Generalizing Symbolic Execution to Library Classes

Sarfraz Khurshid and Yuk Lai Suen
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
1 University Station C5000
Austin, TX 78712
{khurshid, suen}@ece.utexas.edu

ABSTRACT

Forward symbolic execution is a program analysis technique that allows using symbolic inputs to explore program executions. The traditional applications of this technique have focused on programs that manipulate primitive data types, such as integer or boolean. Recent extensions have shown how to handle reference types at their representation level. The extensions have favorably been backed by advances in constraint solving technology, and together they have made symbolic execution applicable, at least in theory, to a large class of programs. In practice, however, the increased potential for applications has created significant issues with scalability of symbolic execution to programs of non-trivial size—the ensuing path conditions rapidly become unfeasibly complex.

We present Dianju, a new technique that aims to address the scalability of symbolic execution. The fundamental idea in Dianju is to perform symbolic execution of commonly used library classes (such as strings, sets and maps) at the abstract level rather than the representation level. Dianju defines semantics of operations on symbolic objects of these classes, which allows Dianju to abstract away from the complexity that is normally inherent in library implementations, thus promising scalable analyses based on symbolic execution.

Categories and Subject Descriptors

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

Keywords

Symbolic Execution, Korat, Test Input Generation, Model Checking

1. INTRODUCTION

Forward symbolic execution is a program analysis technique that allows using symbolic inputs to explore program executions [7, 21]. The traditional applications of this technique have focused on programs that manipulate primitive data types, such as integer or boolean. Recent frameworks have generalized symbolic execution to handle reference (and array) types [2, 3, 8, 11, 13, 16, 20, 22, 27, 33–36, 38].

Some of these frameworks simply combine symbolic execution of primitives together with concrete manipulations of heap, while others show how to treat both primitives and references symbolically. A common aspect of these generalizations, however, is that they perform at the representation level (i.e., the level of class declarations in the source code) of the data they manipulate. In particular, symbolic manipulations of references are defined at the level of equality comparisons among references or destructive updates of relevant fields. For example, manipulations on an object are defined in terms of updates to its fields.

The generalizations have favorably been backed by advances in constraint solving technology [4, 29], and together they have made symbolic execution applicable, at least in theory, to a large class of programs, particularly those written in modern programming languages, such as Java. In practice, however, the increased potential for applications has created significant issues with scalability of symbolic execution to programs of non-trivial size—the ensuing path conditions rapidly become unfeasibly complex.

We believe a key obstacle to the scalability of symbolic execution is performing it at the representation level. Indeed if the *entire* implementation needs to be checked, such execution is necessary. However, it is redundant if the aim is to only check the client code. Consider as a simple illustration, a program that uses as a client a library implementation of the abstract data type set. Let's say the method adds an element, say `e`, to a set, say `S`, and then checks whether `e` belongs to `S`. Executing this program using representation level manipulations involves executing methods that implement the add and membership check operations. For performance reasons, sets are typically implemented using complex data structures, such as balanced binary search trees, which involve complex (balancing) operations. Symbolically executing these operations (even to small bounded iteration depths) rapidly generates large path conditions which quickly lead to infeasible analyses.

We present Dianju, a new technique that aims to address the scalability of symbolic execution. The fundamental idea in Dianju is to perform symbolic execution of commonly used library classes (such as strings, sets and maps [31]) at the *abstract* level rather than the representation level. Dianju defines semantics of operations on symbolic objects of these classes, which allows Dianju to abstract away from the complexity that is normally inherent in implementations of library code, thus promising scalable analyses based on symbolic execution. To solve path conditions that repre-

sent constraints on symbolic objects, Dianju provides specialized constraint solvers for a variety of constraints on strings, sets and maps (in addition to simple enumerators for primitives).

Dianju can also be used for efficient test input generation. In addition to benefits that existing testing frameworks can reap from generalizing symbolic execution to library classes, Dianju provides two novel techniques for generating inputs. One, Dianju provides specialized generators for objects of library classes, including strings, sets, and maps. The set generator, for example, can enumerate sets (as defined by abstract mathematical objects). Given a bounded domain of possible elements of the set, Dianju set generator directly enumerates all sets on those elements in a highly optimized fashion. These abstract mathematical objects are then translated into concrete Java objects (such as those of the class `java.util.HashSet` using a sequence of method invocations of `add` starting from an empty-set. Thus objects of a class that declares a field of a supported library type can be enumerated in an asymptotically optimal fashion.

