TestLocal: Just-In-Time Parametrized Testing of Local Variables

Marcel Heimlich
Computer Science Department
Texas Tech University
marcel.heimlich@ttu.edu

Akbar Siami Namin
Computer Science Department
Texas Tech University
akbar.namin@ttu.edu

ABSTRACT

Variables local to methods play an important role in implementing not only the logic of desired algorithms but also the control of global variables. The values of these variables dictate the paths to be taken while executing the program. While testing is an effective way to exercise the programs under test, the values of these local variables determines the reachability of execution paths. Hence, it is possible that these local variables control the execution of the program and thus may result in some paths never to be tested adequately. This paper introduces a methodology and a prototype tool, called “TestLocal”, to enable testing of methods by changing the values of local variables at runtime thus enabling traversing the paths that are hard to execute. By changing the values of local variables during the execution, the unexpected paths are intentionally forced to be executed. In comparison with the existing approaches (e.g., Microsoft IntelliTest), the proposed methodology requires little to no modifications to the given code. The novelty of the proposed approach is that it integrates a constraint solver and a debugger to forcefully execute unreachable paths.

CCS CONCEPTS

• **Software and its engineering** → **Software defect analysis;**
Software testing and debugging;

KEYWORDS

Debugging, dynamic symbolic execution, SMT solver, unit testing

ACM Reference format:

Marcel Heimlich and Akbar Siami Namin. 2019. *TestLocal: Just-In-Time Parametrized Testing of Local Variables*. In *Proceedings of The 34th ACM/SI-GAPP Symposium on Applied Computing, Limassol, Cyprus, April 8–12, 2019 (SAC ’19)*, 4 pages.
<https://doi.org/10.1145/3297280.3297613>

1 INTRODUCTION

Locally declared variables are key components of programs. These variables are usually utilized by software developers for implementing complex algorithms and controlling the execution of the program. The enforcement of program counters to forcefully proceed with certain path conditions through the values of locally declared variables is still a challenging problem.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SAC ’19, April 8–12, 2019, Limassol, Cyprus

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-5933-7/19/04.

<https://doi.org/10.1145/3297280.3297613>

The process herein helps to test these local variables by stressing the values to which they hold. With the technique introduced in this paper, conditional statements will be examined so that the conditions required in order to make the statements execute will be *forcefully* satisfied and thus it may expose possible bugs contained therein. The work is novel in that an integration of a constraint solver and a debugger is proposed in which values generated by the solver is given to a debugger to execute certain paths.

This paper introduces a Just-In-Time (JIT) methodology for testing local variables. The proposed process is very similar to parameterized unit testing. The process described herein may be used during the software development process and more specifically during white box testing in order to test the robustness of software. The proposed process will change the values of variables with inferred values or values which may cause the executions of expected and unexpected paths. These inferred values are those in which the software is unable to generate and handle by itself and which by no other means may or may not cause specific paths to be executed.

The remaining of this paper is organized as follows: Section 2 motivates the needs for testing local variables. The methodology is presented in Section 3. Section 4 presents the architecture of *TestLocal*. Section 5 concludes the paper.

2 MOTIVATIONAL EXAMPLE

During the software life cycle of a module, and over the course of many iterations, different developers make changes to the code base. If proper measures are not taken to ensure all code paths are considered then drastic results may arise. Thus we need a more automated way to check as many paths as possible, specially those paths which may never be executed. There exist techniques that allow us to check paths (e.g., symbolic execution (SE)).

In software engineering many may refer to some of these cases as “edge cases” because they are on the edge of the software process and many do not expect them to be encountered very often. Many a time, it is cost prohibitive to write software to handle each and every “edge case.” At times, these instances might be a missing *case* or a missing *else* part of an *if* statement. Structural testing, and in particular branch coverage criterion, handles the branches that are missing but for the ones that do exist and are never executed, how would one test them? Although structural testing or branch coverage may test many of the cases of a particular set of code, there are still some cases, which cannot be tested. For example:

Listing 1: Unreachable code.

```
1. int a = 0;
2. if (a != 0) { //code }
```

There is no amount of testing that can force the $a \neq 0$ branch to be executed. With the technique in this paper, the code in the unreachable branch will be executed. The value of variable ‘a’ needed,

in order to force the path to execute, will be automatically found and applied. It should be noted that the code inside the “if” statement may be very complex. Even though we can force the “if” statement to execute there is no means by which all code paths may be covered. As modern technology allows more techniques to be applied, more of the code may be covered so that we may try to attain the as yet unattainable 100% branch or path coverage we so deeply desire. At the heart of the process in this paper is a process to check paths which have remained unreachable through current techniques.

