Automatic Generation of Load Tests

Pingyu Zhang, Sebastian Elbaum, and Matthew B. Dwyer
University of Nebraska - Lincoln
{pzhang,elbaum,dwyer}@cse.unl.edu

***Abstract*—Load tests aim to validate whether system performance is acceptable under peak conditions. Existing test generation techniques induce load by increasing the size or rate of the input. Ignoring the particular input values, however, may lead to test suites that grossly mischaracterize a system's performance. To address this limitation we introduce a mixed symbolic execution based approach that is unique in how it 1) favors program paths associated with a performance measure of interest, 2) operates in an iterative-deepening beam-search fashion to discard paths that are unlikely to lead to high-load tests, and 3) generates a test suite of a given size and level of diversity. An assessment of the approach shows it generates test suites that induce program response times and memory consumption several times worse than the compared alternatives, it scales to large and complex inputs, and it exposes a diversity of resource consuming program behavior.**

***Keywords*-Load testing, symbolic execution**

I. Introduction

Load tests aim to validate whether a system's performance (e.g., response time, resource utilization) is acceptable under production, projected, or extreme loads. Consider, for example, an SQL server that accepts queries specified in the standard query language to create, delete, or update tables in a database. Functional tests would validate whether a query results in appropriate database changes. Load tests, however, would be required to assess whether, for example, for a given set of queries the server responds within an expected time.

Existing approaches to generate load tests induce load by increasing the input size (e.g., a larger query or number of queries) or the rate at which input is provided (e.g., more query requests per unit of time)[19]. Consider again the SQL server. Given a set of tester supplied queries, a load testing tool might replicate the queries, send them to the SQL server at certain intervals, and measure the response time. When the measured response time differs from the user's expectations, which might be expressed as some upper bound determined by the size or complexity of the queries or underlying database, a performance fault is said to be detected.

Current approaches to load testing suffer from four limitations. First, their cost-effectiveness is *highly dependent on the particular values* that are used yet there is no support for choosing those values. For example, in the context of a SQL server we studied, a simple selection query operating on a predefined database can have response times that vary by an order of magnitude depending on the specific values in the select statements. Clearly, a poor choice of values could lead to underestimating system response time thereby missing an opportunity to detect a performance fault.

Second, *increasing the input size may be a costly means to load a system*. For example, in the context of a compression application we studied, we found that to increase the response time of the application by 30 seconds one could use a 75MB file filled with random values or a 10MB file if the inputs are chosen carefully. This is particularly problematic if increasing the input size requires additional expensive resources in order execute the test (e.g., additional disk space, bandwidth).

Third, increasing the input size may just force the system to perform *more of the same* computation. In the worst-case, this would fail to reveal performance faults and, if a fault is detected, then further scaling is likely to repeatedly reveal the same fault. In functional testing, diversity in the test suite is desirable to achieve greater coverage of the system behavior. Load suites that *cover behaviors with different performance characteristics* are not a focus of current tools and techniques.

Finally, while most load testing focuses on response time or system throughput, there are many other *resource consumption measures* that are of interest to developers. For example, mobile device platforms place a limit on the maximum amount of memory or energy that an application can use.

In this paper we present an approach for the automated generation of load test suites that starts addressing these limitations. It generates load test suites that (a) induce load by using carefully selected input values instead of just increasing input size, (b) expose diversity of resource consuming program behavior, and (c) target a range of consumption measures.

The approach leverages recent advances in symbolic execution to perform a directed incremental exploration of the possible program paths. The approach is *directed* by a specified resource consumption measure to search for inputs of a given size or structure that maximize consumption of that resource (e.g., memory, execution time). The technique is *incremental* in that it considers program paths in phases. Within each phase it performs an exhaustive exploration. At the end of each phase, the paths are grouped based on similarity, and the most promising path from each group, relative to the consumption measure, is selected to explore in the next phase.

We have implemented the approach in the JPF infrastructure [21] and assessed its effectiveness on `JZLib`, `SAT4J`, and `TinySQL`. The results indicate that the proposed approach can produce suites that induce response times several times greater than random input selection and can scale to millions of inputs, increase memory consumption between 20% and 400% when compared with standard benchmarks, and expose different ways to induce high consumption of resources.

978-1-4577-1639-3/11/$26.00 © ASE 2011, Lawrence, KS, USA

II. Background and Overview

Test engineers have at their disposal many black box approaches to support load test suite generation which are based primarily on the manipulation of the input size. Overlooking the program's implementation details, however, means a loss of insights into how the structure of the input affects program performance, which can obfuscate the assessment. In this section, we provide an overview of our approach to develop load tests and begin with background on techniques that we use to realize that approach.

A. Mixed Symbolic Execution

Symbolic execution [11] simulates the execution of a program using *symbolic values* instead of actual values as inputs. Symbolic values are unknown and invariant – any computation with them must interpret them as any legal value and that value cannot change throughout the computation. For example, the symbolic execution of:

```
y = x;
if (y > 0) then y++;
return y;
```

would introduce a symbolic variable X to denote the value of variable `x` on entry to the code fragment. It then performs what amounts to a depth-first search (DFS) of the executable paths through the program. For this fragment, there are two such paths: (1) when $X > 0$ the value $X + 1$ is returned and (2) when $\neg(X > 0)$ the value X is returned. We note that the DFS nature of symbolic execution arises when symbolic values are tested in branch conditions. The extension of the path along the true branch will be symbolically executed, then backtracking will occur which will cause the extension along the false branch to be executed.

Paths are uniquely defined by a conjunction of constraints. Each constraint encodes a branch decision made along the path and is defined as a boolean expression over concrete and symbolic values. The conjunction is called the *path condition* (PC). The PCs for paths (1) and (2) are $X > 0$ and $\neg(X > 0)$, respectively. Each *symbolic state* in a symbolic execution defines a PC and a set of possible values (defined as expressions over concrete and symbolic values) for each program variable.

The simplicity of this example belies the true complexity of performing symbolic execution on real programs. Researchers have extended the basic symbolic execution approach, which was defined for integer variables, to treat heap allocated data and reference types [10]. While this allowed the application of symbolic execution to a wider range of programs, controlling the cost of symbolic execution has required mixing symbolic and concrete execution, e.g., [13]. In this approach, some inputs are given symbolic values and the rest are given actual data values. This significantly speeds up processing since the non-symbolic data values cause no increase in the space of paths that are explored, i.e., branches dependent only on non-symbolic values do not require backtracking.

