JST: An Automatic Test Generation Tool for Industrial Java Applications with Strings

Indradeep Ghosh*, Nastaran Shafiei**†, Guodong Li*, Wei-Fan Chiang**‡

*Software Systems Innovation Group, Fujitsu Laboratories of America, Sunnyvale, CA, USA
†Department of Computer Science and Engineering, York University, Toronto, ON, Canada
‡School of Computing, University of Utah, Salt Lake City, UT, USA

***Abstract*—In this paper we present JST, a tool that automatically generates a high coverage test suite for industrial strength Java applications. This tool uses a numeric-string hybrid symbolic execution engine at its core which is based on the Symbolic Java PathFinder platform. However, in order to make the tool applicable to industrial applications the existing generic platform had to be enhanced in numerous ways that we describe in this paper. The JST tool consists of newly supported essential Java library components and widely used data structures; novel solving techniques for string constraints, regular expressions, and their interactions with integer and floating point numbers; and key optimizations that make the tool more efficient. We present a methodology to seamlessly integrate the features mentioned above to make the tool scalable to industrial applications that are beyond the reach of the original platform in terms of both applicability and performance. We also present extensive experimental data to illustrate the effectiveness of our tool.**

I. Introduction

With the ubiquitous presence of software programs permeating almost all aspects of daily life, it has become a necessity to provide robust and reliable software. Traditionally, software quality has been assured through manual testing which is tedious, difficult, and often gives poor coverage of the source code especially when availing of random testing approaches. This has led to much recent work in the arena of formal validation and testing. One such formal technique is symbolic execution [1], [2], [3], [4] which can be used to automatically generate test inputs with high structural coverage for the program under test.

Symbolic execution is a model checking technique that treats input variables to a program as unknown quantities or symbols [5]. It then creates complex equations by executing all possible finite paths in the program with the symbolic variables. These equations are then solved through an SMT (Satisfiability Modulo Theories, [6]) solver to obtain test cases and error scenarios, if any. Thus this technique can reason about all possible values at a symbolic input in an application. Though it is very powerful, and can lead to detection of interesting input values that uncover corner case bugs, it is computationally intensive. Hence some techniques and methodology are needed to make symbolic execution scale to industrial size applications.

Even though traditional symbolic execution has mainly dealt with numeric input variables, industrial Java applications are different as almost all inputs to these applications are strings. While some string inputs are used and manipulated as strings inside these applications, other inputs are converted to integers or floating point numbers after extensive format checking in the string domain. Such back-and-forth conversions pose unique challenges to the symbolic execution tool which has traditionally handled only numeric constraints well. Even in the numeric domain traditional SMT solvers cannot handle non-linear equations as solving such equations is undecidable in general. However, the industrial examples that we deal with routinely have non-linear operations which we have to tackle during symbolic execution. Instead of giving up on such situations, we have devised some techniques that are able to circumvent the problem in certain situations which we describe in this paper. Finally, symbolic execution suffers from the path explosion problem which is even more acute in large industrial examples. In this paper we describe effective steps to eliminate large portions of the symbolic search tree thus making the analysis engine scale to large examples.

The tool that we present in this paper, JST (Java String Testing), is a comprehensive Java testing tool that addresses the above described issues in traditional symbolic execution engines. It has extensive support for string operations and complex interactions between strings and numbers. For example, the JST symbolic executor can handle virtually all Java string operations, regular expressions, as well as string-number conversions. In addition, we have also added support for symbolic container classes like *Maps*, *Arrays*, *etc.*, and other numeric data structures like *BigDecimal* and *BigInteger* which are widely used in financial Java applications.

JST is based on the Java PathFinder [1] (JPF) model checker and its symbolic execution extension, Symbolic PathFinder [2], [7]. JPF consists of a highly configurable and easy to extend toolkit which is the main reason for using it as our underlying platform. JPF implements its own Java virtual machine (JVM) to execute Java bytecode which we have extended to handle all Java primitive types and Strings. We have addressed many bottlenecks in this process through a variety of innovations, each of which is essential to achieve

** Contributed to this work during an intership program at Fujitsu Laboratories of America, Sunnyvale, CA, USA

[1]http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-core

scalability of the tool and the desired quality of results. We summarize these experiences in the following sections.

We organize the paper as follows. We first give the background and a motivating example. This is followed by the details of the symbolic execution framework and the string-numeric hybrid solver. Finally, after presenting experimental results, we end with the discussion and conclusion.

II. Background and Motivating Example

Symbolic execution is used to achieve high code coverage by reasoning on all possible input values. It characterizes each program path it explores with a path condition encoded as a conjunction of Boolean clauses. A path condition denotes a set of branching decisions. When the execution is finished, multiple path conditions may be generated, each corresponding to a feasible execution path with respect to the symbolic inputs. The solutions to path conditions are the test inputs that will assure that the program under test runs along a particular concrete path during concrete execution. Exhaustive testing is achieved by exploring all true paths.

In addition to comprehensiveness, symbolic execution has other benefits. First, it actually "executes" the code in a real environment, hence eliminating the need to build models, or apply abstract analysis. Second, it is highly automated, producing test cases without requiring the intervention from users [1], [8]. This makes it a good choice to apply symbolic execution to realistic programs such as web applications.

On the other hand, there exist various problems that preclude its widespread use. The main issue is poor scalability due to the problem of path explosion. This is because the engine creates a new path for every comparison, or branching instruction and may create thousands of paths. What is worse, each branching or assertion check (*e.g.* on memory out-of-bound accesses) in the program will invoke the solver for a satisfiability check. The vast number of invocations to the solver makes constraint solving the main bottleneck of a symbolic execution tool. This necessitates good solving techniques which is one focus of this work. Specifically, we have developed an in-house string-numeric solver to cater to the special needs of Java web applications.

Now, consider an example with its corresponding branching tree shown in Figure 1. As the tree shows, at each branching point (if statement) the path condition splits into two, and each branch extends with new constraints. Notation `L(q)` represents the length of string `q`. This example first checks whether the symbolic input `s` starts with '-' such that `s` may represent a negative number. If so, it checks whether `s` is of a popular format represented by a regular expression (*e.g.* a string starting with '-', followed by at least one digit, and a comma, and then 3 digits). Then it checks whether ',' appears in `s`. If not then `s` is converted into a BigDecimal number. Otherwise, the substring after character ',' is taken and converted into an integer `x`. Then the computation continues by checking whether `x` is not less than 100. This is a typical computation sequence in web applications. The application first performs format checking on input strings. Then it converts the strings to numbers and finally, branches over arithmetic operations on those numbers. For example, a valid test case for path 4 is "-1,000,200". It is not trivial to automatically cover all the branches through constraint solving, especially when the string formats and the numeric computation are very complicated.

