SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. (2007)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr.380

Automated discovery of state transitions and their functions in source code

Neil Walkinshaw[1,*,†], Kirill Bogdanov[1], Shaukat Ali[2] and Mike Holcombe[1]

[1]*Department of Computer Science, The University of Sheffield, Regent Court, 211 Portobello Street, S1 4DP Sheffield, U.K.*
[2]*Department of Systems and Computer Engineering, Carleton University, Ottawa, Ont., Canada*

SUMMARY

Finite-state machine specifications form the basis for a number of rigorous state-based testing techniques and can help to understand program behaviour. Unfortunately they are rarely maintained during software development, which means that these benefits can rarely be fully exploited. This paper describes a technique that, given a set of states that are of interest to a developer, uses symbolic execution to reverse-engineer state transitions from source code. A particularly novel aspect of our approach is that, besides determining whether or not a state transition can take place, it also identifies the paths through the source code that govern a transition. The technique has been implemented as a prototype, enabling its preliminary evaluation with respect to real software systems. Copyright © 2007 John Wiley & Sons, Ltd.

Received 17 September 2007; Accepted 19 September 2007

KEY WORDS: symbolic execution; finite-state machines; reverse engineering; model checking

1. INTRODUCTION

Perceiving software as a state machine enables the developer to design, document and rigorously test a program in terms of its behaviour. A system can be decomposed into a set of states, where each state characterizes the system at a particular point in its execution. The behaviour of the system is determined by a set of state transitions, where each transition leads from one state to another

*Correspondence to: Neil Walkinshaw, Department of Computer Science, The University of Sheffield, Regent Court, 211 Portobello Street, S1 4DP Sheffield, U.K.
†E-mail: n.walkinshaw@dcs.shef.ac.uk

Contract/grant sponsor: EPSRC; contract/grant number: EP/C511883/1

and is governed by some trigger (e.g. a user input) and, depending on the modelling technique, a (partial) function that attributes semantics to the state transition. Several powerful techniques have been developed to produce test sets for systems that have been modelled as state machines. (Lee and Yannakakis provide a comprehensive overview [1].)

One notable weakness with conventional state-based software modelling techniques is that they tend to specify only the input/output behaviour of a system and fail to provide insights into *how* and *why* state transitions take place. This can render them difficult to read, understand and validate, which in turn undermines the integrity of any test sets that are generated from them. This has inspired the development of more expressive state-based modelling techniques such as X-Machines [2], Abstract State Machines [3] and Extended Finite State Machines [4]. These introduce the notion of state *transition functions*, which enable the developer to precisely specify the functionality that governs state transitions. State machines with transition functions are easier to understand because, even if the states of the system are unintuitive, the semantics that underlie the state changes are made explicit.

To illustrate some of the problems that arise with conventional state-based specifications, Figure 1 shows a simple state machine for a binary search tree. Transitions are annotated with the inputs that trigger them. (Note that two transitions with the same trigger do not necessarily exhibit the same transition behaviour—this is elaborated below.) In practice the developer may ask questions such as

How exactly does behaviour of $Empty \overset{find}{\rightarrow} Empty$ *differ from* $Populated \overset{find}{\rightarrow} Populated$?

or

Tests indicate that a fault in the implementation results in the transition $Empty \overset{insertNode}{\rightarrow} Empty$ *being executed instead of* $Empty \overset{insertNode}{\rightarrow} Populated$. *Where is the fault in the source code*?

Conventional state-based modelling techniques fail to answer these questions because they can only identify the states and *whether* state transitions take place. They fail to provide any insights into *how* and *why* the system transitions from one state to another.

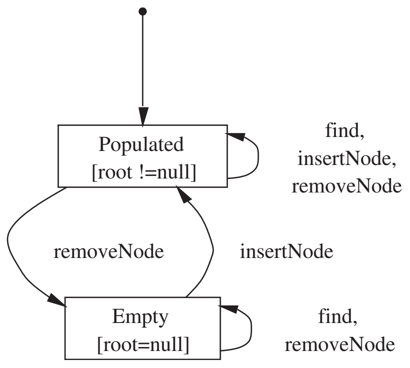


Figure 1. State machine for a simple binary search tree.

The ability to mine an implementation for its state machine would enable the developer to produce state-based functional test sets for the system at any time throughout its development [1,2]. This has spurred the development of several approaches to reverse-engineer state-based specifications. Although existing reverse-engineering techniques can suggest states or detect whether a state transition can take place [5–8], their practicality is limited for the reasons presented above; they do not provide any insights into the semantics of the state transitions.

This paper introduces a technique that not only reverse-engineers state transitions but also annotates each transition with its respective source code. The source code for a state transition is referred to as the *state transition function*. The technique is based upon the observation that the start and exit points of a state transition function can usually be mapped to particular syntax elements of the source code. In UML state charts, for example, state transitions are labelled with the name of the method that initiates the state change. A state transition function for a state chart transition $A \xrightarrow{f()} B$ consists of the source code that is executed between the entry and exit points of method $f()$ (i.e. the outermost '{' and '}'), when it is called in the execution context represented by state A.

A state transition function makes an explicit connection between the design and the source code, thus aiding the traceability from requirements to the implementation [9] and helping the developer to answer questions such as those presented above. This makes the technique particularly useful for understanding, inspecting and testing the dynamics of software that is designed in terms of states and transitions. This is especially the case for state machines with potentially complex state transition semantics, such as X-Machines and Abstract State Machines.

This paper makes the following three principal contributions:

1. It introduces a technique to reverse-engineer state transitions from source code, along with their respective transition functions.
2. It describes the implementation of the technique as a proof of concept, using a novel approach to program conditioning.
3. It evaluates the performance and scalability of the approach by applying the implementation to a small selection of case studies.

The following section provides a brief introduction to program conditioning and symbolic execution, the analysis technique upon which our approach is based. Section 3 provides an overview of our technique. Section 4 shows how the technique has been implemented. Section 5 contains a small example that demonstrates how the technique can be used to answer questions about the binary search tree presented above, as well as a larger example based on the JHotDraw drawing framework. Section 6 provides an overview of related work and Section 7 discusses the future work and conclusions.

2. PROGRAM CONDITIONING AND SYMBOLIC EXECUTION

There are a number of approaches that could be adopted to identify the statements that correspond to a state transition function. The most straightforward approach would be to simply execute the program, and to record those statements that are executed between the transition function entry and exit points. Although the results would be precise, determining the necessary set of input sequences to exhaustively execute the relevant paths in the source code is widely acknowledged as a laborious and expensive task. Conventional static analysis techniques such as data flow analysis [10] are also difficult to apply effectively, because it is challenging to restrict the results to only those statements that are executed from a specific execution context (or program state).

Program conditioning [11] is a novel program analysis technique that circumvents the aforementioned problems. Danicic *et al.* [12] define program conditioning as follows: '*Conditioning is the act of simplifying a program assuming that the states of the program at certain points in its execution satisfy certain properties*'. In other words, it identifies those statements that are executed, provided that certain conditions expressed as constraints on program variables at particular program points hold true.

In the context of this work, a state transition is embodied by a set of paths through the program that cause the system to change state. This paper is concerned with trying to identify these paths. Program conditioning is a useful technique to achieve this; given a transition $A \xrightarrow{f()} B$, if A and B can be encoded as conditions on the program variables at particular points during its execution, a program conditioner will remove those lines of code that do not contribute to the process of reaching B from A; it will identify the transition function $f()$.