Two, Dianju exploits relationships among different fields of an object. Usually classes that declare several fields require all their objects to satisfy certain invariants, known as class invariants [23]. Dianju symbolically executes the class invariant (if provided by the user) and extracts constraints on the field values of the supported library types. Dianju feeds these constraints to its generators, which enables the generators to focus their enumeration on objects that satisfy the class invariants.

Dianju builds on ideas we have developed in our previous work on Korat [6] and generalized symbolic execution [20]. A key distinction of Dianju from that work is that Dianju does not require detailed class invariants. For example, in Korat, if a field is declared to be of type `java.util.Set`, the user must give class invariants for the actual class that implements the desired sets. Writing invariants for classes such as `java.util.TreeSet` or `java.util.HashSet` correctly can be rather challenging. Dianju's direct support of library classes not only eliminates the need for detailed invariants, but it also reduces the number of inputs generated.

As a comparison with Korat, consider generating `HashSet` objects with 9 elements. Korat evaluates over 3 million candidate structures (of which it generates 26,687 as valid structures) whereas Dianju evaluates exactly the $2^9 = 512$ sets that can be (mathematically) constructed from 9 elements. Korat's generation of `HashSet` objects respects non-isomorphism with respect to the (buckets of linked lists) implementation. However, Dianju's generation respects non-isomorphism with respect to the mathematical notion of a set. While the former notion is necessary to systematically check library implementations, it is the latter notion that we really need in order to check client code.

Even though Dianju's test generation is exhaustive, it employs some simple heuristics, which allow it further enhance its efficiency. For example, the Dianju constraint solver makes use of the literals in the constraints that symbolic execution builds to generate candidate solutions: if a string literal is used in a comparison constraint, that literal is added to the set of candidate values for symbolic strings.

The observation that the efficiency of program analyses depends on the implementation choices that programmers make is not new, e.g., model checking literature advocates the use of types that support efficient checking [17]. Dianju, however, does not propose to discourage the use of types that optimize implementations, since programmers must be allowed to use efficient library implementations. Instead, Dianju identifies and supports their use through providing analyses targeted on commonly used libraries.

This paper makes the following contributions:

- It introduces the idea of generalizing forward symbolic execution of Java programs to directly support commonly used library classes.
- It introduces the idea of specialized generators for objects of library classes in conjunction with the use of (partially specified) class invariants.
- It introduces the idea of using symbolic execution to extract constraints from class invariants of complex structures to tune their generation.
- It presents a series of small illustrative examples that display the optimization potential of Dianju.

2. BACKGROUND: SYMBOLIC EXECUTION

Forward symbolic execution is a technique for executing a program on symbolic values [21]. There are two fundamental aspects of symbolic execution: (1) defining semantics to operations that are originally defined for concrete values and (2) maintaining a *path condition* for the current program path being executed—a path condition specifies necessary constraints on input variables that must be satisfied to execute the corresponding path.

As an example, consider the following program that returns the absolute value of its input:

```
       int abs(int i) {
L1.        int result;
L2.        if (i < 0)
L3.            result = -1 * i;
L4.        else result = i;
L5.        return result;
       }
```

To symbolically execute this program we consider its behavior on a primitive integer input, say `I`. We make no assumptions about the value of `I` (except what can be deduced from the type declaration). So, when we encounter a conditional statement, we consider both possible outcomes of the condition. To perform operations on symbols, we treat them simply as variables, e.g., the statement on line 3 updates the value of `result` to be `-1 * I`. Of course, a tool for symbolic execution needs to modify the type of `result` to note updates involving symbols and to provide support for manipulating expressions, such as `-1 * I`.

