Compositional Load Test Generation for Software Pipelines

Pingyu Zhang
University of Nebraska-Lincoln
Lincoln, NE, U.S.A.
pzhang@cse.unl.edu

Sebastian Elbaum
University of Nebraska-Lincoln
Lincoln, NE, U.S.A.
elbaum@cse.unl.edu

Matthew B. Dwyer
University of Nebraska-Lincoln
Lincoln, NE, U.S.A.
dwyer@cse.unl.edu

ABSTRACT

Load tests validate whether a system's performance is acceptable under extreme conditions. Traditional load testing approaches are black-box, inducing load by increasing the size or rate of the input. Symbolic execution based load testing techniques complement traditional approaches by enabling the selection of precise input values. However, as the programs under analysis or their required inputs increase in size, the analyses required by these techniques either fail to scale up or sacrifice test effectiveness. We propose a new approach that addresses this limitation by performing load test generation *compositionally*. It uses existing symbolic execution based techniques to analyze the performance of each system component in isolation, summarizes the results of those analyses, and then performs an analysis across those summaries to generate load tests for the whole system. In its current form, the approach can be applied to any system that is structured in the form of a software pipeline. A study of the approach revealed that it can generate effective load tests for Unix and XML pipelines while outperforming state-of-the-art techniques.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification, Performance, Experimentation

Keywords

Load Testing, Symbolic Execution, Compositional Analysis

1. INTRODUCTION

Load testing aims to validate whether a system's performance (e.g., response time, throughput, resource utilization) is acceptable under production, projected, or worst-case inputs. Traditional load testing approaches induce load by increasing the input size or rate under which the program is executed, treating the program as a black-box.

Emerging symbolic execution based load testing techniques complement traditional approaches by performing a systematic exploration of the program structure. They enable the automatic selection of input values that expose worst-case performance scenarios (WISE [5]), and can produce load test suites that execute diverse paths, increasing the chance of detecting performance faults (SLG [22]).

Through a combination of sophisticated search strategies and mixed symbolic-concrete execution, these emerging techniques have proven successful on algorithms and small systems or system components. However, as the programs under analysis or their required inputs increase in size and complexity, the analyses required by these techniques either fail to scale up or do so at the expense of the generated test's effectiveness in inducing load.

In this paper we propose a *compositional* approach that addresses these scalability and cost-effectiveness issues. The approach builds on the aforementioned techniques to collect the path constraints that would lead to an effective load test suite for each system component. The collected path constraints constitute a performance summary for each component. The approach is compositional in that it analyzes the components' summaries and their connections to identify compatible paths, as defined by their constraints, that can be solved to generate load tests for the whole system. Key to this process is how path constraints across components must be weighted and relaxed in order to derive test inputs for the whole system while ensuring that the most significant constraints, in terms of inducing load, are enforced.

Our work is the first to use compositionality to generate tests aimed at validating non-functional goals. In its current form, the approach can be applied to systems structured in the form of a software pipeline (such as a Unix pipeline or an XML pipeline) which are beyond the scalability of existing approaches. Our contributions are:

1) The idea of generating load tests in a compositional fashion to improve the scalability and cost-effectiveness of the state-of-the-art techniques.

2) A method for systematically weighing and relaxing the constraints of each component according to their impact on the load of the whole system with the support of a constraint solver and the analysis of unsatisfiable cores of constraints.

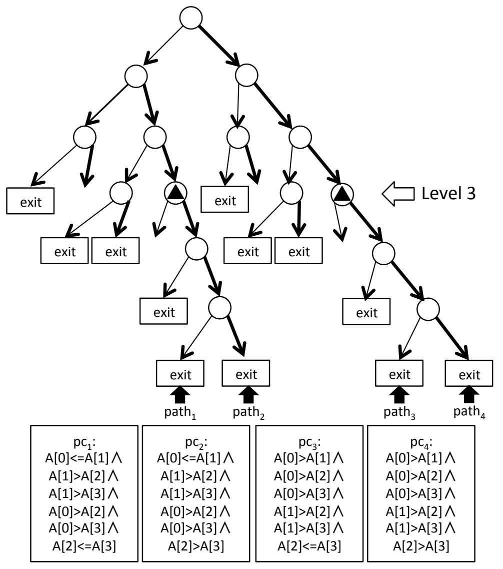
3) An implementation and evaluation illustrating the scalability of the compositional approach on pipelines that SLG could not process, an effectiveness gain of 288% over Random when given the same time to generate tests.

2. BACKGROUND AND OVERVIEW

In this section we review symbolic execution based load testing techniques and introduce a compositional approach to improve their cost-effectiveness.

2.1 Symbolic Execution based Load Test Generation

```
 1 public void bubbleSort(int[] A)
 2     n=A.length;
 3     while(n!=0){
 4         newn=0;
 5         for(i=1; i<n; i++){
 6             if(A[i-1]>A[i]){
 7                 swap(A[i-1], A[i]);
 8                 newn=i;
 9         }     }
10         n=newn;
11 }}
```



exit
Level 3
path$_1$
path$_2$
path$_3$
path$_4$
pc$_1$: A[0]<=A[1]∧ A[1]>A[2]∧ A[1]>A[3]∧ A[0]>A[2]∧ A[0]>A[3]∧ A[2]<=A[3]
pc$_2$: A[0]<=A[1]∧ A[1]>A[2]∧ A[1]>A[3]∧ A[0]>A[2]∧ A[0]>A[3]∧ A[2]>A[3]
pc$_3$: A[0]>A[1]∧ A[0]>A[2]∧ A[0]>A[3]∧ A[1]>A[2]∧ A[1]>A[3]∧ A[2]<=A[3]
pc$_4$: A[0]>A[1]∧ A[0]>A[2]∧ A[0]>A[3]∧ A[1]>A[2]∧ A[1]>A[3]∧ A[2]>A[3]

Figure 1: Code snippet of `bubble sort` and the computation tree for input size 4. Some shorter paths are omitted to simplify the presentation

We use an example to illustrate the core ideas of symbolic execution based load testing techniques. The top of Figure 1 shows the code snippet of a bubble sort algorithm. We use symbolic execution to enumerate all feasible paths of the code with an input size of 4 and show them in the form of a computation tree in the middle of Figure 1, where A[0], A[1], A[2] and A[3] are the four symbolic input variables. Each node in the tree corresponds to the conditional statement at Line 6. Edges are depicted bold or thin to indicate whether the true or false branch is taken. Note that if the true branch is taken, the `swap` method at Line 7 is executed subsequently (corresponds to a bold edge).

There are 24 paths in the tree, of which 12 are shown. To simplify the presentation, some shorter paths are omitted, but all paths that reach the bottom of the tree are shown. The one that induces the most load in terms of response time is the one that executes the largest number of conditional statements and swaps (marked as $path_4$, with 6 conditions and 6 swaps). It also corresponds to the *worst-case* complexity of the program. However, in terms of load, there are other interesting paths in the tree as well. For example, $path_2$ contains 6 conditions and 5 swaps, only one less swap than $path_4$. In fact, $path_2$ will always execute one less swap than the worst-case scenario but it is interesting in that it diverges from $path_4$ starting at the first branch which may expose different behavior.

The boxes at the bottom of Figure 1 contain the conjunction of branch predicates, called a *path constraint*, that must hold (pc_i) for the paths to be taken ($path_i$). A constraint solver can be used to solve a path constraint and obtain a concrete test that guarantees the execution of the path. In the example, pc_4 is solved to get input [4, 3, 2, 1], which ensures the execution of the path.

