# Verification of Architectural Constraints on Interaction Protocols Among Modules

Stuart Siroky, Rodion Podorozhny, Guowei Yang

Computer Science Department
Texas State University
San Marcos, TX 78666, USA
Email: {cs1773, rp31, gyang}@txstate.edu

*Abstract*—**The importance of adhering to an adopted architectural style throughout software development and maintenance has been long recognized. This paper introduces an approach to efficiently checking the correspondence of architectural constraints on sequences of method invocations, i.e., interaction protocols involving more than two modules. Our approach combines parameterized slicing and non-deterministic symbolic execution. Slicing produces an executable portion of the bytecode of the system under analysis relevant to the given architectural property, and symbolic execution is applied to the slice to check all paths and all interleavings for any violations of the given architectural constraints. We have implemented our approach in a prototype, where IBM WALA library is used for slicing, Javassist is used to aid in the mocking of the unused code, and Symbolic PathFinder is used for symbolic execution. Two case studies on verification of Model-View-Controller systems have demonstrated the usefulness of our approach. In particular, property guided automatic slicing, in some cases, significantly reduces the size of the input to the symbolic execution, resulting in a reduction of verification time.**

***Keywords–verification; architecture; symbolic execution; call graph.***

## I. INTRODUCTION

Software architecture was defined as a set of constraints on components, form and rationale by Perry and Wolf [1]. The importance of adhering to an adopted architectural style throughout software development and maintenance has been recognized by the software engineering community. It helps avoid architectural erosion and drift, so that the chosen architectural style continues providing its benefits and ensuring its 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 interaction 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 ArchJava 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.

In this work, we introduce an approach to checking correspondence of architectural constraints on sequences of method invocations, i.e., interaction 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 implementation of the approach for validation uses Java programming language. Thus, the analysis system processes bytecode for a Java Virtual Machine.

First, the approach uses multi-parameter slicing [6] so that to reduce the amount of the code to be analyzed. The prototype uses IBM's WALA [7] library to perform slicing. Next, mocking is performed with the help of Javassist bytecode manipulation library [8] to make the produced bytecode slice executable. Next, the approach uses symbolic execution [9], [10] via Symbolic PathFinder [11] to systematically explore all paths connecting the initial and final methods of interest so that to check if any architectural constraints are violated by the implementation.

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. In our approach, the search is directed only in the sense that the paths that do not connect the initial and final methods are not traversed. The symbolic execution traversal uses results of the slicing and mocking to explore a smaller state space. The symbolic execution is non-deterministic in relation to those variables whose values would not be fixed due to the choice of a source method. Thus, in case of a concurrent system, all interleavings in the slice would be traversed, increasing the assurance level of the analysis results.

We implement our approach in a prototype, where we use WALA [7] for calculating the slice, Javassist [8] to aid in the mocking of the code not contained in the slice and use Symbolic PathFinder (SPF) [11] for symbolic execution. Evaluation based on two case studies demonstrates the usefulness of our approach. In particular, property guided automatic slicing, in some cases, can significantly reduce the size of the input to the symbolic execution, resulting in a reduction of verification time.

The rest of the paper is organized as follows. Section II discusses related work. Section III presents our approach to verification of architectural constraints. Section IV evaluates our approach using two case studies. Section V concludes the paper with a discussion of some future work.

## II. RELATED WORK

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

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-458-9 

First is the work of Jaco Gendenhuys et al. titled "Probabilistic Symbolic Execution" [12]. 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 differs from our focus on reducing the state space 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 et al. in "Directed Symbolic Execution" [13]. In this work, the authors 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 is that they are trying to create test cases and the constraint is satisfied with a single path; while in our case, all possible paths between two points must be explored and the constraint must hold for all such paths.

The two projects 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 research.

Next, related work in the software architecture is overviewed. The most prominent initial work on a formal architectural description language (ADL) is that by R. Allen on the Wright ADL [14] 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 [15]. Another related work is by Jonathan Aldrich on a system for verifying consistency between a specification in a formal ADL and source code [3]. 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 [16]. The author creates an extension to LTSA model checker [17] by Jeff Magee and Jeff Kramer that is aimed at verifying for the lack of a deadlock, race conditions and event sequence properties based on protocols defined in sequence diagrams. The approach presented in this work 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.

## III. Approach

Our approach combines property-guided slicing and symbolic execution. The kinds of properties we focus on are constraints on interaction protocols among modules. Such constraints are often part of architectural constraints on connectors.

