Verification of Architectural Constraints on Sequences of Method Invocations

Stuart Siroky
Texas State University
601 University Dr
Computer Science Department
San Marcos, TX 78666
cs1773@txstate.edu

Rodion Podorozhny
Texas State University
601 University Dr
Computer Science Department
San Marcos, TX 78666
rp31@txstate.edu

Guowei Yang
Texas State University
601 University Dr
Computer Science Department
San Marcos, TX 78666
gyang@txstate.edu

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 backward and forward search is applied on the call graph of the system, to find all potential paths between the initial method and the final method prescribed in the architecture. 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 Soot and Symbolic PathFinder (SPF), and demonstrate the usefulness of our approach using a case study.

Keywords

Verification, architecture, symbolic execution, call graph

1. INTRODUCTION

The notion of software architecture has been defined by Perry and Wolf as a set of constraints on components, form and rationale [8]. 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 ArchJava tool [1] 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 [6, 4].

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 [5, 3]. 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. [7]. The goal of their work is to generate a suit of testcases 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 connect an initial and final methods are not traversed. The symbolic execution traversal uses the call graph to avoid following method calls that do not correspond to allowed transitions. Unlike [7], 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 Soot [10] for generating call graph 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: approach is described in Section 2, a case study in Section 3, and conclusion in Section 4.

2. APPROACH

The goal of the approach is to check if none of the paths that lead from the initial method to the final method violate the given constraints on sequences of method invocations.

It is important to note that properties of this kind have source and sink methods. The source method is not invoked a second time in the transitive closure formed by potential method invocation relation. For instance, for the MVC property we checked, the source method is an actionPerformed method invoked in response to pressing a UI button. That method is not supposed to be invoked again along the method invocation sequences that originate from it.

We evaluated two variations of the approach: one uses the SPF alone to do the traversal and one that uses the static call graph.

Our goal was to use the results of the static call graph analysis to direct the non-deterministic execution by the JPF. It has not been reached yet because of the difficulties of accomplishing it with the control flow graph traversal by the JPF. It revisits methods due to its traversal and we need to distinguish between a method invocation due to revisitation or legal execution path so that not to traverse method calls that are not on paths obtained from the call graph analysis.

Hence, we performed two analyses: one with call graph analysis only and one with JPF traversal. The call graph analysis needs to use some way of checking feasibility of paths inside the control flow graphs of methods so that to determine if a violation occurs on a feasible path (i.e. avoid a spurious result). We intended to use non-deterministic execution to perform analysis only on the feasible paths.

Below we describe the call graph analysis algorithm. First, all of the paths in a call graph between initial and final methods need to be found. The initial call graph is created by Soot. This call graph contains potential transitions because the path feasibility is not checked. Next, the initial call graph is fed to a search/reduce algorithm. The algorithm is designed to prune the graph between the two methods (initial and final), reducing the size of the call graph. The result is a smaller call graph of the program execution that contains all of the paths between the initial and final methods.

The search/reduce algorithm itself consists of two phases.

The first phase in finding the reduced graph is to start from the node corresponding to the final method and do a breadth first search of the graph backwards on the potential transitions. The pseudo-code of this search is shown in Figure 1. The transitions of the call graph have already been determined, so this phase can perform a backward traversal. During this backward traversal, the visited nodes of the call graph get annotated with lists of their immediate successors. This traversal stops when the node corresponding to the initial method is encountered.

The second phase is to perform a breadth first search on the results of the first search in a forward manner, starting from the initial method. This traversal uses the lists of immediate

```
BFS_reverse(graph, final_node) {
  rev_list = get_reverse_transition_list();
  new_trans_list;
  queue.add(final_node);
  while(!queue.empty()) {
    node = q.pop();
    for(transitions_from_node) {
      if(from_node ! added)
          queue.push(from_node);
      new_trans_list.add(from_node, node);
    }
  }
  return new_trans_list;
}
```

Figure 1: Reverse BFS on call graph

```
manage_search(initial_function, final_function) {
  create_worklist();
  //Depth First Search of call graph backward
  //and forward between points of interest
  reduce_2_relevant_graph();
  manage_targets(node);
}

manage_targets(node) {
  while(!worklist.empty()) {
    fromlist = node.get_call_from();
    if(node == final_function)
      create_path(from,to);
    else if(hasPath(node)) {
      for(paths_of_node) {
        if(path_feasible(from,to))
          Update_paths(from,to,path);
      }
    }
  }
}

update_paths(from,to,topath) {
  if(has_path(from))
    paths = get_path(from);
  newpath = topath;
  newpath.addpath(from);
  forall(paths:p) {
    if(p == new path)
      contains_path = true;
  }
  if(!contain_path) {
    addnewpath(newpath);
    addtoworklist(from);
  }
}
```

Figure 2: Forward traversal of the reduced call graph

successors obtained during the backward search. Thus, this traversal is only done along those paths that will eventually lead to the final method.

In the reverse search there are two major steps:

1. Reversing the adjacency list
2. Creating a new adjacency list starting from the final node using the breadth first search

The newly created adjacency list replaces the original for the second search pass.