Given an input size, a cost-effective load testing approach is one that captures most load-sensitive paths *without* enumerating them all. As mentioned earlier, there are two recent approaches to generate load tests. First, Burnim et al. proposed an approach called WISE that aims to capture the path that represents the worst-case scenario [5]. The approach observes what branches are taken on the worst-case paths of small inputs (e.g., arrays of sizes 1-4). These observations are then generalized so that the search can be more focused in the presence of larger inputs. For the bubble sort example, the generalization is that the branch at Line 6 always evaluates to true for a worst-case path. The cost-effectiveness of the technique depends on the quality of the generalization, which requires the user to have extensive knowledge of the program.

Recently, we introduced a second approach called Symbolic Load Generation (SLG) [22]. Instead of capturing a worst-case, SLG generates a load test suite representing a variety of load-sensitive paths that execute the program in different ways ($path_1$ and $path_4$ in Figure 1). SLG performs a guided mixed symbolic and concrete execution while collecting path metrics that serve as a proxy for load. These metrics are then used to prune the less promising paths at given intervals defined by a path diversity assessment. In bubble sort, the load proxy can be defined, for example, as the number of swaps or branches traversed. For bubble sort SLG performs a full symbolic execution up to level 3. At that level the explored paths show a desirable diversity (in terms of their prefixes' differences given predefined thresholds), so SLG chooses the subset of paths (as many as the desired number of tests) that execute the largest number of swaps (the two marked with triangles in the example), and continues the search along just those paths. In the following sections we will use SLG as a representative white-box load testing techniques because it does not require system knowledge for its application, but either approach could be used in our proposed compositional load testing.

2.2 Software Pipelines

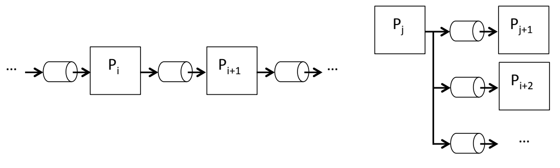
Many complex systems are formed by composing several smaller programs. The constituent programs may be composed in many different ways. For example, two programs may be chained sequentially so that the output of one is the input of the next; or they could form a call-chain relation in which one program is called inside the other as a subroutine.

Sequential program composition plays an important role in quickly assembling functionality from several processing elements to achieve a specific new functionality. One popular implementation of sequential program composition is called a pipeline [15]. Formally, a pipeline is defined as follows.

Table 1: SLG on the pipeline `grep [pattern] file | sort` with increasing input size

Program	Input (# lines)	Gen. (min)	Load (response time in sec)
grep | sort	1000	109	4.2
grep | sort	5000	581	14.7
grep | sort	10000	1355	37.4

Definition 2.1. *Pipeline: A collection of programs connected by their input and output streams. We consider two types of connections in a pipeline –* **linear** *and* **split**. *In a linear pipeline a set of programs $P_1, \cdots, P_n$ are chained so that the output of each P_i feeds directly as input to P_{i+1} (Figure 2(a)). In a split pipeline the output stream of P_j simultaneously feeds into programs $P_{j+1}, P_{j+2}, \cdots$ (Figure 2(b)).*



(a) Linear (b) Split

Figure 2: Pipeline Structures

The *Unix pipeline* is one common instantiation of software pipelines. Each Unix command can be treated as a standalone program executing in a shell environment, or can be chained with others to achieve a combined functionality. For example, the pipeline `grep [pattern] file | sort` achieves the functionality of retrieving information from a file according to a specific pattern, then sorting the results in ascending order. A slightly more complicated example is `find /dir -size 10K | grep -v '.zip' | zip`, which selects files larger than 10K and zips them if they are not zipped already. This pipeline involves three programs and treats not only bytes but also the file system structure.

XML transformations, used extensively on web applications, are another popular type of pipeline. A chain of XML transformations is defined as an *XML pipeline*, expressed over a set of XML operations and an XML transformation language. For example, a three-component linear XML pipeline could 1) validate a webpage against a predefined W3C schema (VALID component); 2) apply a styler transformation for the Firefox web browser (STYLER component); and 3) export the webpage content to an ODF format file that is accessible via OpenOffice (ODF component). The resulting pipeline product is a webpage in ODF format displayed in a way as if it was rendered by a Firefox browser [16].

2.3 Load Testing of Software Pipelines

Even for the simplest Unix pipeline example, generating cost-effective load tests is not an easy task. Either an increase in input size or program complexity or both can make state of the art load test generation tools fail to scale. To illustrate these points, we use a Java implementation of the Unix shell environment called JShell [13], which includes a set of common Unix commands and provides a bash-like notation for chaining them into pipelines (details of this artifact can be found in Section 5.1).

To show how SLG perform when facing larger inputs, we run a pilot study on the pipeline `grep [pattern] file | sort`, treating it as a single program with a single input (the input to `grep`). We initialize the input as a file of various lines numbers of lines (1000, 5000, 10000), each line containing 10 symbolic input values of byte type, and the `[pattern]` in `grep` corresponding to the format of a 10-digit telephone number. We then use SLG to generate ten load tests that attempt to maximize response time on the whole pipeline.

Table 2: SLG on pipelines of increasing complexity

Program	Input (# lines)	Gen. (min)	Load (response time in sec)
grep	5000	247	5.8
grep | sort	5000	581	14.7
grep | sort | zip	5000	1516	21.5

Table 1 shows that, on an input with 1000 lines, we obtain ten tests after 109 minutes. The average response time of the ten tests is 4.2 seconds. The generation time is 581 minutes for 5000 lines and goes up to 1355 minutes for an input of 10000 lines. For a simple artifact like this, it takes almost 24 hours for SLG to produce tests for inputs of 100KB. The approach struggles to scale up in terms of input size when the whole pipeline is considered at once.

Let's now consider the complexity of the programs under analysis. In Table 2 we fix the input size at 5000 but use a more complex artifact on each row. The results suggest that test generation time grows superlinearly with the number of pipeline components. However, the load is not increasing at the same pace. In this case the technique is challenged by the increasing pipeline complexity.

Now, if each pipeline component can be handled by existing approaches, an estimate of the "worst" overall performance may be obtained by adding the longest response times from each component. In practice, however, such attempts can result in gross overestimations of the pipelines performance as inputs that may drive one component to larger response times may not have the same effect on other components. Still, as we shall see, this line of thought of reusing the results from each component to generate load tests for the whole program will be central to our approach.

2.4 A Compositional Approach

Although a large complex program imposes many challenges for a symbolic load test generation approach, if the complex program is composed of several simpler programs, such as the pipeline examples shown in the last section, and each constituent program can be handled by existing approaches, we may devise a new approach that uses the *divide and conquer* strategy to address the illustrated challenges.