In spite of the successes of mixed symbolic execution to generate functional tests, its application to the generation of load tests is not cost-effective. To illustrate this point we apply mixed symbolic execution to assess the response time of `TinySQL`, an SQL server that we study in some depth later in Section V. Assume that the goal is to validate whether the server can consistently provide response times below 90 seconds for a common query structure like the one in Figure 1 operating on a benchmark database. Under this setting the query structure is fixed and the database has concrete data, but the query's parameters are marked as symbolic. Even though this is rather a simple query, after 24 hours a mixed symbolic execution for test generation will still be running. It will have generated 274 tests by solving the paths conditions associated with an equal number of paths.

SELECT <SYM_FLD> <SYM_FLD>
FROM <SYM_TAB>
JOIN <SYM_TAB> **ON** <SYM_COND>
WHERE <SYM_COND> **OR** <SYM_COND>

Fig. 1. SQL Query template

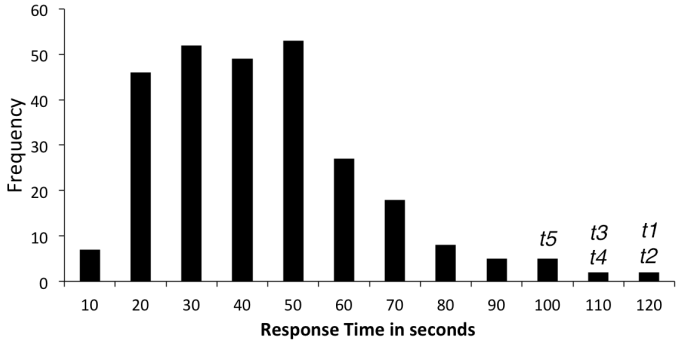

Fig. 2. Histogram of Response Time for TinySQL

Figure 2 shows a snapshot of the tests generated as a histogram where the x-axis represents `TinySQL` response times in seconds, and the y-axis represents the number of tests that caused that response time. We note that most tests cause a system response time of less than 50 seconds, but there is significant variability across tests ranging from 5 seconds to over 2 minutes. Of all these tests, we are interested in the ones on the right of the Figure — the ones causing the largest response times and most likely to reveal performance faults. The question we address is how to direct path exploration to obtain just those tests.

B. Proposed Approach to Induce Load

Rather than performing a complete symbolic execution, we perform an *incremental* symbolic execution to more quickly discard paths that do not seem promising, and *direct* the search towards paths that are distinct yet likely to induce high load as defined by a performance measure.

The approach requires for the tester to: (1) mark the variables to treat as symbolic just like in standard mixed symbolic execution [13], (2) specify the number of tests to be generated, denoted with $testSuiteSize$, and (3) select the performance measure of interest, denoted with $measure$, from a predefined and extendable set of measures. In addition, to control the way that the symbolic execution partitions the space of program

paths into *phases* an additional parameter, $lookAhead$, is needed. $lookAhead$ represents the number of branches that the symbolic execution looks ahead in assessing whether paths are diverse enough and of high load. (We discuss the selection of values for $lookAhead$ in the next section).

Given these developer supplied parameters, the approach performs an iterative-deepening mixed symbolic execution that checks, after a depth of $lookAhead$ branches, whether there are $testSuiteSize$ diverse groups of program paths and if so it then chooses the most promising path from each group based on $measure$.

Promoting diversity. After every $lookAhead$ branches, diversity is assessed on the paths reaching the next depth (short paths are deemed as having less load-generating potential and are hence ignored). The approach evaluates diversity by grouping the paths into $testSuiteSize$ clusters based on their common prefixes. A set of clusters is judged to have enough diversity when no pair of paths from different clusters have a common prefix whose length, in terms of branches taken, is within $lookAhead/testSuiteSize$ of either path in the pair. The intuition is that forcing clusters to diverge by more than $lookAhead/testSuiteSite$ branches will drive test generation to explore different behaviors that incur high load. If the diversity among clusters is insufficient, then exploration continues for another $lookAhead$ branches. Otherwise, the approach selects the most promising path from each cluster for further exploration in the subsequent symbolic execution phase, and the rest of the paths are discarded.

Favoring paths according to a performance measure. The selection of paths at the end of each phase is performed according to the chosen performance measure. So, for example, if the user is interested in response time, then paths traversing the most time consuming code are favored. If the user is interested in memory consumption, then the paths containing memory allocation actions are favored. Independent of the chosen measure, what is important is that each path has an associated cost performance summary that acts as a proxy for the performance measure chosen by the user. This summary is updated as a path is traversed and it is used to select promising paths for load testing.

Illustration. Figure 3 illustrates this approach with $testSuiteSize = 2$ and $lookAhead = 6$. These values mean that the common prefix of any pair of paths from two clusters must differ in more than 3 branches for the clusters to be considered diverse enough to enable path selection. The approach starts by performing an exhaustive symbolic execution up to depth 6. Then, it clusters the paths reaching that depth into two clusters – the grayed boxes labelled C1 and C2 at depth 6 in the Figure. These clusters, however, are deemed insufficiently diverse since a common prefix—whose end is marked with a triangle—is found that has a branch length of 2. Hence, the symbolic execution continues until depth 12 where the clustering process is repeated. In this case the diversity check is successful because the common prefix—whose end is marked with a square—differs in 4 branches which is more than $lookAhead/testSuitesize$.

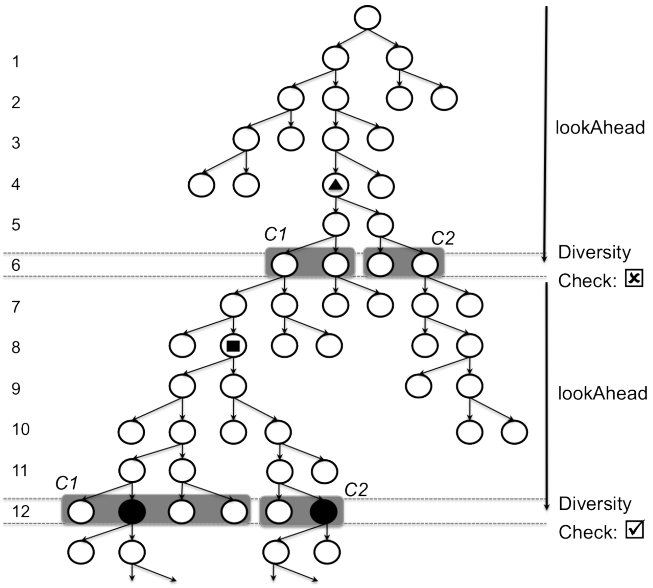

Fig. 3. Iterative-Deepening Beam Symbolic Execution

The approach in practice. Continuing with `TinySQL` and its query template, when the approach is configured to find values for the query template, to generate 5 tests, to use a $lookAhead = 50$, and to target the maximization of response time, our approach takes 159 minutes to generate a set of tests (11% of the cost of the 24 hours of symbolic execution).

