VERIFICATION OF ARCHITECTURAL CONSTRAINTS ON INTERACTION PROTOCOLS AMONG MODULES

by

Colin Stuart Siroky, B.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Software Engineering
May 2015

Committee Members:

Rodion Podorozhny, Chair

Guowei Yang

Anne Ngu

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

ACKNOWLEDGMENTS

I would like to thank Dr. Podorozhny for his help and direction on this research. I would also like to thank Dr. Yang for his input and advice.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

ABSTRACT

The importance of correspondence between the architectural prescription and implementation has been long recognized. This thesis presents an approach to verification of constraints on method invocation chains prescribed by an architectural style. It consists of two key steps. One, static slicing is applied to the code from a given final method in the system. The resulting slice information is then used to create a smaller executable program by mocking out the methods that are not contained in the slice. For the second step, symbolic execution is used for the verification and the application of architecturally defined constraints. We implement our approach in a prototype based on Wala, Javassist and Symbolic PathFinder (SPF), and demonstrate the usefulness of our approach using case studies.

1. INTRODUCTION

The notion of software architecture has been defined by Perry and Wolf as a set of constraints on components, form and rationale (Perry & Wolf, Oct 1992). The importance of adhering to an adopted architectural style throughout software development and maintenance has been recognized. It helps avoiding architectural erosion and drift, thus making sure that the chosen architectural style still provides its benefits and ensures correspondence to requirements.

A number of formal architectural description languages (ADL) have appeared over the years. For instance, Wright (Allen, January 1997), is an example of ADL with an emphasis on specification of communication protocols between modules as part of abstract behavior specification of components and connectors. Further work in the area of software architecture paid attention to automation of checking correspondence between the architectural prescription and implementation. The Arch-Java tool (Aldrich, Chambers, & Notkin, 2002) can serve as an example in this direction. The tool and associated ADL allows for definition of component ports and connectors and type checking of combinations of ports and connectors. It also allows for checking correspondence of a given implementation against an architectural specification. This work suggests an approach to checking correspondence of architectural constraints on sequences of method invocations, i.e. communication protocols involving more than two modules. For example, constraints of this kind are defined in a popular Model-View-Controller (MVC) architectural style (Hunt) (Krasner & Pope, Aug. 1988).

The suggested approach is to try and reduce the problem size so that the method of verification is more scalable and time efficient. The reduction is done via slicing. Once a slice has been determined for a final point of interest, a final Java method, along some path of interest, the code is mocked to create an executable program slice. Once the code has been mocked to create a sliced version of the code it can be run using symbolic execution (Clarke, 1976) (King, 1976). The symbolic execution traversal checks if there are feasible paths that will break the constraints on legal method invocation sequences and builds path conditions to allow for test case generation along the legal invocation chains. Algorithmically, it is similar to the work by Kin-Keung Ma et al. (Ma, Phang, Foster, & Hicks, 2011). The goal of their work is to generate a suit of test cases that reach a given statement (line reachability problem), which is achieved by directed search guided by a variety of heuristics. In our approach, the search is directed only in the sense that paths that do not reach the final method are not traversed. The symbolic execution traversal uses results of the slicing and mocking to explore a smaller state space. Unlike (Ma, Phang, Foster, & Hicks, 2011), our approach needs to traverse all possible paths between initial and final methods to show there is no violation.

We implement our approach in a prototype, where we use Wala for calculating the slice, Javassist to aid in the mocking of the code not contained in the slice and use Symbolic PathFinder (SPF) (Păsăreanu & Rungta, 2010) for symbolic execution. We describe several case studies on the approach to demonstrate its usefulness. In the future we would like to perform a quantitative comparative analysis to similar techniques and application to a number of examples. Other work in the area of slicing paired with symbolic

execution is that of Gendenhuys, Dwyer and Visser (Gendenhys, Dwyer, & Visser, 1012) in the paper “Probabilistic Symbolic Execution”. In this work they propose making the symbolic execution more efficient by calculating probabilities for paths and slicing the code based on these probabilities to reduce the code.

This thesis is organized as follows: theoretical background information, description of related work, the approach taken, case studies and results, future work and conclusion.

2. THEORETICAL BACKROUND

This thesis is aimed at the problem of reducing the software system to be verified to make subsequent verification scalable and efficient. A great deal of work has been done in the area of slicing. Efficient and accurate slicing is the keystone step in reduction of the analyzed software system to enable a more efficient verification via symbolic execution. Next, we will overview the basic method for static slicing and places where changes to the algorithm may customize it for the purpose of verification of constraints on an invocation of methods between modules.

Code Slicing

From the paper by Frank Tip on a survey of slicing techniques (Tip, 1994) we take a general description of the approach to slicing.

Weiser's original definition of program slicing (Weiser, Program Slicing, 1984) is based on iterative solution of dataflow equations. Weiser defines a *slice* as an *executable* program that is obtained from the original program by deleting zero or more statements. A *slicing criterion* consists of a pair (n, V) where *n* is a node in the CFG of the program, and *V* a subset of the program's variables. In order to be a slice with respect to criterion (n, V), a subset S of the statements of program P must satisfy the following properties: (i) S must be a valid program, and (ii) whenever P halts for a given input, S also halts for that input, computing the same values for the variables in V whenever the statement corresponding to node *n* is executed. At least one slice exists for any criterion: the program itself. A slice is *statement-minimal* if no other slice for the same criterion contains fewer statements. Weiser argues that statement-minimal slices are not

necessarily unique, and that the problem of determining statement-minimal slices is undecidable.

For each edge $i \rightarrow_{\text{CFG}} j$ in the CFG:

$$R_C^0(\text{i}) = R_C^0(\text{i}) \cup \{ v \mid v \in R_C^0(j), \text{v} \notin \text{DEF}(i) \} \cup \{ v \mid v \in \text{REF}(i), \text{DEF}(i) \cap R_C^0(j) \neq \emptyset \}$$

$$S_C^0 = \{ i \mid \text{DEF(i)} \cap R_C^0(j)) \neq \emptyset, i \rightarrow \text{CFG j} \}$$

Figure 1: Equations for determining directly relevant variables and statements

$$B_C^k = \{ b \mid \exists i \in S_C^k, i \in \text{INFL}(b) \}$$

$$R_C^{k+1}(i) = R_C^k(i) \cup \cup_{b \in B_C^k} R_{(b,REF(b))}^0(i)$$

$$S_C^{k+1} = B_C^k \cup \{ i \mid \text{DEF}(i) \cap R_C^{k+1}(j) \neq \emptyset, i \rightarrow \text{CFG } j \}$$

Figure 2: Equations for determining indirectly relevant variables and statements

Weiser describes an iterative algorithm for computing approximations of statement-minimal slices. It is important to realize that this algorithm uses *two* distinct "layers" of iteration. These can be characterized as follows:

1. Tracing transitive data dependences. This requires iteration in the presence of loops.
2. Tracing control dependences, causing the inclusion in the slice of certain control predicates. For each such predicate, step 1 is repeated to include the statements it is dependent upon.

The algorithm determines consecutive sets of *relevant variables* from which sets of *relevant statements* are derived; the computed slice is defined as the fix point of the latter set. First, the *directly relevant variables* are determined: this is an instance of step 1 of the iterative process outlined above. The set of directly relevant variables at node *i* in the CFG is denoted R_C^0 (*i*). The iteration starts with the initial values R_C^0 (*n*) = V, and R_C^0 (*m*) = 0 for any node $m \neq n$. Figure 1 shows a set of equations that define how the set of relevant variables at the *end j* of a CFG edge $i \rightarrow_{\text{CFG}} j$ affects the set of relevant variables at the *beginning i* of that edge. The least fix point of this process is the set of directly relevant variables at node *i*. From R_C^0, a set of *directly relevant statements*, S_C^0, is derived. Figure 1 shows how R_C^0 is defined as the set of all nodes *i* that define a variable *v* that is relevant at a CFG-successor of *i*.

As mentioned, the second "layer" of iteration in Weiser's algorithm consists of taking control dependences into account. Variables referenced in the control predicate of an **if** or **while** statement are *indirectly* relevant, if (at least) one of the statements in its body is relevant. To this end, the *range of influence* INFL(b) of a branch statement b is defined as the set of statements control dependent on b. Figure 2 shows a definition of the branch statements B_C^k that are indirectly relevant due to the influence they have on nodes i in S_C^k. Next, the sets of *indirectly relevant variable* $R_C^{k+1}$are determined. In addition to the variables in $R_C^k(i)$, $R_C^{k+1}(i)$ contains variables that are relevant because they have a transitive data dependence on statements in B_C^k. This is determined by performing the first type of iteration again (i.e., tracing transitive data dependences) with respect to a set of criteria (*b*, REF(*b*)), where b is a branch statement in B_C^k (see Figure 2). Figure 2 also

shows a definition of the sets S_C^{k+1} of *indirectly relevant statements* in iteration k + 1. This set consists of the nodes in B_C^k together with the nodes *i* that define a variable that is R_C^{k+1} -relevant to a CFG-successor *j*.