Figure 3 shows the key concepts of a compositional load test generation approach. The approach works in four steps. 1) The set of programs (e.g. $P_1, P_2, \cdots$) that form a software pipeline are analyzed by a symbolic execution based load testing tool (such as the ones mentioned in Section 2.1) to produce a library consisting of *performance summaries* for each program. A performance summary characterizes one program with a set of path constraints, each representing a program path that induces load according to a performance measure. 2) The approach examines how the constituent programs are composed together, and uses this information as a guideline for selecting compatible path constraints from the library. It also generates a set of *channeling constraints* for each pair of programs. Channeling constraints are equal-

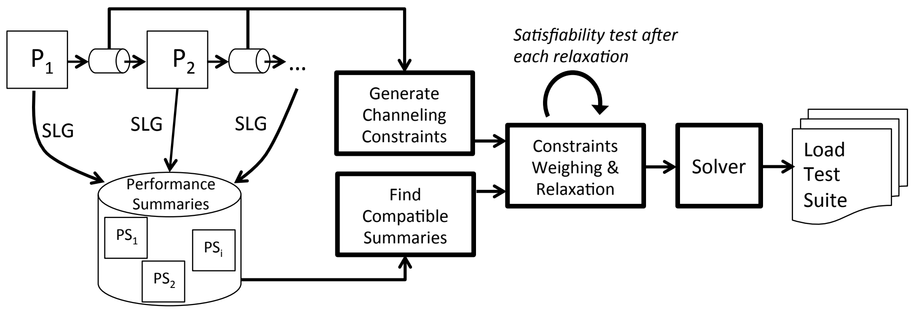

P1
P2
...
SLG
SLG
SLG
Performance Summaries
PS1
PS2
PSi
Generate Channeling Constraints
Find Compatible Summaries
Satisfiability test after each relaxation
Constraints Weighing & Relaxation
Solver
Load Test Suite

Figure 3: Illustration of a compositional approach

ity constraints that bridge the variable sets of two performance summaries computed independently. 3) The selected path constraints and the generated channeling constraints are joined together by conjunction to form a constraint set for the whole program. The resulting constraint set may not be solvable because each constituent constraint encodes a load test of one component, which may not be compatible with the load tests of the other components. Therefore, a constraint weighing and relaxation process is used to remove incompatible conjuncts, starting from the weakest in terms of their contributions to the program load. The weighing and relaxation process tests the satisfiability of the remaining constraint set after each removal. It iterates until the end product becomes solvable. 4) A solver is used to find a solution that satisfies the constraint set, leading to a load test for the whole program. Steps 2-4 are repeated until a load test suite of desired size is generated.

The approach in practice. Continuing with the `grep [pattern] file | sort` example, we use the same settings as before, only this time we collect performance summaries with SLG, then apply the compositional generation approach on them.

Figure 4 illustrates the approach[1]. For steps 1-2, we generate the performance summaries for both `grep` and `sort`, choose one path constraint from each summary (starting from the worst performing ones) and make sure they are compatible by checking for matching output and input sizes (so that they can be executed in a pipeline). The two selected path constraints are shown in Figure 4 on the leftmost and rightmost columns. The center column shows the channeling constraints that bridge the two path constraints (which are computed by tracking the output expressions of `grep` in terms of its input variables). Section 3.3 describes in detail how to compute the channeling constraints.

The third step involves the conjunction of the three constraint sets, and applying a weighing and relaxation process to remove incompatible conjuncts. The process starts by removing those conjuncts that make the least contribution to the load, so that the loss in load is minimized, and testing the satisfiability of the remaining ones after each removal. Figure 4 depicts a situation where the incompatible conjuncts have been removed (crossed out), and the remaining constraint set is satisfiable.

Table 3 shows the results of redoing the first pilot study (Table 1) with the compositional approach. On 1000 lines of input, the approach generated tests of similar quality (4.2 sec vs 4.1 sec) while the generation time is 32% lower. The savings are more impressive as the input size goes up. On an input of 10000 lines, the previous run used almost 24 hours, while the compositional approach finishes in 11 hours and generates tests that induce similar load.

[1]To simplify presentation, Figure 4 depicts a case with the input size of 10 lines, where in the pilot studies the input size varies between 1000 to 10,000 lines.

Table 3: Compositional approach on the pipeline `grep [pattern] file | sort` with increasing input size. % in Gen. and Load shows comparison to Table 1

Program	Input (# lines)	Gen. (min)	Load (response time in sec)
grep | sort	1000	74(68%)	4.1(98%)
grep | sort	5000	342(59%)	13.6(94%)
grep | sort	10000	615(45%)	36.1(96%)

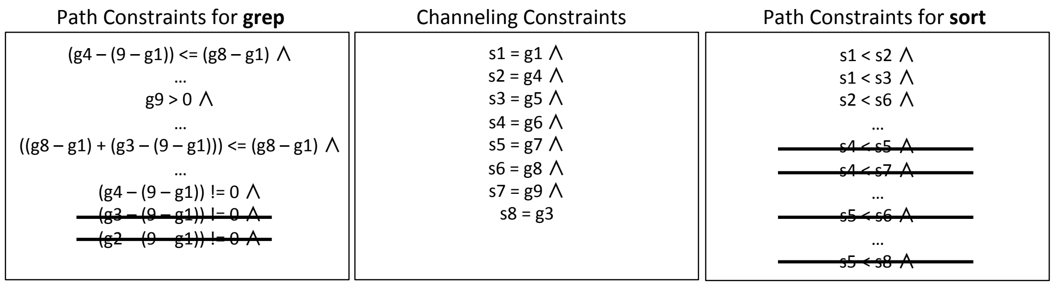
Table 4: Compositional approach on pipelines with increasing complexity Previously computed summaries are reused. % in Gen. and Load shows comparison to Table 2

Program	Input (# lines)	Gen. (min)	Load (response time in sec)
grep	5000	247(100%)	5.8(100%)
grep | sort	5000	153(26%)	14(95%)
grep | sort | zip	5000	179(12%)	19.6(91%)

Redoing the second pilot study yields more impressive results. Since the programs under study use the same input size on the three pipelines, the approach is able to reuse the performance summaries collected from a previous artifact to achieve more savings. For example, on analyzing `grep | sort`, we can reuse the summaries computed for `grep` and only need to compute summaries for `sort`. Table 4 shows that, in case of `grep | sort | zip`, the compositional approach takes only 12% of the effort of a full SLG, but yields comparable tests.

In practice, the gains in efficiency can in turn improve effectiveness if the testing effort is bounded. For example, assume that a tester is given three hours of test generation time, using SLG on the pipeline `grep [pattern] file | sort | zip` with 5000 lines of input will produce load tests that induce 8.3 seconds in response time, while the compositional approach in the same time yields tests that drive load 2.4X higher (last line of Table 4).

These data sets clearly convey that efficiency and scalability gains can be achieved by exploring programs independently and then composing their performance summaries, instead of exploring the whole system space at once. Further efficiency gains can be obtained when reusing previously computed components' summaries to save on repetitive computations. Last, when there are limited resources, the gains could translate into increased effectiveness as well.


Path Constraints for **grep**

(g4 – (9 – g1)) <= (g8 – g1) ∧
…
g9 > 0 ∧
…
((g8 – g1) + (g3 – (9 – g1))) <= (g8 – g1) ∧
…
(g4 – (9 – g1)) != 0 ∧
~~(g3 – (9 – g1)) != 0 ∧~~
~~(g2 – (9 – g1)) != 0 ∧~~

Channeling Constraints

s1 = g1 ∧
s2 = g4 ∧
s3 = g5 ∧
s4 = g6 ∧
s5 = g7 ∧
s6 = g8 ∧
s7 = g9 ∧
s8 = g3

Path Constraints for **sort**

s1 < s2 ∧
s1 < s3 ∧
s2 < s6 ∧
…
~~s4 < s5 ∧~~
~~s4 < s7 ∧~~
…
~~s5 < s6 ∧~~
…
~~s5 < s8 ∧~~

Figure 4: Path constraints computed for `grep [pattern] file | sort`, with the pattern being a 10-digit phone number. Three sets of constraints are shown, one for `grep`, one for `sort`, one for the channeling constraints connecting them. Crossed out constraints have been removed in order to find a solution

3. APPROACH

We now present a compositional approach for generating load tests, discussing its components and its application.

3.1 Overview

A compositional load test generation approach takes *performance summaries* collected on each program in a pipeline, and composes them according to the pipeline structure. We first defines key terms, then outline the approach and its components.