Symbolic execution of the above program explores the following two paths:

```
path 1:
    [I < 0] L1 -> L2 -> L3 -> L5
path 2:
    [I >= 0] L1 -> L2 -> L4 -> L5
```

Note that for each path that is explored, there is a corresponding path condition (shown in square brackets). While execution on a concrete input would have followed exactly one of these two paths, symbolic execution explores both. In general, symbolic execution can systematically explore all program paths up to a bounded path length. Tools that implement such exhaustive exploration often use iterated depth first search [15], where the search depth is iteratively increased, until a sufficiency criterion for exploration has been met.

3. EXAMPLES

We present two small examples that illustrate the potential optimizations that Dianju can provide. The first example illustrates how Dianju generalizes symbolic execution over library classes. The second example illustrates how Dianju's specialized generators and constraint solving ability can even be used to check formulas that express equivalences on sets.

3.1 CoverMe

As an illustration of Dianju's symbolic execution, consider the class `CoverMe` that is inspired by the implementation of another of our software testing frameworks that we internally call CoverMe. As its name suggests this framework aims to automatically achieve a desired level of code coverage for a given program. Consider the following class declaration:

```
class CoverMe {
    String jar;
    Set cover;
    Map b2s;

    // class invariant
    boolean repOk() {
 1:     if (jar == null || cover == null || b2s == null)
 2:         return false;
 3:     if (cover.contains(null))
 4:         return false;
 5:     if (b2s.keySet().contains(null))
 6:         return false;
 7:     if (b2s.values().contains(null))
 8:         return false;
 9:     if (!b2s.values().containsAll(cover))
10:         return false;
11:     return true;
    }
}
```

Objects of class `CoverMe` represent a code coverage criteria. The field `jar` names the relevant jar file that contains the code. The field `cover` represents a set of sourcecode line numbers to cover. The field `b2s` is a mapping between line numbers in the bytecode and the sourcecode.

The method `repOk` represents the class invariant of `CoverMe`. For a set, the method `contains` checks if the set contains the given element; the method `containsAll` checks if the set contains all of the elements in the given collection. For a map, the method `keySet` returns a set view of the keys contained in the map; the method `values` returns a collection view of the values contained in the map. `CoverMe.repOk` uses these methods that are defined in the Java Collection Framework to require that all fields are non-null, and `null` is not a value in the set `cover`, or the set of keys or the set of values in the map `b2s`. The method returns false, if any of these constraints is violated and returns true otherwise.

The implementation of `repOk` is deceptively simple. Notice particularly the use of (`keySet` and `values`) views that are backed by the underlying map. Constraints in path conditions that symbolic execution builds involve these views and therefore have a uniquely complex nature.

Consider symbolically executing `repOk`, which has one input parameter, i.e., the implicit parameter `this`. Notice that all fields of `CoverMe` have library types that Dianju supports. Dianju initializes each field to a new symbolic value of the appropriate type. Consider the case when symbolic execution explores the path that reaches statement 4 (i.e., the path `1 -> 3 -> 4`), where `repOk` returns false. For this path, symbolic execution builds the following path condition:

```
[ SymbolicString_0 != null &&
  SymbolicSet_1 != null &&
  SymbolicMap_2 != null &&
  SymbolicSet_1.contains(null) ]
```

where `SymbolicString_0` represents the value of `jar`, `SymbolicSet_1` represents the value of `cover`, and `SymbolicMap_2` represents the value of `b2s`. For these constraints, Dianju's constraint solver generates the following satisfying assignment:

```
[ SymbolicString_0 = ''jar'',
  SymbolicSet_1 = { null },
  SymbolicMap_2 = {} ]
```

If we execute `repOk` on the object corresponding to this assignment, `repOk` will follow the path `1 -> 3 -> 4` and return false.

Given the `repOk`, test generation aims at enumerating inputs for which `repOk` returns true. In the `repOk` for `CoverMe`, there is one execution path on which this method returns true: `1 -> 3 -> 5 -> 7 -> 9 -> 11`. For this path, Dianju builds the following path condition:

```
[ SymbolicString_0 != null &&
  SymbolicSet_1 != null &&
  SymbolicMap_2 != null &&
  !SymbolicSet_1.contains(null) &&
  !SymbolicSet_3.contains(null) &&
  !SymbolicSet_4.contains(null) &&
  SymbolicSet_4.containsAll(SymbolicSet_1) ]
```

where `SymbolicString_0` represents the value of `jar`, `SymbolicSet_1` represents the value of `cover`, `SymbolicMap_2` represents the value of `b2s`, `SymbolicSet_3` represents the `keySet` view of the map, and `SymbolicSet_4` represents the `values` view of the map.

