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

Guowei Yang

Anne Ngu

**COPYRIGHT**

By

Colin Stuart Siroky

2015

**FAIR USE AND AUTHOR’S PERMISSION STATEMENT**

**Fair Use**

This work is protected by the Copyright Laws of the United States (Public Law 94-553, section 107). Consistent with fair use as defined in the Copyright Laws, brief quotations from this material are allowed with proper acknowledgement. Use of this material for financial gain without the author’s express written permission is not allowed.

**Duplication Permission**

As the copyright holder of this work I, Colin Stuart Siroky, authorize duplication of this work, in whole or in part, for educational or scholarly purposes only.

**ACKNOWLEDGMENTS**

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

**TABLE OF CONTENTS**

[LIST OF TABLES vi](#_Toc414458542)

[LIST OF FIGURES vii](#_Toc414458543)

[ABSTRACT viii](#_Toc414458544)

[1. INTRODUCTION 1](#_Toc414458545)

[2. RELATED WORK 3](#_Toc414458546)

[CODE SLICING 3](#_Toc414458547)

[MOCKING- CODE MODIFICATION 6](#_Toc414458548)

[3. APPROACH 8](#_Toc414458549)

[4. RESULTS AND ANAYLYSIS 12](#_Toc414458550)

[CASE STUDY: Calculator MVC 12](#_Toc414458551)

[RESULTS: Calculator MVC 14](#_Toc414458552)

[CASE STUDY: RWGUI 15](#_Toc414458553)

[RESULTS: XXXX 15](#_Toc414458554)

[5. FUTURE WORK 15](#_Toc414458555)

[6. CONCLUSION 16](#_Toc414458556)

[REFERENCES 18](#_Toc414458557)

# LIST OF TABLES

[Table 1: Calculator MVC Results 14](#_Toc414448253)

[Table 2: Calculator MVC Slicing Results 15](#_Toc414448254)

# LIST OF FIGURES

[Figure 1: Equations for determining directly relevant variables and statements 4](#_Toc414448268)

[Figure 2: Equations for determining indirectly relevant variables and statements 4](#_Toc414448269)

[Figure 3: Code example for mocking [12] 6](#_Toc414448270)

[Figure 4 : Implementation Class Diagram 9](#_Toc414448271)

[Figure 5: Code to be sliced and mocked 10](#_Toc414448272)

[Figure 6: Code after slicing and Mocking 10](#_Toc414448273)

[Figure 8: Calculator MVC 13](#_Toc414448274)

# ABSTRACT

The importance of correspondence between the architectural prescription and implementation has been long recognized. This paper 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 tin the system, backward and forward search is applied on the call graph of the system, to create a smaller runnable program by mocking out the unused methods. Two, symbolic execution is applied to check the feasibility of those potential paths and generate tests for all feasible ones to check the correspondence. We implement our approach in a prototype based on Wala, Javassist and Symbolic PathFinder (SPF), and demonstrate the usefulness of our approach using a case study.

# INTRODUCTION

The notion of software architecture has been defined by Perry and Wolf as a set of constraints on components, form and rationale [1]. 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 [2], 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 [3] 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 [4] [5].

The suggested approach is essentially a traversal of the call graph that, first, identifies the part of the call graph containing only the paths connecting an initial and final Java methods and, next, does a traversal of implementations of the methods inside that part of the call graph using symbolic execution [6] [7]. 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. [8]. 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 [8], our approach needs to traverse all possible call graph paths between initial and final methods to show there is no violation.

We implement our approach in a prototype, where we use Wala for generating the slice, Javassist to aid in the mocking of the unused code and use Symbolic PathFinder (SPF) [9] for symbolic execution. We describe a preliminary case study 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.

This paper is organized as follows: description of related work, the approach taken, case studies and results, future work and a conclusion.

# RELATED WORK

This work began with the idea of reducing the problem set to be verified to make the solution scalable and efficient. Lots of work has been done in the area of slicing. Efficient and accurate slicing is the keystone step in creating a reduction in the problem size to allow symbolic execution to work on the problem in a more efficient manner. Taking a look at the basic method for static slicing and where changes to the algorithm may be implemented to for the purpose of verification of constraints.

## CODE SLICING

From the paper by Frank Tip on a survey of slicing techniques [10] we take a general description of the approach to slicing.