DEFINITION 3.1. *Path Constraint (pc): A conjunction of the necessary conditions for a program path to be taken. A condition is a symbolic expression over symbolic input variables $\{\Phi_P^1, \Phi_P^2, \cdots, \Phi_P^n\}$ to program P.*

DEFINITION 3.2. *Performance Summary (PS): A set of path constraints $PS_j^s = \{pc_1^j, \cdots, pc_n^j\}$ for program P_j caused by inputs of size s that induce load according to a performance measure. Each pc_i^j contains the following tags to assist future analysis:* **size** *of output, and the* **weight score** *indicating its load. A PS_j^s has a tag on the type of input it handles.*

A PS_j^s can be computed by applying any symbolic execution based load testing approach (such as SLG or WISE) to program P_j with a fixed input size s (to simplify the explanation, we assume the input has one type). A weight score is an indicator of how much impact a path has in terms of a performance measure (e.g., the response time for a test that traverses that path). For a set of programs $P_1 \cdots P_n$ and input sizes $size_1, size_2, \cdots$, we compute a library of PS as

$$PSLib = \{\{PS_1^{size_1}, PS_1^{size_2} \cdots\} \cdots \{PS_n^{size_1}, PS_n^{size_2} \cdots\}\}$$

Our compositional analysis takes path constraints generated for different programs and composes them together. First, we need to identify the compatible path constraints that can be stitched. We define compatibility as follows.

DEFINITION 3.3. *Compatibility: For two programs P_i and P_{i+1} in a pipeline $P_i|P_{i+1}$, two path constraints $pc^i \in PS_i^{size_m}$ and $pc^{i+1} \in PS_{i+1}^{size_n}$ are compatible if $|O(pc^i)| = size_n$, where $O(pc^i)$ refers to the set of output variables that are confined to the path defined by pc^i.*

Next, we need a way to express compatible path constraints in the same *namespace*. This is done through the introduction of *channeling constraints.* A channeling constraint is an equality constraint that connects two variables defined over different contexts [18]. Formally, we define:

Algorithm 1 CompSLG($PSLib, T$)

```
1: $\overline{testSuite} \leftarrow \emptyset$
2: while $|\overline{testSuite}| < T$ do
3:     $compPC \leftarrow$ selectCompatiblePC($PSLib$)
4:     $compCC \leftarrow$ genChannelConstraints($compPC$)
5:     $UC \leftarrow$ relaxConstraints($compPC, compCC$)
6:     newTest $\leftarrow$ solve($UC$)
7:     $\overline{testSuite}$.add(newTest)
8: end while
9: return $\overline{testSuite}$
```

Algorithm 2 selectCompatiblePC($PSLib$)

```
1: $compPC \leftarrow \emptyset$
2: $pc^1_{worst} \leftarrow$ worst-case($PSLib[PS_1]$)
3: $compPC$.add($pc^1_{worst}$)
4: $PSLib$.remove($pc^1_{worst}$)
5: for $i \in (2, n)$ do
6:     $pc^i \leftarrow$ worst-compatible($PSLib[PS_i], pc^{i-1}$)
7:     if $pc^i \neq null$ then
8:         $PSLib$.remove($pc^i$)
9:     else
10:        $pc^i \leftarrow$ SLG($P_i, |O_i|$)
11:        if $pc^i = null$ then
12:            break
13:        end if
14:    end if
15:    $compPC$.add($pc^i$)
16: end for
17: return $compPC$
```

DEFINITION 3.4. *Channeling Constraints (CC): Given two programs P_i and P_{i+1} in a pipeline $P_i|P_{i+1}$, and $pc^i \in PS_i$ and $pc^{i+1} \in PS_{i+1}$, channeling constraints CC are equality constraints that map the symbolic input variables of pc^{i+1} to the corresponding output expressions of pc^i.*

These constraints eliminate the need to solve for ps^{i+1}'s variables, which are now expressed in terms of ps^i's input variables.

DEFINITION 3.5. *Unified Constraint Set (UC): Given path constraints $pc^1 \cdots pc^n$, where $pc^1 \in P_1, \cdots, pc^n \in P_n$, and channeling constraint sets $CC_1 \cdots CC_{n-1}$, a unified constraint set is obtained through the conjunction $pc^1 \wedge CC_1 \wedge pc^2 \wedge \cdots \wedge CC_{n-1} \wedge pc^n$.*

In the following sections we will refer to the process of removing individual conjuncts from the constraint set as *relaxation*, since each removal makes the constraint set easier to satisfy.

Algorithm 1, CompSLG, outlines the process for generating load tests for programs $P_1|\cdots|P_n$ forming a pipeline.

The algorithm takes the pre-computed $PSLib$ as input, and has access to the pipeline structure. It also allows the user to specify the number of load tests (T) to generate. The algorithm operates iteratively, generating one test after each iteration of lines 2-8. It first selects a set of compatible path constraints (*compPC*). Then it generates channeling constraints $compCC$ according to the path constraints selection. The conjunction of *compPC*s and *compCC*s are checked for satisfiability. If they are not satisfiable, they are relaxed to ensure that the resulting UC is satisfiable. The UC is subsequently concretized into a test.

In the following sections we will introduce each of these components in detail. We assume a linear pipeline structure in Sections 3.2 - 3.4 for a more concise explanation, and briefly discuss the split pipeline structure in Section 3.5.

3.2 Selecting Compatible Path Constraints

Algorithm 2 shows the process for selecting compatible path constraints. The process takes $PSLib$ as input and starts by selecting a pc^1 from $PSLib$ that represents the worst-case for P_1 (the first component of the pipeline). Then, the process examines the output size associated with pc^1 and finds a pc^2 from the summaries of P_2 with a matching input size at Line 6 of Algorithm 2. The function `worst-compatible` will find a match for pc^2, and if multiple summaries are matched, it will select the one with a higher weight score first. The process operates in this way to find for each pc^{i-1} a matching pc^i. This process continues until all programs have selected compatible *pc*s. If CompSLG cannot find a compatible *pc* for a program, it will call the SLG to generate one with the desired attributes, or stop if no such summary can be generated.

Note that we start the selection of compatible paths constraints with the first component of the pipeline and move forward matching pc^{i-1} to pc^i so that, if a compatible pc^i cannot be found, the approach can instruct SLG on the input size to use to produce a compatible load test.

3.3 Generating Channeling Constraints

Assume that program P_i takes symbolic variables $\{\Phi^1_{P_i}, \Phi^2_{P_i}, \cdots, \Phi^n_{P_i}\}$ as inputs, traverses a path and collects path condition pc^i. After symbolic execution completes, the output of P_i, $\{O^1_{P_i}, O^2_{P_i}, \cdots, O^m_{P_i}\}$, can be expressed in the following way

$$O(pc^i) = \begin{cases} O^1_{P_i} = expr_1(\Phi^1_{P_i}, \Phi^2_{P_i}, \cdots, \Phi^n_{P_i}) \wedge \\ O^2_{P_i} = expr_2(\Phi^1_{P_i}, \Phi^2_{P_i}, \cdots, \Phi^n_{P_i}) \wedge \\ \cdots \\ O^m_{P_i} = expr_m(\Phi^1_{P_i}, \Phi^2_{P_i}, \cdots, \Phi^n_{P_i}) \end{cases} \quad (1)$$

where $O(pc^i)$ means the output variables are confined to one path defined by pc^i and $expr_k$ stands for the expression over input symbolic variables $\Phi^1_{P_i}, \Phi^2_{P_i}, \cdots, \Phi^n_{P_i}$ that represents the value of $O^k_{P_i}$.

