Project Number. GFP1103

Software Agitation of a Dynamically Typed Language

A Major Qualifying Project Report:

submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by:

Austin Noto-Moniz

Date: April 26, 2012

Approved:

Professor Gary F. Pollice, Major Advisor

Abstract

Much research has been conducted on automated testing tools. Recently, Agitar Technologies developed Agitator™, which combines input generation and invariant detection into “software agitation”. Agitator analyzes code to automatically detect and report likely program invariants. However, Agitator only operates on Java, a statically typed language.

I apply Agitator’s techniques to Python, a dynamically typed language, in a tool called PyStick. PyStick implements some alternative techniques to Agitator due to the differences between Java and Python. Since PyStick is a proof of concept, it does not cover the entirety of Python. It has nonetheless generated very promising results, and demonstrates the power and viability of software agitation on a dynamically typed language.

Acknowledgements

I would like to thank Gary Pollice, my advisor for this project. He conceived the project initial idea, provided guidance as I defined the details and dealt with problems, and helped me to not take on more work than I had time for.

Table of Contents

List of Figures

List of Tables

Introduction

Research indicates that true software quality comes from the development phase, which is achieved with developer testing [1]. Correcting a bug following this phase incurs an extra cost. By some estimates, “approximately fifty percent of the elapsed time and more than fifty percent of the total cost were expended in testing the program” [2]. Thus, it is important that developers perform unit testing properly and accurately during the development phase in order to maximize the benefit of the time and minimize the money spent. However, unit testing is hard to do well, and is often executed poorly, if at all.

In an effort to shift some of the unit testing responsibility off of developers, there has been a lot of recent research on automatic testing. One of the newest techniques is known as software agitation, pioneered by Agitar Technologies in AgitarOne Agitator™ [3]. Software agitation unifies automated input generation and dynamic invariant detection into a system for detecting permissible code behavior, presented to the programmer in the form of invariants. This serves to advance testing techniques, as agitation is a more holistic and qualitative investigation of the code.

Since software agitation is such a new technology, it has yet to be applied to a statically typed language (Agitator only supports Java). My research begins this investigation with the construction of a software agitation system for Python (a dynamically typed language), which I call PyStick. Agitator serves as a direct guide, and PyStick uses its techniques wherever possible, as they are well researched. However, applying Agitator’s approach to Python raises a number of issues, most of which are related to Python’s status as a dynamically typed language. Since Python does not include type information in the source, input generation is not as straightforward as with Java. Additionally, invariants must be handled and reported using alternative methods.

PyStick necessarily attempts to resolve these issues in such a way as they can be applied to any dynamically typed language.

The paper is organized in the following way. Section 2 gives a survey of the import works in the field, including Agitator [3], DART [4] (for input generation), and Daikon [5] (for invariant detection). Section 3 discusses in depth the methods employed to construct PyStick, as well as a number of problems that I encountered and how PyStick solves them. Section 4 summarizes the results, including what was successfully implemented and some basic execution time information. Section 5 names areas of future work, which largely consists of ideas I had while coding and researching that I did not have time to implement.

Background

Agitar Technologies strongly influenced PyStick's development, as they are responsible for the development of software agitation. Their Agitator consists of two main phases: input generation and invariant detection. In this chapter, I investigate their technique for completing both phases of software agitation by looking at their research, as well as the work that served as their main influence (Daikon). Additionally, due to a lack of detailed description of their input generation, I look at a technology that they refer to as an advance in random testing which could be used to improve their algorithms, known as DART.

From Daikon to Agitator

Agitator is designed to perform software agitation on Java code [3]. The goal of software agitation is to provide the programmer with a list of program invariants for their code; that is, statements and expressions that hold no matter what input is provided. These are used to verify that a program behaves as expected in a wide variety of cases, and alert the user to the existence of bugs in the code. Although the idea of program invariants is fairly old and well established, dynamic detection in completed code is a recent development. Dynamically detecting invariants is the process of analyzing code for present invariants. The seminal work in this field, known as Daikon [5], requires the programmer to provide a test suite to exercise the code and identify these invariants; Agitator seeks to automate the process as much as possible. This automation of both invariant detection and input generation is a key benefit of software agitation.

Test data generation is a well-studied problem with many different techniques available, largely centering on static code analysis. Agitator uses a variety of methods in order to increase accuracy and scalability, using a mix of static and dynamic analysis. Dynamic analysis is an iterative process that utilizes data from previous passes in an attempt to direct execution along a particular path, while static analysis solves input value constraints to direct the next execution path. Despite this approach, Agitator does not aim for high code coverage; rather, it looks to maximize invariant detection as well as provide relatively high performance. To this end, Agitator uses some experimentation-based heuristics to take shortcuts and make intelligent guesses of data that will force code execution down alternative paths. This results in an efficient and relatively accurate implementation, although it is not complete.

Invariant detection begins with instrumenting the entry point, exit point(s), and loop headers of each target method, and logging potentially interesting values. Values associated with fields, methods, properties, parameters, expressions, and return values are all tracked. Agitator begins by generating a list of all potential relationships between different values in the code, such as a linear relationship or the result of some comparison. Then, it uses dynamic analysis over the previously generated test data to evaluate the truth of each invariant. If an invariant fails to hold even once, it is found to be false and is consequently eliminated. Once Agitator is satisfied that the code has been fully exercised, the observed invariants are presented to the programmer.

Dynamically Discovering Likely Program Invariants

Program invariants in the form of pre-conditions, post-conditions, and loop invariants are well-known in the computer science field, and are often encoded in comments. Sometimes, they take the form of assertions, which allows programmatic enforcement. However, while some invariants can be easily identified, many others are missed. If an invariant is not encoded in an assertion, it can be difficult to ensure an invariant is always respected. To that end, Michael Ernst developed Daikon, a system of dynamically detecting program invariants for reporting to the programmer.

In order to detect likely invariants, a manually created text suite was created to exercise the code and reveal invariant expressions and values. When Daikon is run, the first thing it does is instrument the program by injecting logging code at the entry and exit points of each method, as well as at the head of each loop. The provided tests are executed, and the values of variables and expressions at each instrumented point are output. This output data is then analyzed for the existence of a variety of preprogrammed variable relationships, such as constant value, range of values, linear relationships, comparisons, and many more. For each variable, Daikon initially generates and tests all possible invariants. A single failure of an invariant to hold renders it false. The likelihood of the truth of each remaining invariant is then calculated, and all invariants that are true above a certain threshold are reported to the programmer.

Invariants are calculated for all variables in scope. Additionally, in order to cover and exploit more relationships, derived variables are used. These are variables that may not explicitly appear in code, but are related to those that do; for example, if an array A appears in the code, it is relatively likely that A[0], A[1], A[-1], and A[-2] are relevant. Additionally, if a variable x appears in the same scope as A, it is conceivable that A[x] and A[x-1] are relevant. These derived variables are stored and treated as regular variables for the purposes of invariant detection.

In terms of constructing a test suite, the goal is to produce as many meaningful invariants as possible. Intuitively, a larger test suite generally means more invariants will be reported, and constraints will be tighter (e.g. an array value range of 0-10 instead of 0-20). Two methods of automatically generating input data were investigated; random testing and grammar-based testing. Random testing generated many of the core invariants, but did not catch some of the less obvious invariants that grammar-based testing did. However, grammar-based testing requires the programmer to define a grammar that defines what constitutes a valid input. That being said, random testing did find some other invariants that grammar-based testing missed, leading Ernst to favor it over grammar-based testing.