An architectural constraint of this kind has a source and sink method invocations. Thus, we would like to determine a slice of the program that contains paths by which the execution threads can move from the source invocation to the sink invocation. Once, given a property, such an executable slice is produced, symbolic execution with constraints on the values of variables that correspond to the property is performed. Thus, our approach starts with a traversal of the analyzed system's call graph that, first, identifies the part of the call graph containing only the paths connecting a source and sink methods and, next, does a traversal of implementations of the methods inside that part of the call graph using symbolic execution [10], [9].

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 at a later time. Thus, slicing prunes the method implementation of calls not relevant to the property. Yet, care must be taken to retain those calls on which there is either control or data dependency. Therefore, we are using a slicing implementation that constructs a proper system dependency graph (SDG). Furthermore, our approach makes sure that the obtained slice is executable so that it can be fed directly to symbolic execution. The symbolic execution traversal uses the call graph to avoid invocation of method calls that do not correspond to allowed transitions. Unlike [13], our approach needs to traverse all possible call graph paths between source and sink methods to show there is no violation.

For the initial validation of the approach we constructed a prototype that uses IBMs WALA library [7] for the slicing, Javassist [8] to aid in the bytecode manipulation when mocking the code to create an executable slice and Symbolic Path Finder (SPF) [11] to perform the symbolic execution for property verification. Even though we used WALA library, a slicing algorithm customized for this kind of architectural property that is explicitly constrained both by a source and a sink method would be more efficient. For instance, such an algorithm can be improved by concurrent traversal of the SDG starting simultaneously from the nodes that correspond to source and sink methods of a given property. These tools however could