Figure 2 illustrates program conditioning on a function that classifies triangles (this is inspired by a similar example by Fox *et al.* [13]). The left column contains the whole (unconditioned) program. By adding the condition that `a` and `c` are never equal, it can be inferred that the predicates in lines 2 and 7 will always execute the false branch. A program conditioner retains the behaviour of the program with respect to the condition by removing these predicates, along with their branches that will not be executed (lines 3 and 8).

A program conditioner determines which statements to retain or remove by symbolic execution [14,15]. When a program is symbolically executed, its input values are substituted with symbolic values. During the symbolic execution program, variables are manipulated as symbolic expressions instead of concrete values. The outputs of a symbolic execution are expressed as a set of constraints on the input symbols (referred to as the 'path condition').

The state S of a symbolically executed program can be denoted as follows:

$$S = (V, PCondition, PCounter)$$

where V represents the (symbolic) values of program variables, *PCondition* is a quantifier-free boolean formula over the symbolic program inputs and *PCounter* represents the program counter that points to the current statement. A symbolic execution can be represented as a tree [15], where nodes are symbolic states, and each leaf node corresponds to the termination of a potential path through the program. The symbolic execution tree for the triangle program is shown in Figure 2. Elliptic nodes correspond to conditional branches (i.e. if predicates) that are encountered during program execution. The statement line number n that corresponds to a given node is shown as $\langle n \rangle$, the path condition is denoted as PC. Dotted edges represent elements that would be removed in the conditioned version of the program.

The program conditioner uses a constraint solver to establish whether the path condition for each state concurs with the user-supplied condition. For example, in Figure 2 the supplied condition is $\neg(a = c)$ and the path condition at the node for line 3 is $(a = b \wedge a = c)$. Because there is no solution to the constraint $\neg(a = c) \wedge (a = b \wedge a = c)$, the conditioner can remove line 3.

Unconditioned

```
1 if(a==b){
2  if(a==c)
3   r=''equilateral'';
4  else r = ``isosceles'';
5 }
6 else{
7  if(a==c)
8   r=''isosceles'';
9  else if (b==c)
10  r=''isosceles'';
11 else r=''scalene'';
12}
```

Conditioned: $\neg(a = c)$

```
1 if(a==b){
2
3
4  r = ``isosceles'';
5 }
6 else{
7
8
9  if (b==c)
10  r=''isosceles'';
11 else r=''scalene'';
12}
```

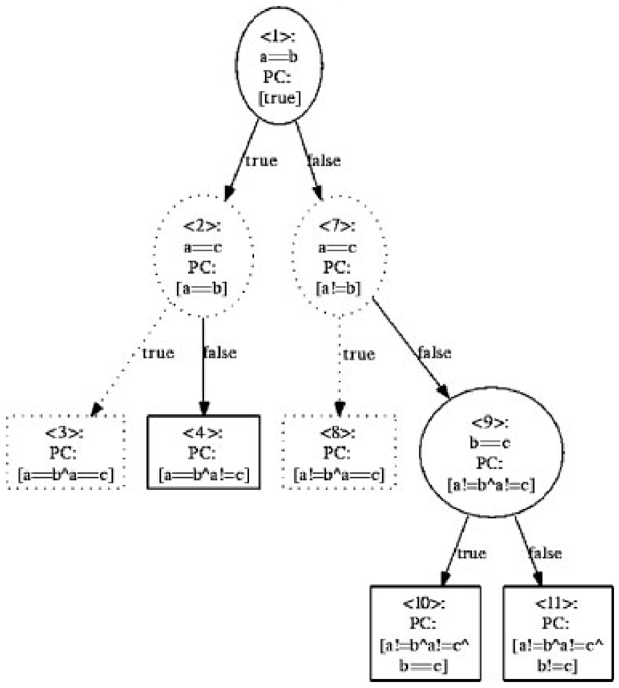


Figure 2. Small example of a conditioned program and its symbolic execution tree.

The potential for a program conditioner to limit the size of the program depends on the capabilities of the underlying theorem prover(s). These are often limited to reasoning about linear constraints, although there has been work on combining symbolic executors with a selection of solvers fitting particular types of problems [16]. It should however be noted that several papers that involve symbolic execution (cf. [14,16]) have commented that it is often the case that, even with complex programs, the control flow itself is often governed by relatively simple conditions for which simple boolean and linear constraint solvers will often suffice.

Problems can arise in the symbolic execution of loops. If the termination of a loop cannot be inferred from the current path condition or the loop has no concrete upper bound, it can result in an infinite execution tree. Current symbolic execution techniques usually address this by either manually inserting upper bounds on specific loops or providing an absolute limit on the depth of the execution tree, regardless of loops. More sophisticated techniques to address this problem will be discussed in Section 7.

3. DISCOVERING STATE TRANSITIONS AND THEIR FUNCTIONS

The main contribution of this paper is a technique that, given a set of rules that indicate when state transitions occur in the source code and (optionally) a set of abstract states, returns the set of all possible state transitions in the program along with the source code that governs their execution. The technique is based on the observation that the start and end points of a state transition usually map directly to particular syntax constructs in the source code. For example, in an object-oriented system, the method calls trigger state transitions. If method $f()$ causes a transition from state A to state B, the state transition function consists of the source code that is executed between the entry and exit points of $f()$, given that the system is in state A when $f()$ commences and in state B when it terminates.

This section provides an overview of the approach. The process of recovering a set of state transitions from an implementation follows the following four steps (Steps 2–4 are illustrated in Figure 3):

1. Identify state transition points in terms of the source code syntax (e.g. method calls, exceptions, etc.).
2. Construct the symbolic execution tree, marking all symbolic execution states that correspond to transition points.
3. Map marked transition points to abstract machine states (these will be referred to simply as 'machine states').
4. Identify state transitions between the abstract states by detecting consecutive marked transition points in the symbolic execution tree. Since every symbolic state corresponds to the execution of a statement in the source code, the statements belonging to a transition function simply correspond to those symbolic states between a pair of marked transition points.

Conceptually, the process of identifying state transitions is relatively simple. Given that we know where a transition can start (a transition point) and end (a subsequent transition point), we establish an abstract machine state for every transition point in the symbolic execution tree. The interval between a pair of transition points thus corresponds to a state transition function. The following subsections elaborate on the four steps presented above.