DART: Dynamic Automated Random Testing

Although known to be beneficial, testing is rarely completed due to its inherent complexity. One increasingly popular solution to this problem is the automation of testing code with minimal human input. The easiest way to do this is to implement random testing, as it is a simple yet powerful testing technique. However, its simplicity also gives rise to a number of limitations. Thus, if this process is to be automated in such a way as to produce useful results, some of these limitations must be dealt with. The Directed Automated Random Test tool (DART) does just that [4]. It uses static analysis for generating initial random test data, symbolic execution, and constraint solving, and dynamic analysis to actually evaluate the code and guide future execution down unexplored paths. DART's execution consists of three steps: interface extraction, random testing, and dynamic analysis to guide future execution.

Interface extraction is performed by statically analyzing the code for anything defined to be an interface, and that interface's type. Interface code consists of the arguments of a user-defined top-level function, as well as any referenced functions or variables that are from external code. It is at this point that instrumentation occurs. DART-instrumentation causes the program to be executed both concretely and symbolically at each interesting step (that is, an assignment or branch). This allows symbolic execution to fall back to a concrete value if it cannot completely solve a constraint, which avoids the pitfalls present in many tools based in static analysis.

Once the interface is extracted, an initial input vector is randomly calculated based on the interface types. Throughout execution, the conditional of each branch taken is tracked. After execution completes, the conditional of the last branch taken is negated, then solved. This way, the next execution is directed down a different path. If the constraint solver cannot exactly solve the constraint, the current concrete value of the constraint is substituted, and the execution is flagged as incompletely solved. Thus, if an error arises, it can be assumed to be due to the incomplete solution, and a random input vector can be generated to force execution of the last unexplored branch.

Using this analysis to explore every conceivable path through the function, DART attempts to detect points in the code that cause an abnormal execution abortion. They claim their mixed system allows a more accurate evaluation of the code, as DART can default to real variable values in the case of hitting an unsolvable constraint, allowing both symbolic and concrete evaluation to continue.

Methodology

Since PyStick's inspiration was Agitator [3], many of the decisions I made were an attempt to imitate their work. PyStick focuses on the software agitation technologies employed by Agitator, while ignoring its features for codify invariants. These additional features are unnecessary for implementing software agitation, and thus do not further the goal of this project. They were omitted with no decrease in the demonstrative power of this prototype.

The approach to implementation of software agitation in PyStick mirrors that of Agitator. There are two core phases: input generation and invariant detection. PyStick aims to implement these two phases using the same technologies as Agitator, since their approach has proven quite successful. Due to time constraints and some lack of detail, PyStick's actual implementation of each phase is inspired by Agitator, but is indeed different.

There is no detailed description of how Agitator accomplishes input generation, leading to a discrepancy between its approach and PyStick's. The one allusion to Agitator's technique seems to imply it uses its own system based off of existing automated test input generation techniques, such as "…symbolic execution, constraint solving, heuristic- and feedback-driven random input generation, and human input specification". Unable to copy their system, I instead base input generation off DART [4]. DART requires limited user interaction by making use of an intelligent random testing approach. Automation is a goal of PyStick, which makes DART a more attractive candidate. For these reasons, it is the basis for PyStick's input generation phase.

The Daikon tool [5] serves as the core influence and inspiration for Agitator. Daikon is a seminal work concerning dynamic invariant detection. As such, it is also the basis for PyStick's invariant detection phase. The invariants supported by PyStick all come from Daikon, except for the ones concerning variable types. Although there is a great amount of detail about each invariant, there is no discussion as to the actual mechanism Daikon uses to detect invariants. There is discussion of where the values are stored and what constitutes an invariant, but not how one is discovered. Some reasonable assumptions of their approach can be made, but the downside is that PyStick cannot leverage all of Daikon's techniques. Nonetheless, Daikon is a tremendous resource, both as an insight into Agitator's operation and invariant detection.

The next sections contain a detailed discussion of my implementation and any differences that exist between it and each technology used.

Python

PyStick agitates Python, which is a prime candidate because it is a popular, dynamically typed language, and my personal language of choice. Thus, it was easy to write examples and effective tests for PyStick.

After much research on the systems I needed to implement, I decided to write PyStick in Python. Python is my strongest language, which makes it the obvious choice, but that alone is not enough to justify its use. Since PyStick agitates Python code, writing it in Python greatly reduced complexity for a number of reasons.

This decision means PyStick can easily execute the code it is analyzing. This greatly reduces the work necessary to implement dynamic analysis, a technique employed by both DART and Daikon. Python includes a *compile* function and an *exec* statement, which together allow a Python program to be compiled and executed from within Python code. PyStick figures out what it needs to dynamically analyze, and also track the state of the environment. As these are already required by DART and Daikon, PyStick does not have to do any extra work to ensure dynamic analysis works.

Due to the extensiveness and power of the Python built-in modules, PyStick is almost entirely free of third-party libraries (python-constraint [6] is used for constraint solving). As such, PyStick has access to a number of modules designed to operate on Python source code. One such module is Python's *ast* module, which PyStick heavily utilizes to parse a valid Python program into an abstract syntax tree (AST), which is used for both code instrumentation and symbolic execution. Both of these are done with relative ease at the source code level due to the power of the *ast* module.

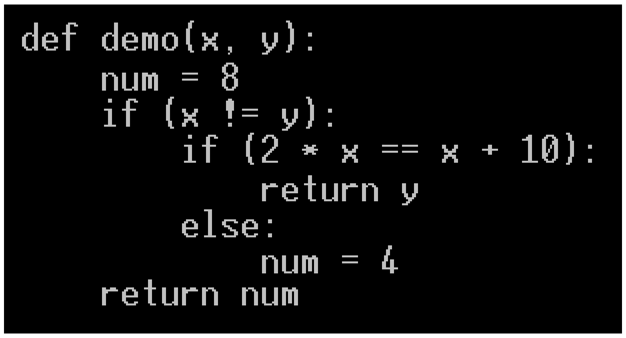
Combining both of these benefits, Python's *compile* function has the ability to compile an AST directly. This eliminates the cumbersome step of turning the AST back into source code, and once again allows PyStick a great deal of flexibility and power by letting Python do all the hard work.

Implementation

PyStick is implemented in Python 2.7, and agitates programs that use a subset of Python 2.7 features. The subset it operates on is relatively small, but it is sufficient to prove that software agitation of a dynamically typed language is feasible using an approach that is similar to Agitator.

PyStick only agitates one function at a time, and that function cannot make any function calls. The file itself may have multiple functions, but the user must tell PyStick which function to agitate. In addition, the function must be written using only the operations supported by PyStick. Currently, they are: assignment, compound assignment, delete statements, if-elif-else statements, while loops, print statements, and return statements. Arithmetic and boolean expressions are also supported. Due to time constraints, there is no support for try-except statements or for-loops, as there are some complexities to input generation with both constructs.

The other large restriction on PyStick is that it can only handle integers. This decision simplifies implementing agitation while preserving the spirit of software agitation. It does not allow for full use of the power and flexibility of a dynamically typed language, but this system may be extended to other types in the future. For now, it only has to show that software agitation can be performed on a dynamically typed language, not that it can handle every aspect of the language. Thus, although this restriction does prevent PyStick from being used on many real world programs, it does not detract from PyStick's goal.



```
def demo(x, y):
    num = 8
    if (x != y):
        if (2 * x == x + 10):
            return y
        else:
            num = 4
    return num
```

Figure 1: A function written for PyStick , using only constructs that it can handle.

PyStick is structured with both subsystems (input generation and invariant detection) as separate, independent pieces. Both systems are capable of operating completely independently from the PyStick tool, and could be provided as separate tools from PyStick. This allows for the greatest modularity and flexibility within PyStick, as its operation does not depend on how either subsystem operates. PyStick simply calls each in turn, stores the output, and hands this output to the next system.