For these constraints, Dianju's constraint solver generates the following satisfying assignment:

```
[ SymbolicString_0 = ''jar'',
  SymbolicSet_1 = {},
  SymbolicMap_2 = {},
  SymbolicSet_3 = {},
  SymbolicSet_4 = {} ]
```

Since Dianju's generation is based on specialized generators, it is easy for Dianju to enumerate all solutions within given bounds[1] For example, another solution that Dianju generates corresponding to the path that returns true is:

```
[ SymbolicString_0 = ''jar'',
  SymbolicSet_1 = { 1 },
  SymbolicMap_2 = { 0=0, 1=0, 2=1 },
  SymbolicSet_3 = { 0, 1, 2 },
  SymbolicSet_4 = { 0, 1 } ]
```

During exhaustive enumeration using a bound of four (i.e., 4 string literals and 4 integer objects that form the elements of the sets, and the keys and the values of the maps), Dianju considers 16,384 distinct candidate structures, of which it generates 6,872 as valid structures (i.e., as satisfying `repOk`). Dianju completes its generation of all inputs in one-tenth of a second (using a Pentium-M processor at 1.7GHz with 512MB of RAM).

3.2 CheckMe

Even though our primary design goal in Dianju is to enable efficient symbolic execution of programs to check their correctness, Dianju has other applications. For example, we can use Dianju's constraint solving capability to check set equivalences. As an example, consider the following implication, which is taken from the definition of equality of two sets, say S and T: $S \subseteq T \wedge T \subseteq S \;\Rightarrow\; S = T$.

We can check this implication (for small sets) by writing it as a Java predicate. For example the following predicate suffices:

```
    boolean checkImplication(Set S, Set T) {
        // =>
1:      if (S.containsAll(T) && T.containsAll(S))
2:          if (S.equals(T))
3:              return true;
4:          else return false;
5:      return true;
    }
```

Dianju can symbolically explore all paths in `checkImplication` and determine (by checking on all small sets) the feasibility of the only predicate path (`1 -> 2 -> 4`) that returns false. In this case Dianju fails to find a satisfying assignment to this path and therefore determines that the path is infeasible, i.e., the implication holds. Using sets of size up to 8, Dianju checks the implication on 65,536 inputs and completes its analysis in one-tenth of a second.

Dianju's ability to solve constraints on sets, maps, strings and integers has obvious applications to checking declarative specifications similar to those given in Alloy [18]. It is indeed very interesting to explore adding a transitive closure operator and quantifiers to Dianju to allow programmers to write essentially in Java Alloy-like specifications and automatically analyze them. Alternatively, since Dianju solves constraints at an abstract level, it could leverage Alloy's translation to SAT and off-the-shelf SAT technology. We plan to explore these relations further in future work.

[1]If no bounds are given, Dianju tries to automatically infer "reasonable" bounds and values (Section 4.6).

4. ABSTRACT SYMBOLIC EXECUTION

Dianju performs symbolic execution by systematically exploring program paths, building path conditions, and checking their feasibility. For test generation using `repOk` (class invariant), Dianju symbolically executes `repOk` and solves constraints represented by path conditions that correspond to paths in `repOk` on which it returns true. The details of the basic symbolic execution, which operates on concrete heap and symbolic primitives and uses off-the-shelf implementations of decision procedures, such as the Omega library [28] and CVC-lite [4], can be found elsewhere [20]. Here we concentrate on the new features that Dianju provides for symbolically executing library classes.

Dianju test generation algorithm proceeds as follows. To generate objects of class `C`, Dianju proceeds as follows.

- It replaces declared types of field in `C` (and transitively replaces declared types of field in those types) with corresponding symbolic field types.
- It replaces operations on original types with its library operations that enable symbolic execution. The symbolic operations build symbolic expressions, which represent values of variables that have symbolic types, and add constraints to path conditions.
- It executes `repOk` on a freshly allocated object of class `C`, initializes the object fields to freshly allocated symbolic objects of appropriate symbolic types.
- It solves the path condition constraints for paths on which `repOk` returns true.

