Identifying State Transitions and their Functions in Source Code

Neil Walkinshaw, Kirill Bogdanov, Mike Holcombe

Department of Computer Science, The University of Sheffield
Regent Court, 211 Portobello Street,
S1 4DP, Sheffield, UK
E-mail: {n.walkinshaw,k.bogdanov,m.holcombe}@dcs.shef.ac.uk

Abstract

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 precisely identifies the path(s) through the source code that govern a transition. The technique has been implemented as a prototype, enabling a preliminary evaluation of our technique with respect to real software systems.

1. Introduction

Perceiving software as a state machine enables the developer to design, document and rigorously test a program in terms of its behaviour. The system is decomposed into a set of states, where each state characterises 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 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 [13].

Despite being useful for a variety of development tasks, conventional state-based software modelling techniques fail to provide insights into *how* and *why* state transitions take place. This can render them difficult to read and understand [19], making it difficult to validate any test sets that are generated from them. This has led to the development of more expressive state-based modelling techniques such as X-Machines [8] and Abstract State Machines [2]. These introduce the notion of state *transition functions*, which enable the developer to 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. Determining state transition functions when specifying the system in the first place is straightforward because they can be obtained via functional decomposition.

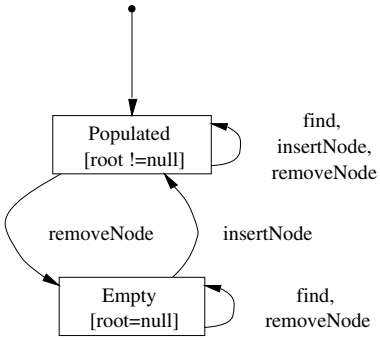

Figure 1. State machine for a simple Binary Search Tree

To illustrate some of the problems that arise with conventional state-based specifications, figure 1 contains a simple state machine for a binary search tree. Transitions are annotated with the inputs that trigger them. 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 the transition $Empty \overset{insertNode}{\rightarrow} Populated$ is missing, where is the fault?"

Conventional state-based techniques fail to answer these questions because they can only identify the states and *whether* state transitions take place. They fail to provide

Proceedings of the Testing: Academic & Industrial Conference – Practice And Research Techniques (TAIC PART'06)
0-7695-2672-1/06 $20.00 © 2006 IEEE

any insights into *how* and *why* the system transitions from one state to the other.

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 [13, 8]. This has spurred the development of several approaches to reverse engineer state-based specifications. Although existing reverse-engineering techniques can suggest states [6, 4], or detect whether a state transition can take place [12, 18], 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 all of the state transitions with respect to a set of states, but also annotates each transition with the relevant source code. The source code for a state transition is referred to as the 'state transition function'. Providing this information is a useful tool for state-based software engineering. It makes explicit links between the design and the source code, thus aiding the traceability of requirements to the implementation and helping the user to answer questions such as those presented above. This makes the technique particularly useful for understanding, inspecting and testing the dynamics of complex extended state machines (particularly X-Machines and Abstract State Machines). The resulting state transition functions are particularly useful for comprehension of both design and source code, state-based testing and debugging, and inspecting source code from the behavioural perspective.

The 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.

Section 2 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, followed by details on our implementation, along with an illustrative example that shows how to answer the questions about the binary search tree presented above. Section 5 evaluates the scalability and performance of our approach. Section 6 provides an overview of related work and section 7 discusses the future work and conclusions.

```
    int x,y;
1   if(x>y){
2    x=x+y;
3    y=x-y;
4    x=x-y;
5    if(x>y)
6     assert false;
    }
```

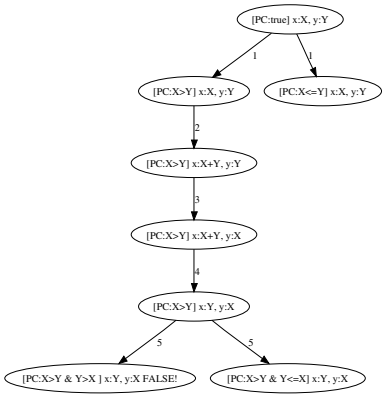

Figure 2. Symbolic execution tree

2. Program Conditioning and Symbolic Execution

Program conditioning forms an important part of our technique (which is presented in the next section). It is used to identify the source code that governs the execution of a given state transition. This section presents an overview of program conditioning and symbolic execution.