As depicted by annotations $t1 \ldots t5$ in Figure 2, the tests identified by the approach do reside on the right hand size of the figure. Two of the generated tests are in the bin labelled 120 seconds (this bin includes all tests that cause response times of 115 seconds or more), 2 are in the 110 seconds bin, and 1 is in the 100 seconds bin. Each of the 5 generated tests takes at least twice the time of the original test from which the template was derived (which took less than 40 seconds).

In addition, as revealed later in the study (Table III), the generated tests are diverse. These five tests use many different fields, tables, and filtering clauses. Furthermore, they represent three types of database join operations, each of which impose heavy load on the system in a different way.

This section has illustrated how the approach works and its potential to generate load tests that induce high response times. In the next section we provide a detailed description of the approach and its parameterization capabilities to address a broad range of load testing scenarios.

III. Approach

This section presents our approach for load test generation, discussing its components, parameters, and application.

A. Symbolic Generation of Load Tests

Algorithm 1, **SymbolicLoadGeneration (SLG)**, details our load test generation approach. Conceptually, the algorithm repeatedly performs a bounded symbolic execution that produces a set of *frontier* symbolic states based on the branch look ahead, *clusters* those states based on the desired number of tests, and then, if the clusters are sufficiently diverse, selects

the most *promising* state from each cluster for further exploration[1]. Like other mixed symbolic execution techniques, e.g., [13], ours represents states as a combination of: (a) symbolic variables and their associated constraints for a selected subset of program inputs in the form of a path condition, and (b) concrete values for the remaining program inputs. In our algorithms, we measure the size of a path condition in terms of the number of constraints, or clauses, which corresponds to the number of branch decisions required along the path.

Algorithm 1 takes 5 parameters: 1) $\overline{init}$, the states from which the search commences, 2) $testSuiteSize$, the number of tests a tester wants to generate, 3) $lookAhead$, the increase in path condition size that an individual symbolic execution may explore, 4) $maxSize$, the size of the path condition that may be explored by the technique as a whole, and 5) $measure$, the cost measure used to evaluate promising states.

Each of the individual symbolic executions is referred to as a *level* and each level is bounded to explore states that add at most *lookAhead* constraints to the path condition of a previously generated state. This is achieved by incrementing *currentSize* by *lookAhead* and using the result to bound the symbolic execution. If the *currentSize* exceeds the *maxSize*, then the algorithm restricts the bound to produce states with path conditions of at most *maxSize*.

The first level begins from a set of states, $\overline{init}$, which forms the initial set of promising states, $\overline{promising}$. A level of the search, which corresponds to a call to $boundedSE()$, attempts to extend states from $\overline{promising}$ to paths of size $currentSize$. As that process proceeds $measure$ is used to update an associated performance estimate with each state.

Each time a $\overline{frontier}$ is formed, the function $frontierClustering$ is called to cluster the frontier states into $testSuiteSize$ clusters. The details of the clustering process are described below.

The goal of clustering is to identify sets of states that are behaviorally diverse—we measure diversity by differences in the branch decisions required to reach a state. If the clusters of states on the frontier are not sufficiently diverse, then we continue with another level of symbolic execution that attempts to extend the entire frontier another $lookAhead$ branches. Should the current $\overline{cluster}$ pass the $diversityCheck$, the function $selectStates()$ selects the state from each cluster with the maximum accumulated value for the performance measure. We discuss several such measures in Section IV.

The iterative-deepening search terminates if no promising state has a path condition whose size is *lookAhead* greater than the previous level's states ($largestSize()$ returns the largest path condition found in $\overline{promising}$).

When the search terminates, the path conditions associated with the $\overline{promising}$ states can be solved with the support of a constraint solver to generate tests as is done by existing automated generation approaches for functional tests.

[1]We describe our algorithm here in terms of symbolic states, but it is understood that each such state defines the end of the prefix of a path explored by symbolic execution.

Algorithm 1 SymbolicLoadGeneration ($\overline{init}$, testSuiteSize, lookAhead, maxSize, measure)

```
currentSize ← 0
promising ← init
search ← true
while search do
  currentSize ← currentSize + lookAhead
  if currentSize > maxSize then
    currentSize ← maxSize
    search ← false
  end if
  frontier ← boundedSE( promising, currentSize, measure)
  cluster ← frontierClustering(frontier, testSuiteSize)
  if diversityCheck(cluster) ∨¬ search then
    promising ← selectStates(cluster, measure)
  else
    promising ← frontier
  end if
  if largestSize( promising) < currentSize then
    search ← false
  end if
end while
return promising
```

B. Parameterizing SLG

In defining $\overline{init}$ a test engineer selects the portion of a program's input that is treated symbolically. Depending on the program one might, for example, fix the size or structure of the input and let the values be symbolic. An example of the former case is load testing of a program that processes inputs of uniform type but of varying size, such as the `JZlib` compression program. An example of the latter case is load testing of a program that processes structured input, such as the SQL query engine we study. For such a program the structure may be fixed, e.g., the number of columns to select, number of where clauses, etc, in the query, but the column names and where clause expressions are symbolic. In general, treating more inputs symbolically will generate more diverse and higher-quality load test suites, but such test generation may also incur greater cost. We expect that in practice, test engineers will start with a small set of inputs as symbolic and then explore larger sized inputs to confirm the observations made on smaller inputs.

The $testSuiteSize$ parameter determines how many tests are to be generated. Varying this parameter helps to produce a more comprehensive non-functional test suite for the application under test. Regardless of the size of the test suite, SLG always attempts to maximize diversity among tests. Exactly how many tests are required to perform a thorough testing on the non-functional aspect of interest of the application, however, is a harder question which cannot be assessed with traditional test adequacy measures, such as code coverage. In practice, selecting a test suite size will likely be an iterative process where test suite size is increased until a point of diminishing returns is reached [15]—where additional tests lack either diversity or high cost.