3 THE METHODOLOGY

In the proposed JIT testing methodology, the testing effort consists of the following steps:

- (1) Build, parse, and process the syntax tree;
- (2) Use an SMT solver to solve path(s) of local variables;
- (3) Use an “automated” debugger to run, break, pause, and change values of variables at runtime.

At the crux of this process is a technique given insight by modern parameterized unit testing (PUT) [7]. Parameterized unit testing is a process by which methods are tested through the introduction of parameters to a method instead of using closed methods, i.e., methods with zero parameters. Testing may then introduce values for the parameters using Dynamic Symbolic Execution (DSE) [1] to evaluate different paths that may be taken by the method.

In implementing *TestLocal*, instead of using DSE, simple Symbolic Execution (SE) [2] is used since the values of variables are determined before running the execution of a certain region of the program. In fact, the values determined will force execution of the otherwise unexecutable paths. Although the values will be determined before entering the region under test, the actual values will be assigned to the variables during runtime, possibly causing unexpected results.

3.1 Approaches for Testing Local Variables

There are two possible approaches to deal with testing local variables: 1) extending the method’s signature, and 2) re-writing the method. In this paper, we also introduce a third approach to ease testing local variables, called 3) manipulating the values at runtime.

3.1.1 Extending the Method’s Signature. An existing method’s parameter list is taken and one more parameter is added to it as a default parameter, i.e., one parameter for each local variable. Accordingly, the declaration of the local variable is moved to the parameter list of the method. If the local variable declaration includes an assignment the assignment statement is then left in place or moved into the default parameter location along with the variable. Once a method is transformed into a version where all local variables are parameters, IntelliTest, the Microsoft’s parameterized unit testing tool, may be used. The advantages of this approach is that similar techniques used in PUT and Symbolic Execution (SE), or even DSE, may be applied directly in defining the values of the variables during test execution.

3.1.2 Re-writing the Method. This process involves taking an existing method and rewriting it for each value in the test. For instance, a declaration of a local variable of “int a = 1;” will be turned into “int a = N;”, where *N* is the value to be exercised in

the test. While this way of modification is simple, in practice, it is quite tedious and laborious as every value to test will need a copy of the method. Each copy will also need to be compiled separately. For each copy made, some type of symbol for the method will need to be created so that the appropriate one will be called during testing. To overcome these problems, we introduce a novel approach called the manipulation of values of parameters at runtime.

3.1.3 Manipulating the Values at Runtime Using Debugger. This technique is proposed in this paper to ease the process. Using an *automated* debugger, a process can be made whereby the application will be paused (using breakpoints) in order to modify the value of variable(s) at runtime. This is analogous to when a user of the debugger would place a breakpoint and then when that breakpoint is hit the modification to the value of the local variable would be made. Along with the re-write method, this is a very time-consuming process but is the most straight forward approach. However, the time and labor-consuming part has been avoided through automation.

3.2 Program Constructs for Manipulations

For simplicity and ease of automation, we limit our scope to methods and the constructs within each method. There are a few simple constructs where local variables can be manipulated at runtime:

- *Conditional and Boolean Logic* (e.g., if statements)
- *Arithmetic Logic* (e.g., bitwise logic operations)
- *Method Calls*, calling a method and using the result
- *Simple Arithmetic Operations* (e.g., Addition, Multiplication)
- *Assignment*, the assignment of any of the above to a variable

The automation process can be simplified for each of these constructs to operations or assignments to a single variable. For example, in the evaluation of a simple “if” statement, we can consider the comparison of two variables such as “if (a = b)”. The same can be said of arithmetic logic, arithmetic operations, and assignment. They can all be broken down until we are left with a simple variable as part of the equation, which in our case is “a” and/or “b”. At each location in the given code where a variable is to be considered, the determined values will be assigned to the variable. For instance, the variable “a” now becomes a constant value of 2.