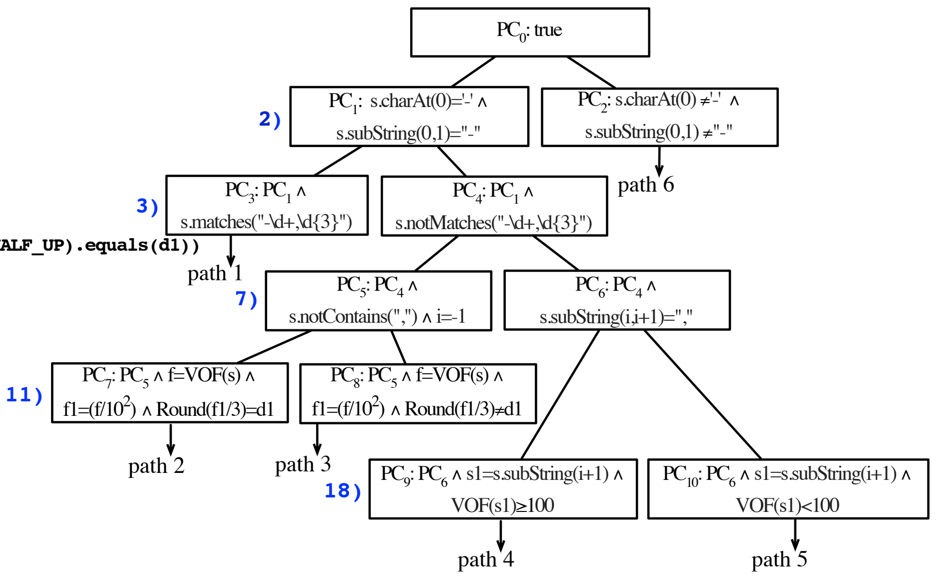
The satisfiability of string+numeric constraints is an undecidable problem in general ([9] shows that a small subset of string operations will result in undecidability of a hybrid set of constraints). Hence practical solutions are important to tackle string-intensive programs. Existing string solvers (see [10] for a comparison of the automaton-based approaches) cannot fulfill our needs completely. These solvers provide no support or only very limited support for hybrid constraints, *i.e.* non-trivial combinations of numeric constraints and string constraints. In contrast, our solver supports almost all Java string operations and, more importantly, hybrid constraints. In addition, we use various techniques to control and optimize the solving during symbolic execution rather than use the solver as a black box.

The motivating example in Figure 1 involves the following techniques:

1) `s.matches(...)` checks whether `s` is of a specific format, and `f=Integer.parseInt(s)` checks whether `s` is an integer. For these cases, we need to intersect the automaton representing `s` and the ones modeling the regular expression and the syntax of valid integers. This may incur complicated automaton refinements.
2) `s.lastIndexOf(...)` may return a symbolic integer whose value is used to break `s` into multiple parts. Some parts may be converted into numbers (*e.g.* through `parseInt`) and then used in intensive computations (*e.g.*, `x>=100`). Hence the path condition may contain hybrid constraints manipulating strings and numbers in tricky ways. Our solver handles these constraints by connecting the string and numeric domains through guided iterations, relational graphs, refinement rules, and other advanced techniques.
3) `f` is further converted into a specific data structure `BigDecimal`, which stores the numeric values in adhoc format. Then some rounding operations are applied. We model many widely-used data structures.
4) In order to make the tool scale, we use *relaxed solving*, a two-tier execution and solving technique, to handle hybrid constraints at run-time. Specifically, we separate the two domains and avoid iterations between them when checking the satisfiability of an intermediate branch, and only apply the iterations at the end of a path. This key optimization enables us to achieve a good balance between accuracy and performance.

Among these, Item 4 has not been used in any prior work, and Item 2 has been studied to a very limited extent. The automaton refinement scheme in Item 1 is quite different from existing automaton-based ones. One of the first attempts at solving hybrid constraints using automata in the string domain is described in our earlier work [11]. This paper

```
 1) String s; // symbolic input
 2) if (s.charAt(0)=='-') {
 3)  if (s.matches("-\d+,\d{3}"))
 4)   ...;  // path 1
 5)  else {
 6)   int i=s.lastIndexOf(',');
 7)   if (i==-1) {
 8)    int f=Long.parseLong(s);
 9)    BigDecimal f1=BigDecimal(f,2);
10)    BigDecimal d1=BigDecimal(-1);
11)    if (f1.divid(3,BigDecimal.ROUND_HALF_UP).equals(d1))
12)     ...; // path 2
13)    else
14)     ...; // path 3
15)   } else {
16)    String s1=s.substring(i+1);
17)    int x=Integer.parseInt(s1);
18)    if (x>=100)
19)     ...; // path 4
20)    else
21)     ...; // path 5
22)   }
23)  }
24)} else
25) ...; // path 6
```



PC_0: true
2)
PC_1: s.charAt(0)='-' ∧ s.subString(0,1)="-"
PC_2: s.charAt(0) ≠'-' ∧ s.subString(0,1) ≠"-"
path 6
3)
PC_3: PC_1 ∧ s.matches("-\d+,\d{3}")
PC_4: PC_1 ∧ s.notMatches("-\d+,\d{3}")
path 1
7)
PC_5: PC_4 ∧ s.notContains(",") ∧ i=-1
PC_6: PC_4 ∧ s.subString(i,i+1)=","
11)
PC_7: PC_5 ∧ f=VOF(s) ∧ f1=(f/10^2) ∧ Round(f1/3)=d1
PC_8: PC_5 ∧ f=VOF(s) ∧ f1=(f/10^2) ∧ Round(f1/3)≠d1
path 2
path 3
18)
PC_9: PC_6 ∧ s1=s.subString(i+1) ∧ VOF(s1)≥100
PC_{10}: PC_6 ∧ s1=s.subString(i+1) ∧ VOF(s1)<100
path 4
path 5

Fig. 1. Our motivating example and its corresponding branching tree with path conditions.

summarizes the advances that we have made in the solver after that. Even with all these techniques, it is still a challenge to handle industrial strength Java applications. We will discuss the difficulties we encountered and the solutions that we finally settled on, in the following sections.

III. THE EXECUTION FRAMEWORK

JST is built on top of the JPF platform, an explicit state model checker that implements a customized Java virtual machine as its core and can execute all Java bytecode instructions. JPF offers a highly configurable structure, and a variety of mechanisms to incorporate extensions. This allows us to integrate our extensions into JPF without tying them too closely to the original implementation. This is important to make our extensions more general. For example our string solver can be used as a stand-alone tool without involving JPF. Figure 2 shows JST's main components. JST's executor implements new instruction semantics, especially for those involving string operations. It uses our own hybrid solver for string-numeric solving.