Input Generation

PyStick's input generation is largely influenced by DART, although a number of liberties were taken for various reasons. DART is written for C and operates on both C code and the corresponding assembly code, necessitating a number of adaptations in order to make it suitable for a source code level tool in Python. As such, the resulting tool is not truly an implementation of DART for Python. Rather, it makes use of many principles set forth by DART, but goes about these principles in a very different way.

The first departure occurs in interface extraction, DART's first step. Part of interface extraction is determining the types associated with each interface to the program. As PyStick deals with a dynamically typed language, type information cannot be easily extracted. This combined with time constraints results in PyStick only considering the function provided by the user. Additionally, it does not attempt to extract any type information from this function. Types are dealt with during input vector seeding.

The biggest difference between PyStick and DART is the timing of concrete execution. DART performs concrete and symbolic execution simultaneously. It operates on the level of assembly code and symbolically executes instructions that it cares about, namely branches and assignments. The code is stepped through address-by-address, and advanced to the labeled address in the case of a jump. DART also manually tracks the environment state both concretely and symbolically, and uses this to handle all variable references. If concrete execution is necessary, the instruction in question is simply executed with the currently stored state. It is a fairly simple and elegant system for execution.

That same system is difficult to setup and execute properly at Python's source code level. The ability of assembly code to execute arbitrary instructions with no regard to context makes switching between symbolic and concrete execution fairly trivial. The source code level inability to ignore context is why this same approach cannot be easily implemented in Python. In order to execute code, Python expects a full, syntactically correct piece of code, complete with an environment that contains all variables referenced. As such, PyStick departs in mechanic from DART at this point. It performs concrete and symbolic execution in two distinct steps.

The general idea with this approach is to first execute the code concretely in order to build up a dictionary containing a snapshot of the environment at each line. The symbolic execution phase follows a similar algorithm to DART's system, but instead of performing any further concrete execution, it simply makes use of the dictionary provided to it.

Concrete Execution

Along with executing the code concretely, PyStick must manually log the environment state. It must do so in a way that not only is suitable for use during symbolic execution, but one that takes Python's somewhat unorthodox scoping rules into account. This is no easy task. It requires extensive instrumentation and code injection.

The function under analysis is first parsed into an AST using Python's built-in *ast* module. Handling the code in this format makes it much easier to analyze as related units since related code will be children of the appropriate parent node of the tree. Additionally, an AST is quite easy to reliably modify in such a way as to produce a legal representation of the code. Finally, the *ast* module provides the *NodeTraversal* class, whose sole purpose is to ease AST traversal (through the visitor pattern) and modification (through modifying return values of the desired node type).

PyStick's implementation of DART defines a subclass of *NodeTraversal* in order to perform all necessary instrumentation and code injection. Code injection is used to prepend the provided program with a number of utility functions. These functions perform various tasks related to propagating scope information in the state dictionary. This is needed because PyStick only updates the state when something changes, such as on an assignment or looping (since iterations are tracked). It does not make sense to update at every line since the state does not often change in a significant way. Updating the state only when it changes requires the same amount of operations, but allows multiple operations to be performed at once.

The other major piece of instrumentation that these functions handle is loop instrumentation. Due to the necessity of tracking the environment state during every iteration, each loop must be instrumented to include an iteration counter, and the state must be associated with the appropriate iteration number. Some of the utility functions PyStick injects serve to ease the transition through loops.

A single dictionary used to track the state is injected at the very beginning of the program. It is this dictionary that is updated to properly track the program state at every line and loop iteration. The utility functions yet again aid with this feature, as they properly initialize the state for every logging operation.

Once all this instrumentation has taken place, the AST is compiled and executed, the state dictionary is extracted from the environment, and the concrete execution step returns this state dictionary for use in the symbolic execution system.

Symbolic Execution

PyStick's approach to symbolic execution is very similar to DART's. The end goal of the symbolic execution phase is to produce a list of interesting input vectors for the function, grouped by which path they will execute. However, despite these similarities, a number of liberties were taken in order to account for the differing approach of DART and PyStick.

To begin, the process is seeded with a randomly generated input vector. This step demonstrates how a dynamically typed language must be handled differently than a statically typed language. DART ensures that all randomly generated inputs are of the appropriate type.

However, PyStick does not know the correct type of input. My intention is for the input generation process to be run for each combination of input types, and invariants will be logged and reported based on the input types involved. Due to time constraints, the prototype does not address this issue, and PyStick can only handle integer inputs. The code is set up such that this could be implemented, but the result of such a process is left for future researchers to discover.

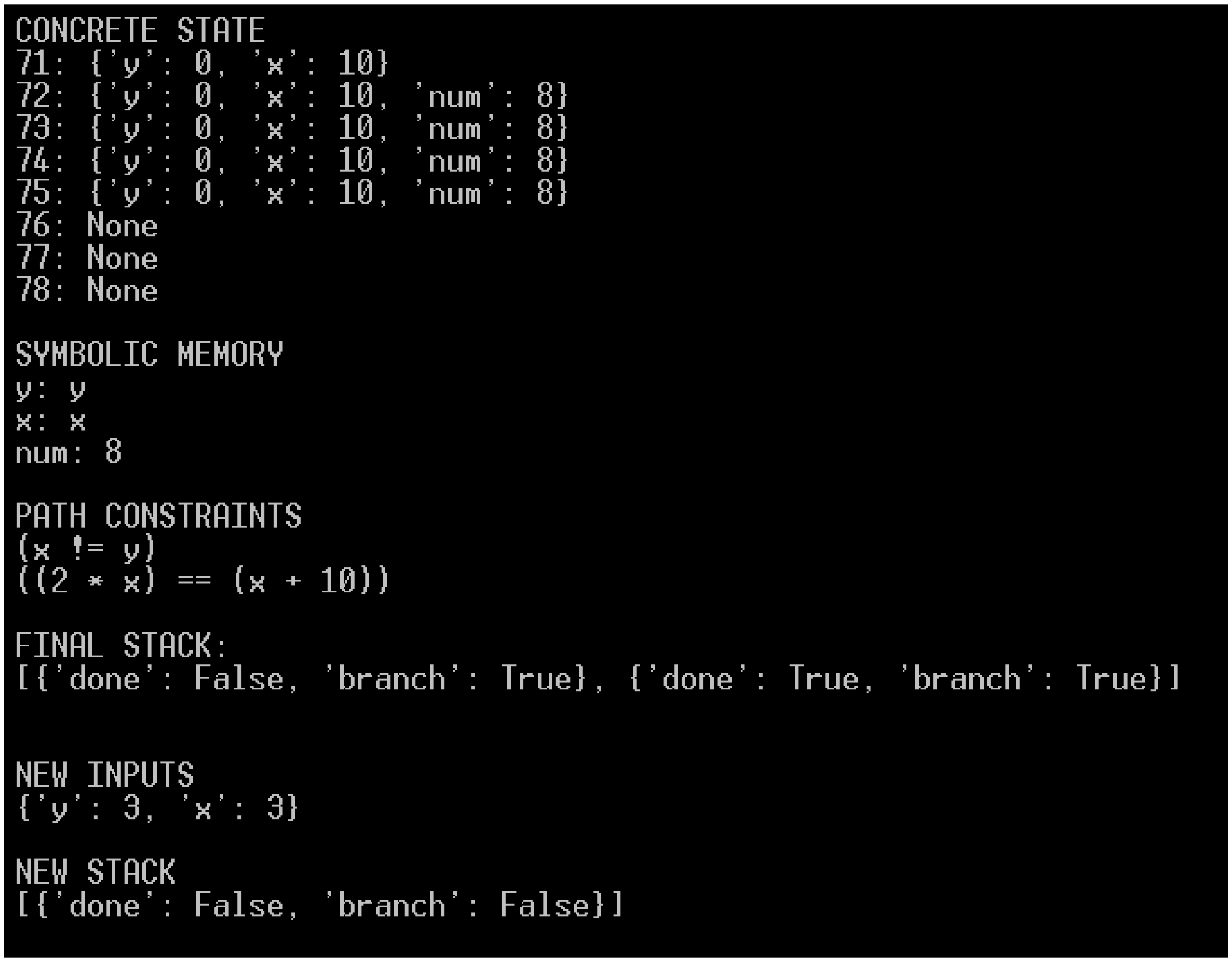
Next, a loop is entered which symbolically executes the program. PyStick replaces DART's assembly level analysis with traversing an AST. At each interesting node (assignments, branches, and loops), the expression is executed symbolically and all necessary logging takes place. Apart from use of an AST, PyStick adheres to DART's established execution scheme, which involves tracking the symbolic state and logging each branch condition that evaluates to true. Both of these pieces of information serve to direct the path taken by the next execution of the code.