We are only considering simple types such as integers and Boolean. So, considering the “if” statement previously and assuming a simple type of integer, “if (a = b)” now will be: 1) “if (1 = b)”, and 2) “if (a = 1)”. This is what would deem a *single factor* application of the introduced process. If we were to extend this modification to multiple variables such as “if (1 = 2)” this would be a *two factor* approach. We can apply this approach to *N* factors where *N* is the number of local variables in the method to be considered.

3.3 The Use of An SMT Solver

In order to apply this process, we need a way to determine and generate the values to assign to the variables automatically. We use Satisfiability Modulo Theories (SMT) and more specifically an SMT solver to generate such values. SMT will be used in order to identify decidable paths in the code that are satisfiable. When we put the code paths through an SMT solver, it provides assignments for the variables which make the path condition (PC) true.

For instance, referring to the earlier example (Listing 1), there is no way to make this path condition to be true without modification of the code and more specifically without changing the assigned value of the variable *a*. SMT will be used to determine the value of the variable in order to satisfy the branch. For the example, the equation is simple when we need to solve $a \neq 0$. Thus any value that is not 0, satisfies the condition. In the simplest case, SE will be applied by determining the path by representing the code in a method as a set of logical expressions and equations. In order to force an “if” or an “else” branch to execute, the formulas of the corresponding branch need to be satisfied. Satisfying the path condition of the branch causes execution of the branch and thus provides the value needed in order to force the branch (i.e., path).

A formula (i.e., expression) will be generated up to the branch statement composed of all formulas and assignments which happened beforehand. The current formula then will be augmented with a condition of the underlying branch. When the condition is added to the existing formula, it is negated. If the resulted formula is satisfiable then the corresponding branch can be taken and the values used to satisfy the formula will be used. In addition to checking negations of branches, the SMT solver needs to apply what is called “*probing*”. This is where at each step, the formula, the condition, and the negation of the condition are applied in order to check the satisfiability of the formula.

As an illustrative example, referring to Listing 1, the value of “*a*” needs to be the value which would force the execution of the condition to be true. For our formula and in order to force the execution of the “if” statement we would have:

$$\exists a : (\sim (a = 0))$$

If there were an “else” part, then if the negation of the formula were satisfied, we would be able to determine the path for the “else”.

4 TESTLOCAL: PROTOTYPE ARCHITECTURE

A prototype tool¹, called *TestLocal*, is developed using Microsoft Visual Studio (VS) to illustrate the introduced JIT testing technique. To develop unit tests, a developer may use commonly available unit testing libraries. Due to the nature of the process contained herein, a unit testing framework was not used but instead, in order to avoid recreating a unit testing library, a simple console application test project was created. Within that project the methods to test were called out explicitly so that when the VS project was executed the test in question was performed automatically.

4.1 Build, Parse, and Process Syntax Tree

The syntax tree Microsoft Code Analysis for C# was used for the parsing purpose (i.e., “CSharpSyntaxTree”). By passing the text of the method to the object, a syntax tree of the method is returned. The syntax tree is then processed in a loop and the identified parts are recorded in order to pass corresponding elements to the Z3 solver [3] and thus find a satisfiable path for the conditions.

There are many elements of a syntax tree, but for simplicity reason, only a few elements were identified and the methods to test conformed to these constraints: 1) Variable declarations, and 2) “if” statements. Through the variable declarations, we can identify the

type of variable declared along with any initialization assignment. Declarations may take the forms of C# primitive types such as DateTime, string, and the type named object. For simplicity, the types of importance in our test were integer, floating point (limited to a single decimal), and characters.

“case” statements, as another important branching logic, can similarly be redefined as a series of if statements. To further simplify the test the left hand side of the equation must always contain the variable; whereas, the right hand side must always contain a constant rather than another variable declaration.