Symbolic execution of a method that takes multiple inputs follows suit: the formal parameters can be treated as fields of a new class and the method can be wrapped into a new method that takes exactly one input, which is an object of the new class. [6].

We have implemented several of Dianju's modules and the tool is currently undergoing further development.

4.1 Building constraints and updating state

Dianju enables symbolic execution via program instrumentation at the bytecode level [14,19,30] using Javassist [10] and the Bytecode Engineering Library [12]. For example, the bytecodes

```
   88:  getfield        #46; //Field cover:Ljava/util/Set;
   91:  invokeinterface #71,  2; //InterfaceMethod java/uti-
l/Collection.containsAll:(Ljava/util/Collection;)Z
```

are replaced with:

```
   88:  getfield        #31; //Field cover:Ldianju/symbolic/
SymbolicSet;
   91:  invokevirtual   #94; //Method dianju/symbolic/Symbo-
licSet.containsAllConstraint:(Ldianju/symbolic/SymbolicSet;)Z
```

We implement systematic path exploration using backtracking. Dianju library predicates that represent predicates of an instrumented

type non-deterministically return true and false, which enables exploration of both branches at each condition. The backtracking algorithm follows a simple state-less search [15].

Dianju declares the abstract class `Constraint` to model constraints. It is subclassed by `StringConstraint`, `SetConstraint`, and `MapConstraint` (besides similar classes that represent constraints on primitives), which are also abstract classes. Each of these is subclassed by concrete classes that represent the constraints that correspond to the API methods that return booleans—it is these methods together with the '==' comparison, which introduce non-determinism and therefore form the basis for generating the path conditions.

Methods which update state are represented by corresponding methods that update symbolic state. For example, the method that adds a new element to a set is modeled by a Dianju method, which performs the addition symbolically and returns a boolean depending on the original symbolic state of the set variable: if Dianju cannot determine with certainty whether the object originally existed in the set it non-deterministically returns true and false. Dianju's exploration can therefore traverse infeasible paths, however, when it solves constraints it can determine whether any input (within the bounds it explores) can actually execute the path.

A limitation of the current Dianju prototype is that it assumes object equality is based on object identity (as is the case for the `Object` class). Since in Java a class may override the `equals` method and re-define equality among its objects, a more sophisticated treatment is required.

4.2 Solving constraints

Dianju essentially uses an enumerate-and-filter approach to solve constraints. However, it optimizes its search using specialized generators for known types and focusing their generation based on the constraints.

Given a path condition, Dianju determines dependencies among symbolic variables. These dependencies arise, for example, when a `keySet` view is taken on a map. Dianju groups together constraints based on their dependency relationship and solves each group of constraints in isolation, and then combines these solutions using cross product. For systematic generation, Dianju exhaustively enumerates the solutions for each group using its specialized generators (Section 4.3) and takes a cross product of the enumerations. Dianju then filters each resulting candidate using the constraint it is solving.

The first step in solving a group of constraints is to identify symbolic variables that must have the same value according to the equality constraints. Dianju builds equivalence classes and solves the given constraints using systematic enumeration for exactly one representative from each equivalence class. Once it finds a satisfying solution with respect to the representatives, it generates a complete solution for all the symbolic variables.

Dianju generates candidate solutions and interprets the constraints in the path condition on each candidate to determine its feasibility. Dianju's enumerative solver can generate one, several, or all satisfying assignments (within a bound) as desired by the user.

4.3 Providing specialized generators

Dianju supports specialized generators for sets and maps (in addition to those for strings and primitives, which perform simple enumeration from a pool of literals). Korat, in contrast, supports specialized generators at the representation level [25].

4.3.1 Set generator

Dianju's set generator uses the well-known mathematical result that the number of subsets of a given finite set of cardinality n is 2^n. The set generator class has a field called `domain`, which represents the pool of elements from which sets are to be generated. (These elements can indeed be symbolic objects.) For a pool of size n, each integer i between 0 and $2^n - 1$ (inclusive) is translated to its binary form, which represents a subset of the pool and is among the generates sets. For example, for the pool $\{hello, world\}$, the binary string "01" represents the set $\{world\}$. The generator supports operations for adding and removing elements from the `domain`, to focus generation on desired sets.