Weiser’s original definition of program slicing [11] 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* → CFG *j* in the CFG:

Figure 1: Equations for determining directly relevant variables and statements

(*i*) = (*i)* ⋃ (*i*)

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 fixpoint 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 (*i*). The iteration starts with the initial values (*n*) = V, and (*m*) = 0 for any node *m ≠ 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* → CFG *j* affects the set of relevant variables at the *beginning i* of that edge. The least fixed point of this process is the set of directly relevant variables at node *i*. From , a set of *directly relevant* *statements*, , is derived. Figure 1 shows how is defined as the set of all nodes *i* that define a variable *v* that is a 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 that are indirectly relevant due to the influence they have on nodes i in . Next, the sets of *indirectly relevant variable*are determined. In addition to the variables in (*i*), (*i*) contains variables that are relevant because they have a transitive data dependence on statements in. 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 (see Figure 2). Figure 2 also shows a definition of the sets of *indirectly relevant statements* in iteration k + 1. This set consists of the nodes in together with the nodes *i* that define a variable that is -relevant to a CFG-successor *j*.

The sets and are nondecreasing subsets of the program’s variables and statements, respectively; the fixpoint of the computation of the sets constitutes the desired program slice. [10]

Modifying the slicing algorithm slightly could have a larger impact on the results. If the INFL set, the set of conditions could be passed through static symbolic evaluation it may be possible to eliminate infeasible paths and create a better answer. Without having to run symbolic execution as the verification step itself.

## MOCKING- CODE MODIFICATION

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. [12]

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 3: Code example for mocking [12]

Using the above code as an example, Figure 3, let's say that when I pass in 1234 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. [12] 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. [12]

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 thru an external means. In other words, you can't 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. [12]

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. [12]

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.

# APPROACH

The approach taken here is to use 3 tool, IBM’s Wala for the slicing, Javassist to do the bytecode manipulation and Symbolic Java Path Finder (SPF) to do the final testing. Wrappers and implementation classes had to be built around these tools. To start a test case CallGraphBuilder is called with the starting point for the slicing and the slicing options. The CallGraphBuilder calls the Slicer that is the wrapper for Wala to implement the code slicing and create a list of the methods that are contained in the slice. The list of MethodSig is then given to the ClassFileModifier that uses Javassist to modify the original bytecode and produce code that represents the slice of interest. The resulting code slice can then be run in Symbolic JavaPathfinder (SPF).