As this second search is performed, during a visit of each call graph node, that correspond to a method call, a check is made whether this call breaks the architectural constraint on method sequence invocation. To be able to do this check, the tool needs to get inside the implementation of the method that corresponds to the currently traversed call graph node. It is needed to build the path conditions and check feasibility of method invocations from the currently traversed call graph node.

At this time we check the sequencing properties by keeping track of a sequence of invocations of methods of interest (mentioned in the property) for each node in the call graph. On each new invocation of a method we check if the possible sequences of previous method invocations contain a required method invocation. For instance, for the MVC, an invocation of a method from the Model must contain in the possible sequences of previous method invocations an invocation to an appropriate method in a controller.

The pseudo-code for finding the feasible execution paths inside the method implementations along the paths of the reduced call graph is shown in Figure 2. This algorithm is based on directed search from [7]. It does not use the heuristics because all the execution paths inside methods encountered along the paths of the reduced call graph need to be traversed to find a violation. As it executes, the algorithm checks for violations of the constraints on sequences of method invocations. It also stores path conditions that correspond to transitions along the call graph (i.e. those paths that would invoke methods corresponding to call graph transitions). The can be used later for testcase generation along the legal call graph paths from the initial to final methods.

3. CASE STUDY

For the 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 bytecode.

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. For this case study, those paths that do not end up on the sink methods are not considered as violations because they may correspond to invocations of methods in response to pressing other buttons. An additional analysis must be made to ensure that all paths originating from the source end up on one of the allowable sink methods. 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.

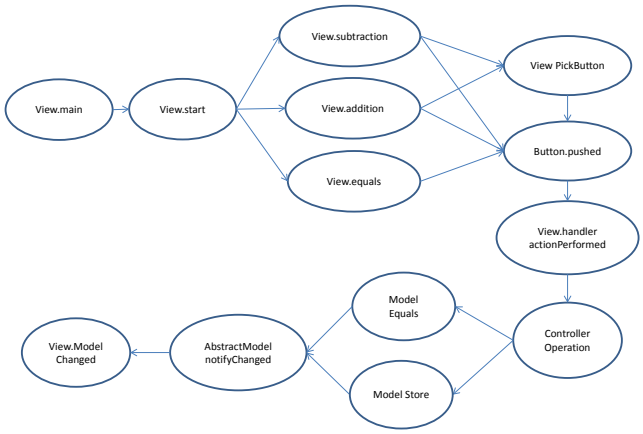

Figure 3: No violation case

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.

Figure 4 shows the case with a violation. Method *actionPerformed* bypassed the *operation* method of Controller class and called a method from the Model directly. Because invocation of the *operation* method of Controller class was bypassed there was a violation. The transition in the call graph due to a violation is highlighted in red in Figure 4.

Two cases were submitted to the analysis: a version with a violation of the constraint on sequence of method invocation and a version without a violation. In both cases there were multiple call graph paths from a given *pushed* method to a relevant method in the CalculatorModel class.

We show the paths found by the tool via manually created visualizations of transition graphs. They were created based on the output of SPF enhanced with the prototype implementation.

Figure 3 shows the case without a violation. In this case there was no path from the initial to the final method that violated the sequence constraint. All paths between the initial and final methods had an invocation of *operation* method of Controller class before the invocation of a relevant method

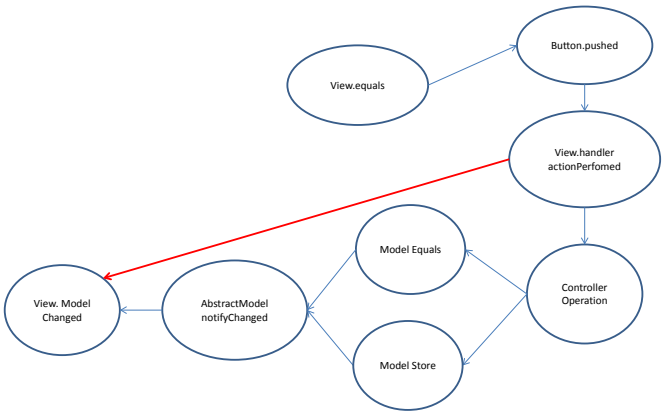

Figure 4: A case with a violation

in the CalculatorModel class. The use of the call graph reduction algorithm reduced the initial call graph from 120 nodes in the whole call graph to 13 nodes encountered all the paths between the initial and final methods.

4. CONCLUSION

In this paper we described preliminary work focused on checking architectural constraints on sequences of method invocations. We gave a brief description of the prototype and a case study. In the future we intend to add a general specification of the constraints via an appropriate logic and a testcase 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.

5. REFERENCES

[1] J. Aldrich, C. Chambers, and D. Notkin. Archjava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 187–197, New York, NY, USA, 2002. ACM.

[2] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[3] L. A. Clarke. A program testing system. In *Proc. of the 1976 annual conference*, ACM '76, pages 488–491, 1976.

[4] J. Hunt. You've got the model-view-controller. *Planet Java*.

[5] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[6] 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.*, 1(3):26–49, Aug. 1988.

[7] 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*, SAS'11, pages 95–111, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, Oct. 1992.

[9] C. S. Păsăreanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *ASE*, pages 179–180, 2010.

[10] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *CASCON*, page 13, 1999.