Assume that program P_{i+1} takes symbolic variables $\Phi^1_{P_{i+1}}$, $\Phi^2_{P_{i+1}}, \cdots, \Phi^m_{P_{i+1}}$ as inputs. Because P_i and P_{i+1} are pipelined, there must a direct mapping between $\Phi^k_{P_{i+1}}$ and $O^k_{P_i}$,

$$I(pc^{i+1}) = \begin{cases} \Phi^1_{P_{i+1}} = O^1_{P_i} \wedge \\ \Phi^2_{P_{i+1}} = O^2_{P_i} \wedge \\ \cdots \\ \Phi^m_{P_{i+1}} = O^m_{P_i} \end{cases} \quad (2)$$

where $I(pc^{i+1})$ represents the list of symbolic input variables in pc^{i+1}. Combining (1) and (2) we obtain the channeling constraint

$$CC = \begin{cases} \Phi^1_{P_{i+1}} = expr_1(\Phi^1_{P_i}, \Phi^2_{P_i}, \cdots, \Phi^n_{P_i}) \wedge \\ \Phi^2_{P_{i+1}} = expr_2(\Phi^1_{P_i}, \Phi^2_{P_i}, \cdots, \Phi^n_{P_i}) \wedge \\ \cdots \\ \Phi^m_{P_{i+1}} = expr_m(\Phi^1_{P_i}, \Phi^2_{P_i}, \cdots, \Phi^n_{P_i}) \end{cases} \quad (3)$$

For a linear pipeline there is one CC between each pair of path constraints. CC allows for all constraints to be expressed in terms of P_i's symbolic variables.

Algorithm 3 relaxConstraints($compPC, compCC$)

```
1: $UC \leftarrow$ union($compPC, compCC$)
2: while $\neg$ satisfiable($UC$) do
3:     $core \leftarrow$ computeUnsatCore($UC$)
4:     $c \leftarrow$ selectWeakestConjunct($core, compPC$)
5:     removeConjunct($c, UC$)
6: end while
7: return $UC$
```

Algorithm 4 selectWeakestConjunct($core, compPC$)

```
1: for all $c \in core$ do
2:     if $c \in compPC$ then
3:         mark($c$)
4:     end if
5: end for
6: $targetPC \leftarrow$ lowestWeight($compPC$)
7: $c \leftarrow$ marked(leastCost($targetPC$))
8: adjustWeight($targetPC$)
9: return $c$
```

3.4 Weighing and Relaxing Constraints

If the set of generated constraints is not solvable, then the approach will systematically remove a constraint until it becomes solvable. During each iteration the approach computes an unsatisfiable core of the constraint system under consideration, and then removes one conjunct that 1) appears in the unsatisfiable core, 2) appears in the *pc* that has the least weight among all *pc*s, and 3) makes the least contribution to the weight of that *pc*.

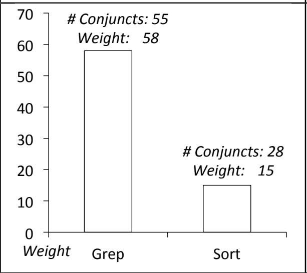
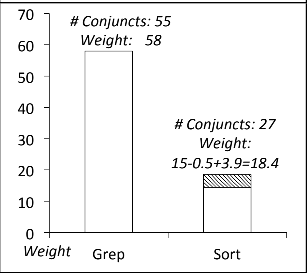
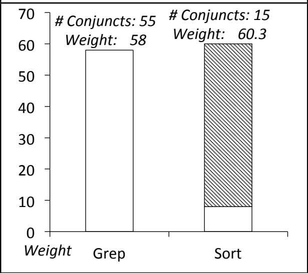
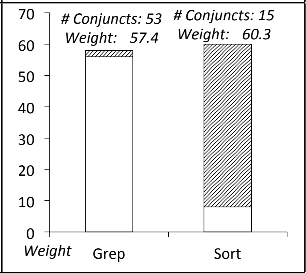
Algorithm 3 shows the procedure for weighing and relaxing constraints on *compPC* collected over a set of linearly piped programs and the corresponding channeling constraints $compCC$. The algorithm starts by performing a conjunction of *compPC* and *compCC* into a unified constraint set and tests for satisfiability. If UC is satisfied, there is no need to relax constraints and the algorithm exits. Otherwise, the algorithm repeatedly follows a three-step relaxation process (lines 2-6) until UC is satisfied.

First, an unsatisfiable core[2] of UC is computed. Then the conjunct $c \in core$, deemed the weakest in terms of its impact on the load of the system, is removed from UC. In the worst-case, UC will be reduced to containing only the constraint from one program, at which time it will be satisfiable since a PS_i contains only feasible paths.

Algorithm 4 details the subroutine for selecting the weakest conjunct. First, the unsatisfiable *core* and *compPC* are

[2]An unsatisfiable core is a subset of UC which preserves the unsatisfiability but is simplified. A minimum unsatisfiable core ensures that removing any one conjunct breaks the unsatisfiability, while a minimal core is the smallest of minimum cores. Here a core is not guaranteed to be either minimum or minimal. As noted in [14], computing such a core is expensive and no practical solver attempts to do so.

Table 5: Illustration of weighing and relaxing constraints. White areas correspond to the original weight, shaded areas correspond to the adjusted weights

Before relaxation	After 1st relaxation	Starting to relax Grep	End of relaxation
			

cross-compared and conjuncts that appear in both sets are marked on *compPC* (Lines 1-5). The algorithm then examines the weight score on each pc^i and selects the one with the lowest weight($targetPC$). If there are more than one pc^i with the same lowest weight, the algorithm will pick one randomly. The algorithm then selects the conjunct c that is both marked and has the lowest cost in $targetPC$ and returns c (the cost of a conjunct c is defined as the difference in the weight of the pc before and after removing c).

Before returning, the weight score for the $targetPC$ must be adjusted (Algorithm 4 - line 8). This step is to ensure that past relaxations have an impact on the current selection of the weakest conjunct, so that we do not keep relaxing the same *pc*. Therefore, the algorithm compensates the weight score for the pc^i whose conjunct is relaxed at the current iteration. For $pc^1 \cdots pc^n$ with associated weight scores $w_1 \cdots w_n$, after the relaxation of $c \in pc^i$, we compensate w_i by a value determined by $w_1 \cdots w_n$. One possible function we later explore is $max\{w_1 \cdots w_n\}/w_i$, where the compensation is determined by w_i's ratio to the largest weight score.

Table 5 illustrates the process of weighing and relaxing of constraints on the running example. The weight scores are shown as bars on the graph, with the white area indicating the original weight, and the shaded area indicating the adjusted weights. The sum of weights and the number of remaining conjuncts are also listed at the top of each bar. We can observe that before relaxation, the original weight scores are 58 for `grep` and 15 for `sort`. After removing the first conjunct on `sort` (which has a lower weight), its weight score is decreased by 0.5 due to the relaxation, and increased by $58/15 = 3.9$ due to compensation. The third cell shows the turning point where after removing 13 conjuncts, the weight on `sort` has been compensated enough to exceed that of `grep`, whose conjuncts are starting to be removed. The fourth cell shows the weight scores at the end of relaxation, with `sort` having 13 conjuncts removed, and `grep` having 2, at which time UC becomes satisfiable.