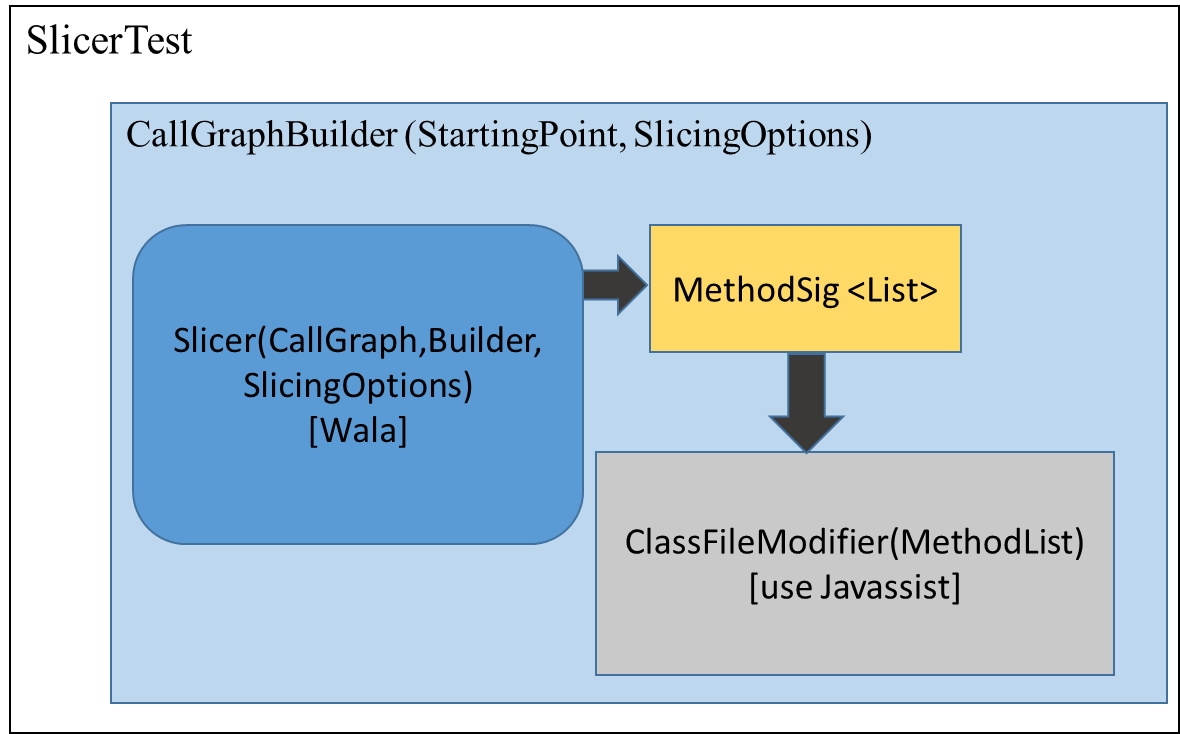


Figure 4 : Implementation Class Diagram

The implementation is done in two parts: The first part is the code analysis and modification. These steps are performed using Wala and Javassist. Example code for looking at the slice for Obj2.method2(). The slice that would be calculated is from 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 5. The final result of what would be seen after the slicing and mocking is seen in Figure 6.