Branch conditions are logged to track the path executed by a given input vector, and ensure a different path is taken on subsequent executions. Each branch executed is pushed onto a stack which will log the condition, which branch was taken (true or false), and whether both branches have been executed. All this information is used in generating the system of constraints to be solved which will result in the next input vector.

Symbolic state is different from concrete state in that if an expression is assigned to a variable, the variable's value is literally that expression, not the expression's value. This allows PyStick to capture a situation where a local variable is dependent on the value of a function argument. As such, the input generation system is quite powerful and expressive, as it can handle complex situations (such as variable aliasing) with no extra work. The symbolic state forms part of the system of constraints that will determine valid inputs for the next execution.

When generating the next input vector, branches are popped off the condition stack until one that is not yet complete (where one branch has not been executed) is discovered. The next input vector will be generated so as to force execution of the other branch of this incomplete condition. To make this happen, the flag indicating which branch of this condition was taken is flipped. For example, if the condition last evaluated to true, the flag is flipped to false. All branch conditions on the stack (including this modified condition) are combined with the symbolic state to form a system of constraints on the next input vector. A constraint solver is then run to generate a solution space that forces execution of an unexecuted branch. In this way, every semantically reachable path through the code is executed.

The DART tool was intended to detect defects in the code for various reasons, and was not designed to log all values that could lead program execution down a certain path. Since PyStick aims to determine program invariants, just traversing every path is not enough. Rather, it must traverse every path as many times as possible in order to gather enough data to accurately infer invariants. While DART discards all solutions not seeding the next execution, PyStick logs all solutions provided by the constraint solver. This is not the entire solution space (as the actual solution space is infinite), but captures all relevant solutions within a user defined range. Note that the solutions are grouped by which path they solve, and returned this way. As such, the system that accepts this input could run a heuristic to pick the optimal set of inputs. It is up to the implementation to do what it needs.



```
CONCRETE STATE
71: {'y': 0, 'x': 10}
72: {'y': 0, 'x': 10, 'num': 8}
73: {'y': 0, 'x': 10, 'num': 8}
74: {'y': 0, 'x': 10, 'num': 8}
75: {'y': 0, 'x': 10, 'num': 8}
76: None
77: None
78: None

SYMBOLIC MEMORY
y: y
x: x
num: 8

PATH CONSTRAINTS
(x != y)
((2 * x) == (x + 10))

FINAL STACK:
[{'done': False, 'branch': True}, {'done': True, 'branch': True}]


NEW INPUTS
{'y': 3, 'x': 3}

NEW STACK
[{'done': False, 'branch': False}]
```

Figure 2: An input generation step. The concrete state is shown at the top, followed by the symbolic memory state, path constraints, and state of the stack. Finally, after solving the constraints, we see the new input vector, and the new stack.

Invariant Detection

Daikon was the inspiration for Agitator, and forms the foundation for PyStick's invariant detection phase. Daikon is written in Java, and does not operate on any specific programming language. Its input is a text file containing information about variable values across by multiple executions of the code using different input vectors. This information comes from the interesting points in the code, defined by Daikon as the function's entry point, exit point(s), and loop headers. The authors specify a format for this text file, which allows anyone to write a front end for Daikon by ensuring their application outputs a file in the correct format. At the time of the original research, they used a Lisp front end for testing, which I will refer to as Daikon's front end. As of this writing, there exist official front ends for Java, C/C++, Perl, Eiffel, and IOA.

While PyStick does not directly use Daikon, it attempts to embody Daikon's approach and extend its established research into the realm of dynamically typed languages. All invariants detected by PyStick come directly from Daikon, except for one, which is exclusive to dynamically typed languages. Code is instrumented at interesting points and then executed, which logs the actual values of each variable. The logged values are then analyzed for invariants, which are printed to the screen. Details follow concerning how this process was completed, as well as if (and how) PyStick's approach differs from Daikon's.

Instrumentation for Logging

Logging variable values is achieved through static analysis, code instrumentation, and concrete execution. Instrumentation forms the core of the logging phase since it allows the values of each variable in scope to be stored at all interesting points. While Daikon's front end also uses instrumentation to log variable values, the key difference is in the storage method.

Daikon's front end writes its logging information to a file, which Daikon uses as input. While this allows invariant detection to be completed independent of which programming language was used, it makes for many writes to disk, which greatly slows execution. To avoid this problem, PyStick stores all variable information in memory, namely a dictionary. This dictionary is then passed to the invariant detection phase. While this binds PyStick's invariant detection to Python, the goal was not to create a language independent system. To make it language independent, this dictionary could be used to output a file in Daikon's format following execution. However, this is left for future research.

PyStick's instrumentation of the code for invariant detection is not very complex. The injected code simply serves to log variable values at interesting points in the function. Thus, each chunk of code injected simply logs all variables in scope according to which kind of point is being logged (entry, exit, or loop) and at which line number this point begins. This injection is achieved through static analysis. The source code is parsed into an AST, which is then traversed to find interesting points. If a node represents an interesting point, then a series of logging statements is injected to log the values of all variables in scope. At all exit points (signified by a return statement), the return type is also logged. Nodes that constitute interesting points are functions, while loops, for loops, and return statements. Try-except statements could also be considered interesting, but introduce a number of problems that must be solved. As they are not necessary for this prototype's goal, they were omitted. Additionally, if the end of a function is semantically reachable, then it is also interesting, as a Python function will implicitly return "None" in this situation.

While not much code gets output, a lot happens during static analysis. Namely, PyStick tracks the scope (which is no small task), and determines if the end of the function is an interesting point.