Danicic *et al.* [5] succinctly 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"*. Using their approach, the user can specify conditions in the program in the form of assertions. Program conditioning removes any statements that cannot be executed when these assertions are true.

A program conditioner determines a set of feasible paths that satisfy user-defined conditions by symbolic execution [11]. When a program is symbolically executed, its input values are substituted with symbolic values. As the program is symbolically executed, its internal variables are manipulated as symbolic expressions instead of concrete values. The outputs of a symbolic execution are expressed as a function of the symbolic values that replaced input values. The state of a symbolic execution consists of a path condition, current symbolic values, and a path counter.

Symbolically executed paths through a program can be summarised in a symbolic execution tree, where each leaf node represents the termination of a potential path through the program. An example of a symbolic execution tree for a program that swaps two integers (taken from Khurshid *et al.* [10]) is shown in figure 2. In the initial state the path condition is set unconditionally to `true`, indicating that it is always feasible. In the following branches however, the path condition reflects the condition in the if statement (so `x>y` or $x \leq y$). A state can be deemed infeasible if its path condition cannot be satisfied, as is the case for statement 6.

Proceedings of the Testing: Academic & Industrial Conference – Practice And Research Techniques (TAIC PART'06)
0-7695-2672-1/06 $20.00 © 2006 IEEE

Note also that the symbolic variables are recorded in terms of their initial/input symbolic variables, hence the symbolic value for variable Y at branch 3 is simply X as opposed to X-Y (from the previous statement).

As the path conditions are generated by the symbolic executor the program is conditioned by employing a theorem prover to reason about the conditions and backtracking as soon as a path becomes infeasible. The potential for the program conditioner to limit the size of the program depends on the capabilities of the underlying theorem prover. These are often limited to reasoning about linear constraints (as is currently the case with our implementation), although there has been work on combining symbolic executors with a selection of solvers fitting particular types of problems [14].

Besides the problems that are introduced by non-linear constraints, another important limitation on symbolic execution is its inability to effectively handle loops. A loop always has to have a concrete upper-bound, otherwise it can result in an infinite execution tree. Current techniques tend to simply apply an artificial limit on the number of times a loop can be artificially executed, but more advanced techniques (e.g. inferring loop invariants as the program is symbolically executed [15]) are being investigated.

3. Discovering State Transitions and their Functions

The key contribution of this paper is a technique that, given a set of program states (in terms of conditions on the program variables), returns the set of all possible state transitions in the program, along with the paths through the source code that are responsible for each state transition. Our technique is based on the observation that the inputs that trigger state transitions (i.e. method calls, exceptions etc.) usually map directly to particular statement types in the source code. Thus a state transition function can be interpreted as the source code that is executed between a pair of transition trigger statements.

This section provides an overview of our approach. Our process of recovering a set of state transitions from an implementation follows the following six steps (Steps 4,5 and 6 are illustrated by figure 3):

1. Identify transition triggers (e.g. method calls, exceptions etc.)
2. Identify appropriate abstract states (in terms of conditions on state variables)
3. Map transition triggers to their respective statements in the source code
4. Construct the symbolic execution tree, marking transition trigger points as they are encountered