4.2 Z3 Solver

By processing all of the elements through the traversal of the syntax tree and before runtime, Z3 [3] elements were created. Through the binary expression of if statements, a variable and a literal were identified and then mapped to a variable and a constant expression in Z3. Together, they were then converted into an Z3 expression which were then constrained by the solver as an assertion. Finally, the Z3 solver was called upon to solve the set of constraints and for which it produced a value for the local variable of interest. The value produced by Z3 would, in turn and after being applied, force the execution of the path of interest. During the debugging step the Z3 produced value would be plugged into the variable directly before the conditional statement is executed.

4.3 Automated Debugging

The Visual Studio (VS) debugger was used for testing. Internally, a breakpoint was placed on the line of the variable declaration. The breakpoint deemed the starting breakpoint placed at the beginning of the method. It allowed the process to start and the workflow to begin. After which, a series of breakpoints were successively and automatically placed, one after another. The final breakpoint was set directly before the comparison as part of an if statement. Therefore, variable value modification could be performed. At the final step, the breakpoint of the debugger was intercepted and through the debugger the value of the declared variable was changed. This allows modification of the value of the variables and in effect modifying the path and execution of the program.

4.4 The Visual Studio Extension (VSIX)

To facilitate the processing of the test, an VS Extension (VSIX) was created. An VSIX allows for the code to become permanently embedded into VS. The VSIX allows modification of the environments’ user interface including tool windows and menus. For the purposes of this paper, the menus have been modified to provide entry points.

4.4.1 The Model-View-View-Model Framework. The framework for creating the VSIX is provided by Microsoft as a template in the VS environment. It provides the mechanics for creating the menus modifications along with the windows. Due to its simplicity in constructing user interfaces, Model-View-View-Model (MVVM) was chosen as the design approach. Windows Presentation Foundation (WPF) is used for the window, which displays the results of the process. A view model was created to encapsulate the relation between the window and the results displayed therein. In order to provide access to the model a view model locator, a *Singleton*, was made so as to provide static access to the view model. The window was

¹<https://github.com/asiamina/TestLocal>

designed using Extensible Application Markup Language (XAML). The MVVM model was then linked in with the VS debugger, which facilitates stop, start, break, and continue.

4.4.2 The Runtime Executor - The Debugger. We use a debugger to provide a runtime environment and modify the values of the variables so as to see the evaluation of the results. By using the debugger to start a program and stop right before the comparison, it would be possible for the runtime environment to be modified accordingly and the value of the local variable to be modified.

The debugger interface is provided as part of the VS Software Development Kit (VS SDK). Thus, through the debugger we can control the runtime environment. We can dictate when the debugging starts, pauses, continues, and stops. For the process in this paper, a simple workflow is considered:

- (1) Set breakpoint before comparison of local variable(s);
- (2) Start Debugging of the method in test;
- (3) Reach breakpoint and modify the value of local variable;
- (4) Continue the execution.

In the debugger, a user could place a breakpoint, push the Start button, and wait for the breakpoint to be hit. Once the breakpoint is hit the value of the variable could then be modified by the user. This is similar to what is referred to as “*changing variables while debugging the code*.” This process has been automated so that tests may be executed more efficiently and through the press of a button.

Along with the debugger command and control (start, stop, etc.), the debugger makes available access to local variables. Since the interest is in local variables they can be found on the debugger’s stack frame. For interrupting control flow, the interface to the debugger implements the “OnEnterBreakMode” event. When the debugger finds a breakpoint the OnEnterBreakMode will occur. The model in the VSIX created will process the break and information on the stack and then notify the debugger to continue.

Along with the event, the debugger notifies the code when it enters the break mode. The breakpoint is invisible to the user as when it is hit, internal code removes it and continues the execution. Thus, one of the reasons the debugger executes the OnEnterBreakMode event is because it has hit a breakpoint. The breakpoint hit causes a context switch which allows the internal code to determine if it is at the beginning of an internal workflow, in which a series of actions are created. Each action is composed of a few properties:

- *Variable*, the name of a local variable of interest, whose value is being changed at runtime.
- *Breakpoint Location*, this may be the location of the comparison of the variable in question or it may be a breakpoint before the comparison, such as the starting breakpoint.
- *Value to Assign*, the value that is assigned to the variable before the comparison operation is performed.