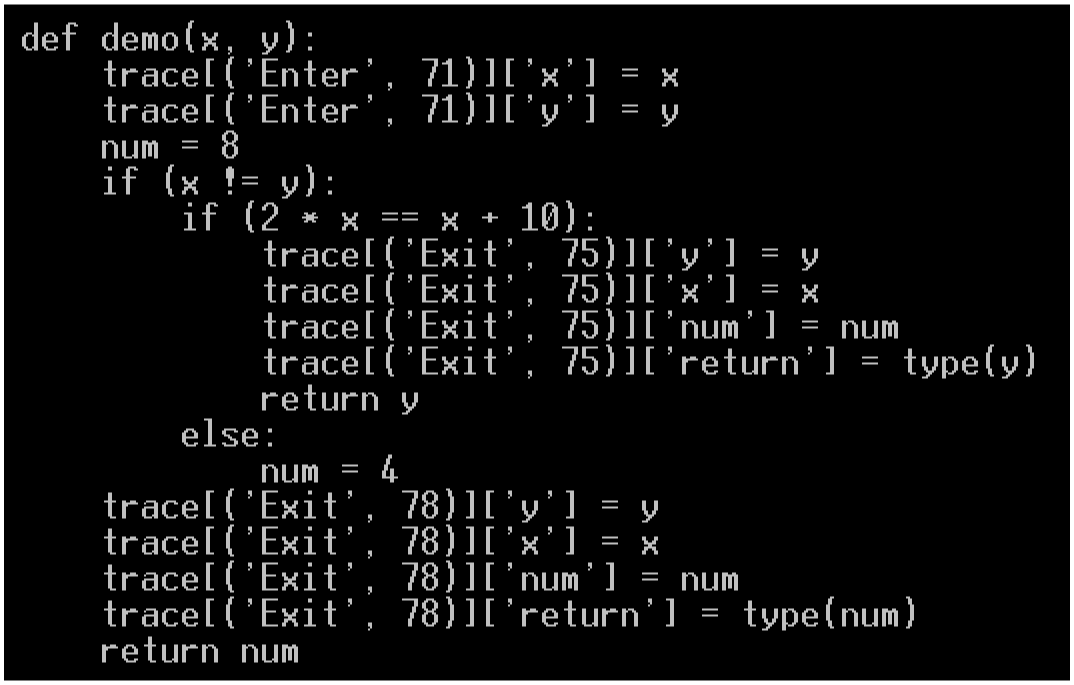
Note that since all logging statements are statically injected into the code, PyStick requires a method of knowing which variables are in scope at each point. Furthermore, it cannot query the program's current state for this information; since code is injected during static analysis, the program is not running, and is thus without state. To solve this problem, PyStick tracks the variables in scope during AST traversal. Any time a variable is created or deleted, the list of variables in scope is updated. Supported control-flow constructs are also analyzed and have their scope tracked in case they contain loops or return statements, which are points of interest, and thus require knowledge of variables in scope. The constructs supported for this purpose include if-elif-else blocks, try-except blocks, with blocks, while loops, and for loops[1].

In many languages, each code block is given its own scope. Thus, the scope that existed before the block's execution is restored upon completion. However, Python's scoping rules do not work this way. In Python, control-flow blocks do *not* get their own scope. Any variables defined within a control-flow block are added directly to the scope of the enclosing function. While this makes the need to reset the scope unnecessary, it introduces problems with logging. If a variable is defined within a block of code that is never executed, but that variable is then referenced later in the code, Python will raise a NameError. This indicates the variable does not currently exist, which is true because its definition was never executed. During static analysis, PyStick cannot know which blocks will be executed, so it cannot know which variables will exist. As such, PyStick must utilize an alternative method for determining this information.

The solution employed by PyStick is to defer the decision to runtime. To do this, upon exiting a control-flow block, any variable defined within the block is marked as a "scoped" variable. When emitting a scoped variable, extra code is first emitted to check if that variable exists in the current scope. If it does, its value is logged. Otherwise, it is logged as "void" at that point. The "void" value is a custom value in PyStick, used to denote a lack of information. Its use lets the user know that this variable could have been defined, but was not. In this manner, Python's unorthodox scoping rules are successfully taken into account, and all scoping information is correctly propagated.

[1] You may notice that some of the "supported" blocks are not supported by PyStick itself due to their absence from the input generation phase. Since invariant detection is independent from input generation, I wrote some of the logging code before I knew exactly what input generation would cover. Once I knew, I saw no reason to remove this code, as it may aid future work based off PyStick.

Static analysis is also responsible for the ability to detect that the end of the function is an interesting point, which is only true if it is semantically reachable. Whenever a return statement is discovered, a flag is set. If the return statement is contained within a control-flow block, the node representing this block has the ability to examine the flag and determine if the block itself returns. A block is said to return if all paths through the block result in a return statement being executed. For example, in the case of an if-else statement, if both the if statement and the else statement return a value, then the if-else statement is said to return. If either one (or both) does not return, then the if-else statement does not return. If a block is determined to return before the bottom of the function is reached, then the bottom of the function is semantically unreachable an uninteresting. Otherwise, it is treated as an exit point, and logging information is emitted. A return type of "void" is logged in order to distinguish an explicit return type of None from the implicit return type of None.



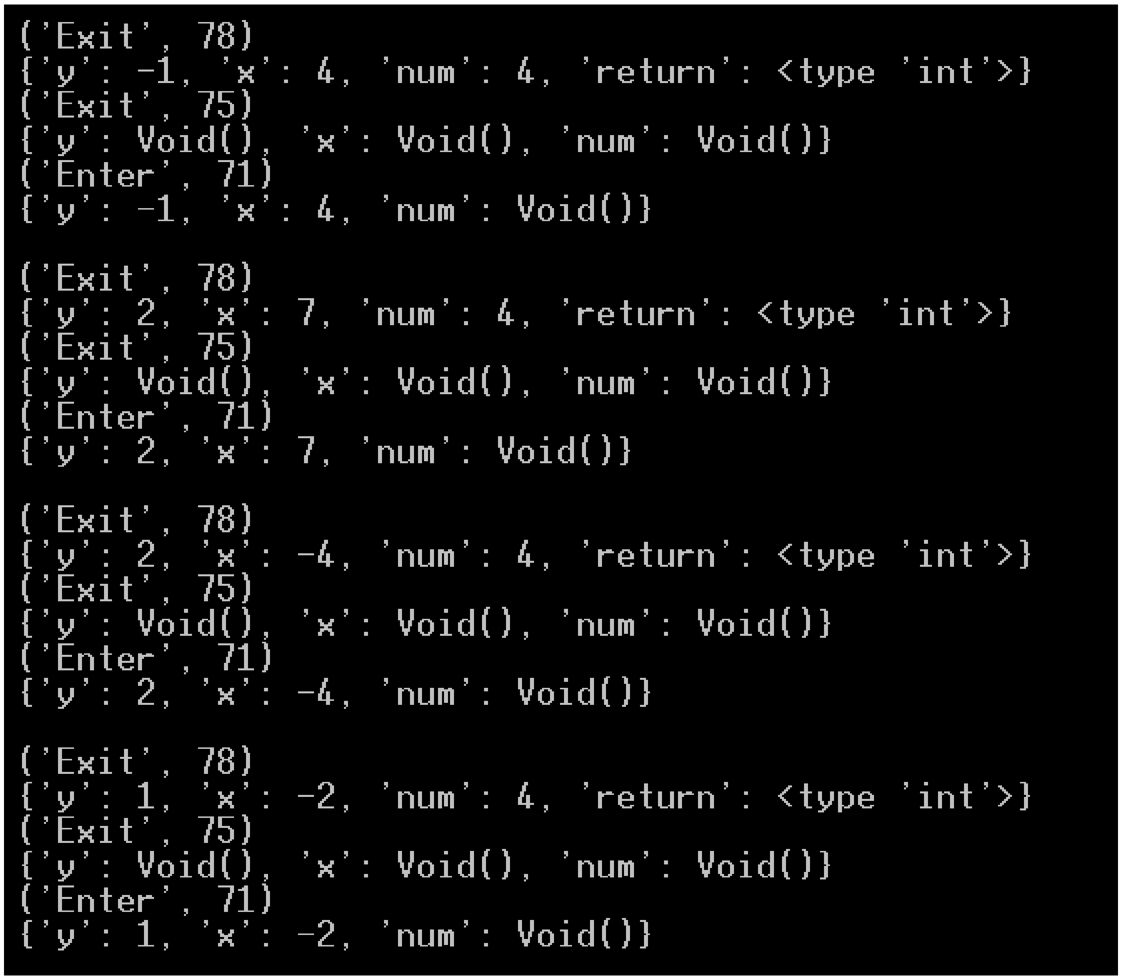
```
def demo(x, y):
    trace[('Enter', 71)]['x'] = x
    trace[('Enter', 71)]['y'] = y
    num = 8
    if (x != y):
        if (2 * x == x + 10):
            trace[('Exit', 75)]['y'] = y
            trace[('Exit', 75)]['x'] = x
            trace[('Exit', 75)]['num'] = num
            trace[('Exit', 75)]['return'] = type(y)
            return y
        else:
            num = 4
    trace[('Exit', 78)]['y'] = y
    trace[('Exit', 78)]['x'] = x
    trace[('Exit', 78)]['num'] = num
    trace[('Exit', 78)]['return'] = type(num)
    return num
```

Figure 3: Code instrumented for logging values for invariant detection.