JST also models many Java libraries to improve the symbolic execution performance. For example, we have created symbolic versions of all the container libraries such as *Set*, *Map*, *etc*. This allows us to model their behaviors in a way that reduces the branching during symbolic execution of these classes. These optimizations are crucial to reduce the number of useless paths.

In customizing the symbolic execution, JST defines different semantics for many bytecode instructions. The JPF framework allows for changing the semantics of a bytecode instruction by implementing the instruction factory as a configurable *abstract factory*. In this way one can choose which `Instruction` classes to use (*i.e.* these classes capture the bytecode instructions and define their execution semantics). JST overrides `Instruction` classes to introduce new or customized semantics for all Java bytecode instructions. Hence JST interprets Java programs in a different way. Though this mechanism is the same as Symbolic PathFinder, JST has different execution semantics for many of the bytecodes than Symbolic PathFinder to handle symbolic strings and their interaction with numbers. For example, when a "method invocation" instruction such as `invokestatic` or `invokevirtual` is executed, we skip its original Symbolic PathFinder processing, and activate the special handling of pre-selected methods for strings, containers, and so on. This approach also speeds up the symbolic execution process and leads to smaller search trees. We handle the methods of about 25 Java classes using this approach, including `String`, `BigDecimal`, `Integer`, `PrintStream`, *etc.*

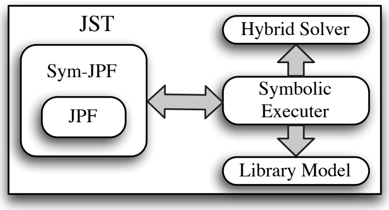


Fig. 2. Main components in JST.

JST is designed to be loosely coupled with the JPF core platform. Main components such as the solver and the library models can be used separately without involving JPF. The executor core is largely independent of JPF too. Hence it will be minimally affected when JPF changes its core. Our experience shows that such design separation is important to develop significant extensions over an evolving platform like JPF.

IV. The Solver

Path conditions usually contain both numeric (integer and real) constraints and string constraints. String constraints impose restriction on string variables, *e.g.* `s.startsWith("ab")` for symbolic string `s`. In our implementation, we separate these two kinds of constraints in a path condition. Numeric constraints are handled using existing solvers like Coral [12], Yices [13], and Choco[2]. We mainly use Yices for linear constraints, and Coral for simple non-linear constraints as we found these two solvers to be superior in performance to most of the other unrestricted use solvers. String constraints, on the other hand, are handled by our in-house solver developed over a Finite State Machine (FSM) package [14]. In addition, we reconcile the automaton-based string constraints and the numeric ones in the solver which is an unique challenge that all prior work [10] fails to address well.

A. Hybrid Solver with Iterations

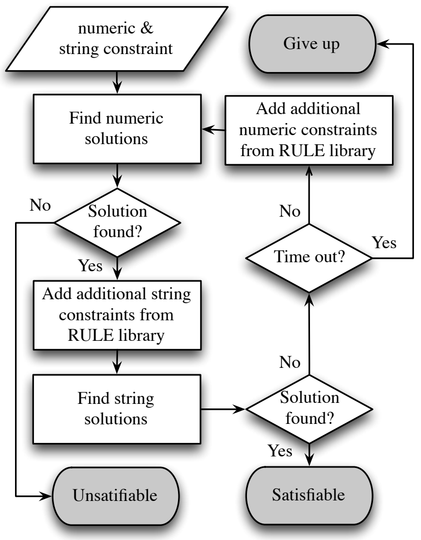

numeric & string constraint
Give up
Find numeric solutions
Add additional numeric constraints from RULE library
No
Solution found?
No
Yes
Time out?
Yes
Add additional string constraints from RULE library
No
Find string solutions
Solution found?
Yes
Unsatifiable
Satisfiable

Fig. 3. The general solving process in JST

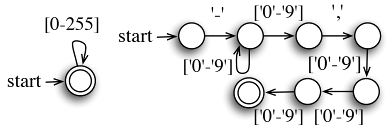
Solving hybrid constraints requires communication between the two domains. This often requires multiple iterations between the domains to reach an agreement on whether the combined constraints are sat (satisfiable) or unsat (unsatisfiable). Our solver in JST is no exception, with the main flow shown in Figure 3. At each iteration, the numeric solver first attempts to solve the numeric constraints. If unsat, then we can safely terminate the entire solver and report unsat. Otherwise, the results of the variables used by both the numeric and the string constraints are sent to the string solver. Other information may also be sent from the numeric solver to the string solver to make the iteration converge faster.

If the string solver can also find a solution, then the path condition is sat and the solver assigns concrete values to (a subset of) numeric and string variables. If no solution is found, a new iteration is performed after the string solver sends the "learned" constraints to the numeric one. This process continues until a solution is found or a pre-defined time or iteration limit is reached.

It should be noted that such an iteration method is not new by itself, SMT solvers [6] use iterations to combine multiple theories. In one sense, our hybrid solver is an SMT solver that combines a string theory and some numeric theories. The key, however, is *how* to perform efficient iterations in such a solver. In subsequent sections we describe a variety of techniques to optimize the iteration process.

B. String Constraint Solving

We use an FSM to represent a symbolic string variable such that all possible values of this variable constitutes the language accepted by this FSM. Our implementation is based on the FSM package `dk.brics.automaton` describe in [14]. Basically, an edge (transition) in an automaton is associated with a character range (interval). When an automaton is refined, we may change not only its nodes and edges but also the edge ranges. For example, the following shows the automata for the symbolic string `s` (left, accepting all values) and the regular expression "-\d+,\d{3}" (right) in the motivating example. For simplicity, we assume ASCII characters ranging from 0 to 255 while our implementation supports Unicode characters.



String operations are modeled by automaton operations. For example, the concatenation of two (symbolic) strings s_1 and s_2 is implemented by the appending s_2's automaton to s_1's, by adding ϵ transitions from all accepting states of s_1's automaton to all initial states of s_2's automaton. Many such operations are supported by the `dk.brics.automaton` package. However, we have to model many more string operations. We have also extended the package for solving rather than static analysis (for which the original package is designed [14]). While static analysis allows over-approximations, string solving must be accurate and report no false solutions. Hence we need to use extra techniques to ensure the preciseness.

Our system supports most of the string operations in the String class in the Java standard library. Some of these operations are applied by performing regular expression operations in the underlying package, such as union, intersection and complement (here we use *automaton intersection* and *language intersection* interchangeably). There are other operations, such as `trim`, `substring`, and `toUpperCase`, that require extra handling. For example, Figure 4 shows how we deal with `substring(2,4)` which returns a substring from begin index 2 to end index 4 (not inclusive). Roughly, we first advance 2 transitions from the start state, then mark the reached states as the new start state s_{new}, and then identify all states reachable from s_{new} in 2 transitions as new accepting states. Finally, we intersect

[2]http://www.emn.fr/z-info/choco-solver/

this automaton with the one accepting all words of length 2 to get the final automaton. There are other operations from the `String` class that need similar extensions, *e.g.* `substring`, `replace`, `replaceAll`, `replaceFirst`, `toUpperCase`, `toLowerCase`, and `split`.