3.1. Identifying state transition points in the source code syntax

This technique is primarily motivated by the observation that, for most modelling approaches, it is straightforward to map the start and end of a state transition to the syntax of the source code. The type of statements that encapsulate a potential state transition depends on the state-based model

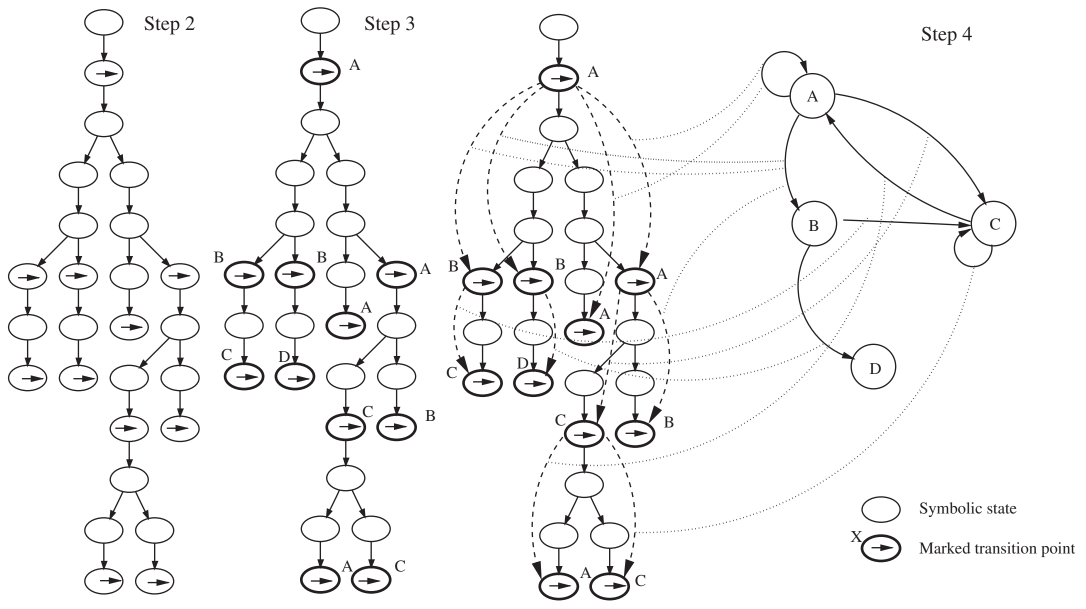


Figure 3. Illustration of the state transition identification process.

that is employed. Identifying them is, however, so straightforward that this step can usually be fully automated. As an example, in UML state charts transitions are triggered by calls to methods. Because the method is therefore responsible for the state change in the system, it makes sense to observe the state of the system before and after each method is executed. Consequently, the state transition points can be identified as the entry and exit points for every method body.

Although we are adopting the aforementioned conventional state-based object model in this paper, it should be noted that it would be straightforward to adapt this technique to more complicated state machine models. As long as state transition points can be mapped to specific control structures in the source code (e.g. a thrown exception or catch point), these can be marked as transition points for this technique. This makes it particularly suitable as a basis for experimenting with new state-based models, for the sake of assessing their utility for testing and comprehension.

3.2. Marking transition points in the symbolic execution tree

The previous step identified those statements in the source code that correspond to transition points. This step maps these transition points to symbolic states in the symbolic execution tree. This is achieved by simply identifying those symbolic states where the program counter matches the transition point statement. As an example, if we know that a state transition happens every time a given method is executed, we identify the symbolic states for every return point of that method. It should be noted that there can be a one-to-many mapping from source code statements to symbolic execution states, because loops can result in a single statement being executed multiple times. In the following sections, the set of symbolic states that correspond to transition points are denoted as T.

3.3. Mapping transition points to machine states

Depending on the underlying state machine model, the definition of what constitutes a state may vary. States often correspond to well-defined equivalence classes of variable values (such as the *Populated* and *Empty* states in Figure 1). Alternatively, in models of control-based systems such as network protocols, for example, states merely act as separators to impose an order on the possible sequence of state transitions.

The approach that is presented in this paper is sufficiently general that it can be applied to either state representation. A function $getState : T \rightarrow M$ is required that identifies a respective machine state M for each transition point T. As mentioned in Section 2, a symbolic state can be represented as the tuple $S = (V, PCondition, PCounter)$. Thus, if M contains a set of abstract data states, the *getState* function uses the symbolic variables T_V. Alternatively, if M consists of a set of control states, the *getState* function will use the program counter $T_{PCounter}$. For the sake of generating a sound state machine, the *getState* function must be total; there must be a member of M for *every* transition point $t \in T$.

As a simple example, in the binary search tree state machine in Figure 1, $M = \{Populated, Empty\}$. For each symbolic state $t \in T$, the symbolic value of *root* can be obtained from t_V. Thus, defining *getState* is straightforward; $getState(root = null) = Empty$ and $getState(root! = null) = Populated$.

Ultimately, it is up to the user to select a suitable set of states that fit their application (it is generally accepted that different applications require different abstractions [17]). Section 4 shows a small example that reverse-engineers the state transition functions for the abstract data states in the binary search tree example, and Section 5 shows a larger example that reverse-engineers state transition functions with respect to control states in the NanoXML application. Potential approaches to (partially) automate the identification of candidate states are elaborated in Section 6.

3.4. Identifying state transitions and their functions

Previous steps have generated the set T of nodes in the tree that correspond to state transition points, along with the function $getState : T \rightarrow M$ that identifies a suitable machine state for every symbolic state that belongs to T. In this final step, the symbolic execution tree is used to identify feasible state transitions, along with the set of statements that are responsible for each transition.

A feasible state transition consists of a pair of subsequent transition points in the symbolic execution tree. These can be identified with conventional tree-search algorithms. For every set of consecutive transition points (a, b) we store the (machine) state transition $getState(a) \rightarrow getState(b)$.

Once a state transition has been identified in the symbolic execution tree, it is relatively straightforward to identify its corresponding state transition function (i.e. the source code that is responsible for the state transition). Given a pair of marked states in the symbolic execution tree, the function corresponds to the source code that is executed between them. Because there is a direct mapping from states in the symbolic execution tree to statements in the source code (via the program counter), the transition function statements can be readily identified as part of the search process that identifies consecutive transition points.

This process is akin to program conditioning (see Section 2). Conditioning conventionally involves the manual insertion of `assert` statements, which contain conditions on program variables at specific points. The approach used here to identify state transition functions specifies conditioning criteria slightly differently; instead of specifying a single program point and a single set of variable constraints, it specifies multiple program points (transition points in T) and multiple sets of variable constraints (defined by the *getState* function). Similar to program conditioning, this approach takes a set of constraints as input from the user, but, instead of specializing a program with respect to these constraints, this approach highlights how the statements belonging to a program are related in terms of the constraints.

4. IMPLEMENTATION

The implementation is designed to compute the possible state transitions and transition functions of Java systems. It uses the Java PathFinder (JPF) explicit state model checker [18] and its symbolic execution extension [19] to construct and traverse the symbolic execution tree. JPF uses a custom-made Java virtual machine to explore the byte code of Java programs. The model is constructed largely automatically by identifying branches in the class files. The implementation consists of two parts: The first part is responsible for instrumenting the program to make a program symbolically executable and inserting annotations at transition points, which communicate the state at that point back to the model checker. The second part is a listener class that monitors the JPF search process and identifies the state transitions as they are encountered.

Before a program can be symbolically executed with JPF, it has to be instrumented. The instrumentation process is detailed by Khurshid *et al.* [19] and simply consists of identifying branches where the predicate is to be added to the path condition, along with which variables should be processed symbolically. The step is relatively straightforward and will be automated in future versions of JPF, hence it is not elaborated in this paper.

During the symbolic execution, whenever the model checker encounters a state transition point, it needs to identify the current abstract state (see step 3 in Figure 3). Our implementation uses the Soot byte code transformation (and instrumentation) framework [20] to insert instructions to communicate the current state to the model checker at each transition point. The rules that state what constitutes a state transition point can thus be stated in terms of the Soot intermediate representation, which is relatively straightforward. The rules presume by default that the source code that is executed between the first and last statement of every public method (possibly including code executed in other methods called by that method) is responsible for a state transition. Therefore, the first and last statements of each method are instrumented and flagged as transition points. (It should be noted that these rules can be customized and extended.)