Execution

Execution is straightforward concrete execution, with one slight twist in order to improve efficiency. The instrumented AST is only generated once: before any execution occurs. Then, to execute the instrumented code with the desired inputs, code is injected that calls the function with the appropriate inputs. For each input vector, this line is changed to reflect the appropriate values.

Other than this little piece of code, execution is simple. It compiles the given AST with the *compile* function and executes it with the *exec* statement. After execution, the produced logging dictionary is extracted and returned.



```
('Exit', 78)
{'y': -1, 'x': 4, 'num': 4, 'return': <type 'int'>}
('Exit', 75)
{'y': Void(), 'x': Void(), 'num': Void()}
('Enter', 71)
{'y': -1, 'x': 4, 'num': Void()}

('Exit', 78)
{'y': 2, 'x': 7, 'num': 4, 'return': <type 'int'>}
('Exit', 75)
{'y': Void(), 'x': Void(), 'num': Void()}
('Enter', 71)
{'y': 2, 'x': 7, 'num': Void()}

('Exit', 78)
{'y': 2, 'x': -4, 'num': 4, 'return': <type 'int'>}
('Exit', 75)
{'y': Void(), 'x': Void(), 'num': Void()}
('Enter', 71)
{'y': 2, 'x': -4, 'num': Void()}

('Exit', 78)
{'y': 1, 'x': -2, 'num': 4, 'return': <type 'int'>}
('Exit', 75)
{'y': Void(), 'x': Void(), 'num': Void()}
('Enter', 71)
{'y': 1, 'x': -2, 'num': Void()}
```

Figure 4: A small sample of logged values, organized by interesting point.

Invariant Detection

PyStick currently detects five different classes of invariants: range, order, unary functions, binary functions, and return type. The first four come directly from Daikon. They were chosen and implemented because they can be applied to integer values. The return type invariant was implemented as a desired and useful addition for a dynamically typed language.

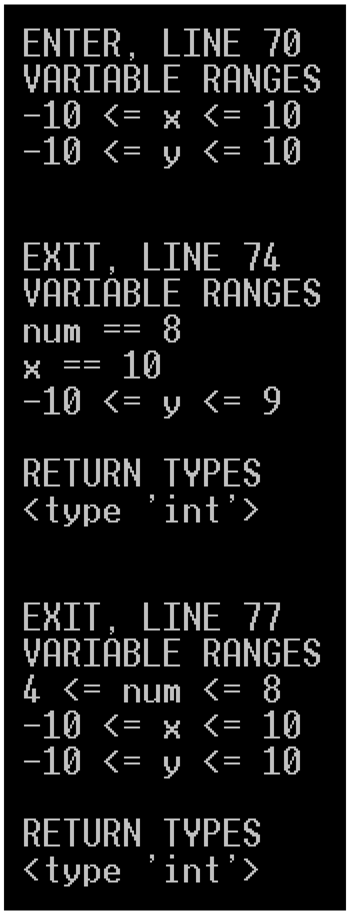
The general approach for detecting each invariant is the same. Each variable present in a section is considered in turn. All values of the variable under analysis are analyzed in order to determine if the property holds. This information is conveyed back to the invariant detection system, which stores it in memory. I will now discuss each invariant and rationale.

The range invariant is one of the most useful ones Daikon provides. PyStick marks each variable one of three ways: ranged, constant, or undefined. A ranged variable's value falls within some range. A constant variable's minimum and maximum values are equivalent. An undefined variable is either marked as "void" in this section or does not contain an integer. Narrowing the possible values of a variable is invaluable information for debugging errors, as it has the ability to exclude certain variables whose type could be causing the error but who cannot take illegal values.

The order invariant is useful in determining how integer variables relate to each other, such as cases where one variable constrains the values of another. A variable can be related to another variable in one of six ways: equal (==), not equal (!=), less than (<), less than or equal to (<=), greater than (>), or greater than or equal to (>=). Note that since a relation must hold in all cases, it is possible for two variables to not be related by any of these operators. Also, two variables will never be reported to be related in more than one way. If two relations apply, then the tighter constraint will be reported. For example, if two variables are related by both less than and less than or equal to, then the reported relationship will be less than.

The unary and binary function invariants are closely related. This invariant applies a unary or binary function to one or two variables (respectively) and checks the result. If the result is always equal to another variable, or if the result is constant, an invariant has been detected. All functions checked are built-in to the language. PyStick checks ten built-in functions, which make up all that take integer inputs. Seven functions are unary, of which only one returns an integer. Of the three that are binary, all return integers. This invariant serves to further the search for a relationship between variables. Additionally, it may imply a source for the value contained in the variable, which could be helpful.

The final invariant is the only one not from Daikon. Looking for an invariant among return types can be immensely helpful for those programming in a dynamically typed language. There is nothing to prevent a programmer from accidentally coding a function to return different types in different situations. Thus, this invariant gives the coder an understanding of all types that may be returned at each exit point from their function. Also note that if the function implicitly returns a None (that is, it exits without a return statement), PyStick will report that return type as “void”. In this way, programmers can differentiate between the two cases and ensure proper one is occurring.



```
ENTER, LINE 70
VARIABLE RANGES
-10 <= x <= 10
-10 <= y <= 10


EXIT, LINE 74
VARIABLE RANGES
num == 8
x == 10
-10 <= y <= 9

RETURN TYPES
<type 'int'>


EXIT, LINE 77
VARIABLE RANGES
4 <= num <= 8
-10 <= x <= 10
-10 <= y <= 10

RETURN TYPES
<type 'int'>
```

Figure 5: Sample PyStick output.

Results and Analysis

The goal of this project was to build a software agitation system to operate on a dynamically typed language using the same principles embodied by Agitator. PyStick took certain liberties to account for the differences between Java and Python, and the lack of detail about Agitator's internal operation. These differences have been detailed in the previous section. Though these differences make PyStick interesting and relevant, they only matter if PyStick achieves its goal. In this section, I evaluate PyStick's success.

Completeness

The first measure of success is PyStick's level of completion. The goal was to implement as much of the system as necessary to show Agitator's approach to software agitation is feasible for a dynamically typed language. PyStick supports one data type and six different language features, which is sufficient to achieve this goal.

PyStick only supports integers, which is all that is required to demonstrate success. Automated input generation is a well-studied problem, so it will not provide any significant obstacles to the system. However, it does prevent PyStick from taking advantage of invariants relating to the input of differing input types. While this does not detract from its success, it implies that PyStick is not reaching its full potential. This consideration is left to future research.

There are a few coding structures that PyStick must handle to be viable as a proof of concept. Assignment forms the basis of software agitation, and was implemented first. Compound assignment and deletion did not require much more work, and increased PyStick's Python coverage. Return statements are another important feature for software agitation, and were implemented following assignment. All this was straightforward and did not take much time. Control structures were more difficult to implement.

Branches are important to software agitation, since local variables will often be modified by a branch. There were a couple difficulties I encountered implementing static analysis. During code instrumentation, I had to make modifications to persisting code scope and marking unexecuted lines.

Loops are a core piece of software agitation, as loop headers are defined as important. For this reason, PyStick can handle while loops. Unfortunately, as implementing loops is a non-trivial problem due to their iterative nature, PyStick cannot handle for loops.