Bounding the depth of symbolic execution is a common technique to control test generation cost — the $maxSize$ parameter achieves this in our approach. The parameter

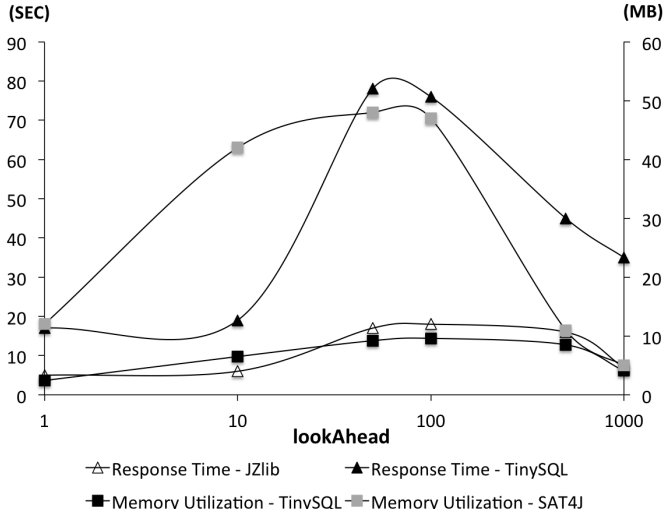

Fig. 4. Quality of load tests as a function of $lookAhead$

$lookAhead$, however, is particular to an iterative-deepening search as it regulates how much distance the search advances in one iteration. The larger the $lookAhead$, the more SLG resembles a full symbolic execution. Normally a smaller value for $lookAhead$ is desired, because a finer granularity would provide more opportunity for state pruning, which is key to the efficiency of our approach. Ultimately state pruning is decided by the diversity among the frontier of states, so a smaller $lookAhead$ alone cannot lead to ill-informed pruning. Setting $lookAhead$ too small may cause efficiency issues — a value of 1 will degrade the search to breadth-first. Figure 4 provides support for this intuition by plotting the quality of tests generated by our approach using $lookAhead$ values of 1, 10, 50, 100, 500, 1000 and 30 minutes of test generation time. The original tests for these programs execute an average of 16,000 branches, the selected $lookAhead$ values allow multiple iterations. The triangle plots show results for response time tests in seconds on the left axis; the black triangle plots are for the `TinySQL` program and the white triangle for `JZlib`. The square plots show results for maximum memory utilization tests in megabytes on the right axis; the black square is for the `TinySQL` program and the gray square for `SAT4J`. In each plot, the quality of the test rises as the $lookAhead$ increases to 50 and then drops off after 100. The reason for such a trend is that when $lookAhead$ is smaller than 50, the approach works less efficiently due to inserting many diversity checks prematurely, and when $lookAhead$ is larger than 100, much effort is wasted in exploring states that are going to be pruned. We use these insights to support the selection of *reasonable* $lookAhead$ values in the technique evaluations in Section V. In practice, a similar process could be performed automatically to select appropriate values for each system under test.

The last parameter, $measure$, defines how the approach should bias the search to favor paths that are more *costly* in terms of the non-functional measure of interest. The details of how the cost for a path is accumulated as the path is traversed and associated with the end state of that path, which is implemented within $boundedSE()$, and then used to select promising states, which is implemented in $selectStates()$, are abstracted in Algorithm 1. In general, our approach can accommodate many measures by simply associating different cost schemes with the code structures associated with a path. In our studies we explore response time and maximum memory consumption measures through the following cost computations:

a) Response Time Cost. This is a cumulative cost, so the maximal value occurs at the end of the path. We estimate response time by accumulating the cost of each bytecode. We use a very simple and configurable platform-independent cost model that assigns different weights to bytecodes based on their associated cost.

b) Maximal Memory Usage Cost. It attempts to record the largest amount of memory used at any point during a program execution by tracking operations that allocate/deallocate memory and increment/decrement a memory usage value by a quantity that reflects the object footprint of the allocated type. The maximal memory value is only updated if the current usage value exceeds it. As with response time, we find that this simple platform-independent approach strongly correlates to actual memory consumption.

Independent of the chosen performance measure and cost, our approach assumes that the measure constitutes part of a *performance test oracle*. We note that there are a large number of measures explored in the literature, and that they are often specified in the context of stochastic and queueing models [3], extended process algebras [9], programmatic extensions [5], or concerted and early engineering efforts focused on software performance [16]. Since such performance specifications are generally still hard to find, we take a more pragmatic approach by leveraging the concept of a *differential test oracle*. The idea behind differential oracles is to compare the system performance across successive versions of an evolving system to detect performance regressions, across competing systems performing the same functionality to detect potential anomalies or disadvantageous settings, or across a series of inputs considered equivalent to assess a systems's sensitivity to variations in common situations.

C. Clustering the Frontier and Diversity Checks

A convenient choice of clustering would be to use the classic *K-Means* algorithm [8] and define the number of unique clauses between two PCs as the *clustering distance*. However, this would require comparing across all pairs of PCs of frontier states and quickly runs into scalability issues. We devised an approximate algorithm that is linear in the number of frontier states. It makes use of the intuition that for frontier states resulting from a depth-first search, a pair of neighboring states are more likely to resemble each other than a pair of distant states. Algorithm 2 details the process. It takes $\overline{frontier}$ and the size of resulting cluster K as input, first computes the gap, in terms of the number of unique clauses, between each pair of $pc(s_i)$ and $pc(s_{i+1})$, then sorts the resulting gap vector to find

Algorithm 2 frontierClustering ($\overline{frontier}$, K)

```
cluster ← ∅
n ← |frontier|
for all s_i ∈ frontier, i ∈ (1, n − 1) do
    gap[i] ← diff(pc(s_i), pc(s_{i+1}))
end for
sortedGap ← descentSort(gap)
largestGap ← sortedGap[1, K-1]
for all s_i ∈ frontier, i ∈ (1, n − 1) do
    if gap[i] ∈ largestGap then
        cluster.createNewPartition()
    end if
    cluster.putInCurrentPartition(s_i)
end for
return cluster
```

Algorithm 3 ConstraintLimitedLoadGeneration($\overline{init}$, testSuiteSize, lookAhead, maxSize, measure, maxSolverConstraints)

```
currentSize ← maxSolverConstraints
while currentSize < maxSize do
    promising = SymbolicLoadGen(init, testSuiteSize, lookAhead, maxSolverConstraints, measure)
    init ← ∅
    for s ∈ promising do
        inputs ← solve(pc(s))
        init ← init ∪ stateAfterReplay(inputs, pc(s))
    end for
    currentSize ← currentSize + maxSolverConstraints
end while
outputTests(promising)
```

the largest $K-1$ gaps. The algorithm then uses the position of these $K-1$ largest gaps to partition the $\overline{frontier}$ into K clusters of various sizes.