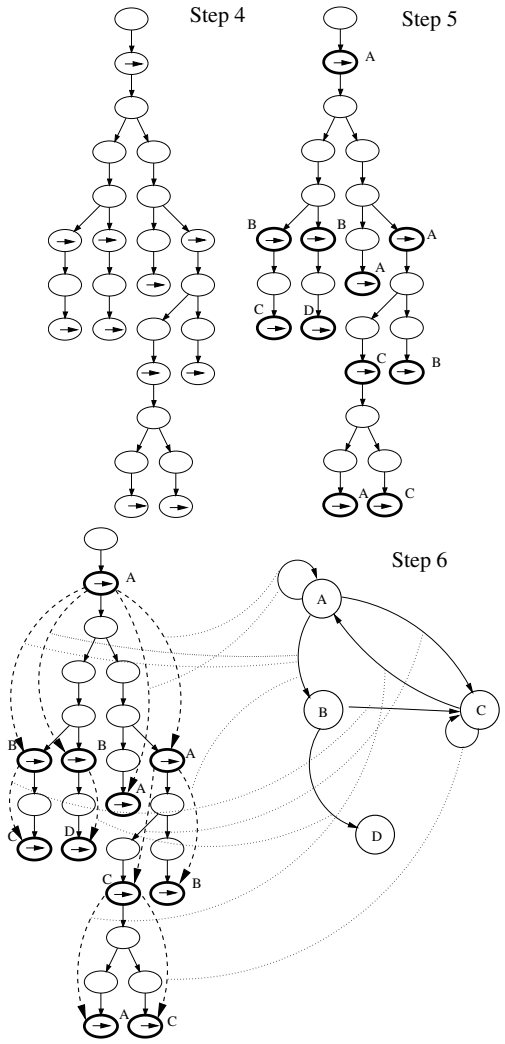

Figure 3. Identifying state transitions

5. For each marked symbolic execution state identify its corresponding abstract state
6. 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 trigger point) and end (a subsequent trigger point), we establish the current abstract state every time a trigger point is (symbolically) executed. The interval between a pair of trigger points thus corresponds to a state

0-7695-2672-1/06 $20.00 © 2006 IEEE

transition function. To become practical however, several important issues need to be taken into account. Section 3.1 considers the problem of expressing abstract states as conditions on symbolic variables. Section 3.2 discusses the identification of statements that encapsulate potential state transitions - the main contribution of this work. Section 3.3 shows how symbolic execution can be used to identify state transitions and their functions.

3.1. Identifying States as Conditions on Symbolic Variables

To ensure that the final state transitions are reverse-engineered at a suitable level of abstraction, our technique currently requires the developer to supply the relevant states. These states have to be provided in the form of expressions of the program's symbolic variables. Each state should be expressed as a quantifier free boolean formula over the state variables, so that it can be derived from symbolic expressions at the transition trigger points.

There are two options to identify the states of a system. They can be identified manually by the developer, which enables the precise specification of the developer's personal state-based perception of the program. Another option would be to automate the identification of potential states via existing automated approaches (e.g. by inferring axiomatic specifications [6, 4]).

3.2. Identifying Statements that Trigger Potential State Transitions

Our technique is primarily motivated by the observation that, for most state-based software modelling approaches, it is straightforward to map state transitions in a state machine to particular statements in the source code. The type of statements that trigger a potential state transition depends on the state-based model that is employed. Identifying them is however so straightforward that this step can (in most cases) be fully automated. As an example, in UML state charts transitions are triggered by calls to methods. Because a state change occurs as the result of a method call, we want to know how that method affects the state of the system; we want to observe the state of the system once the method has executed. Consequently, for now, the statements that trigger the state transitions can simply be interpreted as the last statement of every method body.

Ultimately however we aim to use our technique to construct a more elaborate state-based model (Bogdanov's object machine model [1]), which is particularly useful for testing the responses of an object to various types of triggers. Bogdanov's model is designed to facilitate the thorough state-based testing of communication between objects. It extends the UML model described above by increasing the number of potential triggers that can affect the behaviour of an object. Besides method calls, the behaviour of objects his state-based model also allows for inputs via thrown exceptions (i.e. if a collaborating object throws an exception it may activate some `catch` clause in the object). The extended model also allows for the variation of behaviour according to the values returned by collaborator objects (e.g. if a boolean accessor method is called in a collaborating object, our object may vary its behaviour depending on whether a `true` or a `false` value is returned). If we adopt Bogdanov's more elaborate model, the process of extending the set of statement types that trigger state transitions is straightforward. If a transition is caused by a method call, the trigger of the transition is still the last statement of every method, if the cause is a thrown exception, the trigger is the last statement of every catch clause, and if the cause is a return from a method call, the trigger statement is simply the statement after every method call. Although Bogdanov's model is more complicated than the conventional state-based object model, these additional statements can all be identified automatically by analysing the control structure of the code.

Although we are adopting the straightforward conventional state-based object model in this paper, we want to stress the flexibility of this technique for adopting more complicated models. As long as state transition triggers can be mapped to specific control structures in the source code (e.g. method calls or catch blocks), extending our technique is a relatively straightforward task. 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.