The sets R_C^{k+1} and S_C^{k+1} are nondecreasing subsets of the program's variables and statements, respectively; the fix point of the computation of the S_C^{k+1} sets constitutes the desired program slice. (Tip, 1994)

Modifying the slicing algorithm to take into account constraints imposed by properties to be verified can make it more efficient. If in the INFL set, the set of conditions could be passed through static symbolic execution it may be possible to eliminate infeasible paths and reduce that set. Essentially, such a reduction of the INFL set corresponds to elimination of branching in cases where static symbolic execution can predict the branch to be taken in response to constraints imposed by properties to be verified. Thus a smaller, property specific slide can be created statically.

Next, the computational complexity of the slicing algorithm is given: Weiser's algorithm for *intra*-procedural static slicing based on dataflow equations (Weiser, Program Slicing, 1984) can determine a slice in $O(v * (n + e))$ time, where *v* is the number of variables in the program, *n* is the number of vertices in the CFG, and *e* is the number of edges in the CFG. (Tip, 1994)

A small example program to illustrate slicing on a program that takes a number *n*, and computes the sum and the product for the first *n* positive numbers is presented below in Figures 3 – 6 adapted from (Tip, 1994).

```
(1)    read(n);
(2)    i := 1;
(3)    sum := 0;
(4)    product := 1;
(5)    while i <= n do
       begin
(6)      sum := sum + i;
(7)      product := product * i;
(8)      i := i + 1
       end;
(9)    write(sum);
(10)   write(product)
```

(a)

```
read(n);
i := 1;

product := 1;
while i <= n do
begin

  product := product * i;
  i := i + 1
end;

write(product)
```

(b)

Figure 3: (a) An example program. (b) A slice of the program w.r.t. criterion (10, product).

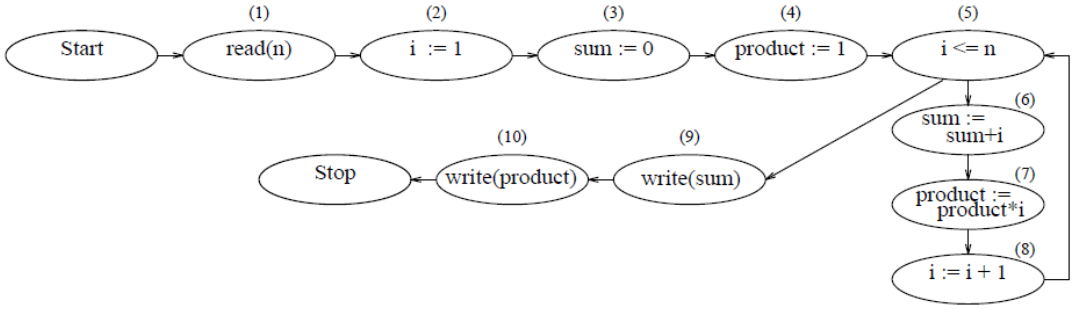

Figure 4: Resulting CFG from sample program

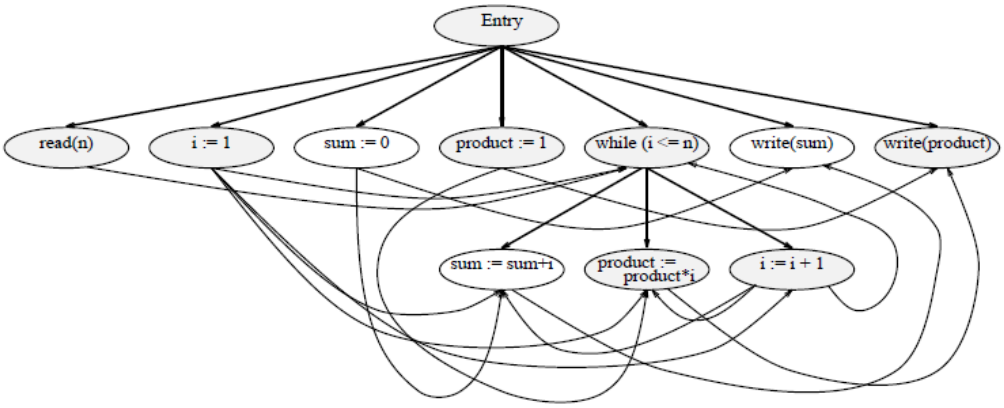

Figure 5: System Dependence Graph (SDG) based on input criterion

NODE #	DEF	REF	INFL	R_C^0	R_C^1
1	{ n }	∅	∅	∅	∅
2	{ i }	∅	∅	∅	{ n }
3	{ sum }	∅	∅	{ i }	{ i, n }
4	{ product }	∅	∅	{ i }	{ i, n }
5	∅	{ i, n }	{ 6, 7, 8 }	{ product, i }	{ product, i, n }
6	{ sum }	{ sum, i }	∅	{ product, i }	{ product, i, n }
7	{ product }	{ product, i }	∅	{ product, i }	{ product, i, n }
8	{ i }	{ i }	∅	{ product, i }	{ product, i, n }
9	∅	{ sum }	∅	{ product }	{ product }
10	∅	{ product }	∅	{ product }	{ product }

Figure 6: Results of Weiser's algorithm with slicing criterion (10,{product})

Mocking – Code Modification

A slicing algorithm is not guaranteed to create an executable slice. It must be possible to execute the produced slice to apply symbolic execution to it to verify the properties.

In this approach mocking is used to create an executable slice from the calculated slice. Next we will describe the general notion of mocking while the particular implementation in this approach will be given in Section 4.

The concept behind mock objects is that we want to create an object that will take the place of the real object. This mock object will expect a certain method to be called with certain parameters and when that happens, it will return an expected result. (Minella, 2008)

```
public class MichaelsAction extends ActionSupport {
    private LookupService service;
    private String key;

    public void setKey(String curKey) {  key = curKey; }

    public String getKey() { return key; }

    public void setService(LookupService curService) { service = curService; }

    public String doLookup() {
        if(StringUtils.isBlank(key)) {
            return FAILURE;
        }
        List results = service.lookupByKey(key);
        if(results.size() > 0) {
            return SUCCESS;
        }
        return FAILURE;
    }
```

```
}
```

Figure 7: Code example for mocking (Minella, 2008)

Using the above code as an example, let's say that when ‘1234’ is passed for my key to the service.lookupByKey call, I should get back a List with 4 values in it. Our mock object should expect lookupByKey to be called with the parameter "1234" and when that occurs, it will return a List with four objects in it. (Minella, 2008) A slightly simpler approach is implemented here by returning the correct type or null instead of a known result for a known input.

There are essentially two main types of mock object frameworks, ones that are implemented via proxy and ones that are implemented via class remapping. Let's take a look at the first (and by far more popular) option, proxy. (Minella, 2008)

A proxy object is an object that is used to take the place of a real object. In the case of mock objects, a proxy object is used to imitate the real object your code is dependent on. You create a proxy object with the mocking framework, and then set it on the object using either a setter or constructor. This points out an inherent issue with mocking using proxy objects. You have to be able to set the dependency up through an external means. In other words, you cannot create the dependency by calling new MyObject() since there is no way to mock that with a proxy object. This is one of the reasons Dependency Injection frameworks like Spring have taken off. They allow you to inject your proxy objects without modifying any code. (Minella, 2008)

The second form of mocking is to remap the class file in the class loader. What happens is that you tell the class loader to remap the reference to the class file it will load. So let's

say that I have a class MyDependency with the corresponding .class file called MyDependency.class and I want to mock it to use MyMock instead. By using this type of mock objects, you will actually remap in the classloader the reference from MyDependency to MyMock.class. This allows you to be able to mock objects that are created by using the new operator. Although this approach provides more power than the proxy object approach, it is also harder/more confusing to get going given the knowledge of classloaders you need to really be able to use all its features. (Minella, 2008)

It needs to be noted that the approach taken by us is partially a hybrid of these two methods of mocking. A by-proxy approach is used to change the methods that are not contained in the slice but this is done directly on the .class files replacing the original code with the mocked code.

Symbolic Execution

In the approach suggested by the thesis, once a property specific executable slice has been produced, symbolic execution is applied to verify if given properties hold.

In this section a general overview of symbolic execution is given.

The following description of symbolic execution implemented by Java Pathfinder tool was adapted from the paper “Evaluation of Java PathFinder Symbolic Execution Extension” by Kari Kähkönen (Kähkönen, June 26, 2007).

In symbolic execution (Khurshid, Pasareanu, & Visser, 2003) the program variables that normally contain concrete values are replaced with their symbolic counterparts that express a range of possible values using symbolic expressions. The states of a single

threaded system now contain the symbolic values of the expressions, a path condition that is a set of constraints that the values have to satisfy for the path being considered and a program counter indicating the next statement to be executed.

When executing a program the path condition is updated on branching points (e.g. when encountering an if-statement). The branching condition restricts the range of values the variables can have if the branch is taken. Knowing the branching condition and the symbolic values of the variables, the path condition can be updated and it can be checked if the new condition is still satisfiable. If it is not, the branch cannot be taken and can be safely ignored. Otherwise the original program can take the branch and that branch belongs to the possible behaviors of the system.