3.5 Handling Richer Structures

We now briefly discuss how to extend the existing approach to enable compositional load test generation for structures other than linear software pipelines. We start with split pipelines, in which the output of P_j is fed simultaneously to $P_{j+1}, \cdots, P_{j+n}$. We treat a split pipeline as a set of parallel linear pipelines $P_j|P_{j+1}, \cdots, P_j|P_{j+n}$ and solve them in a similar way as presented in Algorithms 1 - 4. Although we do not have space to discuss all adjustments in detail, we note that the most noticeable one happens in Line 6 of Algorithm 2 as multiple components may share the same predecessor (the splitting component). To handle this, an additional data structure is maintained to identify the target component on which to invoke *worst-compatible*.

More complex structures beyond the realm of pipelines, such as systems with nested components, introduce additional challenges. The analyses necessary to extract, for example, the channeling constraints will have to consider side effects beyond the outputs (e.g. global and static variables, parameters), which will increase their complexity as well. We leave those for future work.

4. IMPLEMENTATION

We now discuss the most relevant implementation details of our CompSLG test generator and the support needed to handle the symbolic file system for Unix programs that take a file as input.

CompSLG. CompSLG was implemented on top of SLG [22], which is a customized version of the Java PathFinder (JPF) tool. SLG enables iterative-deepening selective symbolic execution by implementing a path replay mechanism and various load proxies to track the progress of each path. It also includes a diversity clustering algorithm to check the diversity of paths efficiently. We build the performance summary libraries by repeatedly invoking SLG on each component of a pipeline. The summaries are stored in XML format. The channeling constraints are computed by implementing a JPF listener that monitors write instructions for the output variables, and traces their symbolic expressions from the stored system states. We use Yices [21] to compute the unsatisfiable core during the constraint weighing and relaxation. We first transform the constraints to a Yices compatible format, and feed it to the solver. Should the solver return unsatisfiable, we then use the `yices_get_unsat_core` API to get the unsatisfiable core of the constraint set.

Symbolic File System Modeling. Certain Unix programs, such as `Find`, take a file directory and a file property as inputs, and browse the directory for files matching such a property. To enable symbolic execution of this type of command, we need to treat the file system symbolically. We implemented several JPF interfaces to handle most file system operations. For each file system operation (e.g. `open`), we check if the action is for a concrete or a symbolic file. For concrete files, we simply invoke the corresponding system call. For symbolic files we emulate the operation with symbolic values (e.g. for `open`, we will return a file handler pointing to a symbolic file with all fields declared as symbolic; for `read`, we will return symbolic bytes of the same size as a concrete read would return). Users can specify size

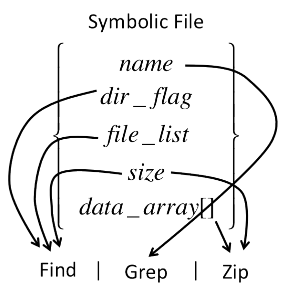


Figure 5: Symbolic file modeling

constraints on the symbolic file system being used as inputs. The two size constraints are the number of file objects (files or directories), and the overall size of these files. A symbolic file system can contain as many levels of directories and as many files as desired, catering to the current analysis progress, as long as the two size constraints are not violated.

Besides the implementation challenges, handling a file system symbolically made us realize that the components in a pipeline may handle different variables so the approach needs to be cognizant of such types. Consider the pipeline in Figure 5 consisting of `find /dir -size 10K | grep -v '.zip' | zip`. The pipeline takes a symbolic file system as input, and each program in the pipeline only accesses part of the symbolic file, collecting constraints just on that part. During constraint weighing and relaxation, only programs that access shared parts of the symbolic file need to participate in the relaxation process. For example, in Figure 5 `grep` is the only program that accesses the `name` of the symbolic file. Thus the performance summaries of `grep` need not be involved in the relaxation of constraints as the analysis proceeds.

5. EVALUATION

We evaluate the cost and effectiveness of CompSLG relative to SLG and Random approaches on various artifacts.

5.1 Artifacts

The goal of CompSLG is to generate load tests for larger pipelined programs by composing performance summaries of the constituent parts. To demonstrate the potential of CompSLG we required a set of artifacts conforming to the pipeline computation model. In this study we explore two types of pipelines, one being the Unix pipelines popular among system administrators, the other being XML pipelines used extensively on web servers and other types of data transformation tools.

As mentioned before, CompSLG makes use of SLG, which was implemented as a customized JPF. This limited the selection of artifacts to Java programs that the JPF symbolic execution engine can handle. With that limitation in mind, we selected the Unix programs from a Java implementation of the Unix shell environment called JShell [13]. We enriched the environment with 2 more commands (`grep` and `sort`) so that we could evaluate a variety of common pipelines by composing these commands in different ways. Table 6 lists the available programs, their LOC, and the type of input data each one takes (to be described in Section 5.3).

Table 7 shows the pipes we study, their description, and how they are initialized for a mixed concrete and symbolic execution. Among them, Artifacts 1 and 2 were distilled from real scripts that we obtained from our Computer Science Department system administrator. Artifact 3 was obtained from a book on Unix system administration [17], and Artifact 4 was abstracted from an instance of the set of security check scripts named COPS [7] (the original script was written in Perl, we extracted the embedded shell pipeline calls that executed the main functionality).

Table 6: Unix and XML Programs

Program	LOC	Description	I/O Type
sort	359	Sorts the input	byte
grep	1051	Retrieves lines according to a pattern	byte
find	1776	Retrieves files according to property	file structure
zip	5313	Compress data	file structure
cat	479	Concatenate and displays files	file structure
cut	894	Extracts information from each line	byte
VALID	850	Validates a webpage	XML
STYLER	2013	Applies a styler transformation	XML
ODF	5512	Transforms a webpage to ODF format	XML

For the XML pipeline we selected a Java implementation called smallx [16] which supports the XML pipeline language XProc [20]. The specific XSLT instance we employed in the study was a three-component linear pipe that: 1) validates a webpage against a predefined W3C schema (VALID); 2) applies a styler transformation for the FireFox web browser (STYLER); 3) exports the webpage content to an ODF format file that is accessible via OpenOffice (ODF). This instance was obtained from the test suite that comes with the smallx package.

5.2 Metrics and Treatments

We assess the effectiveness of a load test by measuring its effect on the *response time* of the target program. We measure costs in terms of the *time* necessary for the approach to generate a test suite.

The main treatment in the study is the CompSLG approach as described in Section 3. We study three variants of CompSLG, described as follows:

- **No-reuse**: all performance summaries are generated for each artifact on the fly, with no reuse of summaries among artifacts.
- **Incremental-reuse**: the artifacts are analyzed in the order defined in Table 6, and summaries are reused when possible. For example, Artifact 2 could reuse the summary generated for `find` in Artifact 1 since they both use the same command and they were invoked with the same input constraints.
- **Full-reuse**: assumes that all summaries were collected previously, so the generation cost only corresponds to the composition effort.

We use two control treatments, SLG and random test generation. SLG is selected as an alternative symbolic load generation approach and Random is used to provide a baseline comparison. For SLG, we apply it to the whole pipeline by treating it as a single program, and we apply it to each component of the pipeline independently, which will reduce the analysis burden, but sacrifices effectiveness in generating a load for the whole pipeline.

