $fail = 0$ then
6          $Pos \leftarrow Pos \cup \{test\}$;
7          if $test \in ProL(PLTS)$ then
8              $PLTS \leftarrow inferPLTS(Pos, Neg)$;
9
10     else
11         $failed \leftarrow sub(test, fail)$;
12         $Neg \leftarrow Neg \cup \{failed\}$;
13         if $failed \in PreL(PLTS)$ then
14             $PLTS \leftarrow inferPLTS(Pos, Neg)$;
15
16     end
17 end
18 return $PLTS$
```

Algorithm 1. Basic iterative algorithm

An initial PLTS is generated by calling the $inferPLTS$ function with $Neg = \emptyset$ and Pos to contain one possible initial trace: the only requirement for the initial trace is that it contains every function in the alphabet of the target machine at least once. For our editor example, the initial sequence could simply be $< load, edit, save, close, exit >$, but any sequence in Σ^* is valid. $inferPLTS$ returns the most general model possible and in this initial iteration will always consist of a single state, with one looping transition that is labelled by the transitions from the trace in Pos. Formally, the resulting PLTS is defined as $A = (Q, \Sigma, \Delta, q_0, \Psi)$ where: $Q = \{q_0\}$, $\forall\sigma \in \Sigma$, $\delta(q_0, \sigma) \rightarrow q_0$ and $\Psi = \emptyset$. The purpose of the ensuing process is to refine this model - to ensure that the behaviour represented by the final PLTS accurately reflects that of the actual implementation. In our example, this initial model is shown in Figure 1(a).

Thus the algorithm iterates. To illustrate this suppose that we have chosen to use the QuickCheck testing framework (i.e. this will provide the functionality of the external functions $generateTests(PLTS)$ and $runTest(test, Prog)$). Being a model-based testing framework, QuickCheck needs a model to generate tests from. For this we use our initial model, the one in Figure 1 (b).

QuickCheck chooses a random test - and may choose to try to execute the sequence $< load, close, close, edit >$ (line 3). This fails, so the variable $fail$, which stores the point of failure in the $test$ is set to the index returned by $runTest$, which is 3. The failing sub-sequence $failed$ is identified by taking the first three elements of the test: $< load, close, close >$ (line 11). This is added to the set of impossible sequences Neg (line 12). Because the sequence $failed$ *is* possible in the current candidate model (belongs to the *prescribed* language), a discrepancy has been identified (line 13). Consequently a new model is inferred, taking the updated set Neg into account, which results in the model shown in Figure 1(c).

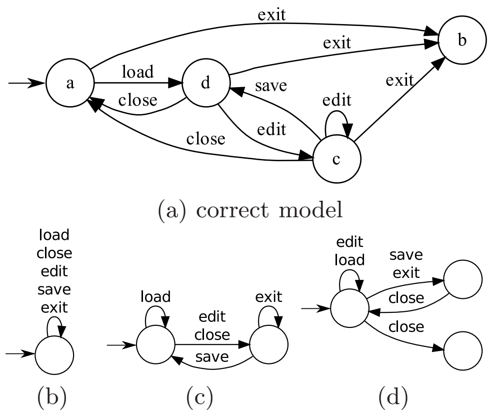


Fig. 1. Inference iterations on editor example

In the next iteration, the updated model is used as a basis for $generateTests$. Suppose that this time it returns the test $< load, edit, exit >$, which when executed does not fail. $fail$ is thus set to 0 (line 4), and the test is added to the set of valid traces Pos (line 6). Since the test has passed, and this is prescribed by the model, there is no disagreement between the test outcome and the model, so another test can be executed. It is important to note that QuickCheck, our tester of choice, will never generate tests that *should* fail according to the provided model, so in our case the branch in line 8 will not occur. Nevertheless, more systematic testing techniques such as the W-Method [15] do attempt tests that should fail; in this case, if a test does not fail when it should according to the current model, a new model has to be generated.