In JPF-SE the idea of symbolic execution is extended to allow execution without initialization of the input variables and to add support for complex data structures. The main task of the extension is to guide the underlying model checker, in this case JPF, to inspect the different paths of a symbolic execution tree. This allows that the normal state exploration techniques of JPF are available to be used, enabling for example symbolic execution of multi-threaded systems as JPF is capable of generating different thread schedules.

To handle uninitialized inputs to the system under verification, JPF-SE uses so called lazy initialization (Khurshid, Pasareanu, & Visser, 2003). The basic idea is that uninitialized variables will be initialized when they are first met during the execution of the program. The initialization is done as follows. If the variable is of a primitive type, it will be initialized to a new symbolic value corresponding to the primitive type. If the

uninitialized variable is a reference variable, it will be non-deterministically initialized to null, to a reference to a new object with uninitialized variables or to a reference to an object created earlier in the process. This way all the possibilities will be taken into account whether the variable points to a new object or is an alias to some existing one. This naturally allows the use of data structures like linked lists and trees to be used as inputs.

For input arrays the lazy initialization proceeds in a similar way (Anand, Pasareanu, & Visser, Symbolic execution with abstract subsumtion cheching , 2006). The array has a symbolic value representing its length and a set of array cells. These cells contain symbolic values for the index of the cell and for the actual data in the cell. When an uninitialized cell is encountered during the symbolic execution, it is initialized non-deterministically to new cell or to a cell that was initialized at some earlier time point. The path condition needs to be also updated so that it can be checked that the index stays within the size of the array. If the uninitialized reference variables are known to have some restrictions to their values, it is possible to use method preconditions to prevent these variables to be initialized in a way that conflicts with the restriction. After the initialization it will be checked if the precondition is broken and if it is, JPF is forced to backtrack and thus the branches with infeasible variable values will not be checked. It is also important to notice that if preconditions are used to limit the possible inputs to the system, these preconditions have to be conservative. This means that when an input structure has uninitialized fields after the lazy initialization, it is not always possible to say if an input structure will satisfy the condition after the field is initialized. In these

cases the inputs have to be seen as valid or otherwise some valid inputs could be excluded from the symbolic execution.

Even if a symbolic state can represent potentially an unbounded number of concrete states, a system may still have an unbounded number of symbolic states. Therefore we need some way of limiting the space of possible symbolic states. One simple way is to use bounded model checking by limiting the input sizes to the system and setting a maximum search depth for the model checker. Of course, this way only a subset of possible behaviors of the system will be checked.

In (Anand, Pasareanu, & Visser, Symbolic execution with abstract subsumtion cheching , 2006) a technique for checking subsumption of two symbolic states was presented. A symbolic state s1 is said to subsume another symbolic state s2 if the set of concrete states represented by s1 contain all the concrete states represented by s2. Now if a state that is found to be subsumed by some other state is encountered, it is safe to backtrack. For example imagine a case where a part of a program has been found free of errors when it is given a symbolic list with one element followed by an uninitialized element. Now if this part is given a longer symbolic list followed by an uninitialized element and the symbolic states are otherwise similar, it is not necessary to check the part of the program again, as all concrete states represented by the symbolic state have already been covered.

The complexity of this algorithm is O(n) but although it guarantees that heap configuration are subsumed if the algorithm return value true, it does not return true for all inputs where the heap configurations are subsumed. For a precise definition of heap

configurations and the algorithm for subsumption the reader is referred to (Anand, Pasareanu, & Visser, Symbolic execution with abstract subsumtion cheching , 2006).

To execute a program symbolically its source code needs to be instrumented first (Khurshid, Pasareanu, & Visser, 2003). The concrete variables and the operations on them have to be changed into their symbolic counterparts. JPF-SE currently provides classes for manipulating symbolic integer expressions (Expression class) and string expressions (StringExpression class). JPF-SE supports also symbolic arrays that contain Boolean or integer type variables. These classes contain the methods that operate on symbolic variables (e.g. different comparison operators, addition and multiplication for integers).

Statements accessing or updating variables need to be instrumented also. Accessing is done by using get methods that use lazy initialization if the variable has not yet been initialized. For updates set methods are used. These methods also set a flag that tells that the variable has been initialized. Simply instrumenting all the variables in a program does not necessary lead to a program that can be symbolically executed. There are two problems with symbolic execution that arise especially with real systems (Anand, Orso, & Harrold, Type-dependency analysis and program transformation for symbolic execution, 2007). First, the decision procedure being used to check the satisfiability of the path conditions might not be able to solve all kind of constrains that will be generated during the symbolic execution. For example an encountered branch condition might use unsupported operations (e.g. use a modulo operator). Second, if a concrete variable is transformed into a symbolic variable, it has to be possible to instrument all the statements

that will use the original variable. This is a problem if for example third party libraries that cannot be instrumented or Java native methods are used.

If these kinds of problems are encountered, they cannot be solved automatically. The user might be able to write the problematic part differently or use some other methods instead of un-instrumented library calls. This however requires that the problematic points of the system have to be identified.

3. RELATED WORK

In this section related work is described. The idea of slicing paired with symbolic execution is not new. Two closely related works that use a combination of slicing and symbolic execution are described below.

First is the work of Jaco Gendenhuys, Matthew Dwyer, and Willem Visser titled "Probabilistic Symbolic Execution" (Gendenhys, Dwyer, & Visser, 1012). In this work the authors explore the adaptation of symbolic execution to perform a more quantitative type of reasoning --- the calculation of estimates of the probability of executing portions of a program. They present an extension of the widely used Symbolic PathFinder symbolic execution system that calculates path probabilities. They exploit state-of-the-art computational algebra techniques to count the number of solutions to path conditions, yielding exact results for path probabilities. To mitigate the cost of using these techniques, they present two optimizations, PC slicing and count memorization, that significantly reduce the cost of probabilistic symbolic execution.

Here slicing and symbolic execution are paired together in a way that uses symbolic execution to calculate path probabilities to aid in the slicing. This contrasts to the work here that focus on reducing the problem with slicing and then using symbolic execution for verification.

Another work that primarily focuses on the line reachability problem is that of Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks in "Directed Symbolic Execution" (Ma, Phang, Foster, & Hicks, 2011). In this paper, they study the problem of automatically finding program executions that reach a particular target line. This problem

arises in many debugging scenarios; for example, a developer may want to confirm that a bug reported by a static analysis tool on a particular line is a true positive. They propose two new directed symbolic execution strategies that aim to solve this problem: shortest-distance symbolic execution (SDSE) uses a distance metric in an inter-procedural control flow graph to guide symbolic execution toward a particular target; and call-chain-backward symbolic execution (CCBSE) iteratively runs forward symbolic execution, starting in the function containing the target line, and then jumping backward up the call chain until it finds a feasible path from the start of the program. They also propose a hybrid strategy, Mix-CCBSE, which alternates CCBSE with another (forward) search strategy. The line reachability problem is very similar to the final point of interest in our problem. The difference being that they are trying to create test cases and are satisfied with a single path. In our case all possible paths between two points must be explored and the constraint must hold for all such paths.

The works mentioned above are related because they combine slicing and symbolic execution. Yet they do not focus on verification of architectural constraints, in the area of software architecture. Below a few related works in the software architecture are overviewed.

The most prominent initial work on a formal architectural description language (ADL) is that by R. Allen on the “Wright” ADL [3] done in the early 1990s. In his work, R. Allen introduces a formal language for definition of protocols assigned to connectors in an ADL. The work itself focuses on description of the suggested formal ADL and does not contain applications of verifiers even though the author does suggest doing such verification with a SPIN model checker in later publications.

Another related work is by J. Aldrich on a system for verifying consistency between a specification in a formal ADL and source code. He developed a tool called ArchJava that allows for verification of topological and component constraints consistency between a prescribed architecture and source code under development. In his work though the protocols for communication among modules are not specified and not verified. The ArchJava stops at defining types of connectors and at checking if topological constraints of a software architectural prescription are adhered to.

Finally, the work by S. Uchitel shows an application of a model checker LTSA to verification of protocols among modules defined in the UML sequence diagram. (Uchitel, 2003). The author creates an extension to LTSA model checker by Jeff Magee and Jeff Kramer from Imperial College, London that is aimed at verifying deadlock, race conditions and event sequence properties based on protocols defined in sequence diagrams. The approach suggested in this thesis differs in that it verifies the protocols on the produced bytecode via symbolic execution and uses slicing to create property specific slice of that bytecode.

4. APPROACH

The approach taken here is to use these tools, IBM's Wala for the slicing, reducing the problem size. Javassist to aid in the bytecode manipulation when mocking the code to create a practical slice and Symbolic Path Finder (SPF) to perform the symbolic execution for checking. Wrappers and implementation classes had to be built around these tools/algorithms. The first step is to define the architectural constraint/s that are of interest. The constraint will provide the final point of interest from which a slice will be calculated. The information from the calculated slice is used to mock the original bytecode into an executable slice. The modified bytecode, (.class files), can then be run in symbolic execution which applies the constraints to all feasible paths in the slice. The implementation of this tool is illustrated below.