The diversity check ensures that the gaps used to partition the $\overline{frontier}$ are of sufficient size to promote paths that have non-trivial behavioral differences. The threshold for such a gap could be defined in any number of ways. We use the heuristic $TH_{minPartitionGap} = \frac{lookAhead}{|\overline{cluster}|}$ since it balances the fact that, in general, larger $lookAhead$ generates greater diversity while larger values of $|\overline{cluster}|$ tends to reduce the difference between groups of states. This threshold is enforced by checking against the least largest gap that is used to define clusters.

D. Dealing with Solver Limitations

When Algorithm 1 returns $\overline{promising}$ a typical test generation technique would simply send the path conditions associated with those states to a constraint solver to obtain the test cases inputs. Because path conditions for candidate load tests often contain many tens of thousands of constraints, this basic approach will quickly expose the performance limitations of existing satisfiability solvers.

We address this by defining an outer search, Algorithm 3, that wraps calls to Algorithm 1 in such a way that the maximum number of symbolic constraints considered within any one invocation of Algorithm 1 is bounded. **ConstraintLimitedLoadGeneration(CLLG)** takes the same parameters as SLG plus an additional parameter $maxSolverConstraints$. This parameter can be configured based on the scalability of the constraint solver used to implement symbolic execution.

Algorithm 1 is invoked with $maxSolverConstraints$ as the $maxSize$ parameter that governs the overall size of the iterative-deepening symbolic execution. When $\overline{promising}$ is returned, Algorithm 3 solves the path constraints of each state in $\overline{promising}$ to obtain the input values necessary to reach those states. Then, it replays the program using those concrete inputs and, when the program has traversed all the predicates in the path condition, the program state is captured and added to $\overline{init}$. The algorithm terminates when the sum of sizes of the path conditions explored in each of the invocations of $SymbolicLoadGen()$ exceeds the $maxSize$.

In essence, Algorithm 3 increases the scalability of our approach by chaining partial solutions together through the use of concrete input values. While this may sacrifice the quality of generated tests, it can help overcome the limitations of satisfiability solvers and allow greater scalability for load test generation. We explore this tradeoff further in Section V.

Finally, we note that when $maxSolverConstraints = maxSize$, $ConstrainedLoadGeneration()$ runs a single instance of $SymbolicLoadGeneration()$. Consequently, we use the former as the entry point for our technique.

IV. Implementation

We have implemented the approach by adapting JPF and its symbc extension [13], which supports symbolic execution. Our modifications enable us to associate a performance measure with a cost for each path, to record the path conditions leading to a state, and to compute diversity, all of which are necessary to implement Algorithms 1 and 3.

Path Performance Estimators. We have implemented two performance cost schemes, one aiming to account for response time and one for maximum memory consumption. For response time, we use a simple weighted bytecode count scheme that assigns a weight of 10 to `invoke` bytecodes and a weight of 1 assigned to all others bytecodes. The implementation allows for the addition of more fine grain schemes (e.g., [28]). The memory consumption costing scheme takes advantage of JPF built-in object life cycle listener mechanism to track the heap size of each path, and associate this value with each path. Neither scheme takes into account the JIT behavior or architectural details of the native CPU. In our experience so far these simple schemes has been quite effective, but this is an area that we have just begun to explore. So we have designed our implementation for other estimators to be easily added. For instance, a cost estimator related to the number of open files in the system can be easily added by tracking open and close calls on file APIs, and multiple measures can be combined by using a weighted sum of individual measures.

Bounded Symbolic Execution. We run JPF configured to perform mixed symbolic execution. When a phase of symbolic execution finishes, the $selectState()$ function is invoked to compute a subset of the frontier states for further exploration. In the current implementation, the branch choices made along the paths leading to those states are externalized to a file.

Then JPF is restarted using the recorded branch conditions to guide execution up to the frontier, and then resume its search towards until the next frontier. This solution is efficient because the satisfiability of the path condition prefix is known from the previous symbolic execution call, thus JPF is simply directed to execute a series of branches and no backtracking is needed for those branches. We note that we have explored alternative mechanisms to avoid recording and replaying path prefixes stored in files and to avoid restarting JPF. We found that none was as efficient in scaling to large programs as the replay approach described above. This approach also has the distinct advantage of allowing the exploration of the recorded path prefixes to proceed in parallel, which is a strategy we plan to pursue in future work. We also use the replay mechanism to implement the $stateAfterReplay()$ function used in Algorithm 3.

Diversity Clustering. Algorithm 2 takes advantage of the JPF backtracking support to efficiently compute the $\overline{gap}$ vector on the fly, since each gap between two states s_i and s_{i+1} equals the number of branches s_i needs to backtrack before it can continue on a new path that leads to s_{i+1}.

Test Instantiation. Generating tests from a path condition requires the ability to both solve and produce a model for a given formula. We have explored the use of several constraint and SMT solvers, including Choco [17], CVC3 [20], and Yices [26]. Our integration of Yices into JPF's symbc extension has greatly improved the scalability of symbc in general and of our technique in particular. The data reported in our study uses this Yices-based implementation.

V. Evaluation

Our assessment takes on multiple dimensions, resources, and artifacts.

First, we assess the approach configured to induce large response times and compared it against random test generation. Then, we explore the scalability of various instantiations of the approach. We do this in the context of JZlib [22] (5633 LOC), a Java re-implementation of the popular Zlib package. This program is well suited for the study because we can easily and incrementally increase the input size (from 1KB to 100MB) to investigate the scalability of the approach and response time is one of its two key performance evaluation criteria. Moreover, we can easily generate what the Spec2000 benchmark documentation [18] defines as the worst load test inputs for JZlib – random values – to increase response time and use those to compare against our approach. Last, we can use the Zlib package as the differential oracle.

Second, we assess an instantiation of the approach to generate a suite of 10 tests that follow a prescribed input structure with the goal of inducing high memory consumption. We conduct this assessment with SAT4J [24] (10731 LOC) which is well suited for the study in that memory consumption is often a concern for such kind of applications. Furthermore, the SAT benchmarks [23] that already aim to stress this type of applications offer a high baseline against which to assess our approach.