Once the system has been instrumented, JPF can traverse a model of its state space. By using the symbolic execution extension, this traversal is akin to a search of the symbolic execution tree. The construction of the actual state machine is implemented as a listener for the model checker that extracts the state machine given that the subject program has been instrumented as above. In terms of the technique overview in Section 3, this corresponds to step 4.

JPF provides several listeners that can be used to monitor the progress of the model checker as it traces along each execution path (see JPF literature for further information [18]). As the model is a symbolic execution tree, the termination of each path corresponds to a leaf node in the tree.

```
procedure getTransitionsFromPath(P)

Input:
$P =< s_0,...,s_n >$:  A list of symbolic states constituting a symbolic execution path.

Output:
$T$:  A set of state transitions, initially empty.

Use:
$getAnnotation(S)$ returns the string used to annotate symbolic state $S$.
$getState(A)$ returns the substring of annotation $A$ that corresponds to the machine state.
$getPC(A)$ returns the substring of annotation $A$ that corresponds to the program counter.
$getCurrentBasicBlock(S)$ returns a string "$a-b$", where $a$ and $b$ represent the first and last lines in the
basic block of source code that corresponds to the execution of symbolic state $S$.
$createTransition(S,S',B,PC)$ returns a new state transition.  $S$ and $S'$ are strings that denote the source and
target states, $B$ is a set of basic blocks and $PC$ is a path condition.

Declare:
$currentState$, $nextState$, $pathCondition$, $annotation$:  Strings
$B$:  list of strings
$\varepsilon$:  empty string

(1) $T \leftarrow \emptyset$
(2) $currentState \leftarrow$"init"
(3) foreach $s_i \in P$
(4)    $annotation \leftarrow getAnnotation(s_i)$
(5)    if $annotation \neq \varepsilon$
(6)       $nextState \leftarrow getState(annotation)$
(7)       $pathCondition \leftarrow getPC(annotation)$
(8)       $T \leftarrow T \cup createTransition(currentState, nextState, B, pathCondition)$
(9)       $currentState \leftarrow nextState$
(10)      $B \leftarrow \emptyset$
(11)   else
(12)      $B \leftarrow B \cup getCurrentBasicBlock(s_i)$
(13)endfor
(14)return $T$
```

Figure 4. Algorithm that processes a path to extract state transitions and their functions.

Every time the search reaches a leaf node it notifies the listener, which stores the execution path. Once the search is finished, every path is processed to extract its state transitions.

This process is detailed in the `getTransitionsFromPath` procedure in Figure 4. The purpose of auxiliary functions is implied by their name; for example, `getState` reads the annotation and returns the substring containing the instrumented state. A path, which is a list of symbolic states, is processed sequentially. If a state is not annotated, it is presumed to be part of the function that governs the state transition; hence, its source code line numbers are added to a temporary list. If the state is annotated, it corresponds to a transition point, and a `StateTransition` is generated from the prior annotated state. The intervening blocks of source code are treated as the transition function, and the symbolic path condition at the annotated state represents the conditions on the inputs required to execute the function.

5. EXAMPLES AND DISCUSSION

The previous section shows how the tool is implemented. This section uses the implementation to demonstrate how it can be used to understand the essential behaviour of two systems. First, a simple binary search tree is analysed. This is used to provide a detailed demonstration of the working implementation, and to answer the two motivating questions posed in the Introduction. In Section 5.2, the technique is applied to the larger JHotDraw system to show how it can be used to understand the behaviour of abstract state transition functions from a model that has been generated *a priori*. Section 5.3 looks into the limitations of the technique.

5.1. Small example—binary search tree

The binary search tree implementation[‡] is shown in Figures 1 and 5. The `BSTree` class provides the interface methods for manipulating or querying the tree and contains a pointer to the root node of the tree. The `BSTNode` class implements the nodes of the tree, and each node contains a pointer to its left node (containing a node with a lower value) and its right node (containing a node with a greater value).

5.1.1. *Obtaining state transitions and transition functions*

This subsection demonstrates how the implementation detailed in Section 4 can be used to extract the transitions and their functions from the binary search tree system. It is presumed that the developer wants to find out precisely how the system behaves in terms of the *Empty* and *Populated* states;

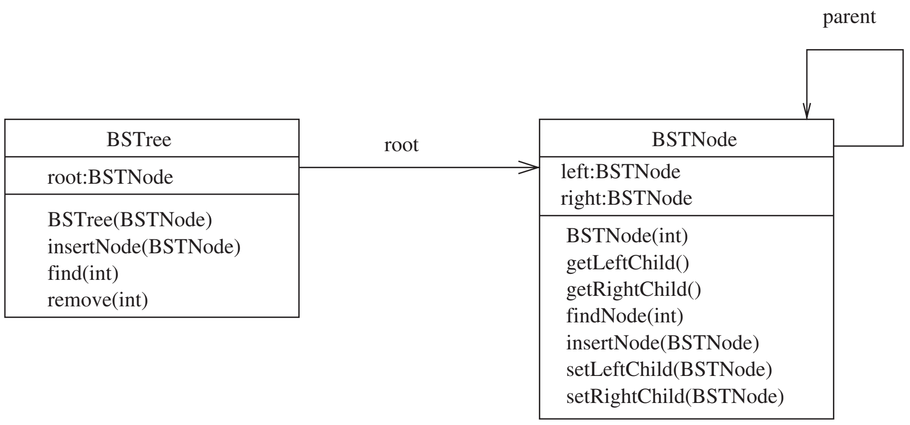


Figure 5. Binary search tree classes.

[‡]The binary search tree source code is too large to include in its entirety. It can, however, be downloaded along with its instrumented version from http://www.dcs.shef.ac.uk/∼nw/autoAbstract/downloads.html.

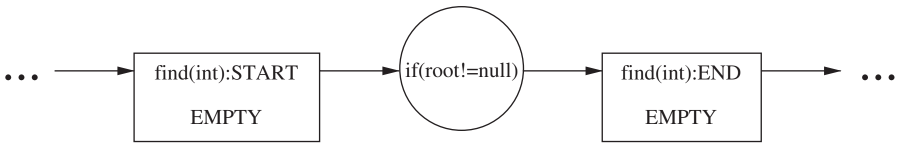


Figure 6. Extract from an annotated symbolic execution path.

hence, these are encoded in the *getState* function. It is also presumed that every public method can be responsible for a state transition.

Having identified the relevant abstract states and state transition points, the program is symbolically executed so that the state transitions can be identified. This involves symbolically executing every possible permutation of calls to `BSTree.find`, `BSTree.insertNode` and `BSTree.remove`, where each combination of calls can reach a specified limit. This can be achieved with the model checker (the approach is documented by Visser *et al.* [21]) by forcing the model checker to search through each permutation, automatically backtracking once each execution has been exhausted.

Transition functions are obtained by determining the set of statements that are executed between a pair of state transition points. Figure 6 shows an extract from an execution path of the binary search tree as the find method is executed. Square boxes correspond to annotated state transition points. The transition function simply consists of the statement `if(root!=null);` (the body of the if statement is not executed because `root==null` and corresponds to the state transition $Empty \xrightarrow{find} Empty$.