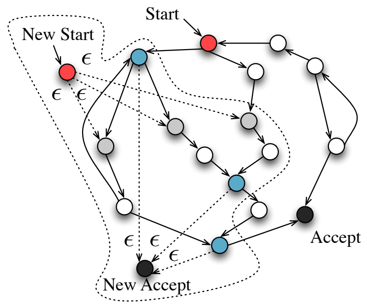
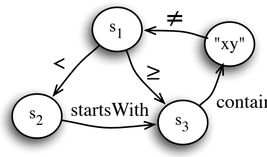


Fig. 4. Model operation substring(2, 4) using automaton.

String constraints depict the relation between strings (and numbers). The `dk.brics.automaton` package does not address string constraints directly. Hence we enhance it to refine string values according to given string constraints. This procedure includes (1) automaton refinement, (2) fix-point calculation, and (3) optimizations to speed-up the convergence. In the motivating example, for constraint `s.charAt(0)=='-'`, we intersect s with an automaton accepting any string starting from character `'-'`. Later on, since a portion of `s` is converted into an integer using `parseInt`, we intersect this portion with the automaton modeling all valid integers; we also need to refine `s` based on the updated portion, *i.e.* intersect `s`'s automaton with the one that accepts strings ending with this portion. Basically, if a constraint enforces a relation over strings s_1 and s_2, *e.g.* s_1`.beginsWith(`s_2`)`, then we refine (1) s_1 by enforcing that it starts with s_2, and (2) s_2 by enforcing that it is the beginning part of s_1. This process is repeated until no more refinement is possible and a fix-point (on the possible string values) is reached. We will skip the details here, but the basic algorithm is similar to abstract interpretation [15], *e.g.* for abstract domains such as integer intervals.

In addition to automaton refinement, we apply special handing to some cases where the pure automaton model is inadequate. Take constraint $s_1 < s_2 \wedge s_2 < s_3 \wedge s_3 < s_1$ for example. It is difficult to model $s_1 < s_2$ using automata, as the automaton capturing the $<$ relation may have a huge size. Furthermore, this constraint should be proven to be false immediately without involving any automaton computation. In our implementation, we introduce two extra models for such constraints. In the first one, each string variable is associated with a "number" representative, *e.g.* s_1's representative is integer i_{s_1}, such that this constraint can be falsified immediately by the numeric solver through checking $i_{s_1} < i_{s_2} \wedge i_{s_2} < i_{s_3} \wedge i_{s_3} < i_{s_1}$. This model allows us to prove unsat cases quickly.

In the second model, we maintain a *relational graph* with the string variables and their relations as the nodes and edges respectively. This graph is used to produce concrete values respecting the logical relations. For example, the following graph indicates that string s_1 is less than s_2, and s_2 starts with s_3, and so on. After all the node automata are refined, we get that $s_2 = s_3\alpha^*$ and $s_3 = \alpha^* xy \alpha^x$ where α denotes any character. Then we can search the graph in a post-topological order to find out a concrete assignment to all the string variables, *e.g.* s_3 = "xy", s_2 = "xyb", and s_1 = "xya". Since the string values have been refined before the search, finding concrete solutions is usually very fast.



C. Numeric and String Solving Interactions

String constraints and numeric constraints must agree on the same value of every shared symbolic variable. Roughly, we have the numeric solver $\mathbb{N}$ give some candidate values, and ask the string solver $\mathbb{S}$ whether these values violate the string constraints. If no violation is found, then these values are valid for both domains. Otherwise, $\mathbb{S}$ gives some feedback, like the conflicts it learns, to $\mathbb{N}$, and asks for other candidates.

Next, $\mathbb{N}$ adds the conflicts and the negation of the current assignment, and then searches for another valid assignment using some heuristics. Note that only the concrete values of shared variables will be passed from $\mathbb{N}$ to $\mathbb{S}$, and $\mathbb{N}$ can learn some string conflicts and passes them to $\mathbb{S}$ too. For example, numeric constraint `s.length() > 5` enforces that the corresponding automaton should be intersected with one accepting strings of length > 5.

Since the two domains interact with each other mainly through concrete values, it is important to avoid using fruitless values. During the iteration we exchange the constraints learned from each domain (called interactive constraints) so as to speed-up the convergence. For example, consider string constraint s_1`=`s_2`.trim()`. A numeric constraint `L(`s_1`)≤L(`s_2`)` is added into $\mathbb{N}$, where `L(`s_1`)` and `L(`s_2`)` are (symbolic) integer variables representing the lengths of s_1 and s_2. In our implementation, interactive constraints are modeled in a RULE library which we describe next.

Interactive Constraint Propagation. The RULE library includes commonly occurring patterns that we observed when applying JST to a wide range of Java applications. These patterns are particularly useful in the web and financial application domain, especially those from $\mathbb{S}$ to $\mathbb{N}$. Table I shows an excerpt of these patterns, The "a" rules have been used by other solvers [9], [16], [17], while the "b" rules are new in JST. One example is `(s.startswith('-')` $\wedge$ `n=Integer.valueOf(s))` where `s` is a string variable and `n` is an integer. This pattern leads to numeric constraint `n<0`. For another example, if the numeric solution found for `VOF(s)` is 5, then string constraint `s.equals("5")` is enforced.

TABLE I
SOME RULES IN THE RULE LIBRARY. HERE s_1, s_2, AND s_3 ARE STRING VARIABLES; i AND j ARE INTEGER VARIABLES; n IS A NUMERIC VARIABLE; c IS A POSITIVE INTEGER CONSTANT.

#	Observed Constraint	Derived Constraints
a.1	s_3=s_1.concat(s_2)	L(s_3)=L(s_1)+L(s_2)
a.2	s_2=s_1.trim()	L(s_2)<=L(s_1)
a.3	s_2=s_1.substring(i,j)	i≥0 ∧ j<L(s_2) ∧ i≤j ∧ L(s_2)= j - i
b.1	i=s.lastIndexOf(c)	s.substring(i,i+1).equals(c)
b.2	s.charAt(0)='−' ∧ n=VOF(s)	n<0
b.3	s.charAt(0)='+' ∧ n=VOF(s)	n>0
b.4	VOF(s)=n	s.equals("n")
b.5	VOF(s)<0	s.charAt(0)='−'
b.6	VOF(s)>c	L(s)≥log(c) ∧ s.charAt(0)!='−'
b.7	VOF(s)<c	L(s)≤log(c) ∧ s.charAt(0)!='−' ∨ s.charAt(0)='−'
b.8	L(s.trim())=c	L(s)≥c

For illustration, consider path 4 in the motivating example. We give below its path condition pc where some additional "𝕊 to ℕ" constraints (#1-#4) have been added into the numeric domain. To solve this pc, the numeric solver ℕ first obtains a valid assignment to numeric variables, *e.g.* L(s)=2, L(s1)=1, i=0, and x=100, then passes these values to the string solver 𝕊. Unfortunately 𝕊 cannot find a solution since constraint $x = \text{parseInt}(s1)$ is unsat. It can ask ℕ to perform new iterations until s1's length is at least 3. Or, the solver uses a rule to derive from "x=parseInt(s1) ∧ x≥100" an additional constraint L(s1)≥3. Moreover, the automata enforce that s's length is at least 2 more than that of s1. With these two new constraints the solver can quickly find a valid solution, *e.g.* s="-,100", s1="100", i=1, and x=100.