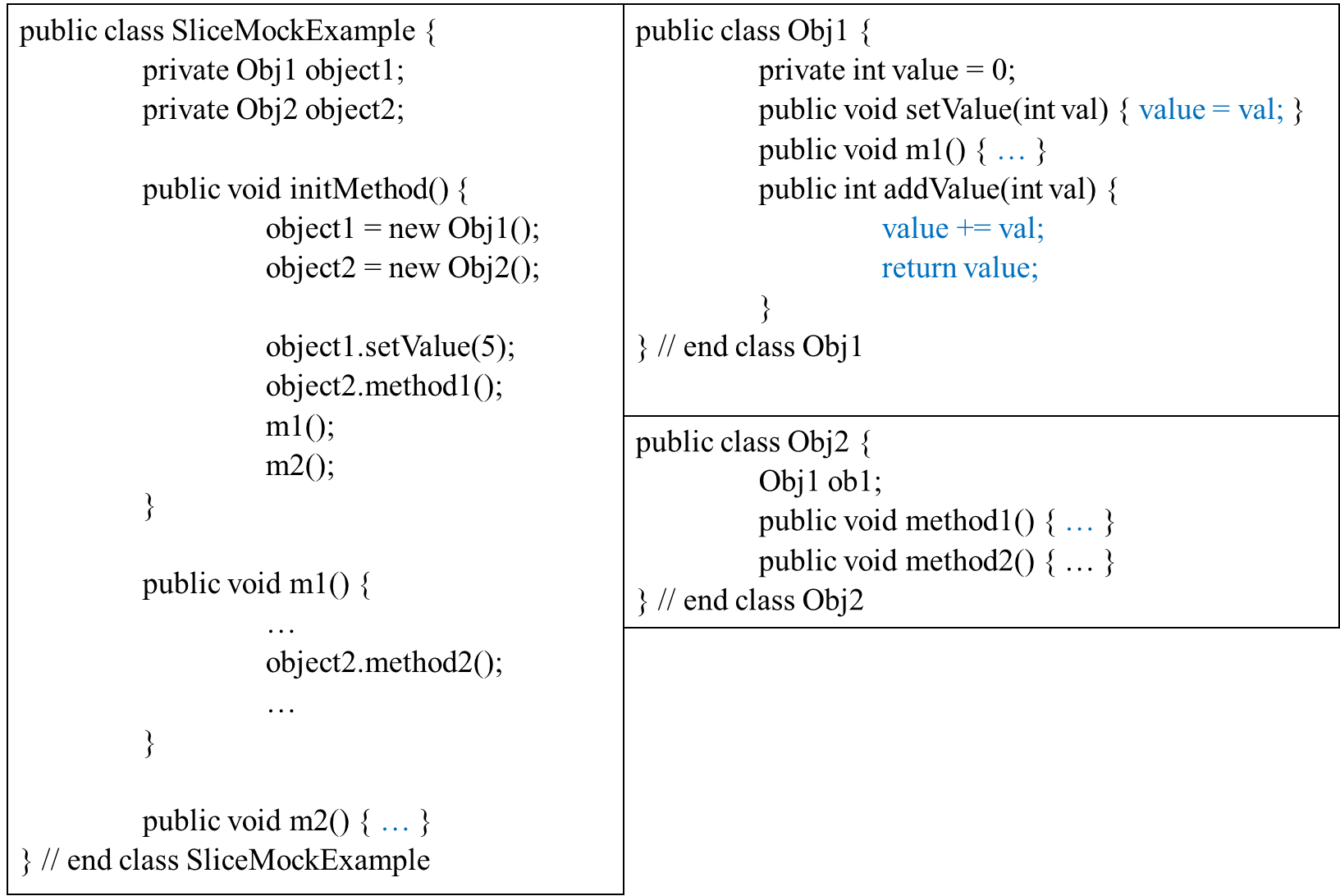


Figure 5: Code to be sliced and mocked

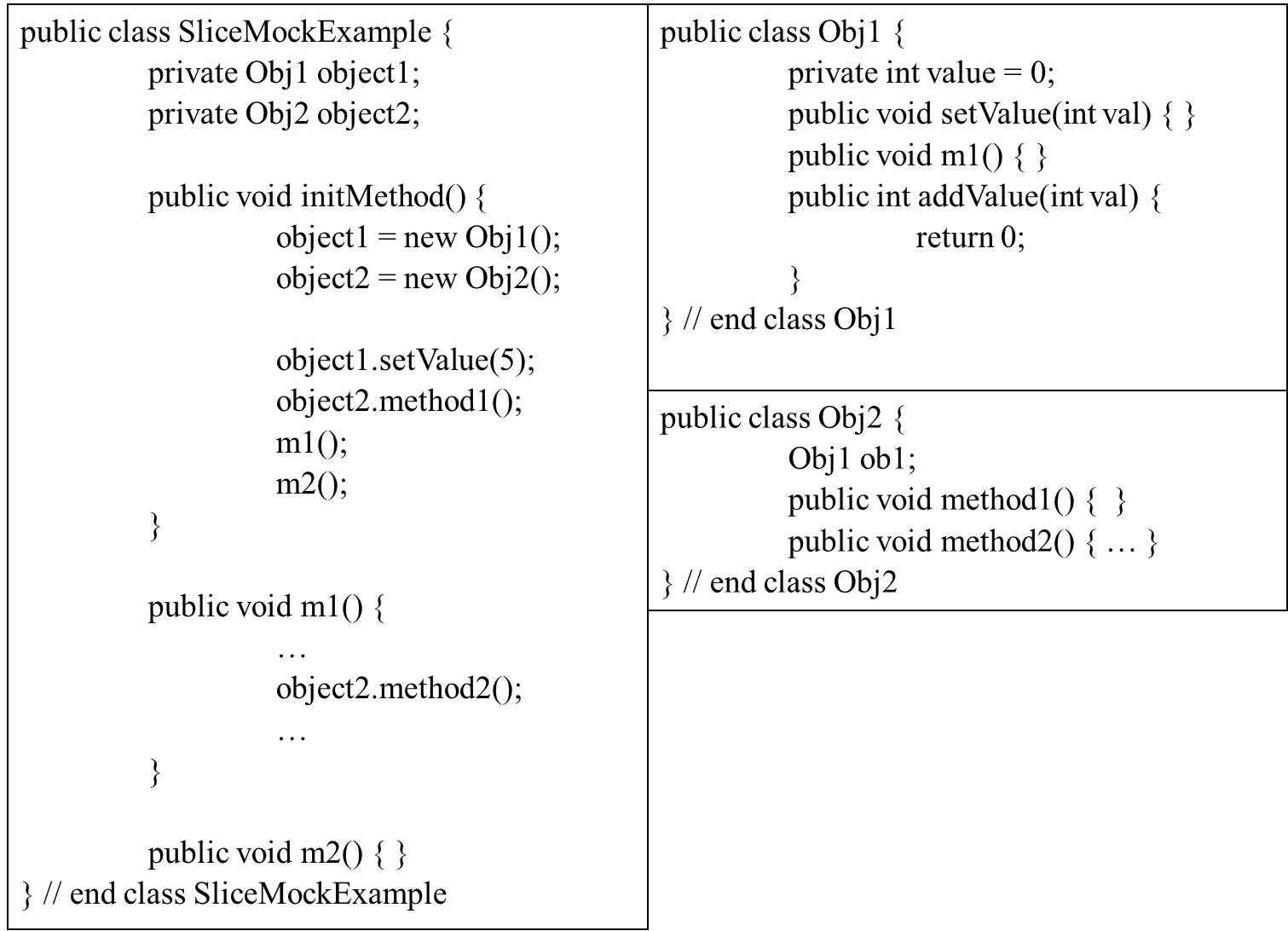


Figure 6: Code after slicing and Mocking

The code to be tested is fed into the CallGraphBuilder along with the point at which you want slicing to start. By default the slicing criteria is full data and control dependence is used to calculate the slice. The point at which you want slicing to start is the final method of interest and the point from which it is called in the protocol constraint. From here a call graph is created and fed to the slicer provided by Wala. The slicer creates a System Dependence Graph (SDG) and calculates the slice from the point of interest. The result of the slice is a set of statements, methods, branches and operations that comprise the code slice. This set of statements is then parsed to find all of the method calls. These method calls are put in a keep list for the slice. A removal list is then created from the list of all methods in the code base. Certain types of methods are additionally excluded at this point. The exclusion list can be modified but initially it contains base library methods, class constructors and abstract methods. The result is a list of methods that will be modified/removed from the code. The list of methods is then fed to the ClassFileModifier. The modifier finds the class/methods in the code and used Javassist to removes the body of the method and if the method returns a value will add an appropriate return type back into the method body. The removal of the method body effectively removed any traversal down that path, this is in the manner of mocking or stubbing a code set. [12] 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 and 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. A more precise approach would be to remove the invocation of the methods in question and the associate 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. The