3.3. Identifying State Transitions by Symbolic Execution

For this step we use the set of 'trigger' statements identified previously, along with the set of abstract states, to determine which state transitions are feasible. For each feasible transition we also obtain the source code that is responsible for its execution via program conditioning. The steps that are described in this subsection correspond to steps 4, 5 and 6 (see figure 3). We use the symbolic execution tree to ascertain the following three properties: (1) potential sequences of state transition trigger points, (2) the abstract states at these transition points and (3) the source code that governs a state transition between a pair of transition points.

First the trigger statements that have been highlighted in the previous step have to be mapped to nodes in the symbolic execution tree (step 4 in figure 3). This may be a one-to-many mapping, as a single statement may be executed multiple times. We assume that this mapping can be established via the symbolic executor (i.e. it is possible to

determine from within the symbolic executor whether the current statement corresponds to a trigger statement such as a method call). Once this is done, we use the symbolic variable values at every marked symbolic state s in the symbolic execution tree to identify its respective abstract state $A(s)$ as supplied by the developer (step 5 in figure 3). Using the annotated symbolic execution tree, we can now determine which states occur in sequence. This is accomplished by traversing every possible path in the tree, identifying every consecutive pair of trigger states as they are encountered (step 6 in figure 3). For every pair of consecutive trigger states (s, t), we store the state transition $A(s) \rightarrow A(t)$.

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 state transition 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, the statements can be readily identified by simply traversing the symbolic states between a pair of trigger points. If there is a branch statement and both branches end up in the same state, this can result in two different state transition functions for the same transition (each function corresponding to a branch). These can simply be unified as a single function.

The above process of identifying transition functions is akin to program conditioning (see section 2). Program conditioning conventionally involves the manual insertion of `assert` statements, which contain conditions on program variables at specific points. Our approach differs from conventional program conditioning. Instead of specifying a single program point and a single set of variable constraints, we supply multiple program points (transition trigger statements) and multiple sets of variable constraints (program states). Instead of producing a single constrained path through the source code, our approach produces multiple paths, which serve to indicate how the program behaves in terms of the criterion states.

4. Implementation and Example

To establish the feasibility of the approach presented in the previous section, it was implemented as a small extension to the Java PathFinder model checker. The implementation details are contained in the following subsection. The example in section 4.2 illustrates how our extension can be used to reverse-engineer the state machine for a binary search tree implementation in Java. Section 4.3 illustrates how the information provided by our approach is useful for software comprehension and debugging.

4.1. Implementation

Our implementation is designed to compute the possible state transitions of an object in Java (the approach itself is not restricted to this level of abstraction). It is built on top of the Java PathFinder model checker and its symbolic execution extension [10]. The model checker enables us to exhaustively execute the program (in practice we restrict the Java PathFinder to considering call strings of a limited length). The symbolic execution extension ensures that the program executions as executed by the model checker are feasible and also enables us to identify those statements that control a given state transition (i.e. they belong to the state transition function).

The implementation currently requires the class under analysis to be instrumented in two steps (this is relatively straightforward and can be largely automated): First, an abstract class (`StateBasedObject`) is inserted so that it is extended by the class under analysis. `StateBasedObject` contains the logic that is necessary to derive the abstract state from the current symbolic variables. Second, each statement that corresponds to a transition point in the class under analysis is instrumented so that it returns the abstract state at that point to our tool. To use the symbolic execution extension to Java PathFinder, the program has to be instrumented further by using a standard set of automatable transformations [10]. The instrumentation process is elaborated in the example in section 4.2.

Section 3 conceptually described our technique as using an existing symbolic execution tree to compute state transitions. This separated the process of identifying the transitions from computing the tree, which is implemented by Java PathFinder and is beyond the scope of this paper. In our implementation the transitions are discovered 'on the fly', as the symbolic tree is constructed.

PathFinder provides several classes that make it possible to monitor the process of tracing along each execution path. As the model checker searches through the set of possible program executions, every time a search event occurs (e.g. a new state is discovered, or it backtracks etc.), it notifies any observers. Our approach has been implemented as a search observer. Every time a new (symbolic) state is encountered, our observer checks whether a trigger point has been encountered (i.e. whether it is instrumented to feed back the current abstract state to our tool). If this is the case its abstract state is read from the annotated symbolic state, state transitions are added from the previous abstract state to the new state, and the new abstract state is stored until a newer abstract state is encountered, where the process is repeated.