5.1.2. Understanding the state transitions of the binary search tree

State transition functions are particularly useful for understanding and debugging software from a state-based perspective. This is demonstrated by using them to answer the two questions about the behaviour of the binary search tree from the introduction. These were

How exactly does behaviour of $Empty \xrightarrow{find} Empty$ *differ from* $Populated \xrightarrow{find} Populated$?

and

Tests indicate that a fault in the implementation results in the transition $Empty \xrightarrow{insertNode} Empty$ *instead of* $Empty \xrightarrow{insertNode} Populated$. *Where is the fault in the source code*?

This section shows that the functions for these transitions can help to answer these questions. The relevant transition functions (as extracted by the JPF extension described previously) are all presented in Figure 7.

Answering the first question, the source code in Figure 7(a) shows that, although it is possible to call the `find` function when the tree is empty, it will simply return `null`. The source code in (b) shows that, if the tree is populated, a call to `find` will traverse the tree until it finds the node that corresponds to the value provided, or return `null` if it finds nothing. Note that in (a) the body for

(a) $Empty \overset{find}{\rightarrow} Empty$

BSTree.find(int val)

```
public BSTNode find(int val){
 if(root != null){}
 return null;
}
```

(b) $Populated \overset{find}{\rightarrow} Populated$

BSTree.find(int val)

```
public BSTNode find(int val){
 if(root != null){
  return root.findNode(val);
 }
}
```

BSTreeNode.findNode(int val)

```
public BSTNode findNode(int val){
 if(val<this.value){
  if(left!=null)
   return left.findNode(val);
  else return null;
 }
 else if (val>this.value){
  if(right != null)
   return right.findNode(val);
  else return null;
 }
 return this;
}
```

(c) $Empty \overset{insertNode}{\rightarrow} Empty$

BSTree.insert(BSTNode node)

```
public void insert(BSTNode node){
 if(root != null){}
}
```

(cropped unconditioned version) ↓

```
public void insert(BSTNode node){
 if(root != null){
  root.insert(node)
 }
}
```

Figure 7. Transition functions.

the if condition is empty because the program conditioner has determined that it cannot be executed for this state transition.

The second question is particularly interesting, as the example stems from a genuine fault in the source code that was discovered during the process of validating the implementation. The state transitions that were reverse-engineered initially did not match the transitions that were predicted (see Figure 1). From the *Empty* state, the *insertNode* state transition remained in the *Empty* state instead of the *Populated* state. To investigate why this is the case, it is straightforward to analyse the transition function that is triggered by calling *insertNode* in the *Empty* state. By analysing the $Empty \xrightarrow{insertNode} Empty$ transition function in (c), the fault becomes immediately apparent; the insertion of a node is guarded by a condition ensuring that `root` is not `null`. If `root` is `null` (as is the case when the tree is empty), the insert method does nothing. A probable reason is the use of two constructors, one of which inserts a root node by default, meaning that the tree starts off in the *Populated* state. If the programmer always used this constructor, and never removed all of the nodes from the tree, the problem of populating the tree from the *Empty* state might never have been encountered. The fault is fixed by inserting an `else` clause that stores the inserted node as the root if the tree is empty. Although this example is relatively simple, it still demonstrates that this approach has the potential to be particularly useful for debugging by improving traceability between the design and implementation.

5.2. Larger example—JHotDraw

JHotDraw[§] is a well-established, open-source Java framework for constructing drawing tools. HotDraw was originally developed in the eighties by Cunningham and Beck as a Smalltalk drawing editor framework. It was then rewritten for Java as JHotDraw, and serves as a showcase to illustrate the use of object-oriented design patterns. JHotDraw is particularly suitable as a case study because it has been extensively explored in other software engineering research projects.

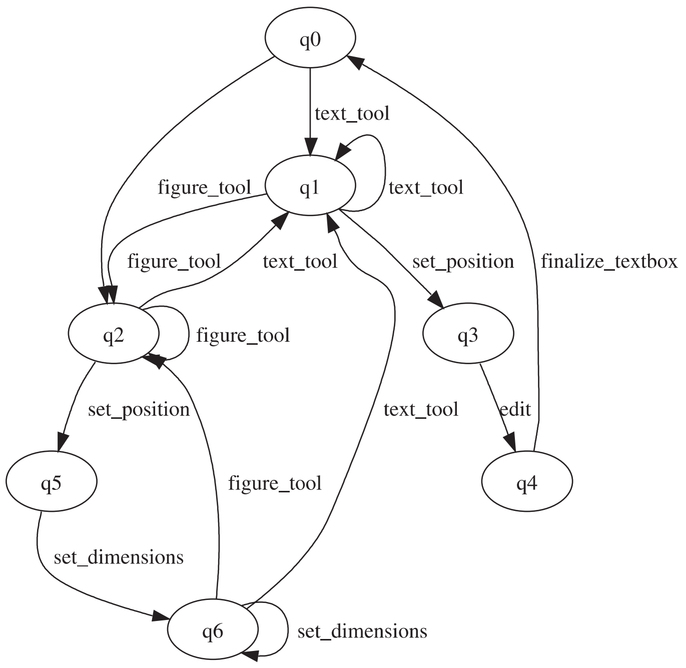
In this case study we concentrate on the specific behaviour of the JavaDraw application, which is included as an example application in the JHotDraw release. A simple state machine of its high-level behaviour, along with a screen shot of JavaDraw, is shown in Figure 8. The behaviour of the system is decomposed into six system functions, which are described in the table below the transition diagram.

5.2.1. Model analysis

In its entirety, JHotDraw is too large to symbolically execute, as is using our JPF-based implementation. For the sake of this experiment, the symbolic execution was restricted to the 10 classes that contribute the core functionality in terms of the diagram shown above. The rest of the system was replaced by stubs using the 'Model-Java Interface' infrastructure provided by JPF.

One inevitable problem that arises, particularly with JHotDraw, is the fact that a large number of methods operate on non-primitive data types. In this case, if the parameter was a relatively simple object (e.g. an Integer or String object), it was replaced with a symbolic counterpart. If the object was more complex, including multiple data members that affect the execution path through the method, a symbolic version was generated, with relevant data members replaced by symbolic counterparts. In certain cases a parameter can contribute to a condition that is beyond the capabilities of the constraint solver, in which case it is simply omitted (at the potential expense of the accuracy of the final model—see the limitations in Section 5.3). Once the stubs and parameters have been set up, the program was instrumented as described in Section 4.

[§] http://www.jhotdraw.org.



System Function	Description
`text_tool`	text tool is selected and activated
`figure_tool`	figure tool is selected and activated
`set_position`	user selects position on canvas
`edit`	user edits text box
`finalize_textbox`	text box is committed to canvas
`set_dimensions`	user sets dimensions of a figure

Figure 8. JavaDraw hypothesis machines and the abstractions.

Every time JPF enters a new symbolic state, the *getState* function checks whether the new state corresponds to one of the states $q_0 - q_6$. This is done by checking the path counter and current variable values. Note that the states in the previous binary search tree example could be ascertained from the variable values alone. The mappings from abstract states to control and data states are shown in Table I. The [RET] means that the state is recorded at the return point of the method.

Having carried out the preliminary steps of preparing the system for symbolic execution, it is now possible to map the high-level system functions as specified in Figure 8 to the blocks of code that implement them, along with the associated conditions on the input variables that govern their execution.

In this example, the model in Figure 8 was generated *a priori*, as part of a case study [22] that illustrates the use of X-Machines to model software systems. The decomposition into these six

Table I. Mapping from abstract states to control and data states as encoded in *getState* function.