modified code can now be fed to the second part of the process, the verification of the architectural constraints. In order to verify the architectural constraints Symbolic Java Pathfinder is employed. A modification of the symbolic listener that comes with SPF is used in the case. The symbolic listener is watching the instructions and when it sees an invoke instruction it puts the call on a stack. When return instruction are seen it removed the call from the top of the stack. This stack is checked against the constraints and violation can be reported. The current constraint is hardcoded in the listener but use of a formal algebra would be preferred.

# 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]. 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.

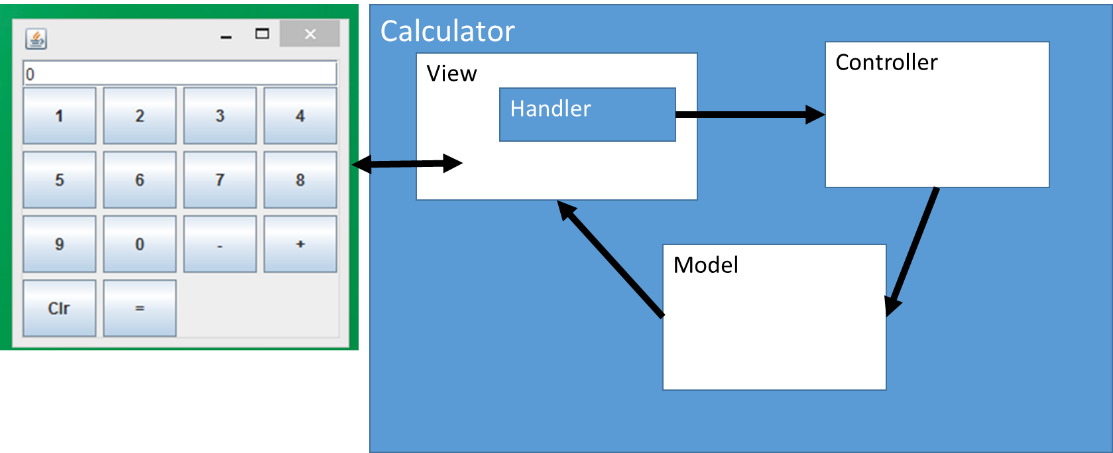


Figure 8: Calculator MVC

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 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 this architecture:

* 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 and operation if an action has not been create by the handler.
* 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 or computationally expensive path in the code that does not affect the path of interest.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 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 |

Table 1: Calculator MVC Results

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

|  |  |
| --- | --- |
| Action | Time (sec) |
| Build Call Graph | 4.37 |
| Create SDG & Slice | 7.78 |
| Modify Byte Code | 6.31 |
| Total | 12.82 |

Table 2: Calculator MVC Slicing Results

Analysis and modification of the code can be expensive but if it allows the removal of computationally complex or 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: RWGUI

The second case study networking planner 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 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.