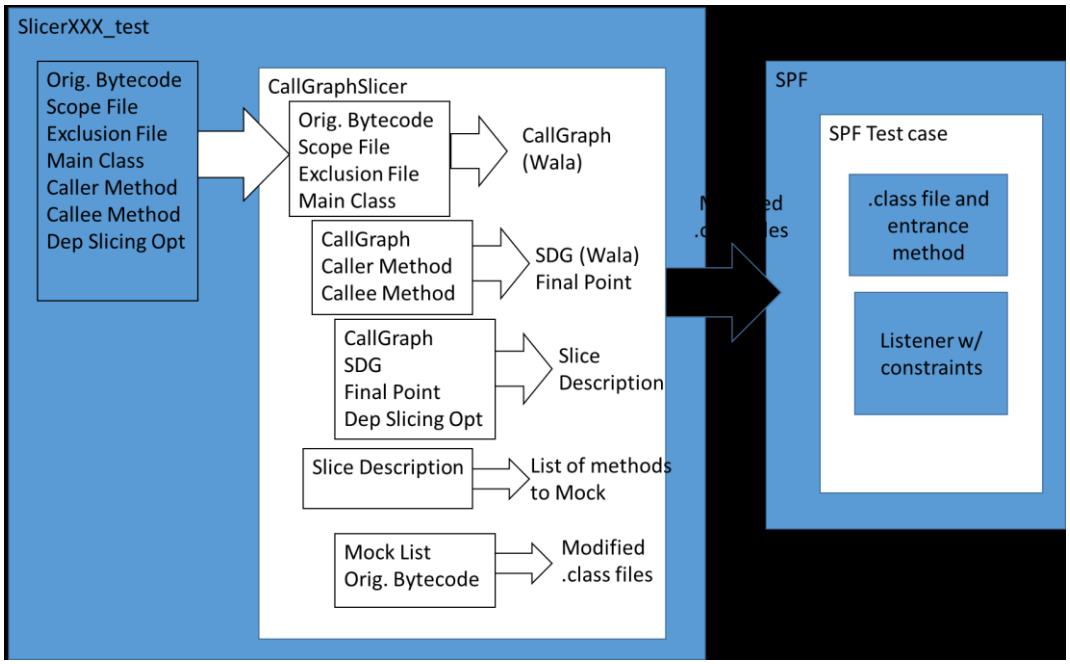

Figure 8 : Implementation Diagram

Let's look at each step in more detail. Once the constraints have been defined it is possible to begin calculating a slice. Several pieces are fed into the slicing. The original byte code, a scope file, exclusion file, and the name of the class containing the main method. The scope file helps to define the set of code to be sliced. The exclusion file is used to ignore libraries and large bodies of code that can be safely ignored. This would include being able to ignore java.math or java.io if the code that is being processed did not use these libraries. Without the exclusion file there is too much library information for Wala to be effective. The name of the file containing the main method is needed so that Wala will know where to begin. With these four inputs Wala represents the code in a call graph.

The call graph along with the final point of interest, the caller and callee method, a system dependence graph (SDG) are created by Wala. The final node of interest is also determined in the graph using the caller and callee methods. Since the final method of interest in our constraint could be called in multiple places the point from which it is called, the callee method, is needed to determine the exact point the slicing should begin from.

The next step is to calculate the actual slice of interest. Using the call graph, SDG, final point of interest and the slicing options, a description of the slice can be calculated. The slicing options allow the user to determine the level of control and data dependence the slicer will use and the direction of the slicing calculation, forward or backward. For all of our experiments backward slicing was used to produce an executable slice of the code. The dependency option used for all experiments was full data and control dependence to help insure that the slice calculated would be an executable slice. The output description

of the slice contains the methods, branches, statements and variables contained in the slice and their relation. This description is not bytecode, nor is it easily transformed to bytecode if at all. Wala also does not guaranteed that the slice will be executable.

The description of the slice is then parsed extracting all of the methods that are contained in the slice. The list of methods contained in the slice and the original bytecode for the program are used to create an executable version of the slice. First, a list of methods to mock needs to be determined. This list starts off by including all methods in the original bytecode. From here the list of methods contained in the slice is removed along with other sets of methods deemed to be needed. This list of methods that is not contained in the slice that is excluded from mocking contains the object constructor methods, any abstract methods, and base library methods. Excluding these methods from mocking eases the modification of the bytecode with little impact to the final result.

The result is a list of methods that will be mocked in the code. The list of methods to be mocked is used to mock the original bytecode, producing modified .class files that can be executed. The mocking step finds the class/methods in the code to be mocked and used Javassist to removes the body of the method and fix up the bytecode. If the method returns a value an appropriate return type will be inserted back into the method body. The removal of the method body effectively removed any traversal further down that path. This method of mocking effectively stubs out the remaining irrelevant code. (Minella, 2008) This is a simplistic and not completely efficient way of modifying the code. One caveat of doing the code modification in this manner is if you have code that returns an object and that object then calls some other function. i.e. getObjA().add(xxx), if the result of the getObjA() method returns null after the mocking the secondary call of

null.add(xxx) with cause a null exception. This can be handled by adding the first method to a list of methods to exclude from mocking. A more precise approach would be to remove the invocation of the methods in question and the associated bytecode. It is sufficient to show the proof of concept for creating a reduced set of code. The modified class files are then written back and can run by any test program just like the original compilation. In our case the modified code is fed to the second part of the process, the verification of the architectural constraints using SPF. In order to verify the architectural constraints Symbolic Pathfinder is employed. A modification of the symbolic listener that comes with SPF is used in the case. The symbolic listener monitors the invocation and the return of methods maintaining a call stack. At the point an invoke instruction is seen in the stack, it is checked against the constraints and violation/pass can be reported. The current constraint is hardcoded in the listener but use of a formal temporal logic appropriate for expressing the given properties would be preferred and left for the future work.

Below is a small example for the slicing and mocking. In this example code a slice is to be calculated for Obj2.method2(). The resulting slice should look something like SliceMockExample.initMethod() → SliceMockExample.m1() → Obj2.method2(). The other methods will be mocked, having the code removed or replaced as is necessary. The code to be removed and replaced is highlighted in blue as seen in Figure 9. The final result of what would be seen after the slicing and mocking is seen in Figure 10.

```
public class SliceMockExample {
        private Obj1 object1;
        private Obj2 object2;

        public void initMethod() {
                object1 = new Obj1();
                object2 = new Obj2();

                object1.setValue(5);
                object2.method1();
                m1();
                m2();
        }

        public void m1() {
                …
                object2.method2();
                …
        }

        public void m2() { … }
} // end class SliceMockExample
```

```
public class Obj1 {
        private int value = 0;
        public void setValue(int val) { value = val; }
        public void m1() { … }
        public int addValue(int val) {
                value += val;
                return value;
        }
} // end class Obj1
```

```
public class Obj2 {
        Obj1 ob1;
        public void method1() { … }
        public void method2() { … }
} // end class Obj2
```

Figure 9: Code to be sliced and mocked

```
public class SliceMockExample {
        private Obj1 object1;
        private Obj2 object2;

        public void initMethod() {
                object1 = new Obj1();
                object2 = new Obj2();

                object1.setValue(5);
                object2.method1();
                m1();
                m2();
        }

        public void m1() {
                …
                object2.method2();
                …
        }

        public void m2() { }
} // end class SliceMockExample
```

```
public class Obj1 {
        private int value = 0;
        public void setValue(int val) { }
        public void m1() { }
        public int addValue(int val) {
                return 0;
        }
} // end class Obj1
```

```
public class Obj2 {
        Obj1 ob1;
        public void method1() { }
        public void method2() { … }
} // end class Obj2
```

Figure 10: Code after slicing and mocking

5. RESULTS AND ANAYLYSIS

Case Study: Calculator MVC

For the first case study we chose a simplified implementation of a calculator that uses the MVC architectural style from [4]. (Figure 11) It was simplified by replacing Swing library calls with stubs so that SPF would not execute the Swing library code. Driver code was also added to the view to mimic user interactions with the calculator.

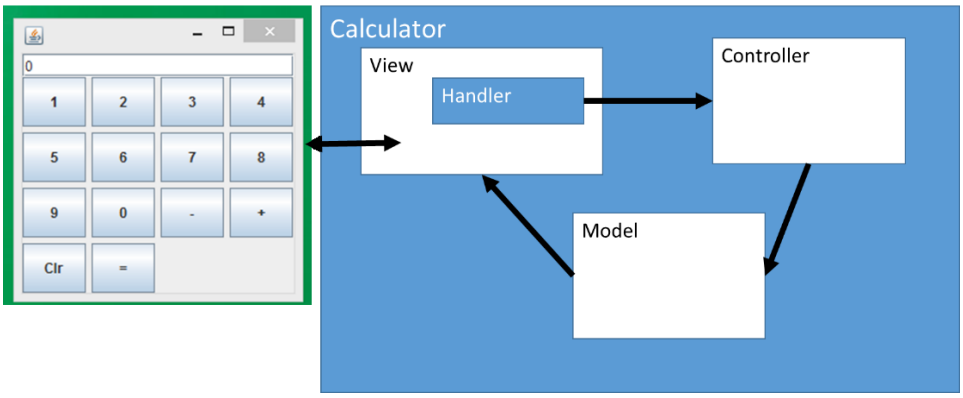

Figure 11: Calculator MVC

The architectural constraint to check for is that a method in the View should not directly invoke a modifying method in the Model. Instead, a View method should invoke a method in the Controller, which in turn should invoke a method in the Model, the Model is responsible for notifying the view of any changes. The diagram below illustrates the architecture of the calculator MVC.