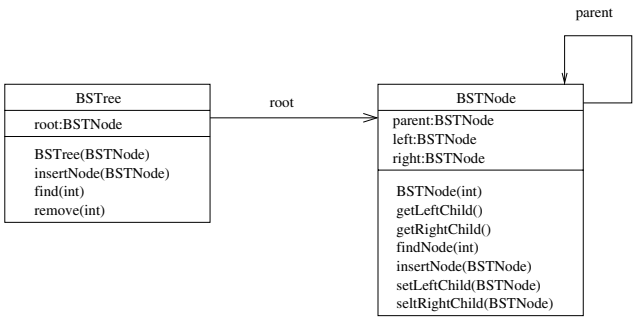

Figure 4. Binary search tree classes

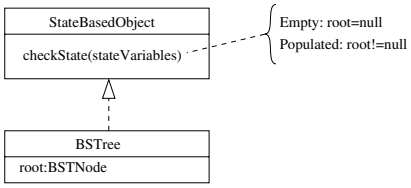

Figure 5. StateBasedObject contains logic to detect the current abstract state of `BSTree`

4.2. Obtaining Transitions and Transition Functions for a Binary Search Tree

Now that we have described the technique and its implementation, we will demonstrate it on the binary search tree example[1] that was used in the introduction. 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).

First we must determine the set of states that are of interest. For this example we suppose that the developer is interested in the system simply in terms of whether it is empty or populated. These must be expressed as conditions on the state variables for the object (its attributes) as shown in figure 5. The `checkState` method implements these conditions and returns a string that gives a name to the current state.

Having established the logic that is responsible for identifying states, we identify the trigger points in the source code for potential state transitions. For this example we assume that state transitions are simply triggered by method calls. Because we are interested in the state of the system once a method has been executed (i.e. how a trigger affects the state of the system) the trigger points are

[1]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

```
1: public SymBSTNode find(Expression val){
2:  if(root != null){
3:   SymBSTNode result = root.findNode(val);
4:   Verify.setAnnotation("find(val) -"+
         checkState(root));
5:   return result;
6:  }
7:  Verify.setAnnotation("find(val) -"+
         checkState(root));
8:  return null;
9: }
```

Figure 6. Example of the fully instrumented `BSTree.find(int)` method

the last nodes of the control flow graph for each method. At the trigger points we need to relay the current abstract state to the model checker. To establish the state, a call to `StateBasedObject.checkState` is inserted, sending the current symbolic values as parameters. The state is returned as a string, which we use as a label for the current symbolic state by passing it to our tool via PathFinder's `Verify` class.

Figure 6 shows an example of the `BSTree.find(int)` method. It is instrumented for the symbolic executor (for example, it takes an argument of type Expression instead of int). The details of the instrumentation required for symbolic execution in Java PathFinder are provided by Visser *et al.* [10]. The most important instrumentations with respect to this work are statements 4 and 7. These are the last control points in the method (the trigger points) and serve to send the current state to the model checker (via the `Verify` class). As an example, if the destination of the last encountered state transition was the $Populated$ state, and abstract state at the new trigger point is annotated as $(Empty, removeNode)$, the resulting transition is $Populated \overset{removeNode}{\rightarrow} Empty$.

Now that the first two stages (identifying states and identifying state transition points) have been completed, the program must be symbolically executed so that the state transitions can be identified. This involves symbolically executing every possible permutation of calls to `BSTree.find`, `BSTree.insert` and `BSTree.remove` (up to a specified limit). This can be achieved with the model checker (the approach is documented by Visser *et al.* [16]), 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. A single pair of transition points corresponds to a single path in the transition function. If the transition function contains a branch statement, the complete transition function is the union of multiple paths between the multiple symbolic states that correspond to the same abstract states. In our implementation we map each transition

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

BSTree.find(int val)