Third, we assess an instantiation of the approach to generate three test suites, one that causes large response times, one that causes large memory consumption, and one that strives for a compromise. We use TinySQL [25] (8431 LOC) and its sample queries and database, as described earlier. Since the tests for this artifact can be easily interpreted (they are SQL queries), we perform a qualitative comparison on their diversity.

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. We configure it to use $lookAhead = 50$ as per the description in Section IV and $maxPCSize = 30000$ in order to keep within the capabilities of the various constraint solvers we used [2].

A. Revealing Response Time Issues

Our study considers two load test case generation techniques as treatments: CLLG and Random. Each test suite consists of 10 tests. CLLG is configured to target response time. We imposed a cap of 3 hours for all the test generation strategies. Strategies requiring more time were terminated and not reported as part of the series. For Random test generation we used the whole allocation of 3 hours to simplify the study design, which is conservative in that it may overestimate the effectiveness of the Random test suites in cases were the CLLG suites for the same input size took less than 3 hours to generate. For the Random treatment, we use a random byte generator to create the file streams to be expanded. Another consideration with Random is that, unlike our approach with its built-in selection mechanism, it does not include a process to distinguish the best 10 load tests. This is important as Random can quickly generate millions of tests from which only a small percentage may be worth retaining as part of a load test suite. To enable the identification of such tests, we simply run each test generated by Random once and record its execution time. This execution time, although generally brief, is included in the overall test case generation time.

We first compare the response times of JZlib caused by tests generated with our two treatments. We use Zlib, a program with a similar functionality, as part of a differential oracle that defines a performance failure as: $responseTime_{JZlib} - responseTime_{Zlib} > \delta$. Figure 5 describes, for input sizes ranging from 100Kb to 1MB, the ratio in response times between JZlib and Zlib when a randomly generated suite is used (light bars) versus when inputs generated by CLLG are used (dark bars). (The values shown are averaged across the ten tests). We see that CLLG generates inputs that reveal greater performance differences, more than twice as large as random with the same level of effort. It is also evident that, depending on the choice of δ, our approach could increase fault detection. For example, if δ = 1 sec, then using random inputs will not reveal a performance fault with the input sizes

[2]This number could be adjusted depending on the time available and the selected constraint solvers. As noted earlier, however, all solvers would eventually struggle to process an increasing number of constraints so selecting this bound is necessary.

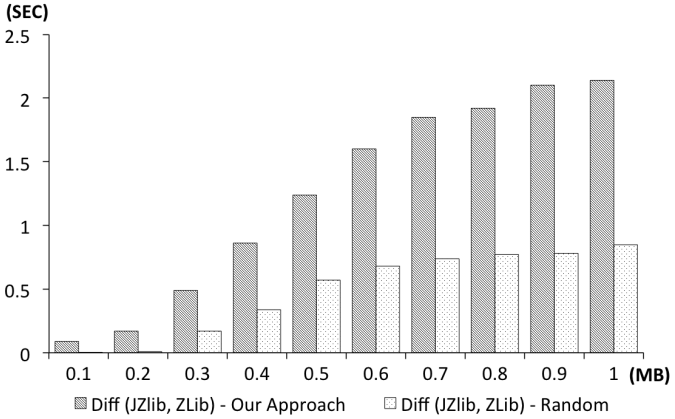

Fig. 5. Revealing performance issues: response time differences of JZlib vs Zlib when using testing suites generated by CLLG vs Random

being considered while the tests generate by CLLG would reveal the fault with an input of 0.4MB or greater.

Scalability. To better understand the scalability of the approach and the impact of the $maxSolverConstraints$ bound on the effectiveness of CLLG, we increased the input size up to 100MB and used $maxSolverConstraints$ values of 100, 500, and 1000. We again imposed a cap of 3 hours for all the test generation strategies. Strategies requiring more time were terminated and not reported as part of the series.

The scalability results are presented in Figure 6, which plots the response times averaged across the tests in each suite. The trends confirms the previous observations. The response time of JZlib is several times greater for the test suites generated with CLLG strategies than with those generated by Random.

This is more noticeable for CLLG test suites with greater $maxSolverConstraints$. For example, for an input of 10MB, the suite generated with CLLG-1000 had an average response time that was approximately five times larger than Random and two times larger than that with CLLG-100. However, this strength came at the cost of scalability as the former strategy could not scale beyond 15MB. We note similar trends for the other test suites generated with the CLLG strategies, where each eventually reached an input size that required more than the 3 hour cap to generate the test cases. Only the CLLG-100 is able to scale to the more demanding inputs of up to 100MB. Still, the response time of JZlib under the test suite generated with this strategy is more than 3 times greater than the one caused by the Random test suite for approximately the same generation cost. Furthermore, to generate a response time of 40 seconds, a randomly generated test would require on average an input of more than 100MB while CLLG-100 would require a file smaller than 25MB. More importantly, this figure offers evidence of the approach configurability, through the $maxSolverContraints$ parameters, to scale and yet it can outperform an alternative technique that for this particular artifact is considered the worst case [18].

B. Revealing Memory Consumption Issues

We now proceed to assess 5 test suites of 10 tests generated by our approach to induce high memory consumption in SAT4J. We randomly chose 5 benchmarks from the SAT

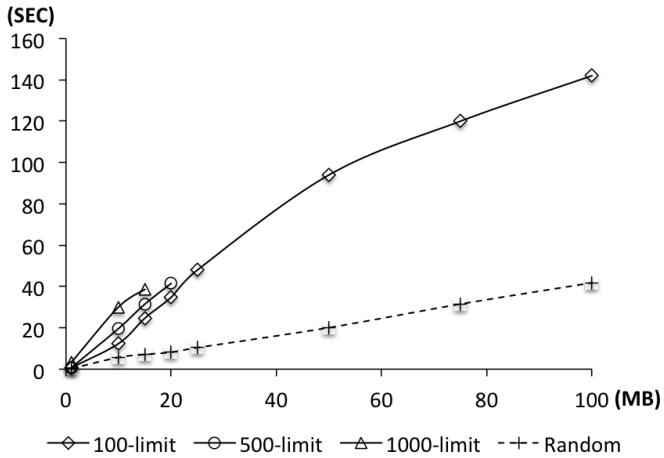

Fig. 6. Scaling: response times of JZlib with test suites generated by CLLG and Random