Numeric	String
$(1: \ L(s) > 0)$	$s.charAt[0] = \text{‘}-\text{’}$
$(2: \ i = -1 \ \vee \ 0 \leq i < L(s))$	$i = s.\text{lastIndexOf(‘,’)}$
$(3: \ i + 1 + L(s1) = L(s))$	$s1 = s.\text{substring}(i + 1)$
$(4: \ L(s1) > 0)$	$x = \text{parseInt}(s1)$
$i \neq -1 \ \wedge \ x \geq 100$	$\neg(s.\text{matches}(-\backslash d+, \backslash d\{3\}))$

When 𝕊 cannot find a valid solution, additional constraints are passed to ℕ and then a new iteration starts by throwing away previous candidate values. For example, suppose that numeric shared variables a, b, and c with the solutions 5, -7, 0 are found unsat in 𝕊, then we will need to add the numeric constraint $\neg(a = 5 \wedge b = -7 \wedge c = 0)$ or $a \neq 5 \ \vee \ b \neq -7 \vee c \neq 0$.

Clearly this may lead to an exponential number of case splittings. To mitigate the blow-up, we apply an optimization through utilizing a feature provided by the Yices solver: Yices returns satisfiable values closest to zero for numeric variables. Hence we can safely try three narrower cases: $a > 5$; $b < -7$; and $c \neq 0$, which searches the same space but can converge faster. We also consider another case, $(a > 5 \wedge b < -7 \wedge c \neq 0)$, in hope of finding a valid solution rapidly. Our experience demonstrates that such simple improvements and heuristics can have considerable effects in practice.

D. Relaxed Solving and Execution

One of the main bottlenecks of our hybrid solver is that it may need many iterations to find a solution or derive unsat. The situation becomes worse when the solver is invoked multiple times. Suppose a program contains n satisfiable branches, then $O(2^n)$ paths can be generated and $O(2^n)$ queries are invoked on the solver. If m iterations are needed for each query, then $O(2^n m)$ iterations happen in total. Fortunately, we have observed in real applications that many path constraints will turn into unsat in subsequent computations quickly leading to false paths. We also know that only the solved values at the end of a symbolic path will contribute to valid test cases.

Based on these observations, we apply a two-phase solving technique during symbolic execution. For the intermediate branching nodes, we use *relaxed solving* which checks only whether the string and the numeric constraints are sat without multiple iterations. At the end of each path, we use the usual regular solving with full iterations and constraint propagation. This method dramatically improves the performance since we not only avoid expensive solving for intermediate nodes but are able to quickly cut out many false paths using the over approximation techniques of *relaxed solving*.

The relaxed solving process is similar to the regular one but without multiple iterations. It starts with solving the numeric constraints. If a solution is found, it passes additional constraints (*e.g.* those in Table I) but not the values of shared variables to the string solver. Note that these additional constraints represent a strict over-approximation of the set of solutions possible in the numeric domain. If the string solver cannot find a solution, the path condition pc is unsatisfiable because no matter what other solution is passed from the numeric domain it will satisfy the over-approximation constraint and hence be unsat in the string domain. Otherwise, pc is regarded to be sat. Thus in the relaxed mode the two domains do not communicate through concrete values.

The main disadvantage of relaxed solving is that it may explore infeasible paths whose path conditions are sat in the relaxed mode while unsat in the regular mode. This seems to increase the number of intermediate paths. However, this happens rarely in practice since (1) we still use the string solver 𝕊 and the numeric solver ℕ to rule out most infeasible paths, and (2) an infeasible path is often falsified by subsequent relaxed solving at intermediate nodes. Thus relaxed solving can be very effective in pruning infeasible paths early.

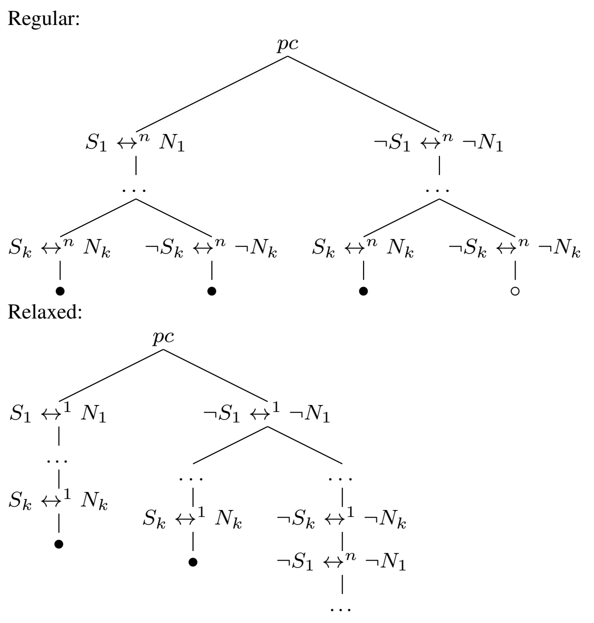

Regular:
pc
$S_1 \leftrightarrow^n N_1$
$\neg S_1 \leftrightarrow^n \neg N_1$
$S_k \leftrightarrow^n N_k$
$\neg S_k \leftrightarrow^n \neg N_k$
$S_k \leftrightarrow^n N_k$
$\neg S_k \leftrightarrow^n \neg N_k$
Relaxed:
pc
$S_1 \leftrightarrow^1 N_1$
$\neg S_1 \leftrightarrow^1 \neg N_1$
$S_k \leftrightarrow^1 N_k$
$S_k \leftrightarrow^1 N_k$
$\neg S_k \leftrightarrow^1 \neg N_k$
$\neg S_1 \leftrightarrow^n \neg N_1$

Fig. 5. Comparing regular solving and relaxed solving.