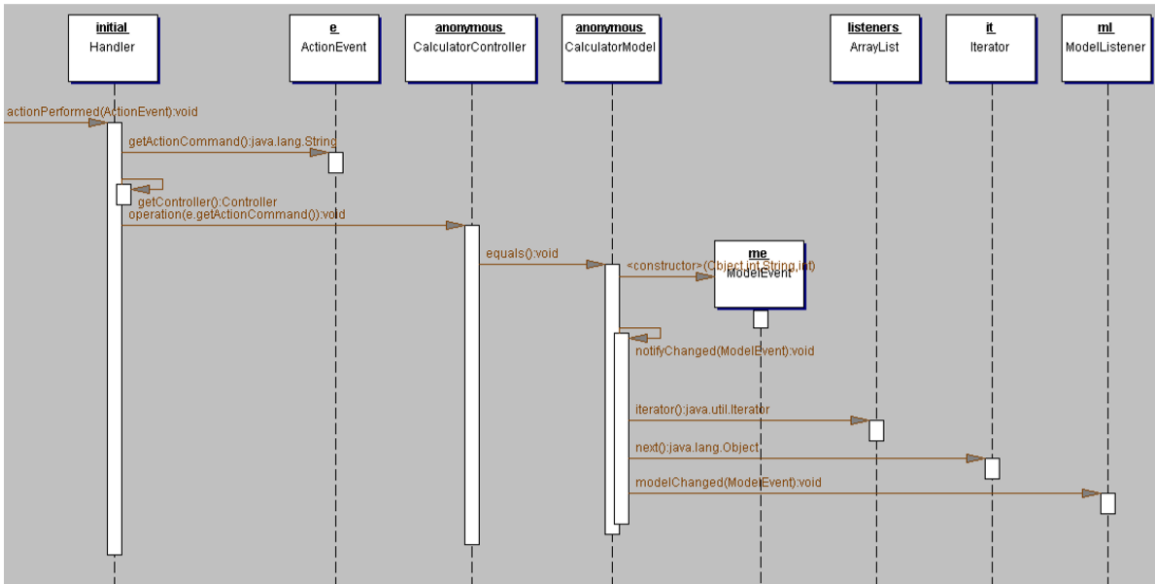

Figure 12: MVC Sequence Diagram (Hunt)

The particular names of methods are used when checking this constraint. Specification of the constraint was implemented programmatically. We understand that a generalized specification via an appropriate temporal logic is needed. This is left for future work.

The constraints derived from the sequence diagram in Figure 12:

- The view should never update without notification from the model.
- The model should never notify if the controller has not issued an operation.
- The controller should not issue an operation if the handler has not created an action.
- The handler should not create an action if there is no activity in the view.

The pertinent classes of the calculator implementation under analysis include CalculatorView, CalculatorController and CalculatorModel. The whole codebase under

analysis contains many more classes, but these contain the methods used in the architectural constraint. The CalculatorView contains main method that mimics pressing a button by invoking pushed method on an instance of a given button. It also contains an inner class Handler with an actionPerformed method. It is this method that is invoked in response to a pushed method. The actionPerformed is supposed to invoke the operation method from the CalculatorController class. The operation method, in turn, is supposed to invoke a relevant method from CalculatorModel. The buttons correspond to basic calculator operations: addition, subtraction, store, and equals (to get result). The constraint is that actionPerformed should not invoke the CalculatorModel methods directly.

Results: Calculator MVC

The results are as follows: Results for the original MVC, results for the MVC with additional threads added into the code, results with slicing and automatic code modifications to reduce the state space. The addition of threads to the MVC is used to mimic a more complex, non-deterministic or computationally expensive set of interleaving's of instructions in the code that does not affect the path of interest.

Below is a summary of the results.

Table 1: Calculator MVC Results

Code Type	# Paths Searched	SPF Runtime	# State	Max Search Depth	Memory	# Instructions
Orig. No Slicing	90	3 sec	225	5	78MB	82563
w/ 2 threads No Slicing	90	4min 47 sec	39947	17	132MB	133433898
w/ Slicing	3	1sec	17	5	60MB	5626

For the case with the slicing the time to examine and modify the code needs to be taken into account as well:

Table 2: Calculator MVC Slicing Results

Action	Time (sec)
Build Call Graph	4.37
Create SDG & Slice	7.78
Modify Byte Code	0.631
Total	12.82

The design contains 787 total methods in the project, 99 of which are created and not inherited. After the slicing the mocking removed 77 method bodies, a reduction of 77.78%. The size of the original code (.class files) was 42.5 KB (43,577bytes), this was reduced to 29.9 KB (30,654 bytes), 70.3% of the original code size.

From the results little gain can be seen between the original code and the sliced code other than a reduction in the number of paths that SPF traversed. The total time to verify

including the time slice and modify the code was expensive almost 4 times what the original code took to verify. However when you compare the code with additional threads to the sliced code the advantage of reducing the problem can be seen. There is a savings in total time of verification, and the number of paths explored. The sliced code was able to eliminate the extra threads simplifying the code greatly.

Analysis and modification of the code can be expensive but if it allows the removal of computationally expensive code the benefit can be substantial, as seen when comparing the sliced version of the code with the version of the code that has additional threads added to mimic code complexity. In this case you can save an immense amount of resources on the verification of the code. From 4min and 47sec to only a second.

Case Study: ModelCheckCTL

The second case study CTL based model checker implemented in the MVC architectural style. As with the calculator example a similar type of architectural constraints are to be verified in that the implementation matched the MVC architectural style. This example is about twice as larger the calculator example. The model itself has more parts in that it contains a representation of the model in a Kripke Structure and applied a given CTL

formula and reports the results

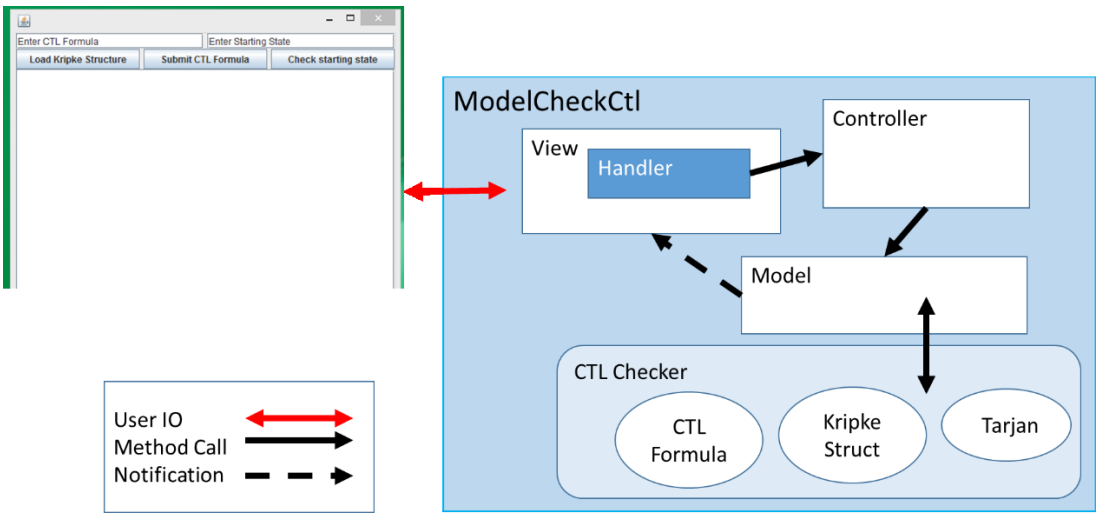

Figure 13: ModelCheckCTL MVC

The architectural constraint to check for is that a method in the View should not directly invoke a modifying method in the Model. Instead a View method should invoke a method in the Controller, which in turn should invoke a method in the Model with the Model notifying the View of any changes.

Once again the particular names of methods are used when checking this constraint. Specification of the constraint was implemented programmatically.

The constraints derived from this architecture are similar to those in the first case study:

- The view should never update without notification from the model.
- The model should never notify if the controller has not issued an operation.
- The controller should not issue an operation if the handler has not created an action.

- The handler should not create an action if there is no activity in the view.