Actions are then put into a queue so they may be processed in a First-In-First Out order. After the initial breakpoint, all breakpoints become an integral part of the process as they signal a new action to be taken. Conversely, at every breakpoint a new action is signaled by setting the next breakpoint.

5 CONCLUSION AND FUTURE WORK

This paper introduced *TestLocal*, a methodology and a prototype tool to test local variables. The methodology introduced the integration of a constraint solver and a debugger to forcefully satisfy path conditions and thus execute certain hard to reach paths. The proposed just-in-time methodology utilizes several testing and program analysis techniques including parameterized unit testing, symbolic execution, and an SMT solver to identify the required values to test in-time local variables. The paper demonstrated that the proposed just-in-time technique has the advantages of changing the values of local variables at any time thus forcing the program execution to execute previously unreachable paths.

There are several research directions that can be explored as future work: First, the feasibility of the proposed methodology is demonstrated for conditional statements with only one single variable. In practice, more than one variable might be used in each conditional statement and thus it would be interesting to investigate the effectiveness of the proposed methodology for conditions with more than one variable. Second, it is also beneficial to investigate whether the proposed methodology can be used in improving debugging and coverage-based fault localizations [8, 9]. Third, the introduced methodology may improve mutation-based testing [10] and test case generations [11] and it would be useful to explore these applications of the methodology. Fourth, it is also interesting to observe the performance of some other solvers such as non-monotonic solvers (e.g., Answer Set Programming) and find out if there is any improvement [4, 5]. The automation of the debugger may also be enhanced through machine learning approaches [6].

ACKNOWLEDGMENT

This research is supported in part by National Science Foundation under the grant number 1821560.

REFERENCES

- [1] Thomas Ball and Jakub Daniel. 2015. Deconstructing Dynamic Symbolic Execution. In *Dependable Software Systems Engineering*. Vol. 40. IOS Press, 26–41.
- [2] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic execution for software testing in practice: preliminary assessment. *ICSE* 33 (2011), 1066–1071.
- [3] L. de Moura L and N. Bjorner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer.
- [4] Sara Sartoli and Akbar Siami Namin. 2015. Poster: Reasoning Based on Imperfect Context Data in Adaptive Security. In *37th IEEE/ACM International Conference on Software Engineering, ICSE, Volume 2*. 835–836.
- [5] Sara Sartoli and Akbar Siami Namin. 2019. Modeling Adaptive Access Control Policies Using Answer Set Programming. *International Journal of Journal of Information Security and Applications* 44 (2019), 49–63.
- [6] Sima Siami-Namini, Neda Tavakoli, and Akbar Siami Namin. 2018. A Comparison of ARIMA and LSTM in Forecasting Time Series. In *International Conference on Machine Learning and Applications (ICMLA)*, Florida, USA. 306–311.
- [7] Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. 2010. Parameterized unit testing: theory and practice. *2010 ACM/IEEE 32nd International Conference on Software Engineering 2* (2010), 483–484.
- [8] Xiaozhen Xue and Akbar Siami Namin. 2013. Measuring the Odds of Statements Being Faulty. In *Reliable Software Technologies - Ada-Europe 2013, 18th Ada-Europe International Conference on Reliable Software Technologies, Germany*. 109–126.
- [9] Xiaozhen Xue, Yulei Pang, and Akbar Siami Namin. 2014. Feature Selections for Effectively Localizing Faulty Events in GUI Applications. In *13th International Conference on Machine Learning and Applications (ICMLA)*, USA. 306–311.
- [10] Xiaozhen Xue, Sima Siami-Namini, and Akbar Siami Namin. 2018. Assessing the Effectiveness of Coverage-Based Fault Localizations Using Mutants. *International Journal of Software Engineering and Knowledge Engineering* 28, 8 (2018).
- [11] Xiaozhen Xue, Sima Siami-Namini, and Akbar Siami Namin. 2018. Testing Multi-Threaded Applications Using Answer Set Programming. *International Journal of Software Engineering and Knowledge Engineering* 28, 8 (2018).