Consider Figure 5 which contains two branching trees starting from a node with path condition pc. Each tree branches over a sequence of conditions $S_1 \leftrightarrow^m N_1, \ldots, S_k \leftrightarrow^m N_k$, where $S_i \leftrightarrow^m N_i$ denotes the i^{th} condition such that its numeric constraint N_i and string constraint S_i iterate m times. When $m = 1$ the relaxed mode is used. The regular mode assumes $m = n$ where n is the average iteration number. The "regular" tree is of height k, incurring $(2^{k+1}-2)n$ iterations in total. Suppose all the leaf nodes except the rightmost one are invalid (marked by $\bullet$) and the rightmost path is valid (marked by $\circ$). In this case exploring the others is fruitless. This can be avoided through the relaxed mode. Assume that in all the paths except for the rightmost one, S_i and N_i may conflict with S_j and N_j for $i \neq j$, *i.e.* the paths become unsat in the relaxed solving phase. Hence there is no need to perform expensive iterative solving on those paths.

The number of iterations now is reduced to $2^{k+1} - 2 + kn$, an improvement of $\frac{(2^{k+1}-2)n}{2^{k+1}-2+kn} \approx n$ times (for large k) over the regular mode. That is, if the average iteration number is 1000, then relaxed solving can produce a speed-up of 1000x. This is validated in the experimental results section.

E. Handling Some Nonlinear Operations

Some applications that we tested generate path conditions involving nonlinear numeric constraints. Solving them is undecideable in general and beyond the capabalities of SMT solvers. For these constraints, we tried the Coral solver [12] which is a randomized solver that uses machine-learning algorithms to search solutions for complex non-linear constraints.

Unfortunately, using such a random solver considerably slows down the solving process (it has been reported that Coral can be $>$ 100x slower than SMT solvers for linear constraints [12]). To mitigate this problem, we transform some nonlinear constraints to equivalent linear ones, which can then be solved using faster SMT solvers. One example is the `round` method in the Java class `java.lang.Math` which returns the closest integer to the given floating point number. If the nearest integers to the given number are equidistant, this operation returns the greater integer (*e.g.* `round(2.5)` becomes 3). Specifically, for the operation `round(e)`, we add the following linear constraint to the numeric set, where e is a real variable, and $x1$, $x2$, and $result$ are introduced integer variables. $result$ represents the value of `round(e)`, and it replaces all its occurrences in numeric constraints.

When e is positive, the constraint enforces $e - 0.5 < x1 \leq e + 0.5$, *e.g.* when $e = 1.6$, constraint $1.1 < x1 \leq 2.1$ implies that $result = 2$. Variable $x2$ is for the case when e is negative.

$$\begin{array}{l} ((e - 0.5 < x1) \ \wedge \ (e + 0.5 \geq x1)) \ \wedge \\ ((e - 0.5 \leq x2) \wedge (e + 0.5 > x2)) \ \wedge \\ (result = (if \ (e > 0) \ x1 \ x2)) \end{array}$$

Similarly, we replace other operations in the `Math` class, such as `ceil` and `floor`, with their equivalent linear constraints. Many rounding methods in the `BigDecimal` class are also handled in this manner. While most conversions are not complicated, this represents one important enhancement we use to scale JST to handle industrial applications. It is obvious that this transformation can only work for non-linear functions that are piecewise linear. Otherwise we still need to use Coral and abort in case of a time-out.

V. Evaluation

We evaluate JST on three string-intensive benchmark examples whose characteristics are described in Table II. They represent many other similar applications we have tested. For each benchmark we create a driver and and some stubs to produce a closed system on which JST can be run. Since the first two benchmarks are very large and relied on multiple external libraries, packages and jars, and a lot of source code is missing, we focus on the parts where the core logic are implemented. Despite their small sizes (1-4k lines), they represent the most complicated string and numeric computations in the applications. Note that, in order to to test the core logic, we have to symbolically execute the whole application leading to huge symbolic paths through the complete application. We only measure the coverage of the core logic although many other parts of the applications are also covered (many of which are not sensitive to the symbolic inputs). We run the experiments on a Ubuntu Linux Machine with quad core 3.4Ghz Intel core i7 processor and 8GB of RAM.

We could not compare our results with any other freely available tool because we found all of them to be completely inadequate in handing all the String operations that existed in our examples and the specific interactions between the numeric and string domain. There is also currently no standard format for expressing String constraints. Thus it is not possible to translate all the complex constraints to operations that other String-Numeric constraint solving tools can understand.

Table III shows the result of running JST on Example A while gradually increasing the number of symbolic inputs to 4. We also present some results for different combinations of

TABLE II
BENCHMARK STATISTICS

Application	Description	#classes	#bytecode	#lines of core logic	# Inputs
A	Business to Business Ordering	2,801	1,226K	3,379	4 (strings)
B	Financial Application 1	5,232	2,704K	2,613	5 (strings)
C	Financial Application 2 Snippet	3	1,672	1,157	3 (strings)

symbolic variables. All these inputs are string variables though some are converted to integers and double values within the application. The number of true paths in the table corresponds to the actual test cases generated at valid end states in the program. However, there are also some unhandled exception paths that are encountered during the symbolic execution like `NumberFormatException`, `ArithmeticExecption`, *etc.* which are shown in the Column 3. These exception paths usually indicate that the application is missing the error handling code. Manual investigation indicates that a small portion of them are "real" issues (wrong assumptions or bugs).

In Column 4 the maximum number of iterations required between the numeric and string domains to arrive at a valid solution is shown. As expected this number does increase as we have more symbolic inputs interacting with each other but due to the optimizations mentioned in subsection IV-B, JST is able to keep the number of iterations to a reasonable limit. There are no unsolved constraints in this case and with 4 symbolic inputs we can achieve a greater than 80% code coverage on the core logic part of the application. We will discuss in Section VII the reasons for the coverage not attaining 100%.

TABLE III
EXPERIMENTAL RESULTS FOR EXAMPLE A

#Sym.Vars	#Paths		Time	#Iterations
	True	Exception		
1	6	1	8s	1
2 (subset 1)	27	5	16s	1
2 (subset 2)	35	9	55s	2
3 (subset 1)	595	16	3:24m	5
3 (subset 2)	493	25	4:01m	5
4	1,971	95	11:08m	24

Table IV shows the effectiveness of the relaxed solving described in subsection IV-D. In this experiment we run multiple experiments on Example B first by using the relaxed mode (Columns 2-4) and subsequently turning this feature off (Columns 5-7). The hybrid solver is allowed 2,000 iterations before giving up. The complete time out of a run is set to 48 hours. Again we gradually increase the number of symbolic variables from 1 to 5 and again we report results for multiple combinations of same number of variables. In the relaxed mode run for the 5 variable case, JST finishes after 1 day with a total of about 28,000 test cases. Again, we can achieve about 80% code coverage on the core logic code. All valid path conditions can be solved or proved unsat and the maximum number of iterations needed among the solvers is 268. However, when we turn off that feature the effect is a dramatic reduction in effectiveness of JST. JST times out with even as little as 3 symbolic variables and even for the cases it finishes, it is unable to find many solutions within the stipulated 2,000 iterations. The reasons for this are twofold (see Section IV-D too). First, the relaxed mode is able to prove some intermediate path conditions unsat in the middle of the symbolic execution tree thus pruning off large portions of the symbolic search tree. This speeds up the exploration of the complete symbolic execution tree. Second, even at a true leaf node, the rule based constraint exchange between the numeric and string domains is able to quickly converge to a solution whereas without that feature the hybrid solver often hits the 2,000 iteration limit without finding a solution. Thus a lot of time is wasted without producing any new results. Note that the relaxed mode does not lead to over-approximations since full solving is applied on the leaf nodes to eliminate false "tentative" paths.