(ADD second case study)

## RESULTS: XXXX

ADD results here

# 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 starting points, the final and initial point of interest it may be possible to further reduce the code and speed up the slicing. 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 evaluation 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 probably more important.

The modification of the bytecode could be handled in many different way. 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 speciation 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.

# CONCLUSION

In this paper we described preliminary work focused on checking architectural constraints on sequences of method invocations. We gave a description of the prototype and a case studies. The initial approach was to begin by representing the code with a call graph. The graph could then be pruned by finding all the predecessor from a final point of interest and all the successor from an initial point of interest. Doing this greatly reduced the size of the graph and so the state space. After working down this path a major flaw was discovered with this approach. The call graph representation did not have the contextual information needed. That is the dependence information, control and data, was missing. Without the missing 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 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. Slicing can solve the problem but it turned out that it was not as big of a win in efficiency and scalability as we would have liked to see. More work needs to be done in the area of slice calculation. For certain problem this approach looks promising. Problem where large amounts of computationally expensive code are removed or multi-thread application that can be removed from the path of interest both have good results. The preliminary work shows some promise but needs more work to make it mainstream.

. Slicing not as big a win. Multipoint slicing,

# REFERENCES

|  |  |
| --- | --- |
| [1] | D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Software Eng. Notes,* vol. 17, no. 4, pp. 40-52, Oct 1992. |
| [2] | R. Allen, "A Formal Approach to Software Achitecture," Issued as CMU Technical Report CMU-CS-97-144, January 1997. |
| [3] | J. Aldrich, C. Chambers and D. Notkin, "ArchJava: Connecting software architecture to implementation," in *24th International Conference on Software Engineering ICSE* , New York, NY, USA, 2002. |
| [4] | J. Hunt, "You've got the model-view-controller".*Planet Java .* |
| [5] | G. E. Krasner and S. T. Pope, "A cookbook for using the model-view-controller user interface paradigm n smalltalk-80," *JOOP,* pp. 26-49, Aug. 1988. |
| [6] | L. A. Clarke, "A program testing system," in *In Proc of the 1979 anual conference*, 1976. |
| [7] | J. C. King, "Symbolic execution and program testing," in *Communications of the ACM*, 1976. |
| [8] | K. K. Ma, K. Y. Phang, J. S. Foster and M. Hicks, "Directed Symbolic Execution," in *In Proceedings of the 18th International Conference on Static Analysis*, Berlin, 2011. |
| [9] | C. S. Păsăreanu and N. Rungta, "Symbolic PathFinder: symbolic execution of Java bytecode," *ASE,* pp. 179-180, 2010. |
| [10] | F. Tip, "A survey of program slicing Techniques," CWI (Centre for Mathematics and Computer Science), Amsterdam, the Netherlands, 1994. |
| [11] | M. Weiser, "Program Slicing," *IEEE Transactions in Software Engineering,* pp. 352-357, 1984. |
| [12] | M. Minella, "Michael Minella.com," [Online]. Available: http://www.michaelminella.com/testing/the-concept-of-mocking.html. [Accessed 3 March 2015]. |
| [13] | R. Vallée-Rai, E. G. P. Co, L. J. H. P. Lam and a. V. Sundaresan, "Soot - a java bytecode optimization framework," *CASCON,* p. 13, 1999. |
| [14] | M. Weiser, "Program Slices: formal, phychological, and practical investigations of an automatic program abstraction method," University of Michigan, Ann Arbor, 1979. |
| [15] | T. Reps and W. Yang, "The Sematics of Program Slicing," Computer Sciences Technical Report #777, June 1988. |
| [16] | M. Weiser, "Programmers use slices when debugging," *Communication of the ACM,* pp. 446-452, 1982. |
| [17] | L. M. Ottenstein and K. J. Ottenstein, "The program dependence graphi in a foftware development environment," in *In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1984. |
| [18] | P. Dinges and G. Agha, "Targeted Test Input Generation Using Symbolic-Concrete Backward Execution," in *IEEE/ACM International Conference on Automated Software Engineering*, Vasterias, Sweden, 2014. |
| [19] | H. Cui, G. Hu, J. Wu and J. Yang, "Verifying System Rules Using Rule-Directed Symbolic Execution," in *ASPLOS'13*, Houston, TX, 2013. |
| [20] | M. Seemann, "Mock Objects to the Rescue! Test Your .NET Code with NMock," *MSDN Magazine,* October 2004. |