State	Data	Control
q_0	—	—
q_1	selectedTool = 1	PaletteButton.mouseReleased [RET]
q_2	selectedTool = 2	PaletteButton.mouseReleased [RET]
q_3	selectedTool = 2	StandardDrawingView.mousePressed [RET]
q_4	selectedTool = 1	StandardDrawingView.mousePressed [RET]
q_5	—	TextTool.beginEdit [RET]
q_6	—	CreationTool.mouseDrag [RET]

Transition: $q_1 \overset{setPosition}{\rightarrow} q_3$
PC: selectedTool = 1 (text)

```
1.   $anchor$ = co-ordinates of mouse click
2.   $figure$ = find figure at $anchor$
3.   if $figure$ != null
3.1    $textHolder$ = get text holder from $figure$
4.   if $textHolder$ != null
4.1    edit $textHolder$
5.   else
5.1.   $figure$ = generate new text box at $anchor$
5.2.   $textHolder$ = get text holder from $figure$
5.3.   edit $textHolder$
```

Transition: $q_2 \overset{setPosition}{\rightarrow} q_5$
PC: selectedTool =2 (figure)

```
1.   $anchor$ = co-ordinates of mouse click
2.   $figure$ = generate new figure at $anchor$
```

Figure 9. Pseudocode of transitions $q_1 \overset{setPosition}{\rightarrow} q_3$ and $q_2 \overset{setPosition}{\rightarrow} q_5$.

abstract system functions matches the developer's perception of the behaviour. Thus, according to the diagram, one would assume that transitions $q_1 \overset{setPosition}{\rightarrow} q_3$ and $q_2 \overset{setPosition}{\rightarrow} q_5$ should have a similar underlying implementation.

By using the state mappings in Figure 1, the tool automatically identifies the source code that implements the two state transitions. For the sake of readability, the source code identified by the tool for the two transitions has been summarized in pseudo-code form in Figure 9. Contrary to what might be expected from the abstract functional decomposition shown in Figure 8, although the two transitions are labelled with the same abstract function, their behaviours are quite different. If the text tool is selected and the mouse is clicked on the canvas, the *setPosition* function checks whether the mouse has been clicked on a figure and whether that figure has a text holder; if this is the case it activates the editor. (This leads to the *edit* transition function.) If there is no figure, or the clicked figure does not contain a text holder, a new text box figure is created, and its text holder is made available for editing. If, on the other hand, the figure tool is selected, the previous steps of checking the mouse click location are skipped, and the tool simply generates a new figure straight away.

Although this example is relatively simple (small scale, and only computed with respect to a small number of symbolic variables), it demonstrates the benefits of being able to relate abstract state transition functions to their implementations. The benefits are obvious for tasks such as software design and code inspections [23], general program comprehension and also for identifying test inputs to execute sequences of abstract state transitions identified by transition-testing techniques such as the W-Method [1,24].

5.3. Limitations

The previous sections have demonstrated that, although the approach is feasible, there are certain limitations common to most symbolic-execution-based techniques that must be discussed. Constraining the size of the symbolic execution tree can result in an incomplete set of state transitions and transition functions. Also, certain path conditions can be too expensive to solve with conventional constraint solvers, and can thus result in the inclusion of infeasible state transitions and transition functions. These two problems are elaborated below, and potential means of addressing them are discussed in Section 7. Coward [14] provides a more comprehensive discussion of both of these problems.

5.3.1. Missing or incomplete state transitions

The size of the symbolic execution tree increases exponentially, especially in the presence of constructs such as loops and `if` predicates. As a result, it becomes necessary to impose certain constraints on the symbolic execution. These constraints are usually limits either on the number of times individual loops can execute (i.e. execute every loop three times), or on the depth of the symbolic execution tree as a whole (regardless of the number of times individual loops execute). In the context of this technique, such limitations mean that, depending on the abstract states that may be of interest to the developer, certain state transitions or transition functions may never be symbolically executed. If, for example, a state of interest is reached only after a loop is executed a substantial number of times and the symbolic tree depth limit is relatively low, it may not be observed. If the state transition is identified, it could still be the case that certain blocks of code only contained at a certain depth of the execution tree are not executed and thus not correctly attributed to the transition.

5.3.2. Infeasible state transitions

Most symbolic executors are supplied with theorem provers that deal only with linear or boolean constraints. These are straightforward to solve, with a relatively low computational overhead. Nonlinear constraints (e.g. constraints that contain variables that are squared) are more challenging and usually require specialized theorem provers that require the developer to provide additional information about the problem domain. It should be noted that the JPF symbolic executor does provide the facility to interface with nonlinear solvers, but these are not integrated with the current implementation. The current inability to eliminate infeasible paths with nonlinear path conditions can result in the possible inclusion of infeasible state transitions and transition functions. Whenever a nonlinear transition is encountered during symbolic execution, JPF alerts the user (by printing the warning to the screen) to permit a manual inspection of transitions with nonlinear path conditions.

5.3.3. Manually identifying abstract states

In its current form, the approach relies on the developer identifying a suitable set of states. This can become problematic if the developer identifies conflicting states; for example, the developer may identify two 'overlapping' states, such that the system can be in two abstract states at the same time. At the moment, the approach would simply choose one of the two states, and the resulting machine would be incorrect. This can be avoided only if the user chooses a suitable set of initial states. Nonetheless, there are a number of more systematic abstract state identification approaches (for example, see the work by Dallmeier *et al.* [25]), which can be used in conjunction with this technique.

6. RELATED WORK

Section 6.1 puts this work into the context of existing approaches to reverse-engineer state-based specifications. Section 6.3 discusses this technique in the context of program conditioning and symbolic execution.

6.1. Reverse-engineering axiomatic specifications

There exist several approaches to reverse-engineering invariants and axiomatic specifications from software. The information provided by these techniques can be seen as complementing the technique presented in this paper, because they are useful for constraining the size of the symbolic execution tree (e.g. with loop invariants), as well as for providing hints at abstract states that may be of interest to the developer, which can be used to define the *getState* function for this technique.

Ernst *et al.* [8] have developed a tool called Daikon to reverse-engineer invariants from source code. Using their approach the subject program is (automatically) instrumented and executed using a test set. Daikon can then be used to infer invariants at particular points in the program, such as procedure entry and exit points as well as loop invariants.

Henkel and Diwan [26] have developed a dynamic technique to reverse-engineer algebraic specifications from Java classes (axioms are reverse-engineered at an interface level as opposed to the lower-level Daikon invariants). They heuristically generate a large number of terms, which are sequences of method invocations that do not result in an exception. The outcomes of terms are compared with each other, generalizing these comparisons to axioms, and then term rewriting is used to eliminate redundant axioms.

Tillmann *et al.* [27] note that dynamic approaches rely on a comprehensive test set, which can often be difficult to produce. Their approach uses symbolic execution to discover axiomatic class specifications. For a given class they identify its modifier methods (methods that modify the state of the class). For each modifier method they also identify a set of observer methods, which reflect the state change caused by the modifier method. They then symbolically execute each modifier method, recording the symbolic paths. For each path they symbolically execute the observer methods at the initial and final execution states. The symbolic terms over the implementation state are then mapped to equivalent terms of the observer methods. Ultimately this results in multiple path-specific axioms in terms of the observer methods.