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-458-9 

```
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 1. Mocking illustration

not be used for our prototype without modification to deliver an executable slice.

The summary of the steps of our approach is as follows. The first step is to define the architectural constraints that are of interest. The constraints will provide the source and sink methods based on 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 can then be non-deterministically processed by the symbolic execution component which applies the constraints to all feasible paths in the slice. The implementation of this tool is illustrated below.

Once the constraints have been defined it is possible to begin calculating a slice. Several inputs are fed into the slicing component. 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 call and callee methods, a system dependence graph (SDG) are created by WALA. The final node of interest is also determined in the graph using the call 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 ensuring 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 output description produced by WALA is not bytecode, nor is it easily transformed to bytecode, if at all. WALA also does not guarantee 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 the 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 (i.e., the methods whose complete implementation is needed for a slice to be executable, but that were not placed into the slice by WALA) 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 bytecode files that can be executed. The mocking step finds the classes and methods in the code to be mocked and uses Javassist to remove the bodies of the methods and fix up the bytecode to keep it executable. If a method returns a value, an appropriate return type will be inserted back into the method body. The removal of the method body effectively removes any traversal further down that path. This approach to mocking effectively stubs out the remaining irrelevant code [18].

This is a simplistic and not completely efficient way of modifying the code. One caveat of doing the code modification in this manner is that if there is code that returns an object and that object then calls some other function. i.e., `getObjA().add(xxx)`, and then if the result of the `getObjA()` method returns a null after mocking the secondary call of `null.add(xxx)` then this invocation sequence will cause a null exception. Such a situation 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 be run by any test program just like the originally complicated code. In our case, the modified code is fed to the second part of the process, the verification of the architectural constraints using Symbolic PathFinder (SPF).

We implement a Java PathFinder [19] listener to monitor invocations and returns of methods maintaining a call stack. At the point where an invoke instruction is seen in the stack, it is checked against the constraints and a violation or a pass can be reported. Currently, constraint definition is done programmatically, i.e., it is hardcoded into the listener, albeit the use of a formal temporal logic appropriate for expressing the given properties would be preferred. This is left for future work.

Here, a small example for the slicing and mocking is described and illustrated in Figure 1.

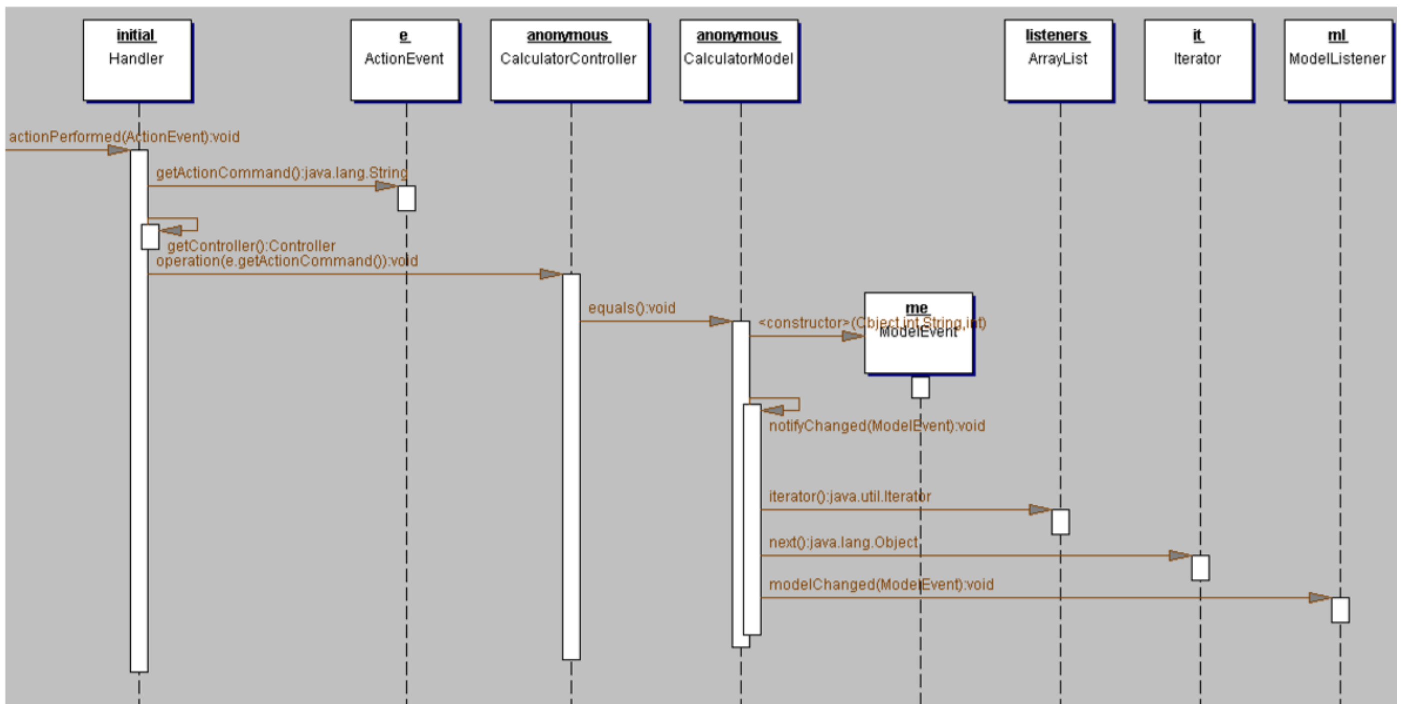


Figure 2. MVC Sequence Diagram (by John Hunt)

TABLE I. CALCULATOR MVC SYMBOLIC EXECUTION RESULTS

| Techniques | # Paths | Runtime (SS) | # States | Max Depth | Memory (MB) | # Instructions |
|---|---|---|---|---|---|---|
| Orig; no Slicing | 90 | 3 | 225 | 5 | 78 | 82,563 |
| W 2 threads; no slicing | 90 | 287 | 39,947 | 17 | 132 | 133,433,898 |
| W 2 threads; w slicing | 3 | 1 | 17 | 5 | 60 | 5,626 |

TABLE II. CALCULATOR MVC SLICING RESULTS

| Actions | Time (SS) |
|---|---|
| Build Call Graph | 4.37 |
| Create SDG & Slice | 7.78 |
| Modify Bytecode | 0.631 |
| Total | 12.82 |

In this example code, a slice is to be calculated for `Obj2.method2()`. The resulting slice should contain bytecode for the following sequence of method invocations: `SliceMockExample.initMethod()` $\rightarrow$ `SliceMockExample.m1()` $\rightarrow$ `Obj2.method2()`. The other methods will be mocked, having the code removed or replaced as necessary. The code to be removed and replaced is highlighted in blue in Figure 1. Even though the removed code in this illustration is small, a real world implementation will have more complicated and larger method implementations to be removed and mocked. There may also be many mocked methods. Thus, reduction in the amount of code to be removed from the analyzed system can be noticeable.

## IV. EVALUATION

### A. Case Study 1: MVC Calculator

For the first case study we chose a simplified implementation of a calculator that uses the Model View Controller (MVC) architectural style from [5]. The MVC architectural style can be considered to be composed out of the following design patterns: Component, Strategy, and Observer. Its implementation was simplified by replacing Swing library calls with stubs so that SPF would not execute the Swing library code. The test driver code was also added so that to mimic a scenario of user interactions with the calculator.