3.2 Model Inference

The inference process, which is called by the $inferPLTS$ function in the algorithm, is based upon the EDSM / blue-fringe state-merging method [16,8,1]. A brief illustration will be provided with respect to the editor example. As described above, the algorithm gradually gathers a set of traces that are either valid, or invalid. The purpose of $inferPLTS$ is to infer a state machine from these, that is a suitable generalisation - i.e. will correctly classify previously unseen traces as either valid or invalid.

To do this, the two sets of traces Pos and Neg are aggregated into a single tree - referred to as an *augmented prefix tree acceptor (APTA).* This tree represents the most specific and precise machine possible, that exactly corresponds to the provided sets of traces. For example, suppose the model-based tester has selected the test $< load, edit, edit, save, load >$ for the next iteration from the model in Figure 1 (c). This test fails, because only one file can be opened at a time. $InferPTA$ is called to build a new model, incorporating this test. Figure 2 (a) shows the corresponding APTA (valid traces are listed under S^+, and invalid

Initial traces
S^+:
$< load, edit, save, close, exit >$
$< load, edit, exit >$
S^-:
$< load, close, close >$
$< load, edit, edit, save, load >$

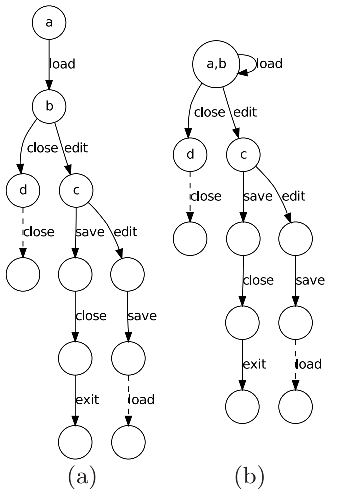


Fig. 2. Augmented Prefix Tree Acceptor and illustration of merging

traces are listed under S^-). Dashed lines in this tree represent paths in the tree that are invalid.

The goal of the inference is to identify states in this tree that are actually equivalent, and to merge them. In doing so, this will collapse the machine down to a minimal machine that is a generalisation of the set of traces. The merging process is iterative - lots of subsequent merges are required to reach the final machine. At each iteration, a set of state-pairs is selected using the Blue-Fringe algorithm [16], a colour-based breadth-first traversal algorithm (a description of this is beyond the scope of this paper). Each candidate pair is assigned a score, which indicates the likelihood that the states are equivalent. The score is computed by comparing the extent to which the suffixes of each state overlap with each other[2]. Any pairs with non-negative scores can potentially be merged. A pair of states is incompatible if a sequence is possible from one state, but impossible from the other - this leads to a score of -1. Once the scores have been computed, the pair with the highest score is merged, and the entire process starts afresh, until no further pairs can be merged.

To illustrate the scoring process, we refer back to the example prefix-tree in Figure 2 (a). Initially, the Blue-Fringe algorithm suggests only one pair of states (a,b). They have a score of zero, so they can be merged. The result is shown in Figure 2 (b). In the next iteration, we are given the option of selecting to merge either pairs ((ab),c) or ((ab),d). This is where the scoring comes into play - we want to select the pair that are most likely to be equivalent. In this case it is straightforward because the score for ((ab),d) is -1; a close is possible from state ab, but not from state d, ruling out that merge. The score for ((ab),c) is 2; both

[2] The Blue-Fringe algorithm ensures that the suffixes of one state are guaranteed to be a tree without loops, which facilitates this score computation.

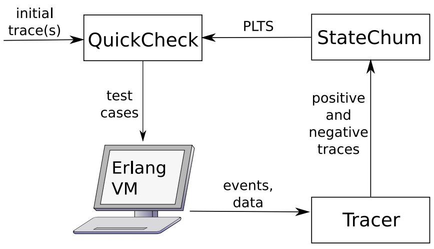


Fig. 3. Schematic overview of implementation

states share the suffix $< edit, save >$, so this is chosen to be merged. This is how the merging process continues until no more merges can be carried out. The resulting machine is shown in Figure 1(d).

3.3 Implementation of the Technique

We have implemented the technique for use on programs written in Erlang. However, the approach is essentially a black-box one, and is not tied to a specific language or paradigm. A schematic overview of the key components is given in Figure 3, and the tools that are used for the inference, testing and tracing are briefly described below.