The pertinent classes of the calculator implementation under analysis include ModelCheckCTLView, ModelCheckCTLController and ModelCheckCTLModel. The whole codebase under analysis contains many more classes, but these contain the methods used in the architectural constraint. Unlike the first case study there is more potential code that can be activated by the model. The path selected to test in this instance should avoid this extra code, thus eliminating a large and possibly complex portion of the code. The ModelCheckCTLView contains main method that mimics pressing a button by invoking pushed method on an instance of a given button. It also contains an inner class Handler with an actionPerformed method. It is this method that is invoked in response to a pushed method. The actionPerformed is supposed to invoke the operation method from the ModelCheckCTLController class. The operation type method, in turn, is supposed to invoke a relevant method from ModelCheckCTLModel. A combination of buttons and text boxes make up the input. When a button is pressed the input from the associated text box is read in and the appropriate action performed on that input. For instance when the 'Load Kripke Structure' button is pressed a file containing a representation of a model is read in and a graph structure is created. When the 'Submit CTL Formula' button is pressed the text representing a CTL formula is read in and parsed for correctness. The final input of 'Check Starting State' will take the text with the starting state of the graph and apply the CTL formula to the graph and respond with the results. If any inputs are malformed an error message results.

Results: ModelCheckCTL MVC

The path of interest was that of pressing the 'Submit CTL Formula' in the View to the point of the Model notifying the View of the change.

Table 3: ModelCheckCTL MVC Results

Code Type	# Paths Searched	SPF Runtime	# State	Max Search Depth	Memory	# Instructions
Orig. No Slicing	4	2 sec	5	2	76MB	74851
w/ Slicing	4	1sec	5	2	76MB	16510

Table 4: ModelCheckCTL MVC Slicing Results

Action	Time (sec)
Build Call Graph	3.54
Create SDG & Slice	19.8
Modify Byte Code	0.224
Total	24.22

The design contains 1097 total methods in the project, 258 of which are created and not inherited. After the slicing the mocking removed 196 method bodies, a reduction of 75.97%. The size of the original code (.class files) was 96.4 KB (98,802 bytes), this was reduced to 64.1 KB (65,736 bytes), 66.5% of the original code size.

A significant amount of code was removed due to the slicing but the cost of the slice far outweighed the final verification results in this case. The state space that was made available to SPF was small enough and code simple enough to not have a significant impact on the runtime. Without code structure that would cause indeterminate results, such as threads, the SPF was able to solve the problem without much effort and the reduction was not needed.

Case Study: RWGUI

The third case study is a networking planner implemented in the MVC architectural style. As with the calculator example a similar type of architectural constraints is to be verified in that the implementation matched the MVC architectural style. This example is much larger and more complex than the calculator example. This program allows the user to add networking resources and connect the resources creating a network.

Results: RWGUI

In testing the RWGUI a similar type of path to the previous case studies was the target of the exploration. When a button was pushed we wanted to see the View activate the Controller, the Controller activate the Model and the Model notify the View of the change. I this example the hope was for a greater than 70% reduction in code size.

In this example it was not targeted to be run in SPF so slicing was attempted on the original code containing all of the java.swing library code. Wala was able to take the code and create the call graph and system dependence graph but it was unable to calculate the slice after running for more than 24 hours. It is suspected that something in

the swing library code causes the slicing algorithm to become stuck. It may also be that the machine the example was run on did not have enough memory to support the calculation.

6. FUTURE WORK

This work is preliminary and shows proof of concept in creating testing efficiency and scalability. There is more that could be done here. Currently the flow uses Wala to do the code slicing and it may be possible to make a more efficient/precise slicing algorithm to meet the needs of reducing the code. If the slicing was modified to take two parameters, the initial and final points of interest it may be possible to further reduce the code and speed up the slicing. The original goal is to verify a protocol between two modules, a source and sink relationship. The idea would be that the slice is computed from the initial point of interest and then a second slice would be computed on the final point of interest using the results of the initial slice. Any node already marked in the initial slice would end the slicing exploration down that path. Implementation of this could also be done concurrently further reducing the time.

The addition of static symbolic execution into the slicing algorithm at the points of decisions has a possibility of also reducing the code size by removing the infeasible paths. This not only reduces the resulting code size but should also reduce the search and calculation time for the slicing which is probably more important.

The modification of the bytecode could be handled in many different ways. Instead of simply mocking the methods that are not in the portion of the code that is of interest the actual invocations and associated code could be removed instead along with other non-method code. A mocking tool may also be used at this point to modify the code to reflect the slice of interest. In the future we intend to add a general specification of the constraints via an appropriate logic and a test case generation capability. We also would like to perform a quantitative comparative analysis against similar approaches and

improve the efficiency of the analysis algorithms. In addition, we would like to validate the prototype by applying it to analysis of larger systems.

7. CONCLUSION

In this thesis we described preliminary work focused on checking architectural constraints on sequences of method invocations. We gave a description of the prototype and case studies. The initial approach was to begin by representing the code with a call graph. The call graph could then be pruned by finding all the predecessors from a final point of interest and all the successors from an initial point of interest. Doing this greatly reduced the size of the graph and, consequently, the state space. After working along this direction a major flaw was discovered with this approach. The call graph representation did not have the contextual information needed. Without the dependency information, the resulting code left was not a true representation of the program and may not even be runnable. At this point an investigation into using program control flow graphs and program dependency graphs was looked at and slicing was determined to be the solution that met the needs of the problem. While the approach was validated on MVC systems, it can be applied to systems that must conform to other similar architectural constraints. For instance, in a cyber-physical system, there might be properties that stipulate that if a sensor value was changed then a control output must be calculated by a particular module on all paths between some "onSensorChanged()" method invocation and "controlOutput()" method invocation in some actuator control module. Briefly put, a sensor value change must result in a properly calculated actuator control output on all execution paths between those two method invocations. The suggested approach would make verification of such properties more efficient by reducing the application to a potentially much smaller executable slice for a given increment of a software development process, thus significantly cutting down on time taken by regression testing

both during development until release and maintenance after the release of the analyzed software system.

Using slicing for the reduction of the problem size seems to hold the most promise for code that has low coupling and applied to a problem that would eliminate complex or non-deterministic code.

APPENDIX SECTION

APPENDIX A

The work on this thesis involved three tools, SPF, Wala and Javassist, along with the code to make them all work together. All work was done in Eclipse. The following is a how to and what is needed to run the test cases in this paper. The following projects need to be downloaded, configured and built in Eclipse:

Java Path Finder: a tutorial on how to do this is located on YouTube at :

https://www.youtube.com/watch?v=6vVAYT4yMfw

The copy of Java Symbolic Path Finder from my github:

https://github.com/stuartsiroky/jpf_symbc.git

This has the listener changes and test cases to run.

Wala from github: https://github.com/wala/WALA.git

My code that implements the Wala slicing and Mocking using Javassist on github:

https://github.com/stuartsiroky/WalaSliceByteMod.git

Code for test cases on git: https://github.com/stuartsiroky/modelCheckCTL_noSwing.git

https://github.com/stuartsiroky/NoSwingCalc.git

Wala will need to be setup. Here is a link to the getting started but there is only a few modification:

http://wala.sourceforge.net/wiki/index.php/UserGuide:Getting_Started

copy com.ibm.wala.core (project) dat/wala.properties.sample to dat/wala.properties and set the java_runtime_dir and output_dir

com.ibm.wala.core.tests (project) copy dat/wala.example.properties.sample to dat/wala.examples.properties

Add the following projects to this projects build path:

com.ibm.wala.core

com.ibm.wala.core.tests

com.ibm.wala.util

You may also have to change some of the paths in the examples from my local paths to your local paths in the top level file in the SliceTest pkg the test should be there.

The test in SPF are in the example directory and can be run before slicing the code to see the results. The user can then run the slicer test to modify the code and rerun SPF to see the impact of the slicing and mocking.