The constraint on an interaction protocol to be checked is due to the Strategy design pattern used by MVC.

It requires that a user gesture represented by an invocation of a method in the View should not directly invoke a method that modifies state of the Model. Instead, a View method should invoke a method in the Controller, which, in turn, should invoke a method in the Model. Thus, a Controller encapsulates strategies that map user gestures to manipulation of the Model. The Model is responsible for notifying the view of any changes. The sequence diagram in Figure 2 illustrates the event-based notification communication according to MVC [5].

Our implementation of the approach uses particular method names specific to this implementation when checking this

TABLE III. MODELCHECKCTL MVC SYMBOLIC EXECUTION RESULTS

| Techniques | # Paths | Runtime (SS) | # States | Max Depth | Memory (MB) | # Instructions |
|---|---|---|---|---|---|---|
| Orig. No Slicing | 4 | 2 | 5 | 2 | 76 | 74,851 |
| W slicing | 4 | 1 | 5 | 2 | 76 | 16,510 |

TABLE IV. MODELCHECKCTL MVC SLICING RESULTS

| Actions | Time (SS) |
|---|---|
| Build Call Graph | 3.54 |
| Create SDG & Slice | 19.8 |
| Modify Bytecode | 0.224 |
| Total | 24.22 |

constraint on interaction protocol between components. Specification of the constraint itself is currently implemented programmatically. The following constraints on interaction protocol were verified:

- 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 a method that mimics pressing a button by invoking a "button 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 "button pushed" method. The `actionPerformed` is supposed to invoke an 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 perform the previously chosen operation between two operands). The constraint requires that an `actionPerformed` method should not invoke the `CalculatorModel` methods directly.

The verification tool prototype has been applied to several variants of the Calculator application to collect performance data. The tool with both slicing and symbolic execution was run on:

- the Calculator application that does not violate the given constraints
- the Calculator application that violates the given constraints
- the Calculator application that violates the given constraints and has a potentially long running execution paths that are not of interest to the given constraints (internal concurrency via multi-threading was used)

Also, the tool with symbolic execution alone was run on the `Calculator` variants with a violation and without a violation of the constraint as a baseline.

The results of the first case study are shown in Tables I and II. The codebase for the `Calculator` contains 787 total methods in the project, 99 of which are created and not inherited from the external libraries. After the slicing, the mocking removed 77 method bodies, a reduction of $77.78\%$. The size of the original bytecode was $42.5$ KB ($43,577$ bytes), and was reduced to $29.9$ KB ($30,654$ bytes), $70.3\%$ of the original bytecode 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 four times what the original code took to verify. However, when one compares the variant of the analyzed system with internal concurrency to the sliced code, the advantage of reducing the problem is more noticeable due to a large number of interleavings removed from non-deterministic execution by SPF. There are savings in total time of verification and the number of paths explored. In this case, slicing happened to remove a code block with internal concurrency. Analysis and modification of the code can be expensive; however, if it allows the removal of computationally expensive code, the benefit can be substantial. In this experiment, the symbolic execution time was reduced from $4$ minutes and $47$ seconds ($287$ seconds) to only one second.

### *B. Case Study 2: ModelCheckCTL*

The second case study uses Computation Tree Logic (CTL) based model checker implemented in the MVC architectural style. The architectural constraints to verify are the same as for the first case study as enforced by MVC. This example is about twice as large compared to the `Calculator` example in terms of the size of the codebase. The Model part of the MVC itself has more components in that it contains classes for representation of a Kripke Structure, a CTL formula and a result representation. The `Calculator` example only contains one class in the Model part that encapsulates the calculator's current computation result and previously entered operand.

For brevity, we do not describe the implementation details for the constraint definition in this case study. The model checker under analysis provides a user with a GUI for definition of a CTL formula in a text field, choice of a file with a Kripke structure to be analyzed via a file chooser, and a text area that displays, textually, the result of model checking (i.e., whether a property holds and if not - a textual representation of a counterexample). The GUI has a number of buttons that are used to run the analysis, to clear the text areas, and to close the application. As the system was built according to MVC

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-458-9 