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

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

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;
}
```

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

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

function to a set of basic blocks in the source code. Each time a given state transition is encountered in the symbolic execution tree, its transition function is united with any additional basic blocks that have been identified by other pairs of trigger points that correspond to the same (abstract) state transition.

4.3. Understanding the State Transitions of the Binary Search Tree

In the introduction we noted that although existing state-based reverse engineering techniques can identify the states of a system, they provide little information about the nature of the state transitions. Therefore they provide little information about the behaviour of the system as a whole. Our technique provides this behavioural information in terms of the state transition functions.

This subsection demonstrates how the state transition functions by our technique are useful for understanding and debugging software from a state-based perspective. We demonstrate this by answering the two questions about the behaviour of the binary search tree from the introduction. To recap, the two questions were:

"How does the transition $Empty \overset{find}{\rightarrow} Empty$ differ from $Populated \overset{find}{\rightarrow} Populated$?"

and

"Tests indicate that the transition $Empty \overset{insertNode}{\rightarrow} Populated$ is missing, where is the fault?"

Here we demonstrate how transition functions help to answer these questions. The relevant functions are all presented in figure 7.

Answering the first question, the source code in (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 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. We discovered it because the state transitions that were reverse engineered did not match the transitions we predicted (see figure 1). In the $Empty$ state, the $insert$ state transition remained in the $Empty$ state instead of the $Populated$ state. To investigate why this is the case, we can analyse the transition function that is triggered by calling $insertNode$ in the $Empty$ state. By analysing the $Empty \overset{insertNode}{\rightarrow} Empty$ transition function in (c), the reason 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. The programmer has forgotten to insert 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 our approach is particularly useful for debugging by improving traceability between the design and implementation.

5. Evaluation

The previous section shows how we implemented the approach and used it to reverse engineer the state transitions for a binary search tree. This demonstrated that the technique is feasible. However as the approach relies on symbolic execution, which is generally perceived to be expensive, the evaluation in this section is intended to provide some insight into the performance of our technique. The

	Methods	Complexity	MethLOC
SimpleStack	push, pop	1.75	19
LinkedList	add, get, remove, size	1.1846 (6 loops)	61
BSTree	insertNode, removeNode, removeMax, find	3.294	223

Figure 8. Systems with the mean cyclomatic complexity per method and MethLOC

results in this section are preliminary, and can only serve as an *indicator* of the technique's performance and scalability. Our future work (section 7) will involve a more comprehensive evaluation that will also evaluate other factors, such as the soundness of the technique and the practical applicability of the technique to software testing and comprehension.

There are two main factors that impact on the scalability and performance of our approach. The most significant factor is the size of the symbolic execution tree; if the subject system is complex, with a large number of branches, the symbolic execution tree will be large. This in turn affects the amount of time our approach takes to traverse the tree. A further important factor is the number of times a trigger point is encountered in the tree. Each time such a point is encountered, the conditioned segment of the program has to be extracted as part of the transition's function.

To obtain an idea of how these two factors impact our technique, we have selected three small programs. These are listed in figure 8 along with their cyclomatic complexity and the number of non-comment / non-blank lines of code belonging to methods (MethLOC[2]). The cyclomatic complexity metric lists the average number of individual paths through each method. This provides a crude indicator[3] of the expense of the symbolic execution, which has to take every permutation of paths through all of the methods into account. `SimpleStack` is the system that was introduced in the previous section. `LinkedList` is particularly interesting because of its extensive use of `for` and `while` loops, which tend to be particularly challenging for symbolic execution. `BSTree` is an implementation of a binary search tree (a corrected version of the one considered in the previous section) and is larger than the other two, containing many nested branches.