TABLE IV
EXPERIMENTAL RESULTS FOR EXAMPLE B. #T.P AND #E.P. DENOTE (#TRUE PATHS) AND (#EXCEPTION PATHS) RESPECTIVELY. IN RELAXED MODE, #T.P. CONTAINS "#FINAL_PATHS/#TENTATIVE_PATHS"

# Sym.	With Relaxed Mode			Without Relaxed Mode		
Vars	#T.P.	#E.P.	Time	#T.P.	#E.P.	Time
1	3/5	9	7s	3	9	35s
2 (subset 1)	15/26	15	20s	15	15	2:37h
2 (subset 2)	20/36	18	22s	20	18	3:05h
3 (subset 1)	178	183	7:47m	-	-	TO
3 (subset 2)	253	301	9:42m	-	-	TO
4 (subset 1)	890	2,535	4:31h	-	-	TO
4 (subset 2)	1,430	3,242	5:54h	-	-	TO
5	7,156	20,530	28:50h	-	-	TO

Finally, we present results to demonstrate the effectiveness of non-linear constraint modeling as described in subsection IV-E. In this case we run experiments of example C which is carved out of a second financial application for unit testing. These classes consist of a lot of different types of rounding operations on the Java `BigDecimal` data type. We make multiple runs with the non-linear modeling turned on (Table V, Columns 2-4) and then off (Table V, Columns 5-7). When the non-linear modeling is turned on all the resulting path conditions can be solved by the Yices [13] solver but when it is turned off we have to invoke the non-linear solver Coral [12]. Again it is evident from the table that we can get orders of magnitude improvement in runtimes and dramatic improvements in the quality of results. Since this is unit testing, with all input variables symbolic, we can obtain 100% code coverage in the whole example. However, when we turn off this feature we run into frequent time-outs inside Coral. This is to be expected as the Coral algorithm is by definition incomplete and weaker than SMT solving which relies on efficient algorithms for solving Boolean and numeric constraints. Of course there will be cases when it is impossible to model non-linear constraints with piecewise linear operations. In such cases we do invoke Coral as a last resort and give up in case Coral times out without finding a solution.

TABLE V
EXPERIMENTAL RESULTS FOR EXAMPLE C. #T.P AND #E.P. DENOTE THE NUMBERS OF TRUE PATHS AND EXCEPTION PATHS RESPECTIVELY.

# Sym. Vars	With Nonlinear Models			Without Nonlinear Models		
	#T.P.	#E.P.	Time	#T.P.	#E.P.	Time
1	5	0	1s	3	0	9s
2 (subset 1)	35	10	19s	28	8	57:1m
2 (subset 2)	29	22	23s	24	7	45:1m
3	175	50	1:39m	31	10	6:37h

Overall, our experience indicates that "automaton+iteration" is a promising way to handle hybrid constraints. In addition, other key features that make the tool practical include: (1) use of relaxed solving to prune paths and speed-up the solving, (2) convert non-linear expressions to linear ones whenever possible, (3) optimize Java libraries to cater for the need of symbolic execution, and (4) applying divide-and-conquer techniques to divide the state space.

VI. RELATED WORK

Table VI gives a brief comparison on string solvers in terms of string model, support for basic string operations, string relations and regular expressions, support for string and numeric interactions, using rules during the iterations, and the targeted applications. For instance, "Reln" denotes relational constraints such as $s_1 > s_2$. Roughly, the solvers can be divided into two kinds: bitvector based and automaton based.

Microsoft's solver [9] encodes string operations with bitvectors. Constraints over string lengths are derived from string constraints before the bitvectors are instantiated and solved. It does not support string relations and regular sets, and does not apply constraint propagation from the numeric domain to the string domain. Hampi [16] handles fixed-size strings but supports constraints checking membership in regular languages and fixed-size context-free languages. It converts string constraints into fixed-size bitvectors. Numeric constraints are not supported (except for the simple ones over the lengths). Kudzu [17] uses bitvector encoding and derived length constraints. It uses the Hampi solver, and translates constraints into the input of Hampi. Kudzu provides support for many basic string operations such as concatenation, plus limited support for symbolic numbers and their interactions with strings.

A recent evaluation of automaton-based string solvers is given in [10]. Among these tools, the Rex tool [18] uses automata and an SMT solver, and represents automaton transitions using logical predicates. This allows it to handle numbers using the SMT solver. Its encoding is shown be too inefficient, and slower than Hampi by several orders of magnitude [10]. Stranger [20] use an automaton-based method to model string constraints and length bounds for abstract interpretation. It models many string operations including replacement but no interactions with numbers.

TABLE VII
DEVELOPMENT SIZE OF JST. WE COUNT ONLY OUR EXTENSIONS.

Main Component	Extend Over	l.o.c
Execution Engine	Sym. JPF [2]	23,418
Automaton Package	JSA [14]	4,546
Hybrid Solver	–	8,784
Nonlinear Solver	Coral [12]	1,180
Others	–	1,500
Total		39,428

A lazy solving technique is proposed in [23]. It uses a graph to represent constraints over string variables, and searches the graph with back-tracking to guess solutions. It can find a solution (if it exists) fast but the unsat case is much more difficult. However, it handles only a very limited set of string and regular operations, and allows no symbolic numbers.

In [21], [22] a string solver similar to ours is implemented for JPF. This solver uses range automata and bitvectors to model a subset of string operations and regular languages. It supports fewer string operations, has a limited set of iteration rules, and maintains weak connection between the automaton domain and the numeric domain. It does not support many of the features described here such as relaxed solving.

As for symbolic executors, some widely used symbolic execution engines [1], [2], [3], [4] handle high level languages such as C and Java. Recently we have extended KLEE [1] to handle C++ programs [8] and GPU programs [24]. All the tools have no built-in string solvers. Hence they rely on specific extensions such as Rex for Pex [4], and Hampi for KLEE. Note that, only JST uses relaxed solving and bi-directional flow of derived constraints.