6.2. Reverse-engineering state machines

There have been several attempts to reverse-engineer state machines from software systems. Biermann and Feldman [28] developed a dynamic analysis technique that reverse-engineers machines with states that are points of control in the source code. Their *k-Tails* technique merges two states together if for both states the following *k* successor states are identical. Lorenzoli *et al.* [29] combine Bierman's algorithm with Daikon [8] (discussed above) as a basis for reverse-engineering state machines, where state transitions between the control states are annotated with the data conditions that govern them.

Kung *et al.* [6] have developed a technique that is based on symbolic execution to reverse-engineer state machines from an object-oriented system. Besides identifying possible state transitions, their method also includes a technique to infer possible abstract states from the symbolic path conditions. However, they do not consider abstract states that are composed of multiple variables.

Whaley *et al.* [30] describe an approach to extract component interfaces as state machines. As with the Bierman and Lorenzoli approaches, states are based entirely on the control/data flow of the system, as opposed to the variable values. They process a system in two steps, using static flow analysis to indicate potential states and transitions, and then confirming these by attempting to execute them.

Xie *et al.* have produced a substantial amount of work on reverse-engineering state machines of classes, using various analysis approaches. Their Obstra tool [31] uses dynamic analysis to reverse-engineer Object State Machines (OSMs). The abstract states of an OSM is described in terms of a set of observer methods of the class under analysis. Owing to the inherent problems of dynamic analysis discussed in the previous subsection, they extend their approach to use symbolic execution with the Symstra tool [32].

Dallmaier *et al.* [25] also use dynamic analysis to reverse-engineer state machines of objects. Their approach is motivated by the need to find 'meaningful' abstract states. To achieve this, they extract the state of an object via its observer methods.

The key difference between all of those techniques and the technique presented in this paper is the fact that the technique presented in this paper emphasizes on *how* a program reaches one state from another, as opposed to simply determining whether this is possible. Instead of simply indicating what the states are, and whether it is possible for one to reach the other, our technique specifies when and why states interact with each other as a transition function. A number of papers mentioned above do, however, employ more sophisticated means to abstract states (as embodied in our *getState* function), and it should certainly be noted that these could be combined with our approach. Nonetheless, it is this shift in emphasis from states to state transitions as the principle entities in a state machine that underpins our ongoing research into the use of more advanced state machine representations such as X-Machines [2] and Abstract State Machines [3,33].

6.3. Symbolic execution and program conditioning

There has been a substantial amount of progress in the field of software model checking and symbolic execution. Sophisticated model checkers such as JPF [18,19] and Bogor (with its Kiasan extension [34]) have made it possible to prove the feasibility of symbolic-execution-based techniques such as this one, which might have been considered too expensive and infeasible in the past. Bogor is particularly interesting, because it dispenses with the need to instrument the source code for symbolic execution. (This is currently also being implemented for the JPF symbolic executor.) There are a number of other similar projects (such as Microsoft's XRT model checker [35]), which similarly enable the symbolic execution of a variety of other languages.

A major problem that arises with model checkers is the fact that the number of possible program states to be explored rapidly becomes intractable with complex programs. A substantial amount of research is concerned with identifying strategies to reduce this by means of novel state abstraction and search techniques. These techniques have not been extensively explored with respect to this implementation, but provide a potent avenue for improving its scalability.

Program conditioning has become established as a means for specializing a program to increase the accuracy of static analysis techniques such as conditioned code slicing [11,12]. Conventionally, a program has been conditioned with respect to its inputs. Fox *et al.* [36] have proposed the use of 'backward conditioning' as a means to impose conditions on the values of variables at arbitrary points in the program. Each approach requires a set of conditions as well as a point where these conditions are true. (In conventional conditioning the point is implicitly the start of the program.) The technique presented in this paper is essentially a state-based approach to program conditioning. Once we have observed that state transitions can (usually) be tied to particular points in the source code, the developer can supply a means of identifying the state at a given point (the *getState* function), and the technique provides those statements that are responsible for each state transition.

7. CONCLUSIONS AND FUTURE WORK

This paper presents a technique for reverse-engineering transitions from source code. It uses program conditioning to identify branches of source code that are responsible for the execution of a particular transition. This allows a developer to determine which states of the system they are interested in, and our (largely automated) technique will state whether or not, and in what manner, these states are related to each other. This is particularly useful for software testing, documentation and comprehension.

The approach has been implemented as an extension of the JPF model checker as a proof of concept. The results demonstrate that the technique is feasible, but also show that the technique becomes very expensive as the subject program increases in complexity. This is largely due to an inherent weakness of symbolic execution—namely, that it becomes expensive in the presence of nested branches and loops.

There have been no significant attempts to improve its performance and scalability yet, but there are a number of readily available model-checking techniques that offer much potential in this area. A key problem with respect to poor performance in symbolic execution is the presence of unbounded and potentially infinite loops. To address these problems, future work will concentrate on the following areas:

1. Inferring loop invariants from symbolic executions. (This has already been investigated by Pasareanu and Visser [37].)
2. The use of loop-squashing program transformations to replace loops with simple conditionals. (This is based on the work by Hu *et al.* [38] with respect to amorphous program slicing.)

3. The use of slicing to restrict the program to what is relevant. (This has already been investigated by Dwyer *et al.* [39].)
4. Combining symbolic execution with dynamic analysis to reduce the symbolic search space. (This is influenced by Sen *et al.*'s work on Concolic execution [40].)

Currently, the approach has been used in conjunction with two notions of state: abstract data states that are supplied by the user and control states that simply correspond to transition points in the source code syntax. The benefit of the abstract states is that the resulting state machine is presented in the developer's terms; transitions are reverse-engineered at a useful level of abstraction, producing results that are easier to inspect. The downside, however, is that it requires a prior knowledge about the system and its potential states. The benefit of the control states is that they are easy to identify, but the downside is that the abstraction level of the resulting machine may be too low to be of much practical use to the developer.

Future work will investigate the use of different state abstractions. Prenninger and Pretschner [17] note that the desired type of abstraction depends on the ultimate application of the machine. Abstract data states of the type used previously to model the binary search tree are already useful for manual validation, because they present the system in the developer's terms. Ultimately, however, the aim is to reverse-engineer machines that form a suitable basis for rigorous model-based testing [2,41]. For this purpose, states will inevitably need to combine data and control elements to fully characterize the intended functional behaviour of the system.

ACKNOWLEDGEMENTS

The authors thank Phil McMinn and Qiang Guo, the anonymous STVR and TAIC PART referees, for their valuable comments on an earlier draft which have made this a much improved paper. We also thank Willem Visser and the members of the NASA Java PathFinder project for providing us with advice and support for developing our JPF extension, as well as Saswat Anand for contributing a preliminary version of the symbolic execution instrumenter. This work is supported by EPSRC grant EP/C511883/1.