Although for loops are unsupported, the successful inclusion of branches and while loops demonstrate the feasibility of agitating a dynamically typed language.

Testing

Ideally, PyStick's success would be evaluated by running it on arbitrary code. However, since it handles a small subset of the Python language, this is not feasible. Even the simple functions put forth by Daikon are outside of PyStick's range, as they require contain arrays. To this end, I evaluated PyStick's success through the use of two different test suites. The informal test suite is a collection of small functions using the features PyStick can support, while the formal test suite evaluates each PyStick supported structure in isolation.

The informal test suite consists of fifteen tests. All tests in this suite were intended for personal use during development. PyStick consists of four main pieces: concrete execution, symbolic execution, logging, and invariant detection. These pieces were developed in isolation as much as possible, and as such required a method of verifying their operation independent of any other system. I wrote some small functions to exercise each PyStick feature I added support for. They were not carefully constructed, and do not cover any special cases. Rather, they exercise the basic operation of a number of features within PyStick. They are informal, and behave much like an arbitrary piece of code. Many of these functions are somewhat complex, or exercise many different features at once. In this way, they differ from rigorous tests.

The formal test suite consists of nineteen tests, and is a more traditional style test suite. It provides a number of very small test functions, each with a very specific purpose. Each feature supported by PyStick has its own dedicated test. Special cases are also exercised, such as parameter-less functions. These were all written following the completion of development, since it was not until then that I knew which constructs PyStick could handle.

Each test suite is a collection of functions, and each has a test runner. Both runners operate in the same manner. They run PyStick on each function in the test suite multiple times, defined by the user. On each run, standard out and standard error are written to file for later inspection. Additionally, execution of each test is timed and then averaged across each run to give a rough evaluation of performance. By default, each test is run five times.

In order to evaluate the success of each test, I inspected the output and compared it to the invariants I expected. Any discrepancy required closer investigation. First, I checked to ensure the expected result was correct. If it was, the next step was to further investigate the code and correct the bug. A number of bugs were resolved in this manner.

Timing

In addition to verifying correctness, these tests perform a basic and informal analysis of execution time. I was surprised with the speed that PyStick appears to operate at. Since I paid little attention to efficiency while coding, I was expecting that it would operate quite slowly in all but the simplest of cases. While it is not fast, these basic tests imply that PyStick's speed is decent.

One run of the test sweet reported the average execution time for a test in the formal test suite was about 0.53 seconds. All times were between 0.12 seconds and 2.72 seconds. Although the tests in the formal test suite are quite small (the longest is six lines) with a maximum of three variables, it was unexpectedly fast. This shows that PyStick works very well on short programs that are only a few lines in length.

The informal test suite experienced more interesting results. The average execution time was about 3.43 seconds, with all tests falling between 0.14 seconds and 27.71 seconds. At first glance, this shows a must slower trend than the formal test suite. However, the maximum time function involves the execution of every control structure PyStick supports, including a loop which executed nine times. If we exclude this function from the data, the average drops to about 1.7 seconds per test, with a maximum duration of 5.43 seconds. Although there remains much room for improvement, I was expecting much slower results.

One possible explanation is that my test functions are not complex enough to see the slowdown I was expecting. The execution time of my most complex function may show that. However, further testing is needed to confirm that this is indeed true.

A telling trend is the tendency for execution time to increase with the number of function parameters. From the formal test suite, the average execution time for the fifteen functions with only one parameter was 0.23 seconds. Meanwhile, the average execution time for the three functions with two parameters was 2.15 seconds. The same trend exists in the informal test suite. Functions without parameters average 0.14 seconds; single parameter functions take about 0.33

seconds; and two parameter functions take about 2.18 seconds. This implies that much of PyStick's time is spent executing the generated input vectors. As such, it may make sense to research ways to select inputs in such a way that the input vector set is minimized, but the coverage does not suffer.

In watching an execution of the complex test function, I noticed the majority of its time is spent attempting to solve path constraints, even though there are only two. The reason seems to be a fault of the way PyStick interacts with its constraint solver. At that point in the code, there are five live variables. All five are passed to the constraint solver as needed for solving, even though only two of those variables appear in the constraints. This was lazy coding and can easily be fixed, which would improve the efficiency in cases such as thus.

This test function may also imply that PyStick does not handle loops very well. The biggest slow down on this test occurs when the loop is executed, so it may be beneficial to further investigate the input generation to force the proper execution of different paths through loops.

Future Work

The product of this research is a prototype representing an area lacking in extensive research. As such, there are many opportunities for further work.

Little attention was paid to the efficiency of PyStick's code during development, leading to much room for improvement. The initial prototype traverses a function's AST three times: twice to instrument the code (for input generation and invariant detection), and once to symbolically execute it. This is a waste, especially since the traversals largely focus on the same aspects of the code. The traversal from the input generation phase and the invariant detection phases could be blended to reduce this redundancy.

The other primary concern of efficiency exists when logging variable values. Storing the values in memory and calculating invariants following execution works well for small programs. However, large programs may significantly slow down execution and inflate memory usage. This is due to a likely increase in the number of variables and interesting points. Daikon's approach is to write all logged values to a file. Then, to calculate invariants, it simply reads from that file. Although this also works, the authors have expressed concern over the decreased efficiency from the many file operations required. One possible solution to both these problems would be to calculate invariants on-the-fly. Since an invariant must hold true across every execution, simply tracking the current state of the invariant and updating it on each execution would be enough to ensure its correctness. This technique would need to be evaluated for effectiveness in increasing performance, as it currently exists only as a proposition.

PyStick supports a very small subset of the Python language. For this tool to realize its full potential, it must be able to handle much more of the language, with the final goal being the ability to handle the entirety of Python. It would not be necessary for PyStick to agitate every aspect of Python. The ability to process and skip over constructs it cannot handle would be sufficient to allow PyStick to analyze real-world code.

In its current form, PyStick cannot handle function calls, whether user-defined, built-in, or third party. It is a difficult problem to solve, made even more difficult by the lack of type information concerning the return type. Further research could be put into how to handle these three different classes of functions in order to enable PyStick to effectively analyze them. Two proposals come from my own work: on-the-fly invariant detection, and agitating the callee in order to determine input and return types.

PyStick also must be extended to operate on more data types. While integers are sufficient to imply that software agitation of Python is feasible, the tool is not currently useful for arbitrary code. PyStick should at least be able to handle Python's most commonly used built-in types, such as floating-point numbers, strings, booleans, lists, tuples, sets, and dictionaries. Adding more types to PyStick necessitates the handling of a few related issues.

The most straightforward of these issues will be the need to add more invariants. This is likely the easiest to solve and implement. Since PyStick's purpose is invariant detection, an inability to calculate invariants on a given type renders the inclusion of that type useless. Included in this is the task of enabling the invariant detector to ensure all values of a variable at a given interesting point are the same type. Additionally, the invariant detector could look for invariants only on variables of a type that matches the invariant. However, it may be interesting to see what invariants appear when variables of unexpected types are analyzed.

On-the-fly invariant calculation has some problems of its own, but it is an interesting idea due to the problems it could solve. The inclusion of more types is another problem it could aid. Calculating type invariant on-the-fly would have benefits for input generation, even if no other invariants were detected at the same time. These type invariants could be used to determine legal input types for the function being analyzed. This information could be used to ensure input generation is only performed for legal types.

Analysis and detection of invariants would also require some further decisions, namely how to organize these invariants with differing input types. It may not make sense to analyze invariants over the entire program without considering input type. Since only one input type will apply to each program execution, combining all of them will likely not be the optimal configuration for the programmer.