Model Inference: StateChum[3] is an open-source model inference framework that has been developed by the authors [1]. It implements a state-merging approach as described in the previous section. The tracing mechanism (described below) has been augmented with a small script that translates traces into suitable input files.

Tracing: Erlang has a wide array of tracing tools, many of which are included in the standard Erlang OTP libraries. The traces used in this work are however laid out in a particular format, to facilitate the application of other trace-analysis tools such as Daikon [17]. To this end, a small Erlang tracing module was developed[4], which runs as an independent Erlang process. The source code is instrumented at the exit points of functions that are of interest, such that every time an instrumented point is executed, it sends the relevant details (function name and variable values) to the trace process. The tracing process produces an output in the form of a Daikon trace file. It is optionally possible to add abstractions, which map the traces from lower level events to sequences of higher-level program functions.

Testing: As discussed earlier QuickCheck is particularly suited to this work because it can incorporate finite state machine (FSM) specifications[5]. The

[3] `http://statechum.sourceforge.net/`

[4] `http://www.dcs.shef.ac.uk/~nw/Files/FM2009/dtraceGenerator.erl`

[5] `http://quviq.com/eqc_fsm%20example/index.htm`

```
-record(state,{openfile}).
% openfile stores the name of
% the open file
%===== initial state =====
initial_state() -> a.
initial_state_data() ->
  #state{openfile=[]}.
%===== data generator =====
filename() -> elements([``test1'',
  ''test2'',''test3'']).
% data transformations
%===== state transition system =====
```

```
a(S) [{b,{call,editor,exit,[]}},
 {d,{call,editor,load,[filename()]}}].
b(S) [].
c(S) [{c,{call,editor,edit,[chars(4)]}},
 {a,{call,editor,close,[S#state.openfile]}},
 {b,{call,editor,exit,[]}},
 {d,{call,editor,save,[]}}].
d(S) [{c,{call,editor,edit,[chars(4)]}},
 {a,{call,editor,close,[S#state.openfile]}},
 {b,{call,editor,exit,[]}}].
```

Fig. 4. Example QuickCheck FSM specification of text-editor

specification of an FSM is essentially divided into four parts: The initial state specification, the state transition system, the data transformations and the data generators. A small example specification is shown in Figure 4. It should be noted that this example only contains the essential information for a FSM construction; QuickCheck supports a variety of other constructs (such as pre/post-conditions), which are omitted here.

Currently, all of the steps in Figure 3 are automated. With our implementation the user to provide an initial trace, along with a parameter stating the number of tests that should be executed for each candidate model. This will cause the entire process to iterate, terminating once it has produced a model that does not disagree with any tests.

4 Case Study

The case study revolves around a simple FTP-client that is a modified version of the `ssh_sftp`, which is part of the Erlang OTP ssh libraries (version 1.0.2) released by Ericsson[6]. For reference the main files involved in the tracing and testing are available on the web[7].

Subject system and set-up. The main functions of the FTP client are presented in Table 1. In this model we will only consider the operation of the FTP client with respect to a single file. The client has been deliberately designed to incorporate some reasonably intricate state-based rules. Only one file-handle can exist at a given time. Since a file has to be opened in either ‘read’ or ‘write’ mode, this means that a file can not be written to and read from at the same time. We have added the constraint that it is impossible to read from an empty file, and it is impossible to write to, or read from, a specific position in the file without having explicitly obtained the position using the *write_position* / *read_position* files first.