For the Random treatment, we treat the pipeline as a single program and we randomly generate one type of input

Table 7: **Summary of Pipeline Artifacts**

No.	Program	Source	Description	Initialization
1	find /dir -size 10K | grep -v '.zip' | zip	System admin	Backup script: Finds files larger than 10K and zips them if they are not zipped already	File system size 100, 1MB of data
2	find /dir -name 'mbox' | tee > zip > grep '[[:alnum:]+._-]*@ [[:alnum:]+._-]*' | sort –unique	System admin	Email server management script: Finds all mail box files, zips them, and lists email addresses of all users	File system size 100, 1MB of data
3	find /dir -name '.txt' | cat | grep [[:digit:]]{3}[-] ?[[:digit:]]{4} | xargs -ifoo grep foo recordsBook	Unix textbook [17]	Information retrieval script: Filters results through grep of grep	File system size 100, 1MB of data. recordsBook is a concrete file.
4	cat /var/log/secure.log | grep -i 'user' | cut -d ':' -f 5 | xargs -ifoo grep foo /etc/passwd	COPS [7]	Security check script: It finds user who accidentally typed password in the id field	1MB of data for secure.log, /etc/passwd is a concrete file
5	VALID | STYLER | ODF	smallx [16]	It validates the content of a webpage and transforms it into ODF format	Sizes vary between 70KB to 280KB

that feeds into the first component. We adopt a generate-and-run scheme for retaining the most effective tests as they are being generated. Random is run for as long as the compositional approach to enable a comparison with the same baseline cost.

We generate 10 tests for all treatments and report average measures of the 10.

5.3 Experiment Setup

For Unix pipelines taking a file system as input (e.g. via the `find` command), we initialize it with a symbolic file system and constrain the total number of files and directories it can access. For pipelines taking a single file as input, we simply constrain the size of that file. In Table 7, column "Initialization" specifies the *size* constraints we use for the analysis on the Unix pipelines.

For the XML pipeline, we select an input size of 140K, which is reported to be the average size of webpages that exist on the Internet [19]. To explore how CompSLG performs with various input sizes, we also generate tests for two more input sizes, which are 50% and 200% of 140K. In addition to input sizes, we also impose a set of structural constraints on the input to the XML pipeline program. We specify that the input webpage can contain six types of tags (paragraph, heading, hyperlink, image, list, and table tags), and that each of those tags has an upper count of 160. The choice of tags and the upper count are based on the characteristics of the test suite of smallx. This set of constraints is to ensure that the approach is not stuck in a local optimum, and the resulting tests are more diverse.

We use SLG as a subroutine for the main treatment (as described in Section 4) and as a standalone control treatment. We use the same SLG configuration as in previous empirical studies [22], with iterative searches of 50 branches in length and maximum path size of 30000 in order to keep within the capabilities of the various constraint solvers we used. We enforce a generation time cap of five hours across all runs. If exceeded, the analysis is terminated and no results are reported. Throughout the study we use the same platform for all programs, a 2.4 GHz Intel Core 2 Duo machine with Mac OS X 10.6.7 and 4GB memory. We executed our tool on the JVM 1.6 with 2GB of memory.

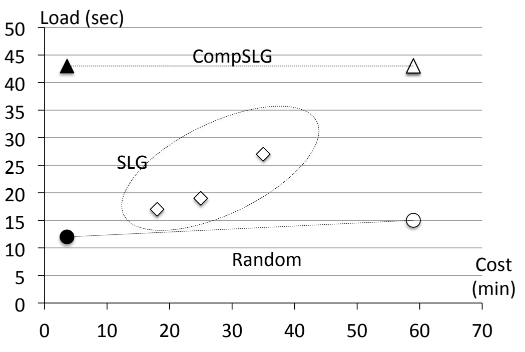
5.4 Results and Analysis

Figure 6 shows results on the Unix pipelines. Each figure corresponds to one artifact specified in Table 7. The x-axis shows the cost of generating load tests measured in terms of generation time in minutes, and the y-axis shows the effectiveness of running those tests, measured in terms of the average response time in seconds, induced by the 10 tests produced by each treatment. On each figure we plot three sets of data points corresponding to the three treatments used in the study. The triangles refer to CompSLG, with white triangles indicating No-reuse, grey triangles indicating Incremental-reuse, and black triangles indicating Full-reuse; the diamonds refer to the SLG treatment applied to a program in the pipeline to generate input for the whole pipeline, and the circles refer to the Random treatment, with colors matched to each variation of CompSLG. Figure 7 shows the results on the XML pipeline in the same format.

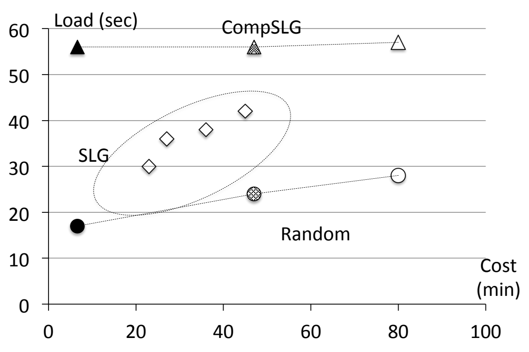
We first applied SLG on the whole pipelines and found that none of them could finish within five hours, so they were terminated and are not depicted in the figures. For the rest of the treatments, we describe our findings in terms of effectiveness first and then cost.

We notice that the various types of CompSLG are all capable of inducing more load than the other treatments across all artifacts. CompSLG produces load tests that generate as much as 4.4X times more load than Random on a Unix pipeline (Artifact 4), and as much as 2.8X on the XML pipeline (280K) with the same cost. We also note that the three variants of CompSLG show almost no variance in terms of effectiveness, indicating that reuse of previously computed summaries does not degrade load test quality. Random, on the other hand, shows certain level of variance as its effectiveness corresponds directly to the allocated generation time. CompSLG also consistently induces more load in every pipeline than SLG applied to any of the involved components. This makes sense as SLG finds the best inputs for one component which may not necessarily be the best for the pipeline. On average CompSLG induces 1.5X time more load over SLG for the Unix pipelines, and 1.3X for the XML pipeline.

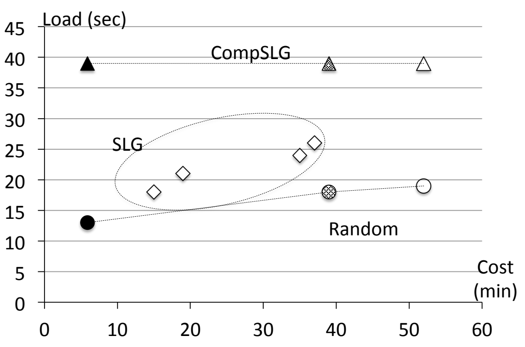
In terms of cost, CompSLG with No-reuse (white triangles) has a higher cost than SLG. However, the cost for CompSLG can be dramatically reduced by reusing perfor-



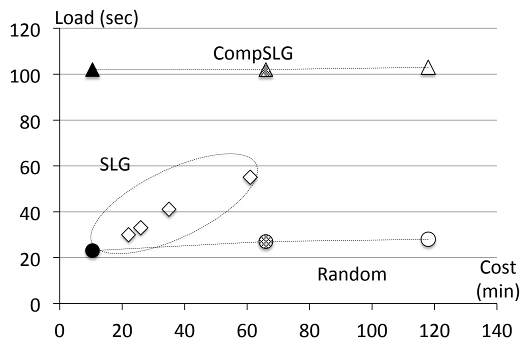
(a) Artifact 1



(b) Artifact 2



(c) Artifact 3



(d) Artifact 4

Figure 6: Cost-effectiveness of treatments applied to Unix pipelines. Triangles represent CompSLG (blank for No-reuse, shaded for Incremental-reuse, solid for Full-reuse), diamonds represent SLG, and the circles represent Random

mance summaries collected from analysis of previous artifacts. Results on Incremental-reuse (grey triangles) indicate that the cost for CompSLG is close to that of SLG, but in all cases achieves a higher load[3]. The savings are more dramatic when we compare the Full-reuse version of CompSLG (black triangles) with other treatments. For the Unix pipelines, on average, CompSLG with Full-reuse took 17% of the time of SLG to generate a load test suite yet achieved an average 45% more load over the non-compositional counterpart. For the XML pipeline, although we do not report any data on Incremental-reuse because size incompatibility prevents such reuse, a similar trend can be observed with Full-reuse, which on average spent 42% of the effort of SLG and achieved 68% more load.