APPENDIX B

Code for the listener that is used in the symbolic execution. The base listener maintains a method call chain by monitoring the invoke and return instructions and applies the given constraints when appropriate. Constraints are defined in a listener that extends the base listener.

```
package gov.nasa.jpf.symbc;

import gov.nasa.jpf.Config;
import gov.nasa.jpf.JPF;
import gov.nasa.jpf.PropertyListenerAdapter;
import gov.nasa.jpf.jvm.bytecode.ARETURN;
import gov.nasa.jpf.jvm.bytecode.DRETURN;
import gov.nasa.jpf.jvm.bytecode.FRETURN;
import gov.nasa.jpf.jvm.bytecode.IRETURN;
import gov.nasa.jpf.jvm.bytecode.JVMInvokeInstruction;
import gov.nasa.jpf.jvm.bytecode.JVMReturnInstruction;
import gov.nasa.jpf.jvm.bytecode.LRETURN;
import gov.nasa.jpf.report.ConsolePublisher;
import gov.nasa.jpf.report.Publisher;
import gov.nasa.jpf.report.PublisherExtension;
import gov.nasa.jpf.search.Search;
import gov.nasa.jpf.symbc.bytecode.BytecodeUtils;
import gov.nasa.jpf.symbc.bytecode.INVOKESTATIC;
import gov.nasa.jpf.symbc.concolic.PCAnalyzer;
import gov.nasa.jpf.symbc.numeric.Expression;
import gov.nasa.jpf.symbc.numeric.IntegerConstant;
import gov.nasa.jpf.symbc.numeric.IntegerExpression;
```

```
import gov.nasa.jpf.symbc.numeric.PCChoiceGenerator;
import gov.nasa.jpf.symbc.numeric.PathCondition;
import gov.nasa.jpf.symbc.numeric.RealConstant;
import gov.nasa.jpf.symbc.numeric.RealExpression;
import gov.nasa.jpf.symbc.numeric.SymbolicConstraintsGeneral;
import gov.nasa.jpf.symbc.numeric.SymbolicInteger;
import gov.nasa.jpf.util.Pair;
import gov.nasa.jpf.vm.ChoiceGenerator;
import gov.nasa.jpf.vm.ClassInfo;
import gov.nasa.jpf.vm.Instruction;
import gov.nasa.jpf.vm.LocalVarInfo;
import gov.nasa.jpf.vm.MethodInfo;
import gov.nasa.jpf.vm.StackFrame;
import gov.nasa.jpf.vm.ThreadInfo;
import gov.nasa.jpf.vm.Types;
import gov.nasa.jpf.vm.VM;

import java.io.PrintWriter;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Stack;
import java.util.StringTokenizer;
import java.util.Vector;


public class SliceSymbolicListener extends PropertyListenerAdapter implements
      PublisherExtension {

   /*
    * Locals to preserve the value that was held by JPF prior to changing it in
    * order to turn off state matching during symbolic execution no longer
    * necessary because we run spf stateless
    */
   protected Stack<String> MethodCallStack = new Stack<String>();// added
   private String CalleeMethod;

   private Map<String, MethodSummary> allSummaries;
   private String currentMethodName = "";

   public SliceSymbolicListener(Config conf, JPF jpf) {
      jpf.addPublisherExtension(ConsolePublisher.class, this);
      allSummaries = new HashMap<String, MethodSummary>();
   }

   @Override
   public void propertyViolated(Search search) {
      …
   }

   @Override
   public void instructionExecuted(VM vm, ThreadInfo currentThread,
         Instruction nextInstruction, Instruction executedInstruction) {

      if (!vm.getSystemState().isIgnored()) {
         Instruction insn = executedInstruction;
```

```
ThreadInfo ti = currentThread;
Config conf = vm.getConfig();

if (insn instanceof JVMInvokeInstruction) {
   JVMInvokeInstruction md = (JVMInvokeInstruction) insn;
   String methodName = md.getInvokedMethodName();
   int numberOfArgs = md.getArgumentValues(ti).length;

   MethodInfo mi = md.getInvokedMethod();
   ClassInfo ci = mi.getClassInfo();
   String className = ci.getName();

   StackFrame sf = ti.getTopFrame();
   String shortName = methodName;
   String longName = mi.getLongName();

   CalleeMethod = mi.getFullName(); //added
   MethodCallStack.push(CalleeMethod); //added
   CheckPath(); //Added

   if (methodName.contains("("))
      shortName = methodName
            .substring(0, methodName.indexOf("("));

   if (!mi.equals(sf.getMethodInfo()))
      return;

   if ((BytecodeUtils.isClassSymbolic(conf, className, mi,
         methodName))
         || BytecodeUtils.isMethodSymbolic(conf,
               mi.getFullName(), numberOfArgs, null)) {

      MethodSummary methodSummary = new MethodSummary();

      methodSummary.setMethodName(className + "." + shortName);
      Object[] argValues = md.getArgumentValues(ti);
      String argValuesStr = "";
      for (int i = 0; i < argValues.length; i++) {
         argValuesStr = argValuesStr + argValues[i];
         if ((i + 1) < argValues.length)
            argValuesStr = argValuesStr + ",";
      }
      methodSummary.setArgValues(argValuesStr);
      byte[] argTypes = mi.getArgumentTypes();
      String argTypesStr = "";
      for (int i = 0; i < argTypes.length; i++) {
         argTypesStr = argTypesStr + argTypes[i];
         if ((i + 1) < argTypes.length)
            argTypesStr = argTypesStr + ",";
      }
      methodSummary.setArgTypes(argTypesStr);

      // get the symbolic values (changed from constructing them
      // here)
      String symValuesStr = "";
      String symVarNameStr = "";
```

```
      LocalVarInfo[] argsInfo = mi.getArgumentLocalVars();

      if (argsInfo == null)
        throw new RuntimeException(
            "ERROR: you need to turn debug option on");

      int sfIndex = 1; // do not consider implicit param "this"
      int namesIndex = 1;
      if (md instanceof INVOKESTATIC) {
        sfIndex = 0; // no "this" for static
        namesIndex = 0;
      }

      for (int i = 0; i < numberOfArgs; i++) {
        Expression expLocal = (Expression) sf
            .getLocalAttr(sfIndex);
        if (expLocal != null) // symbolic
          symVarNameStr = expLocal.toString();
        else
          symVarNameStr = argsInfo[namesIndex].getName()
                  + "_CONCRETE" + ",";
        symValuesStr = symValuesStr + symVarNameStr + ",";
        sfIndex++;
        namesIndex++;
        if (argTypes[i] == Types.T_LONG
            || argTypes[i] == Types.T_DOUBLE)
          sfIndex++;

      }

      // get rid of last ","
      if (symValuesStr.endsWith(",")) {
        symValuesStr = symValuesStr.substring(0,
            symValuesStr.length() - 1);
      }
      methodSummary.setSymValues(symValuesStr);

      currentMethodName = longName;
      allSummaries.put(longName, methodSummary);
    }
  } else if (insn instanceof JVMReturnInstruction) {
    MethodInfo mi = insn.getMethodInfo();
    ClassInfo ci = mi.getClassInfo();

    if (!MethodCallStack.empty()) { //added
      CalleeMethod = MethodCallStack.pop();
    }

    if (null != ci) {
      String className = ci.getName();
      String methodName = mi.getName();
      String longName = mi.getLongName();
      int numberOfArgs = mi.getNumberOfArguments();

      if (((BytecodeUtils.isClassSymbolic(conf, className, mi,
```

```
   methodName)) || BytecodeUtils.isMethodSymbolic(
   conf, mi.getFullName(), numberOfArgs, null))) {

ChoiceGenerator<?> cg = vm.getChoiceGenerator();
if (!(cg instanceof PCChoiceGenerator)) {
   ChoiceGenerator<?> prev_cg = cg
                  .getPreviousChoiceGenerator();
   while (!((prev_cg == null) || (prev_cg instanceof PCChoiceGenerator))) {
      prev_cg = prev_cg.getPreviousChoiceGenerator();
   }
   cg = prev_cg;
}
if ((cg instanceof PCChoiceGenerator)
      && ((PCChoiceGenerator) cg).getCurrentPC() != null) {
   PathCondition pc = ((PCChoiceGenerator) cg)
                  .getCurrentPC();
   // pc.solve(); //we only solve the pc
   if (SymbolicInstructionFactory.concolicMode) {
      SymbolicConstraintsGeneral solver = new SymbolicConstraintsGeneral();
      PCAnalyzer pa = new PCAnalyzer();
      pa.solve(pc, solver);
   } else
      pc.solve();

   if (!PathCondition.flagSolved) {
      return;
   }

   // after the following statement is executed, the pc
   // loses its solution

   String pcString = pc.toString();// pc.stringPC();
   Pair<String, String> pcPair = null;

   String returnString = "";

   Expression result = null;

   if (insn instanceof IRETURN) {
      IRETURN ireturn = (IRETURN) insn;
      int returnValue = ireturn.getReturnValue();
      IntegerExpression returnAttr = (IntegerExpression) ireturn
                        .getReturnAttr(ti);
      if (returnAttr != null) {
                  returnString = "Return Value: "
                                    + String.valueOf(returnAttr
                                                      .solution());
                  result = returnAttr;
      } else { // concrete
                  returnString = "Return Value: "
                                    + String.valueOf(returnValue);
                  result = new IntegerConstant(returnValue);
      }
   } else if (insn instanceof LRETURN) {
      LRETURN lreturn = (LRETURN) insn;
      long returnValue = lreturn.getReturnValue();
```

```
    IntegerExpression returnAttr = (IntegerExpression) lreturn
                        .getReturnAttr(ti);
    if (returnAttr != null) {
                returnString = "Return Value: "
                                    + String.valueOf(returnAttr
                                                        .solution());
                result = returnAttr;
    } else { // concrete
                returnString = "Return Value: "
                                    + String.valueOf(returnValue);
                result = new IntegerConstant(
                                    (int) returnValue);
    }
} else if (insn instanceof DRETURN) {
    DRETURN dreturn = (DRETURN) insn;
    double returnValue = dreturn.getReturnValue();
    RealExpression returnAttr = (RealExpression) dreturn
                        .getReturnAttr(ti);
    if (returnAttr != null) {
                returnString = "Return Value: "
                                    + String.valueOf(returnAttr
                                                        .solution());
                result = returnAttr;
    } else { // concrete
                returnString = "Return Value: "
                                    + String.valueOf(returnValue);
                result = new RealConstant(returnValue);
    }
} else if (insn instanceof FRETURN) {

    FRETURN freturn = (FRETURN) insn;
    double returnValue = freturn.getReturnValue();
    RealExpression returnAttr = (RealExpression) freturn
                        .getReturnAttr(ti);
    if (returnAttr != null) {
                returnString = "Return Value: "
                                    + String.valueOf(returnAttr
                                                        .solution());
                result = returnAttr;
    } else { // concrete
                returnString = "Return Value: "
                                    + String.valueOf(returnValue);
                result = new RealConstant(returnValue);
    }

} else if (insn instanceof ARETURN) {
    ARETURN areturn = (ARETURN) insn;
    IntegerExpression returnAttr = (IntegerExpression) areturn
                        .getReturnAttr(ti);
    if (returnAttr != null) {
                returnString = "Return Value: "
                                    + String.valueOf(returnAttr
                                                        .solution());
                result = returnAttr;
    } else {// concrete
                Object val = areturn.getReturnValue(ti);
```

```
                        returnString = "Return Value: "
                                        + String.valueOf(val);
                        // DynamicElementInfo val =
                        // (DynamicElementInfo)areturn.getReturnValue(ti);
                        String tmp = String.valueOf(val);
                        tmp = tmp
                                        .substring(tmp.lastIndexOf('.') + 1);
                        result = new SymbolicInteger(tmp);

                    }
                } else
                    // other types of return
                    returnString = "Return Value: --";

                pcString = pc.toString();
                pcPair = new Pair<String, String>(pcString,
                        returnString);
                MethodSummary methodSummary = allSummaries
                        .get(longName);
                @SuppressWarnings("rawtypes")
                Vector<Pair> pcs = methodSummary
                        .getPathConditions();
                if ((!pcs.contains(pcPair))
                        && (pcString.contains("SYM"))) {
                    methodSummary.addPathCondition(pcPair);
                }

                if (allSummaries.get(longName) != null) // recursive
                                                                                    // call
                    longName = longName + methodSummary.hashCode(); // differentiate

// the

// key

// for

// recursive

// calls
                allSummaries.put(longName, methodSummary);
                if (SymbolicInstructionFactory.debugMode) {
                    System.out
                                    .println("*************Summary***************");
                    System.out.println("PC is:" + pc.toString());
                    if (result != null) {
                        System.out.println("Return is:  " + result);
                        System.out
                                        .println("************************************");
                    }
                }
            }
        }
    }
}
```

```
    }

    protected void CheckPath() {
    }

    /*
     * The way this method works is specific to the format of the methodSummary
     * data structure
     */

    private void printMethodSummary(PrintWriter pw, MethodSummary methodSummary) {
      …
    }

    // -------- the publisher interface
    @SuppressWarnings("rawtypes")
    @Override
    public void publishFinished(Publisher publisher) {
      …
    }

    protected class MethodSummary {
      …
    }
}


package gov.nasa.jpf.symbc;

import java.util.ArrayList;

import gov.nasa.jpf.Config;
import gov.nasa.jpf.JPF;

public class Equals_SliceSymbolicListener extends SliceSymbolicListener {

    ArrayList<String> gmConst;

    public Equals_SliceSymbolicListener(Config conf, JPF jpf) {
      super(conf, jpf);
      initPathConst();
    }

    protected void CheckPath() {
      boolean result = true;
      Object oA[] = MethodCallStack.toArray();
      ArrayList<String> sA = new ArrayList<String>();
      for (int i = 0; i < oA.length; i++) {
        sA.add((String) oA[i]);
      }

      ArrayList<String> pp = new ArrayList<String>();
      int lastIndex = gmConst.size() - 1;
      String lastMethName = gmConst.get(lastIndex);

      if (sA.contains(lastMethName)) {
```

```
        for (String s : sA) {
          if (gmConst.contains(s)) {
            pp.add(s);
          }
        }

//      System.out.println("STUART path is          " + sA.toString());
//      System.out.println("STUART contraint path is " + pp.toString());

        if (pp.size() != gmConst.size()) {
          result = false;
          System.out.println("Path is " + pp.size() + " expected "
              + gmConst.size());
        }
        if (result == true) {
          for (int i = 0; i < pp.size(); i++) {
            if (!pp.get(i).equals(gmConst.get(i))) {
              System.out.println("Failed to match at " + i + " "
                  + pp.get(i));
              result = false;
              break;
            }
          }
        }
        if (result == false) {
          System.out
              .println("======= FAILED CONSTRAINT PATH ===========");
          System.out.println("\t" + pp.toString());
          System.out
              .println("should have been \n\t" + gmConst.toString());
          System.out
              .println("======= FAILED CONSTRAINT PATH ===========");
        }
      }

    }

    private void initPathConst() {
      gmConst = new ArrayList<String>();
      gmConst.add("calc.view.CalculatorView.equalsMethod(Lcalc/view/CalculatorView;)V");
      gmConst.add("buttons.EqualsButton.pushed()V");

    gmConst.add("calc.view.CalculatorView$EqHandler.actionPerformed(Lcalc/noSwing/ActionEvent;)V");
      gmConst.add("calc.controller.CalculatorController.equalsOperation()V");
      gmConst.add("calc.model.CalculatorModel.equalsOp()V");
    }
}
```