REFERENCES

1. Lee D, Yannakakis M. Principles and methods of testing finite state machines—A survey. *Proceedings of the IEEE*, vol. 84, 1996; 1090–1126.
2. Holcombe M, Ipate F. *Correct Systems—Building a Business Process Solution* (*Applied Computing Series*). Springer: Berlin, 1998.
3. Börger E. Abstract state machines and high-level system design and analysis. *Theoretical Computer Science* 2005; **336**(2–3):205–207.
4. Krishnakumar AS. *Reachability and Recurrence in Extended Finite State Machines*: *Modular Vector Addition Systems*. *Computer-Aided Verification* (*Lecture Notes in Computer Science*, vol. 697), Courcoubetis C (ed.). Springer: Berlin, 1993; 110–122.
5. Chen F, Tillmann N, Schulte W. Discovering specifications. *Technical Report MSR-TR-2005-146*, Microsoft Research, Redmond, October 2005.
6. Kung D, Lu Y, Venugopalan N, Hsia P, Toyoshima Y, Chen C, Gao J. Object state testing and fault analysis for reliable software systems. *Proceedings of the 7th International Symposium on Software Reliability Engineering* (*ISSRE'96*), New York, U.S.A., 1996.
7. Yuan H, Xie T. Automatic extraction of abstract-object-state machines based on branch coverage. *Proceedings of the 1st International Workshop on Reverse Engineering to Requirements at WCRE 2005* (*RETR 2005*), Pittsburgh, U.S.A., November 2005; 5–11.

8. Ernst M, Cockrell J, Griswold W, Notkin D. Dynamically discovering likely program invariants to support program evolution. *Transactions on Software Engineering* 2001; **27**(2):1–25.
9. Gotel O, Finkelstein A. An analysis of the requirements traceability problem. *Proceedings of the First International Conference on Requirements Engineering*, Colorado, U.S.A., 1994.
10. Aho A, Sethi R, Ullman J. *Compilers*: *Principles*, *Techniques*, *and Tools*. Addison-Wesley: Reading, MA, 1986.
11. Canfora G, Climitile A, De Lucia A. Conditioned program slicing. *Information and Software Technology* 1998; **40**(11/12):595–607.
12. Danicic S, Daoudi M, Fox C, Harman M, Hierons R, Howroyd J, Ouarbya L, Ward M. ConSUS: A light-weight program conditioner. *Journal of Systems and Software* 2005; **77**(3):241–262.
13. Fox C, Danicic S, Harman M, Hierons R. CONSIT: A fully automated conditioned program slicer. *Software—Practice and Experience* 2004; **34**(1):15–46.
14. Coward PD. Symbolic execution and testing. *Information and Software Technology* 1991; **33**:53–64.
15. King J. Symbolic execution and program testing. *Communications of the ACM* 1976; **19**(7):385–394.
16. Lembeck C, Caballero R, Mueller R, Kuchen H. Constraint solving for generating glass-box test cases. *Proceedings of the International Workshop on Functional and* (*Constraint*) *Logic Programming* (*WFLP'04*), Aachen, Germany, 2004.
17. Prenninger W, Pretschner A. *Abstractions for Model-based Testing* (*Electronic Notes on Theoretical Computer Science*, vol. 116). Springer: Berlin, 2005; 59–71.
18. Java pathfinder. Available at: http://javapathfinder.sourceforge.net/ [10 October 2006].
19. Khurshid S, Pasareanu C, Visser W. Generalized symbolic execution for model checking and testing. *Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS'03*), Warsaw, Poland, 2003.
20. Vallee R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V. Soot—A Java bytecode optimisation framework. *Proceedings of Cascon '99*, Mississauga, Canada, 1999; 125–135.
21. Visser W, Pasareanu C, Khurshid S. Test input generation with Java PathFinder. *Proceedings of the International Symposium on Software Testing and Analysis* (*ISSTA'04*), Boston, U.S.A., 2004.
22. Walkinshaw N, Ali S, Bogdanov K, Holcombe M, Salahuddin S. Modelling and testing software with x-machines: A case study. *Technical Report*, Department of Computer Science, The University of Sheffield, 2007.
23. Fagan M. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 1976; **15**(3): 182–211.
24. Chow T. Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering* 1978; **4**(3):178–187.
25. Dallmeier V, Lindig C, Wasylkowski A, Zeller A. Mining object behaviour with adabu. *Proceedings of the International Workshop on Dynamic Analysis* (*WODA'06*), Shanghai, China, 2006.
26. Henkel J, Diwan A. *Discovering Algebraic Specifications from Java Classes* (*Lecture Notes in Computer Science*, vol. 2743). Springer: Berlin, 2003; 431–456.
27. Tillmann N, Chen F, Schulte W. Discovering likely method specifications. *International Conference on Formal Engineering Methods* (*Lecture Notes in Computer Science*, vol. 4260). Springer: Berlin, 2006; 717–736.
28. Biermann AW, Feldman J. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers* 1972; **21**:592–597.
29. Lorenzoli D, Mariani L, Pezze M. Inferring state-based behavior models. *Proceedings of the International Workshop on Dynamic Analysis* (*WODA'06*), Shanghai, China, 2006.
30. Whaley J, Martin M, Lam M. Automatic extraction of object-oriented component interfaces. *Proceedings of the International Symposium on Software Testing and Analysis*, Rome, Italy, July 2002.
31. Xie T, Notkin D. Automatic extraction of object-oriented observer abstractions from unit-test executions. *International Conference on Formal Engineering Methods* (*Lecture Notes in Computer Science*, vol. 3308), Davies J, Schulte W, Barnett M (eds.). Springer: Berlin, 2004; 290–305.
32. Xie T, Marinov D, Schulte W, Notkin D. Symstra: A framework for generating object-oriented unit tests using symbolic execution. *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS 05*), Edinburgh (*Lecture Notes in Computer Science*, vol. 3440), Halbwachs N, Zuck LD (eds.). Springer: Berlin, April 2005; 365–381.
33. Gurevich Y. Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic* 2000; **1**(1):77–111.
34. Deng X, Lee J, Robby. Bogor/Kiasan: A *k*-bounded symbolic execution for checking strong heap properties of open systems. *Automated Software Engineering*. IEEE Computer Society Press: Silver Spring, MD, 2006; 157–166.
35. Grieskamp W, Tillmann N, Schulte W. XRT—exploring runtime for NET architecture and applications. *Workshop on Software Model Checking* (*SoftMC'2005*), Edinburgh, U.K. (*Electronic Notes on Theoretical Computer Science*, vol. 144), Cook B, Stoller S, Visser W (eds.). Springer: Berlin, 2006; 3–26.
36. Fox C, Harman M, Hierons R, Danicic S. Backward conditioning: A new program specialisation technique and its application to program comprehension. *Proceedings of the International Workshop on Program Comprehension* (*IWPC'01*), Toronto, Canada, 2001; 89–97.

37. Pasareanu C, Visser W. *Verification of Java Programs Using Symbolic Execution and Invariant Generation* (*Lecture Notes in Computer Science*, vol. 2989). Springer: Berlin, 2004; 164–181.
38. Hu L, Harman M, Hierons R, Binkley D. Loop squashing transformations for amorphous slicing. *Working Conference on Reverse Engineering* (*WCRE'04*). IEEE Computer Society Press: Silver Spring, MD, 2004; 152–160.
39. Dwyer M, Hatcliff J, Hoosier M, Ranganath V, Robby, Wallentine T. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. *Proceedings of the Twelfth International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS'06*), Vienna, Austria, 2006.
40. Sen K, Marinov D, Agha G. CUTE: A concolic unit testing engine for C. *Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*ESEC/SIGSOFT FSE'05*), Lisbon, Portugal, 2005; 263–272.
41. Bogdanov K, Holcombe M, Ipate F, Seed L, Vanak S. Testing methods for X-Machines: A review. *Formal Aspects of Computer Science* 2006; **18**:3–30.