6. RELATED WORK

Until recently, techniques and tools for load test generation have treated the target program as a black box. They either employ user profiles [2] or adaptive resource models [3] to decide how to induce load, but provide limited support for accurately selecting load inducing inputs. Our approach, on the other hand, considers program structure on two levels. On the component level, it explores the program paths systematically with the support of guided symbolic execution to compute performance summaries which encode paths constraints that induce heavy load [22]. On the system level, it examines how the components interact and stitches compatible summaries together in order to generate load tests for the whole system.

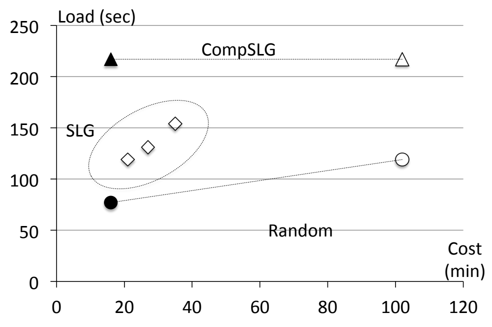
[3]Figure 6 - Artifact 1 does not have a grey triangle because it is the first artifact to be analyzed so we do not have pre-computed summaries to reuse.

The idea of using a compositional approach to relieve the burden of program analysis on the whole system and improve scalability has been explored extensively, notably in the area of compositional verification of concurrent reactive systems(e.g., [4], [9]), inter-procedural static analysis (e.g., [11],[12]), and compositional test generation with symbolic execution (e.g., [1], [8]). In a sense they can be viewed as analyzing individual functions in isolation, summarizing the results of these analyses, and then using those summaries to perform a global analysis across function boundaries. Conceptually, our approach operates in a similar way, but with different goals and mechanisms.

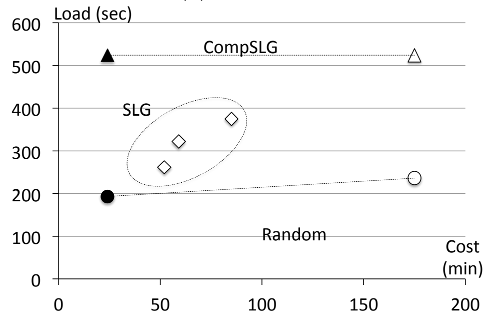
Similarly, Gulwani et al. proposed a compositional approach to characterize computational complexity [10]. The approach instruments a program with counters, uses an invariant generator to compute bounds on these counters, and composes individual bounds together to estimate the upper bound of loop iterations. This approach relies on the power of the invariant generator and the user input of quantitative functions to bound any type of data structures. Our approach, on the other hand, uses symbolic execution and does not require any user input specific to the program under test. To the best of our knowledge, our work is the first one to utilize compositional symbolic execution to generate non-functional tests.

7. CONCLUSION

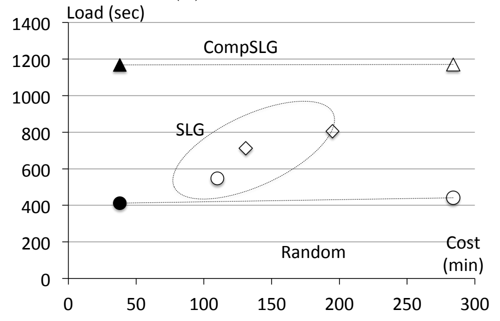
This paper presents a compositional approach, CompSLG, that can automatically generate load tests for complex programs. CompSLG uses emerging symbolic execution based techniques to analyze the performance of each system component in isolation, summarize the results of those analyses,



(a) Size 70K



(b) Size 140K



(c) Size 280K

Figure 7: Cost-effectiveness of treatments applied to XML pipeline. Triangles represent CompSLG (blank for No-reuse, solid for Full-reuse), diamonds represent SLG, and the circles represent Random

and then perform a global analysis across those summaries to generate load tests for the whole system. In its current form, CompSLG can be applied to any system that is structured in the form of a software pipeline (such as a Unix pipeline or an XML pipeline). A study of CompSLG revealed that it can generate load tests for pipelines where SLG alone would not scale up, and it is much more cost-effective than either SLG applied to single components or Random test generation.

Our next step is to investigate how to extend CompSLG to enable its application to systems that are not confined to a pipeline structure. We plan to start this effort by examining how to encode systems that have nested components in its structure, and by exploring whether automatic software partitioning techniques [6] may help identify units that are individually analyzable yet composable.

8. ACKNOWLEDGMENTS

This material is based in part upon work supported by NSF award CCF-0915526 and by AFOSR awards #9550-09-1-0687 and #9550-10-1-0406. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of AFOSR or NSF.

9. REFERENCES

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proc. Int'l Conf. Tools Algo. Cons. Analy. Syst.*, 2008.

[2] A. Avritzer and E. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Trans. Softw. Eng.*, 1995.

[3] M. Bayan, Cangussu, and J. ao W. Automatic feedback, control-based, stress and load testing. In *Proc. ACM Symp. Applied Comp.*, 2008.

[4] T. Bultan, J. Fischer, and R. Gerber. Compositional verification by model checking for counter-examples. In *Proc. Int'l Symp. Softw. Test. Anal.*, 1996.

[5] J. Burnim, S. Juvekar, and K. Sen. Wise: Automated test generation for worst-case complexity. In *Proc. Int'l. Conf. Softw. Eng.*, 2009.

[6] A. Chakrabarti and P. Godefroid. Software partitioning for effective automated unit testing. In *Proc. Int'l Conf. Embed. Softw.*, 2006.

[7] D. Farmer and E. H. Spafford. The cops security checker system, 1994.

[8] P. Godefroid. Compositional dynamic test generation. In *Proc. Symp. Princip. Of Prog. Lang.*, 2007.

[9] O. Grumberg and D. E. Long. Model checking and modular verification. *Trans. Prog. Lang. Syst.*, 1994.

[10] S. Gulwani, K. Mehra, and T. Chilimbi. Speed: Precise and efficient state estimation of program computational complexity. In *Proc. Symp. Princip. of Prog. Lang.*, 2009.

[11] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proc. Prog. Lang. Design and Impl.*, 2002.

[12] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proc. Symp. Found. Softw. Eng.*, 1995.

[13] Jshell project website. `http://geophile.com/jshell/`.

[14] M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints, 2007.

[15] C. V. Ramamoorthy and H. F. Li. Pipeline architecture. *ACM Comput. Surv.*, 1977.

[16] Smallx project website. `http://code.google.com/p/smallx/`.

[17] D. Taylor. *Teach Yourself Unix System Administration in 24 Hours.* Sams Publishing, 2003.

[18] T. Walsh. Permutation problems and channelling constraints. In *Proc. Artificial Intelli. on Logic for Programming*, 2001.

[19] Report on average web page size. `http://www.websiteoptimization.com/speed/tweak/average-web-page/`.

[20] Xproc: An xml pipeline language. `http://www.w3.org/TR/xproc/`.

[21] Yices solver website. `http://yices.csl.sri.com/`.

[22] P. Zhang, S. Elbaum, and M. Dwyer. Automatic generation of load tests. In *Proc. Int'l. Conf. Automated Softw. Eng.*, 2011.