REFERENCES

Aldrich, J., Chambers, C., & Notkin, D. (2002). ArchJava: Connecting software architecture to implementation. *24th International Conference on Software Engineering ICSE* (pp. 187-197). New York, NY, USA: ACM.

Allen, R. (January 1997). *A Formal Approach to Software Achitecture.* Carnegie Mellon, School of Computer Science. Issued as CMU Technical Report CMU-CS-97-144.

Anand, S., Orso, A., & Harrold, M. J. (2007). Type-dependency analysis and program transformation for symbolic execution. *13th International conference on Tools and Algorithms for the Construction and Analysis of Systems.*

Anand, S., Pasareanu, C. S., & Visser, V. (2006). Symbolic execution with abstract subsumtion cheching . *Lecture Notes in Computer Science, 3925*, 163-181.

Clarke, L. A. (1976). A program testing system. *In Proc of the 1979 anual conference* (pp. 448-491). AMC '76.

Cui, H., Hu, G., Wu, J., & Yang, J. (2013). Verifying System Rules Using Rule-Directed Symbolic Execution. *ASPLOS'13.* Houston, TX.

Dinges, P., & Agha, G. (2014). Targeted Test Input Generation Using Symbolic-Concrete Backward Execution. *IEEE/ACM International Conference on Automated Software Engineering.* Vasterias, Sweden.

Gendenhys, J., Dwyer, M., & Visser, W. (1012). Probabilistic symbolic execution. *International Symbosium of Software Testing and Analysis pg 166-176.*

Hunt, J. (n.d.). You've got the model-view-controller. *Planet Java* .

Kähkönen, K. (June 26, 2007). *Evaluation of Java PathFinder Symbolic Execution Extension.*

Khurshid, S., Pasareanu, C. S., & Visser, W. (2003). Generalized symbolic exectution for model checking and testing. *Lecture notese in Computer Science, 2619*, 553-568.

King, J. C. (1976). Symbolic execution and program testing . *Communications of the ACM* (pp. 385-394). ACM.

Krasner, G. E., & Pope, S. T. (Aug. 1988). A cookbook for using the model-view-controller user interface paradigm n smalltalk-80. *JOOP*, 26-49.

Ma, K. K., Phang, K. Y., Foster, J. S., & Hicks, M. (2011). Directed Symbolic Execution. *In Proceedings of the 18th International Conference on Static Analysis.* Berlin.

Minella, M. (2008). *Michael Minella.com*. Retrieved March 3, 2015, from http://www.michaelminella.com/testing/the-concept-of-mocking.html

Ottenstein, L. M., & Ottenstein, K. J. (1984). The program dependence graphi in a foftware development environment. *In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (pp. 177-184).

Păsăreanu, C. S., & Rungta, N. (2010). Symbolic PathFinder: symbolic execution of Java bytecode. *ASE*, 179-180.

Perry, D. E., & Wolf, A. L. (Oct 1992). Foundations for the study of software architecture. *SIGSOFT Software Eng. Notes, 17*(4), 40-52.

Reps, T., & Yang, W. (June 1988). *The Sematics of Program Slicing.* Computer Sciences Technical Report #777.

Seemann, M. (2004, October). Mock Objects to the Rescue! Test Your .NET Code with NMock. *MSDN Magazine*.

Slaby, J., Streicek, J., & Trtik, M. (2013). Symbiotic: Synergy of Instrumentation, Slicing, and Symbolic Execution. *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science Vol 7795*, pp. 630-632.

Tip, F. (1994). *A survey of program slicing Techniques.* Amsterdam, the Netherlands: CWI (Centre for Mathematics and Computer Science).

Uchitel, S. (2003). *LTSA-MSC: tool support for behavior model elaboration using implied scenarios.*

Vallée-Rai, R., P. Co, E. G., L. J., Lam, P., & Sundaresan, a. V. (1999). Soot - a java bytecode optimization framework. *CASCON*, 13.

Weiser, M. (1979). *Program Slices: formal, phychological, and practical investigations of an automatic program abstraction method.* Ann Arbor: University of Michigan.

Weiser, M. (1982). Programmers use slices when debugging. *Communication of the ACM*, pp. 446-452.

Weiser, M. (1984). Program Slicing. *IEEE Transactions in Software Engineering*, pp. 352-357.