Results were collected by symbolically executing the systems with method sequences of length N, where N ranged from 1 to 4. For every value N the symbolic executor would exhaustively explore every possible permutation of N method calls. We measured the amount of time required, the number of states in the symbolic execution tree, the amount of memory consumed and the number of pairs of trigger nodes that were encountered in the symbolic execution tree. These results are presented in figure 9. A logarithmic scale is used to make it easier to distinguish between the lower values.

The results indicate that the space and time required by our approach increases exponentially as N increases. For complex systems with multiple branches and loops our approach rapidly becomes unpractical for long method sequences. It should be noted that no significant attempts have been made to optimise the performance of our tool in any way. A significant part of our future work is to investigate ways to limit the expense incurred by the symbolic execution (see section 7).

6. Related Work

Section 6.1 puts this work into the context of existing approaches to reverse engineer state-based specifications. Section 6.2 discusses this technique in the context of program conditioning and symbolic execution.

6.1. Reverse-Engineering State-Based Specifications

There exist several approaches to reverse engineering invariants and axiomatic specifications from software. They provide (potential) software states that may be of interest to the developer. These states can then be used as input states for our technique, which discovers how they are related to each other. For this reason the technique that we presented in this paper complements these techniques.

Ernst *et al.* [6] have proposed a dynamic approach for reverse engineering axiomatic specifications. Chen *et al.* [4] note that dynamic approaches are, by virtue of being dynamic, only representative of the set of program executions they are based upon. They propose a technique to derive axiomatic specifications via symbolic execution instead. A single symbol represents all of the possible values of the (concrete) variable it represents in the program, which makes it suitable for summarising multiple program executions. As a result they claim that the results produced by their technique are more sound than other dynamic approaches.

There have been few attempts to reverse-engineer state machines from software (i.e. state transitions in addition to the states themselves). Kung *et al.* [12] developed a

[2]Metrics were collected by using the Metrics Eclipse plugin (http://metrics.sourceforge.net/).

[3]Note that, as it is based upon the control flow graph, it does not distinguish loop predicates from conventional `if` statements and therefore makes the (false) assumption that each loop is executed only once.

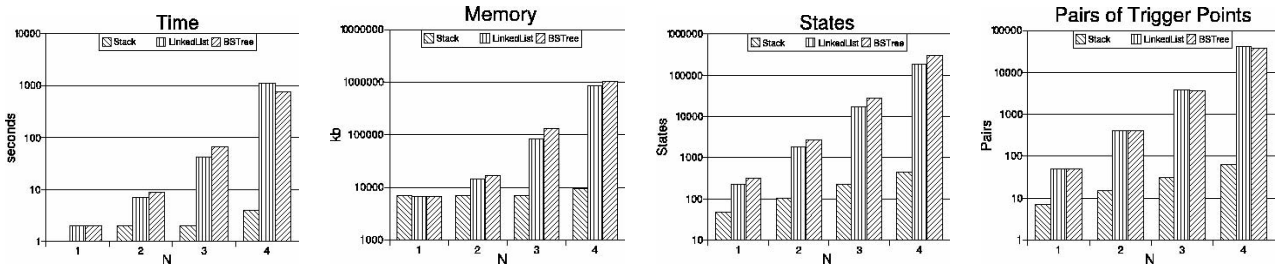

Figure 9. Results from Performance and Scalability Study (logarithmic scale)

technique that is based on symbolic execution to reverse-engineer state machines from an object-oriented system. They provide an automated technique to identifying the system states. They do not consider abstract states that are composed of multiple variables. Whaley *et al.* [17] describe a system to extract component interfaces as state machines to drive testing. They take a more control-centric view of the state of an object and describe it in terms of the method being executed as opposed to the variable values. Yuan and Xie [18] describe a dynamic technique to reverse engineer an object-state machine. They then use an 'abstraction function' to reduce the complexity of the resulting state machine, by mapping multiple concrete states to a sensible set of abstract states.

The key difference between all of those techniques and the technique that is presented in this paper is the fact that our technique considers *how the states are related*. 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. 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-based representations such as X-Machines [8] and Abstract State Machines [2].

6.2. State-Based Program Conditioning

Program conditioning has become established as a means for specialising a program to increase the accuracy of static analysis techniques such as conditioned code slicing [3, 5]. Conventionally a program has been conditioned with respect to its inputs. Fox *et al.* [7] have more recently 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).

Our technique is 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 user no longer needs to supply the point(s) in the source code that are of interest. All they have to do is supply the abstract states. This is particularly beneficial when the user is not intricately familiar with the system at a source code level.

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.

We have implemented our technique as an extension of the Java PathFinder model checker, and used it to gather some preliminary data about its performance. The results demonstrate that our technique is feasible, but also show that the technique becomes very expensive as the subject program increases in complexity. This is largely down to an inherent weakness of symbolic execution; namely that it becomes expensive in the presence of nested branches and loops.

We have not seriously attempted to optimise the performance of our tool yet and believe that there is 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. There are three approaches that we aim to investigate:

1. Inferring loop invariants from symbolic executions (this has already been investigated by Pasareanu *et al.* [15]).

2. The use of loop squashing program transformations to replace loops with simple conditionals (this is based on work by Hu *et al.* [9] with respect to amorphous program slicing).

3. The use of slicing to restrict the program to what is relevant.

Currently the approach presented in this paper relies on the existence of a suitable set of system states. The benefit in this 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 the developer requires prior knowledge about the system and its potential states. We will investigate the automation of state discovery, thus minimising the requirement for prior system knowledge. We will investigate the combination of our technique with existing state discovery tools, such as Daikon [6].

We need to investigate the soundness and precision of the transitions produced by this technique, especially if we use techniques such as invariant detection to impose upper bounds on loops. If the upper bounds set on loops is not sufficiently high, the model can omit state transitions. If on the other hand the symbolic execution considers too many infeasible paths (e.g. because of the inability to handle non-linear proofs), the model can end up being too large, compromising the integrity of a resulting test set.

Ultimately we aim to use our technique as part of a larger framework that enables the developer to use this technique as a basis for reverse-engineering a state-based view of the system at an arbitrary level of abstraction. In line with the X-Machine method [8], state transition functions can themselves be expressed as X-Machines. This results in a functional hierarchy for the entire system, which can be subjected to established state-based testing techniques.

Acknowledgements We thank Phil McMinn and Qiang Guo for their valuable comments on an earlier draft. We also thank Willem Visser for kindly providing us with the JPF symbolic executor and permitting the use of the symbolic execution tree example in section 2. This work is supported by EPSRC grant EP/C511883/1.

References

[1] K. Bogdanov. Testing from Object Machines in Practice. In *Proceedings of UKTEST'05*, Sheffield, 2005.

[2] E. Börger. Abstract state machines and high-level system design and analysis. *Theor. Comput. Sci*, 336(2-3):205–207, 2005.

[3] G. Canfora, A. Climitile, and A. D. Lucia. Conditioned Program Slicing. *Information and Software Technology*, 40(11/12):595–607, 1998.

[4] F. Chen, N. Tillmann, and W. Schulte. Discovering Specifications. Technical Report MSR-TR-2005-146, Microsoft Research, Redmond, October 2005.

[5] S. Danicic, M. Daoudi, C. Fox, M. Harman, R. Hierons, J. Howroyd, L. Ouarbya, and M. Ward. ConSUS: a lightweight program conditioner. *Journal of Systems and Software*, 77(3):241–262, 2005.

[6] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *Transactions on Software Engineering*, 27(2):1–25, Feb. 2001.

[7] C. Fox, M. Harman, R. Hierons, and S. Danicic. Backward Conditioning: A new Program Specialisation Technique and its Application to Program Comprehension. In *Proceedings of the International Workshop on Program Comprenhesion (IWPC'01)*, pages 89–97, Toronto, Canada, 2001.

[8] M. Holcombe and F. Ipate. *Correct Systems - Building A Business Process Solution*. Applied Computing Series. Springer, 1998.

[9] L. Hu, M. Harman, R. M. Hierons, and D. Binkley. Loop squashing transformations for amorphous slicing. In *Working Conference on Reverse Engineering (WCRE'04)*, pages 152–160, 2004.

[10] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, Warsaw, Poland, 2003.

[11] J. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7), 1976.

[12] D. Kung, Y. Lu, N. Venugopalan, P. Hsia, Y. Toyoshima, C. Chen, and J. Gao. Object State Testing and Fault Analysis for Reliable Software Systems. In *Proceedings of the 7th International Symposium on Software Reliability Engineering (ISSRE'96)*, 1996.

[13] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines - A Survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.

[14] R. Mueller, C. Lembeck, and H. Kuchen. A Symbolic Java Virtual Machine for Test Case Generation. In *Proceedings of the IASTED International Conference on Software Engineering*, 2004.

[15] C. Pasareanu and W. Visser. Verification of Java Programs Using Symbolic Execution and Invariant Generation. *Lecture Notes in Computer Science*, 2989:164–181, 2004.

[16] W. Visser, C. Pasareanu, and S. Khurshid. Test Input Generation with Java PathFinder. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'04)*, 2004.

[17] J. Whaley, M. Martin, and M. Lam. Automatic Extraction of Object-Oriented Component Interfaces. In *Proceedings of the International Symposium on Software Testing and Analysis*, July 2002.

[18] H. Yuan and T. Xie. Automatic Extraction of Abstract-Object-State Machines Based on Branch Coverage. In *Proceedings of the 1st International Workshop on Reverse Engineering To Requirements at WCRE 2005 (RETR 2005)*, pages 5–11, November 2005.

[19] M. K. Zimmerman, K. Lundqvist, and N. Leveson. Investigating the readability of state-based formal requirements specification languages. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 33–46, New York, May 19–25 2002. ACM Press.