TABLE I
LOAD TEST GENERATION FOR MEMORY CONSUMPTION

Programs	Description			Memory (MB)	
	Variables	Clauses	Hardness	Orig.	Gen.
aloul-chnl11-13	286	1742	medium	6	22
cmu-bmc-longmult15	7807	24351	easy	35	92
eq-atree-braun-8-unsat	684	2300	medium	11	35
rbcl-xits-06-UNSAT	980	47620	hard	38	45
unif-k3-r4.25	480	2040	medium	6	17

competition 2009 suite to get a pool of potential values for the number of variables, number of clauses and the number of variables within each clause, and we declare the variables themselves as symbolic. Columns 2 to 4 of Table I show some of the attributes of the selected benchmarks including the number of variables, number of clauses, and hardness level (assigned based on the results of past competitions, with instances solved within 30 seconds by all solvers labelled as "easy", not solved by any solver within the timeout of the event labelled as "hard", and "medium" for the rest.)

Table I shows the memory consumption results in the last two columns. Column "Orig." shows memory consumption of SAT4J when these benchmark programs are provided as inputs, and column "Gen." shows the same measure for the average of the ten tests generated with our approach. Overall, the approach was effective in increasing the memory load for SAT4J compared with the original benchmarks. However, the gains are not uniform across instances. The most dramatic gain achieved by our approach, almost a 4x increase in memory consumption, comes from selecting values for `aloul` which is of "medium" hardness. Not surprisingly, the least significant gain comes from `rbcl-xits-06-UNSAT` which is classified as "hard". But even for this input with a very large and challenging space of over 47000 clauses the approach leads to the generation of a test suite that on average consumes 20% more memory.

C. Load inducing tests across resources and test diversity

We now assess the approach in the presence of different performances measures and in terms of the diversity of the test suite it generates.

We generate three test suites with 5 tests each for TinySQL.

TABLE II
RESPONSE TIME AND MEMORY CONSUMPTION FOR TEST SUITES DESIGNED TO INCREASE THOSE PERFORMANCE MEASURES IN ISOLATION (TS-RT AND TS-MEM) AND JOINTLY (TS-RT-MEM).

Test Suite	Response Time		Memory Consumption	
	Seconds	% over Bas.	MB	% over Bas.
Baseline	45	100	6.6	100
TS-RT	105	234	13	196
TS-MEM	98	217	15	219
TS-RT-MEM	96	213	13	201

TABLE III
QUERIES ILLUSTRATING TEST SUITE DIVERSITY

```
SELECT MUSIC_TRACKS.track_name, MUSIC_TRACKS.track_id
FROM MUSIC_TRACKS
JOIN MUSIC_COLLECTION ON
    MUSIC_TRACKS.track_id = MUSIC_COLLECTION.track_id
WHERE MUSIC_TRACKS.track_name <> null OR
    MUSIC_TRACKS.track_id > 0
```

```
SELECT MUSIC_ARTISTS.artst_id, MUSIC_ARTISTS.artst_name
FROM MUSIC_ARTISTS
JOIN MUSIC_EVENTS ON
    MUSIC_ARTISTS.artstid <> null
WHERE MUSIC_ARTISTS.artst_name <> null OR
    MUSIC_ARTISTS.artst_id > 0
```

```
SELECT MUSIC_ARTISTS.artst_name, MUSIC_ARTISTS.artst_country
FROM MUSIC_ARTISTS
JOIN MUSIC_ARTISTS ON
    MUSIC_ARTISTS.artst_name <> MUSIC_ARTISTS.artst_name
WHERE MUSIC_ARTISTS.artst_id > 5 OR
    MUSIC_ARTISTS.artst_country <> null
```

The first, TS-RT, favors response time. The second, TS-MEM, favors memory consumption. The third, TS-RT-MEM, was generated with an equally weighted sum of the cost for response time and memory consumption. Table II shows the performance caused by each test suite averaged across the five tests. We use as baseline the original test from which the test template was derived. We report both response time and memory consumption, along with their respective effectiveness over the baseline, for each suite.

The results show that all three test suites are effective at increasing their respective measures. On average, TS-RT forces response times to rise 234% over the baseline, TS-MEM causes a 217% increase over the baseline in terms of memory usage, TS-RT-MEM increases both response time by 213% and memory by 201% over the baseline. By looking at TS-RT and TS-MEM is clear that favoring response time or memory consumption has an effect on their counterpart. As expected, the TS-RT-MEM suite does in between the other two suites in terms of memory consumption. What was a bit surprising is that TS-RT-MEM average response time was lower than TS-MEM. Although the difference is less than 4%, this indicates that when using combinations of performance measures special care must be taken to integrate the different costs schemes to account for potential interactions.

Test suite diversity. We now turn our attention to the issue of test suite diversity. We note that there are no identical queries – TinySQL tests – within each of the generated test suites. Furthermore, all queries complete in different times (differences in tenths of seconds) and consume different memory (differences in KB). We illustrate some of the differences with the sample queries in Table III. Because of space constraints, we only include 3 of the generated queries which, like all others, have various degrees of difference. Some obvious differences are in the fields selected, tables retrieved, and the type of where clauses specifying the filtering conditions. Others are more subtle but still fundamental. For example, while the first query in the Figure is an inner join (it will return rows with values from the two tables with matching track_id), the second one is a cross-join (it will return rows that combine each row from the first table with each row from the second table), and the third one is a self-join (joining content from rows in just one table.) Although this is just a preliminary qualitative evaluation, it provides evidence that the path diversity pursued by the approach translates into behaviorally diverse tests.

VI. RELATED WORK

There are a large number of tools for supporting load testing activities [19], some of which offer capabilities to define an input specification (e.g., input ranges, recorded input session values) and to use those specifications to generate load tests [12]. A common trait among these tools is that they provide limited support for selecting load inducing inputs as they all treat the program as a black box. The program is not analyzed to determine what inputs would lead to higher loads. Similar trends appear in load testing techniques and processes in general as they use other sources of information (e.g., user profiles [1], adaptive resource models [2]) to decide how to induce a given load, but still operating from a black box perspective. One interesting exception proposed by Yang et al. [27]. Conceptually, the approach aims to assign load sensitivity indexes to software modules based on their potential to allocate memory, and use that information to drive load testing. Our approach also considers program structure, but a key difference in that instead of having to come up with static indexes our approach explores the program systematically with the support of symbolic execution to identify promising paths that we later instantiate as tests.