We start off by identifying the instrumentation points in the source code. This consists of identifying the points in the source code that correspond to exit

[6] `http://www3.erlang.org/documentation/doc-5.6.5/lib/ssh-1.0.2/doc/html/`
[7] `http://www.dcs.shef.ac.uk/~nw/Files/FM2009/`

Table 1. Functions of the FTP client (Σ in the PLTS)

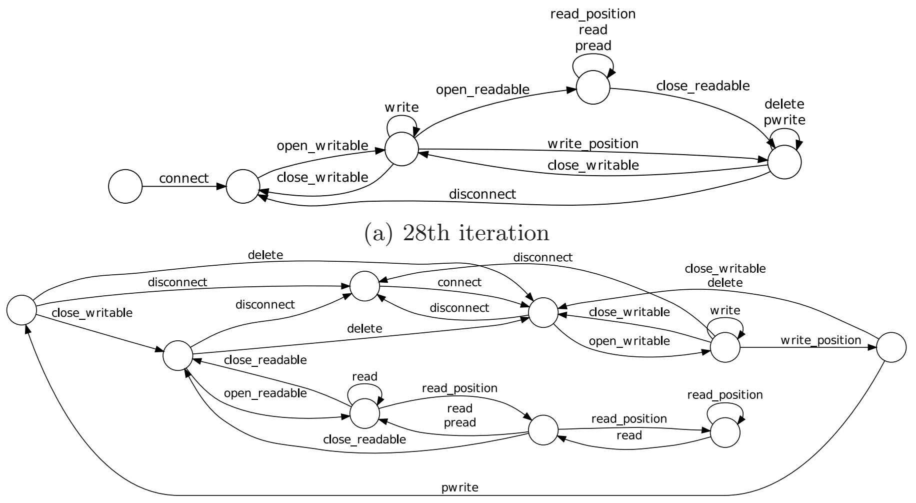
Function	**Description**
connect	connect to server, only one connection permitted at a time
disconnect	disconnect from server
open_writable, open_readable	open file in 'write' or 'read' mode
close_writable, close_readable	close file in 'write' or 'read' mode
write, read	write data to or read data from beginning of the file
write_position, read_position	obtain a specific position for writing to or reading from the file
pwrite, pread	write to or read from a specific position in the file
delete	delete the file

points for the abstract functions. In our case this is straightforward, as the abstract functions all correspond to actual function definitions in `ssh_sftp`. As an example, at the end point of `ssh_sftp.open` we insert a statement to add the name of the function "open", its arguments and the output of the function (see accompanying website for sample files). Depending on the arguments, the tracer either records an execution of the function as "open_new" or "open_existing". Having set up the tracer, all that remains is to set up the QuickCheck tester. StateChum has been augmented to automatically generate the QuickCheck model files from the inferred transition systems.

The inferred models. Figure 5 contains two snapshots from the inference process, which consisted of 57 iterations in total when run using the implementation of our technique described above. The process terminated when the testing process yielded no further tests that revealed faults in the hypothesis. The entire set of iterations is available on the accompanying website. For the sake of readability, proscribed transitions are not shown. Figure 5 (a) shows the model produced after 28 iterations. This model contains a number of errors (e.g., it permits the file to be opened in 'read' mode while it is still open in 'write' mode). The final candidate specification, which is produced after 57 iterations has 9 states, and is the most accurate version produced. Every state transition that is permitted in the model is also possible in the actual implementation (i.e., the inference process has made no over-generalisations).

Degree of accuracy. The key question is: how useful is the technique? One aspect of this is to what extent the inferred model is the actual behaviour of the implementation, although it should be noted that even a partial model will be of use to a developer in increasing their understanding of the system.

Figure 6 shows a "diff" between the final inferred model and the target. The technique used to compute this is defined in [18]. From this comparison it can be established that 19 transitions were correctly inferred by the machine, that 16 transitions are missing, and that 6 superfluous transitions have been added. Most of the missing transitions are of a particular nature: they are events that should be possible from every state in the system and would consequently require a very large number of tests to capture. The events "disconnect" and "delete" should be possible from most states, and account for 9 of the 16 transitions. The remaining missing transitions represent relatively minor differences in behaviour.



(a) 28th iteration

(b) 57th (final) iteration

Fig. 5. Inferred models

The correctly inferred transitions produce a reasonably accurate overview of how the system behaves; the system has its three well defined phases of operation - connection, writing to a file, and reading from that file. The requirement that a file can only be open either in "write" or "read" mode is correctly captured, and a file must be written to before it can be read.