architectural style, the verification performed is very similar to the first case study.

The results of this experiment are shown in Tables III and IV. The codebase for the model checker contains $1,097$ total methods, $258$ 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 bytecode was $96.4$ KB ($98,802$ bytes), and was reduced to $64.1$ KB ($65,736$ bytes), $66.5\%$ of the original bytecode size. A significant amount of bytecode was removed by slicing. Also, we note that multiple properties can be verified on the slice, so even greater benefit can be achieved by performing slicing before symbolic execution.

## V. CONCLUSION

In this paper, we introduced an approach to checking architectural constraints on sequences of method invocations via combination of slicing and symbolic execution. To our knowledge, this is a novel approach to property-guided verification of architectural constraints. Our 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. It is especially effective in cases where the code deemed not needed for property verification is long running and/or contains multi-threading. In such a case, the resultant slice is noticeably smaller than the initial codebase, and furthermore, some potentially long running code not needed for property verification will be removed from the slice.

This work shows proof of concept of the suggested approach, and there are a number of directions for further improvement. First of all, a slicing algorithm that takes multiple criteria (statements for which a slice is calculated) can speed up calculation of a slice. One approach might be to compute one slice using forward traversal starting from the source method of an interaction protocol property and another one using backward traversal from the sink method of the property and then use their intersection as a resultant slice. Any node already marked by the former slicing traversal would end the slicing exploration down that path by the latter slicing traversal and vice versa. Implementation of the slicing algorithm traversal from multiple nodes of an SDG could be done concurrently, further reducing the slicing time. As future work we also intend to add a general specification of the constraints via an appropriate logic and a test case generation capability based on the path conditions created by the symbolic execution of the property-guided slice. We would like to perform a quantitative comparative analysis against similar approaches. In addition, we would like to validate the prototype by applying it to analysis of larger systems.

## REFERENCES

[1] D. E. Perry and A. L. Wolf, “Foundations for the study of software architecture,” SIGSOFT Softw. Eng. Notes, vol. 17, no. 4, Oct. 1992, pp. 40–52.
[2] R. Allen, “A formal approach to software architecture,” Ph.D. dissertation, Carnegie Mellon, School of Computer Science, January 1997, issued as CMU Technical Report CMU-CS-97-144.
[3] J. Aldrich, C. Chambers, and D. Notkin, “Archjava: Connecting software architecture to implementation,” in Proceedings of the 24th International Conference on Software Engineering, ser. ICSE ’02. New York, NY, USA: ACM, 2002, pp. 187–197.
[4] G. E. Krasner and S. T. Pope, “A cookbook for using the model-view controller user interface paradigm in smalltalk-80,” J. Object Oriented Program., vol. 1, no. 3, Aug. 1988, pp. 26–49.
[5] J. Hunt, “You’ve got the model-view-controller,” Planet Java.
[6] F. Tip, “A survey of program slicing techniques,” Journal of Programming Languages, vol. 3, March 1995, pp. 121–189.
[7] “IBM WALA,” http://wala.sourceforge.net/wiki/index.php/Main_Page.
[8] “Javassist,” http://jboss-javassist.github.io/javassist/.
[9] L. A. Clarke, “A program testing system,” in Proc. of the 1976 annual conference, ser. ACM ’76, 1976, pp. 488–491.
[10] J. C. King, “Symbolic execution and program testing,” Communications of the ACM, vol. 19, no. 7, 1976, pp. 385–394.
[11] C. S. Păsăreanu and N. Rungta, “Symbolic PathFinder: symbolic execution of Java bytecode,” in Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ser. ASE ’10, 2010, pp. 179–180.
[12] J. Geldenhuys, M. B. Dwyer, and W. Visser, “Probabilistic symbolic execution,” in Proceedings of the 2012 International Symposium on Software Testing and Analysis, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 166–176.
[13] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, “Directed symbolic execution,” in Proceedings of the 18th International Conference on Static Analysis, ser. SAS’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 95–111.
[14] R. Allen, “A Formal Approach to Software Architecture,” Carnegie Mellon University, Technical Report CMU–CS–97–144, 1997.
[15] G. J. Holzmann, “The model checker spin,” IEEE Trans. Softw. Eng., vol. 23, no. 5, May 1997, pp. 279–295.
[16] S. Uchitel, R. Chatley, J. Kramer, and J. Magee, “LTSA-MSC: Tool support for behaviour model elaboration using implied scenarios,” in Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS. Springer Verlag, 2003, pp. 597–601.
[17] J. Magee and J. Kramer, Concurrency: State Models & Java Programs. New York, NY, USA: John Wiley & Sons, Inc., 1999.
[18] “Mocking,” http://www.michaelminella.com/testing/the-concept-of-mocking.html.
[19] “Java PathFinder Tool-set,” http://babelfish.arc.nasa.gov/trac/jpf.