Techniques for automatic object generation may also be of interest to automate as much of the software agitation process as possible. Agitator often requires the use of factory classes for object creation. While this works well, it is more work on the part of the programmer. The ideal situation would be to completely remove the need for the programmer to do anything extra besides run the program.

As it currently stands, PyStick is merely an interesting research project. However, it has the potential to become a very useful tool for diagnosing issues that exist in Python code. Before it reaches that stage, much more research and work must occur, beginning with the areas laid out above.

Conclusion

The development of PyStick investigated the feasibility of applying software agitation to a dynamically typed language. Specifically, it evaluated the suitability of the approach used by Agitator for this purpose. It addressed some issues that arise when dealing with a dynamically typed language in this manner, and laid the groundwork for even more. In addition, it is intended to serve as the foundation for further research into this understudied area, and will hopefully act as a starting point for future research on this topic.

Although there was only time to apply this approach to a small subset of the Python language, its success implies that taking the Agitator approach to software agitation of a dynamically typed language is feasible. The success of development and application to arbitrary functions (using PyStick's supported features) shows that the system works correctly in all situations it is designed for. A more complete implementation is required in order to make a more complete assessment of the viability, but the initial work looks promising.

References

[1] Petschenik, N. (1985). Building awareness of system testing issues. *Proceedings of the 8th international conference on Software engineering* (pp. 182–188). IEEE Computer Society Press. Retrieved from http://dl.acm.org/citation.cfm?id=319622

[2] Myers, G. J., Sandler, C., & Badgett, T. (2011). *The Art of Software Testing* (p. 256).

[3] Boshernitsan, M., Doong, R., & Savoia, A. (2006). From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing. *ISSTA '06 Proceedings of the 2006 international symposium on Software testing and analysis* (pp. 169-179). Retrieved from http://portal.acm.org/citation.cfm?id=1146258

[4] Godefroid, P., Klarlund, N., & Sen, K. (2005). DART: Directed Automated Random Testing. *ACM Sigplan Notices* (Vol. 40, pp. 213–223). ACM. Retrieved from http://portal.acm.org/citation.cfm?id=1065036

[5] Ernst, M. D., Cockrell, J., Griswold, W. G., & Notkin, D. (2001). Dynamically Discovering Likely Program Invariants to Support Program Evolution. *Software Engineering, IEEE Transactions on*, *27*(2), 99–123. IEEE. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=908957

[6] python-constraint. http://labix.org/python-constraint

Appendix A: Test execution time

Table 1: Formal test execution (seconds)

	RUN 1	RUN 2	RUN 3	RUN 4	RUN 5	AVERAGE
test_assign_const	2.062883	0.173008	0.164737	0.165630	0.192040	**0.551660**
test_assign_expr	0.196885	0.204549	0.197228	0.196789	0.169783	**0.193047**
test_compound_add	0.198192	0.167378	0.177526	0.183870	0.173630	**0.180119**
test_compound_div	0.222598	0.208199	0.188096	0.205139	0.173262	**0.199459**
test_compound_mul	0.176685	0.203896	0.214953	0.184246	0.168472	**0.189651**
test_compound_sub	0.181380	0.194607	0.173252	0.199067	0.197671	**0.189196**
test_delete	0.156428	0.156547	0.152528	0.151552	0.159574	**0.155326**
test_eval_bool	1.657985	1.605594	1.673990	1.758288	1.633656	**1.665903**
test_ex	2.105477	1.884806	2.026968	2.123109	2.312203	**2.090513**
test_if	0.228716	0.249631	0.231156	0.279056	0.236461	**0.245004**
test_no_args	0.130358	0.113249	0.117369	0.126785	0.122779	**0.122108**
test_no_control	0.193162	0.182320	0.181568	0.178608	0.188983	**0.184928**
test_return_from_if	0.222802	0.218475	0.236591	0.217440	0.227621	**0.224586**
test_return_from_while	0.226921	0.224346	0.227636	0.263562	0.234645	**0.235422**
test_return_implicit	0.166882	0.169341	0.171560	0.166274	0.171349	**0.169081**
test_return_no_control	0.158779	0.158272	0.160755	0.155392	0.169300	**0.160500**
test_return_uneven	0.253552	0.214449	0.229881	0.211830	0.222037	**0.226350**
test_two_args	2.591316	2.635859	2.515194	2.664483	3.204716	**2.722314**
test_while	0.419260	0.327074	0.412145	0.439893	0.477731	**0.415221**
AVERAGE	**0.607909**	**0.489032**	**0.497533**	**0.519527**	**0.549259**	**0.532652**

Table 2: Informal test execution (seconds)

	RUN 1	RUN 2	RUN 3	RUN 4	RUN 5	AVERAGE
branching	1.851488	1.831767	1.908082	1.765576	1.646559	**1.800694**
branching_more	0.144265	0.160410	0.146485	0.132162	0.144068	**0.145478**
complete_branching	2.237624	2.084089	2.183840	2.085307	1.873404	**2.092853**
dart21	3.034892	2.816095	3.033769	2.891061	2.750769	**2.905317**
dart24	2.054509	2.029064	2.119138	1.952037	1.877094	**2.006368**
dart24_extended	1.873900	1.995767	1.909370	1.747660	1.758448	**1.857029**
dead_return	0.130102	0.124767	0.153896	0.146210	0.131097	**0.137214**
demo	2.740241	2.810190	2.754288	2.580650	2.502033	**2.677480**
everything	33.771165	33.948558	34.261485	4.490436	32.082941	**27.710917**
func	0.123746	0.137106	0.139074	0.133404	0.146156	**0.135897**
func2	0.417451	0.501758	0.479422	0.487692	0.519400	**0.481145**
loop	0.195978	0.207671	0.176826	0.183905	0.197414	**0.192359**
loop_2_params	2.051417	2.053822	1.983989	1.813325	1.862365	**1.952984**
param_types	1.959999	1.953787	2.013656	1.796314	1.828702	**1.910492**
param_types_branching	7.965827	4.034993	4.029345	7.338196	3.822643	**5.438201**
AVERAGE	**4.036840**	**3.779323**	**3.819511**	**1.969596**	**3.542873**	**3.429629**

Appendix B: Package Structure

pystick\				**1511 lines, 55 functions, 21 classes, 103 methods**
	__init__.py			
	commonast.py			**46 lines, 5 classes, 5 methods**
	main.py			**16 lines, 1 function**
	report.py			**64 lines, 6 functions**
	util.py			**64 lines, 5 functions, 2 classes, 5 methods**
	daikon\			**489 lines, 23 functions, 7 classes, 32 methods**
		__init__.py		
		daikon.py		**13 lines, 2 functions**
		invariants\		**236 lines, 20 functions, 3 classes, 6 methods**
			__init__.py	
			detect.py	**202 lines, 19 functions**
			inv_util.py	**34 lines, 1 function, 3 classes, 6 methods**
		logging\		**240 lines, 1 function, 4 classes, 26 methods**
			__init__.py	
			logging.py	**240 lines, 1 function, 4 classes, 26 methods**
	dart\			**832 lines, 20 functions, 9 classes, 61 methods**
		__init__.py		
		constraint_solver.py		
		dart.py		**231 lines, 7 functions, 1 class, 12 methods**
		hashdict.py		**10 lines, 1 class, 3 methods**
		concrete\		**399 lines, 9 functions, 3 classes, 32 methods**
			__init__.py	
			concrete.py	**338 lines, 2 functions, 3 classes, 32 methods**
			funcs.py	**61 lines, 7 functions**
		symbolic\		**174 lines, 2 functions, 4 classes, 14 methods**
			__init__.py	
			symbolic.py	**119 lines, 2 functions, 4 classes, 14 methods**
			symbol_table.py	**55 lines**