To compare the language of the inferred model with its intended target precision, we adopt a technique that is defined in [19]. Precision denotes the extent to which the language that is represented by the inferred machine represents the language of the target state machine of the actual software system. Recall denotes the extent to which the language that is represented by the target machine is covered by the inferred machine. These measures have been applied to the inferred machine to separately assess the accuracy in terms of both valid and invalid languages. In terms of the valid language of the two machines, the precision is 96.2%, with a recall of 41%. The low positive recall tallies with the large number of missing transitions; a large number of sequences that should be accepted by the inferred machine are missing. In terms of negative precision and recall, the machine has a precision of 85%, and a recall of 99.5%.

In summary, both methods of comparison indicate that the model is precise, and that it captures the essential functionality of the system. Inaccuracies are primarily due to the fact that the model is inferred model can end up missing certain transitions, especially those that are possible from every state (such as "disconnect"). In practice, this can be explained by the choice of testing technique. Current versions of QuickCheck explore the model randomly, and only choose transitions that are in the model - they do not attempt to explore unspecified behaviour. This inevitably leads to some transitions potentially being missed out.

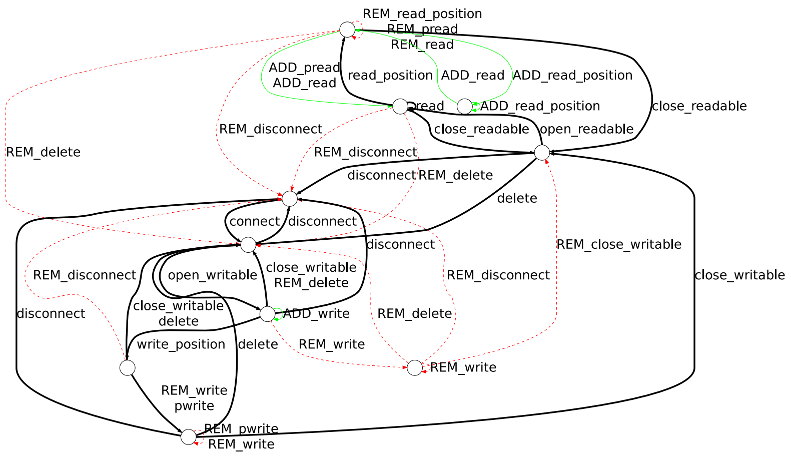


Fig. 6. Difference between final inferred model and reference model - Bold transitions are correct, thin (green) lines are added, dashed (red) lines are missing

A major strength of the technique demonstrated above is that it is based on an inference technique (the EDSM-bluefringe state-merging technique) that excels at dealing with *sparse* samples of program traces or tests. It can arrive at a reasonable hypothesis of how the program behaves, without requiring an exhaustive or impractically large number of traces or tests. One potential weakness of the implementation here is that, if errors are made early on in the state-merging process, they are compounded by future merges, so it relies on a sufficient base of traces to prevent invalid merges from happening. The iterative testing process is responsible for gathering this base of traces in the form of tests.

As a result, the accuracy of the final result is highly dependent on the testing technique that is used - in the name of efficiency we have used a simple random testing approach in QuickCheck. There are however a number of systematic testing techniques, such as the W-method mentioned above [15], and evaluation of how these can be integrated into the process is a key area of future work that we wish to undertake.

5 Conclusions and Future Work

This paper has presented an iterative approach to reverse-engineering labelled transition systems. The approach has been implemented and demonstrated with respect to and Erlang system, but can in princple be applied to any system regardless of the underlying language. The inference engine, along with the various illustratory resources are openly available.

The approach was demonstrated with respect to a small model of a real Erlang implementation of a FTP client. The resulting model is shown to be very precise, both in terms of the graph structure and the language that this represents.

There are a number of ways by which the authors intend to extend this work. A more extensive case-study will be used, to ensure that the approach scales reasonably to larger systems. There is already a lot of experimental evidence from the grammar inference community [16] to suggest that this will be the case. Previous work by the authors has involved the manual provision of selected LTL constraints to increase the efficiency and accuracy of the inference process. It is our intention to integrate these techniques with the current testing infrastructure.

As mentioned in Section 3.3, the traces are produced in the Daikon format [17]. Daikon can infer data constraints on variables from execution traces. We are currently investigating the use of Daikon to infer pre/post-conditions from Erlang executions, which can then be used to annotate the reverse-engineered state machines.