A second thread of related work targets improvements to the performance, scalability, and applicability of symbolic execution More specifically, many efforts have studied how to reduce, guide, and enrich the space explored by symbolic execution [14]. Our approach takes advantage of some of these advances, particularly on the use of mixed symbolic execution, but is unique in its application to load testing, which requires different search heuristics and exposes different tradeoffs and scalability issues.

Our work is also related to research efforts aimed at characterizing the computational complexity of a program as load testing often tries to expose worst case scenarios. Goldsmidth et al. propose an approach that, given a set of loads, executes the program under those loads, groups code blocks of similar performance, and applies various fit functions to the resulting data to summarize the complexity [6]. Critical to the performance of this approach is the user provided workloads. Gulwani et al. take a static approach [7]. Their 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.

The last piece of related work is the closest to our approach in that it uses symbolic execution to identify a worst-case scenario [4]. Our approach is different in two significant respects. First, our goal is develop a suite of diverse tests, not just identifying the worst-case. This requires the incorporation of additional mechanisms and criteria to capture distinct paths that contribute to a diverse test suite, and of a family of performance estimators that can be associated with program paths. Second, the approach is different. Burmin's approach utilizes full symbolic execution on small data sizes, and then attempts to generalize the worst case complexity from those small scenarios. This works well when the user can provide a branch policy indicating what branches or branch sequences should be taken or not in order to generalize the worst-case from small examples, and the authors show sample programs where that is the case. However, for programs like the ones we studied defining even reasonable branch policies that merely approximate the worst-case scenario, much less a test suite, it would require an extremely good understanding of the program behavior and even then it would be challenging. Our approach is different in that we perform an incremental symbolical execution favoring the deeper exploration of a subset of the paths associated with code structures. This removes the requirement for a user provided generator.

VII. Conclusion

Load testing can assist in the detection of performance anomalies and many tools and approaches exist to support the test engineer in load test suite development. Most of those efforts treat the program as a black box, focusing on increasing load by providing larger input sizes. As we have shown, however, size is not all that matters. The selection of the right combination of input values can deliver an equivalent load with smaller input sizes which can reduce testing infrastructure requirements, can provide a more accurate characterization of scenarios where the system behaves poorly, can cover a range of resources, and may be helpful to identify anomalies that are exposed when traversing different execution paths.

Yet, identifying such inputs can be extremely challenging since it requires an understanding of the program internals. To address this challenge for smarter input selection, we developed SLG, an approach that performs a focused form of symbolic execution, utilizing iterative-deepening and beam search strategies, on portions of the system with the aim of discovering execution paths that contribute to high program loads while ensuring path diversity. Our implementation [3] and assessment of SLG shows that it can induce program loads across different types of resources that are significantly better than alternative approaches (randomly generated tests in the first study, a standard benchmark in the second study, and the default suite in the third study). Furthermore, we provide evidence that the approach scales to inputs of large size and complexity and produces functionally diverse test suites.

[3]Source code is available at http://cse.unl.edu/~pzhang/symload

Acknowledgments

This material is based in part upon work supported by NSF award CCF-0915526 and by AFOSR award #9550-09-1-0687. 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. We thank W. Visser and V. Cortellessa for providing feedback on earlier versions of this work.

References

[1] A. Avritzer and E. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Trans. Softw. Eng.*, 21(9), Sept 1995.
[2] M. Bayan, Cangussu, and Jo ao W. Automatic feedback, control-based, stress and load testing. In *Proc. ACM Symp. Applied Comp.*, 2008.
[3] G. Bolch, S. Greiner, H. de Meer, and K. Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. Wiley-Interscience, 1998.
[4] J. Burnim, S. Juvekar, and K. Sen. Wise: Automated test generation for worst-case complexity. In *Proc. Int'l. Conf. Softw. Eng.*, 2009.
[5] S. Frolund and J. Koistinen. Qml: A language for quality of service specification. Technical Report 10, HP Laboratories, 1998.
[6] S. Goldsmith, A. Aiken, and D. Wilkerson. Measuring empirical computational complexity. In *Proc. Euro. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2007.
[7] 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.
[8] J. A. Hartigan and M. A. Wang. Algorithm as 136: A k-means clustering algorithm. *Journal Royal Stat. Society*, 1979.
[9] H. Hermanns, U. Herzog, and J. P. Katoen. Process algebra for performance evaluation. *Theory. Comp. Sci.*, 274(1-2):43–87, 2002.
[10] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. Int'l Conf. Tools Alg. Constr. Anal. Systems*, 2003.
[11] J. C. King. Symbolic execution and program testing. *Comm. of the ACM*, 19(7), 1976.
[12] Silk Performer. http://microfocus.com/products/SilkPerformer.
[13] C. Păsăreanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proc. Int'l Symp. Softw. Test. Anal.*, 2008.
[14] C. Păsăreanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int'l. J. Softw. Tools Tech. Transf.*, 2009.
[15] E. Sherman, M. Dwyer, and S. Elbaum. Saturation-based testing of concurrent programs. In *Proc. Euro. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2009.
[16] C. Smith and L. Williams. *Performance solutions: a practical guide to creating responsive, scalable software*. Addison Wesley Longman Publishing Co., Inc., 2002.
[17] Choco Solver. http://www.emn.fr/x-info/choco-solver/doku.php.
[18] SPEC. http://www.spec.org/cpu2000/CINT2000/164.gzip/docs/164.gzip.
[19] Load Test Tools. http://www.softwareqatest.com/qatweb1.html.
[20] CVC3 Solver Website. http://www.cs.nyu.edu/acsys/cvc3/.
[21] Java Path Finder Website. http://babelfish.arc.nasa.gov/trac/jpf.
[22] JZlib Website. http://www.jcraft.com/jzlib/.
[23] SAT Competition Website. http://www.satcompetition.org/.
[24] SAT4J Website. http://www.sat4j.org/.
[25] TinySQL Website. http://www.jepstone.net/tinySQL/.
[26] Yices Solver Website. http://yices.csl.sri.com/.
[27] C. Yang and L. Pollock. Towards a structural load testing tool. In *Proc. Int'l Symp. Softw. Test. Anal.*, 1996.
[28] L. Zhang and C. Krintz. Adaptive code unloading for resource-constrained jvms. In *Proc. Lang. Compilers Tools Embed. Sys.*, 2004.