VII. DISCUSSIONS AND CONCLUSION

This paper describes the JST tool which is used to uncover bugs and generate test vectors for industrial strength Java applications. Though the tool is based on the well known Symbolic Java PathFinder platform, the extensions needed to make the tool applicable to industrial examples are non-trivial. This is evident from Table VII which enumerates the size of the different blocks that we have implemented in order to achieve our goals.

The primary issues that we have tackled are extensive support for different Java primitives and libraries that are frequently encountered in such examples, support for symbolic strings, and multiple enhancements in the hybrid string-numeric solver to make the approach scalable. While the original platform had support for some numeric and Boolean primitive types, we have implemented support of all symbolic primitive types in Java as well as symbolic strings. While implementing symbolic strings we have had to add support for most of the string manipulation methods in the Java `String`, `StringBuilder` and `StringBuffer` libraries. Even some conversions between String and `CharSequence`, and String and Character arrays, are tackled along with methods

TABLE VI
COMPARISON OF STRING SOLVERS. NOTATIONS "$\oplus$", "$\ominus$" AND "–" INDICATE "STRONG", "WEAK" AND "NONE"RESPECTIVELY.

Solver	String Model	Support			w. Numeral	+ Rules		App.
		Basic	Reln	R.E.		$\mathbb{S} \to \mathbb{N}$	$\mathbb{N} \to \mathbb{S}$	
Microsoft [9]	bitvector	$\oplus$	–	–	$\oplus$	$\ominus$	–	.NET
Hampi [16]	bitvector	$\ominus$	–	$\oplus$	–	–	–	Web,C
Kudzu [17]	bitvector	$\oplus$	–	$\oplus$	$\ominus$	$\ominus$	–	JavaScript
Rex [18]	automaton+SMT	$\oplus$	–	$\ominus$	$\oplus$	–	–	General
Stranger [19], [20]	bit automaton	$\oplus$	–	$\ominus$	$\ominus$	–	–	PHP
Lazy [10]	range automaton	$\ominus$	–	$\oplus$	–	$\ominus$	–	N/A
[21], [22]	range automaton	$\ominus$	$\ominus$	$\oplus$	$\oplus$	$\ominus$	–	Java
JST	range automaton	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	Java

in those classes. Java regular expressions are also supported. In addition we have created symbolic models for different Java container classes like `Vector`, `ArrayList` *etc*. Some Arithmetic classes like `BigDecimal` and `BigInteger` are also modeled as they are used frequently in financial applications. After enhancing the symbolic support to the various data types and libraries that we have encountered, an efficient hybrid numeric-string solver was needed to solve the symbolic constraints arising out of the symbolic execution. Though a lot of work exists in the literature to do this, we have found them inadequate to tackle unique challenges arising out of type conversions using functions like `.valueOf()`, `.parseInt()`, *etc.*. In such cases we have devised a rule based constraint propagation technique between the numeric and string domains so that the solver can quickly decide unsat on a hybrid formula or use fewer iterations to arrive at a solution. We have also modeled certain non-linear constraints in a piecewise linear fashion so that they can be solved quickly by a standard SMT solver.

The effectiveness of JST incorporating the above set of enhancements is shown on two large industrial strength examples. In the results we are not able to achieve 100% code coverage for the sections under test due to the deficiencies in the driver which are manually generated. Note that the complete coverage of the code under test not only depends on the various data values in the inputs that exercises different branches, but also on the different event scenarios that the driver takes the code through. While symbolic execution is very effective in finding data values that cover all branches, it is unable to unearth possible scenarios that the driver has completely missed. This is one of the deficiencies of JST which we are looking into next. The tool is currently being used internally by Java developers for Unit and Component level testing in the order of 2K-20K source lines. System level testing is not advised with JST as the symbolic execution becomes unmanageable even with all the above enhancements. Even though we have described a tool applicable for the Java language, it is possible to use similar techniques to handle C/C++, JavaScript or other popular programming languages.

ACKNOWLEDGMENT

The authors would like to thank Tadahiro Uehara and Shoichiro Fujiwara from Fujitsu Labs Japan for extensive testing of JST and providing examples to make the tool better.

REFERENCES

[1] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008.

[2] C. S. Păsăreanu and N. Rungta, "Symbolic PathFinder: symbolic execution of Java bytecode," in *ASE*, 2010.

[3] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ESEC/FSE*, 2005.

[4] N. Tillmann and J. De Halleux, "Pex: white box test generation for .net," in *International conference on Tests and proofs (TAP)*, 2008.

[5] J. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[6] D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 2008.

[7] C. S. Păsăreanu, P. C. Mehlitz, D. H. 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 *ISSTA*, 2008.

[8] G. Li, I. Ghosh, and S. P. Rajan, "KLOVER : A symbolic execution and automatic test generation tool for C++ programs," in *CAV*, 2011.

[9] N. Bjørner, N. Tillmann, and A. Voronkov, "Path feasibility analysis for string-manipulating programs," in *TACAS*, 2009.

[10] P. Hooimeijer and M. Veanes, "An evaluation of automata algorithms for string analysis," in *VMCAI*, 2011.

[11] D. Shannon, I. Ghosh, S. Rajan, and S. Khurshid, "Efficient symbolic execution of strings for validating web applications," in *2nd Workshop on Defects in Large Software Systems*, 2009.

[12] M. Souza, M. Borges, M. d'Amorim, and C. S. Psreanu, "CORAL: solving complex constraints for symbolic pathfinder," in *NFM*, 2011.

[13] B. Dutertre and L. D. Moura, "The Yices SMT Solver," Computer Science Laboratory, SRI International, Tech. Rep., 2006.

[14] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *SAS*, 2003.

[15] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, 1977.

[16] V. Ganesh, A. Kiezun, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "HAMPI: A string solver for testing, analysis and vulnerability detection," in *CAV*, 2011.

[17] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A Symbolic Execution Framework for JavaScript," in *IEEE Symposium on Security and Privacy*, 2010.

[18] M. Veanes, P. de Halleux, and N. Tillmann, "Rex: Symbolic regular expression explorer," in *ICST*, 2010.

[19] F. Yu, T. Bultan, and O. H. Ibarra, "Symbolic string verification: Combining string analysis and size analysis," in *TACAS*, 2009.

[20] F. Yu, M. Alkhalaf, and T. Bultan, "Stranger: An automata-based string analysis tool for PHP," in *TACAS*, 2010.

[21] G. Redelinghuys, "Symbolic string execution," Master's thesis, University of Stellenbosch, 2012.

[22] G. Redelinghuys, W. Visser, and J. Geldenhuys, "Symbolic execution of programs with strings," in *SAICSIT Conf.*, 2012.

[23] P. Hooimeijer and W. Weimer, "Solving string constraints lazily," in *ASE*, 2010.

[24] G. Li, P. Li, G. Sawaga, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, "GKLEE: Concolic verification and test generation for GPUs," in *PPoPP*, 2012.