There are a number of QuickCheck features that could be used to increase the efficacy of the testing process. As mentioned previously, it is our intention to use the transition-weighting feature to coax the tester towards certain states that would otherwise be in danger of being left unexplored by random tests. It is envisaged that Uchitel's PLTS formalism could be particularly useful in this respect, helping to identify those states with lots of 'unknown' transitions, to automate the assignment of weights in the hypothesis machine.

Acknowledgements. This work is supported by the EU FP7 ProTest project. Walkinshaw is also supported by the EPSRC REGI grant (EP/F065825/1). The authors would like to thank Thomas Arts, John Hughes, Koen Claessen and Hans Svensson at ITU / Chalmers, Gothenburg for their valuable suggestions and comments.

References

1. Walkinshaw, N., Bogdanov, K., Holcombe, M., Salahuddin, S.: Reverse Engineering State Machines by Interactive Grammar Inference. In: 14th IEEE International Working Conference on Reverse Engineering, WCRE (2007)
2. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (July 2007)
3. Claessen, K., Hughes, J.: Quickcheck: A Lightweight Tool for Random Testing of Haskell Programs. In: Proceedings of the International Conference on Functional Programming (ICFP), pp. 268–279 (2000)
4. Walkinshaw, N., Bogdanov, K., Ali, S., Holcombe, M.: Automated discovery of state transitions and their functions in source code. Software Testing, Verification and Reliability 18(2) (2008)
5. Ernst, M.: Static and Dynamic Analysis: Synergy and Duality. In: Proceedings of the International Workshop on Dynamic Analysis, WODA (2003)
6. Walkinshaw, N., Bogdanov, K., Holcombe, M., Salahuddin, S.: Improving Dynamic Software Analysis by Applying Grammar Inference Principles. Journal of Software Maintenance and Evolution: Research and Practice (2008)
7. Gold, E.: Language Identification in the Limit. Information and Control 10, 447–474 (1967)

8. Dupont, P., Lambeau, B., Damas, C., van Lamsweerde, A.: The QSM Algorithm and its Application to Software Behavior Model Induction. Applied Artificial Intelligence 22, 77–115 (2008)
9. Biermann, A., Feldman, J.: On the Synthesis of Finite-State Machines from Samples of their Behavior. IEEE Transactions on Computers 21, 592–597 (1972)
10. Cook, J., Wolf, A.: Discovering Models of Software Processes from Event-Based Data. ACM Transactions on Software Engineering and Methodology 7(3), 215–249 (1998)
11. Angluin, D.: Learning Regular Sets from Queries and Counterexamples. Information and Computation 75, 87–106 (1987)
12. Walkinshaw, N., Bogdanov, K.: Inferring Finite-State Models with Temporal Constraints. In: Proceedings of the 23rd International Conference on Automated Software Engineering, ASE (2008)
13. Uchitel, S., Kramer, J., Magee, J.: Behaviour Model Elaboration using Partial Labelled Transition Systems. In: 4th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 19–27 (2003)
14. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley, Reading (2007)
15. Chow, T.: Testing Software Design Modelled by Finite State Machines. IEEE Transactions on Software Engineering 4(3), 178–187 (1978)
16. Lang, K., Pearlmutter, B., Price, R.: Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm. In: Honavar, V.G., Slutzki, G. (eds.) ICGI 1998. LNCS (LNAI), vol. 1433, pp. 1–12. Springer, Heidelberg (1998)
17. Ernst, M., Cockrell, J., Griswold, W., Notkin, D.: Dynamically Discovering Likely Program Invariants to Support Program Evolution. Transactions on Software Engineering 27(2), 1–25 (2001)
18. Bogdanov, K., Walkinshaw, N.: Computing the Structural Difference between State-Based Models. In: 16th IEEE Working Conference on Reverse Engineering, WCRE (2009)
19. Walkinshaw, N., Bogdanov, K., Johnson, K.: Evaluation and Comparison of Inferred Regular Grammars. In: Proceedings of the International Colloquium on Grammar Inference (ICGI), St. Malo, France (2008)