4.3.2 Map generator

Maps can be viewed mathematically as functions. (A map maps any given key that exists in the map to exactly one value.) Dianju's map generator is based on this observation. The generator uses two domains as a basis for its generation. The field `keyDomain` indicates the pool of elements that represent the keys and the field `valueDomain` indicates the pool of elements that represent the values. For a `keyDomain` pool of size n and a `valueDomain` pool of size m, there are m^n maps. The map generator translates each integer between 0 and $m^n - 1$ (inclusive) into a string that represents the integer as a base-m number. These strings represent the maps. The generator supports operations for adding and removing elements from the `keyDomain` and the `valueDomain`, to focus generation on desired maps.

4.3.3 Generator efficiency

Specialized generators for sets and maps provide highly efficient generation. For example, Dianju can generate all 2^{20} (> 1 million) `java.util.HashSet` objects constructed out of a pool of 20 `java.lang.String` objects ({`"0", ..., "19"`}) in one second (on a Pentium-M processor at 1.7GHz with 512MB of RAM). This really should not come as a surprise, and results from specialized generators should not be compared in the raw with general purpose constraint solvers such as Korat or Alloy, since specialized generators already "know" what to generate and how to generate it.

4.4 Concretizing abstract sets and maps

The set and map generators provide generation at an abstract level. These abstract objects need to be concretized into actual Java objects to test programs. The concretization translation is straightforward and uses API level calls to generate appropriate objects. For example, the following code concretizes the abstract set {`1, 2`} into a java.util.HashSet object:

```
Set s = new HashSet();
s.add("1");
s.add("2");
```

It is worth pointing out that Dianju's test generation combines abstract level generation and concrete level generation. Consider, for example, a class `C` that declares a field `s` of type `Set`. Dianju will generate an object of `C` by first generating a `HashSet` (or `TreeSet`,

if that is desired) object as above and then explicitly setting the field `s` of a freshly allocated object of class `C`.

4.5 Using constraints to focus generation

During symbolic execution, Dianju uses constraints on equality and set membership to make the generators more efficient. For example, consider the constraint `!s.contains(null)` (on a set `s`) in the path condition. Dianju notifies the set generator to remove `null` from its pool of elements. This makes the generator focus on enumerating relevant sets. For the constraints that Dianju cannot convey directly to its generators, it uses exhaustive enumeration.

4.6 Bounding input sizes

Dianju does not require the user to provide bounds on the number of elements to consider in each pool for generation. Instead, it tries to determine automatically from the given constraints what the range of elements should be. For example, if Dianju solver encounters a constraint of the type `s.contains("hello")`, it adds `"hello"` to the domain for generating that set. Additionally Dianju provides some default values that it tries for the elements, based on the element types. As stated before Dianju allows the user to initialize the domains to tune the generation. Dianju also notifies the user of the domains used in generation.

5. DISCUSSION

Dianju has its set of limitations. Some of these arise from the complexity inherent in the Java Collections Framework (JCF). For example, JCF allows views to update the underlying collections. It sometimes becomes infeasible to keep building constraints in order to track all the updates. When Dianju's exploration hits such a control point, Dianju uses the current path condition to systematically concretize symbolic objects into concrete objects. Dianju's symbolic execution then takes the form of bounded exhaustive testing where, say all sets up to a certain size will be explored. Notice that Dianju does not need to concretize all the symbolic objects, it can heuristically select a subset that simplifies the current path condition to a sufficient degree.

Efficient handling of arrays symbolically presents a challenge to Dianju. Symbolic execution of integers together with the use of arrays typically makes the exploration of paths explode (because a symbolic index into an array can refer several array elements and must also remain within the array bounds). We handle arrays by non-deterministically initializing them with small lengths at the point when they are accessed first.

Based on the work we have presented here, we plan to develop an extensible framework that allows users to "plug-in" symbolic execution for the types they desire and to define the semantics of their symbolic execution.

Dianju's solving, in particular, and symbolic execution, in general, can benefit from incremental constraint solving [37]. We can define a notion of similarity on paths based on the length of their common prefix. Then a solution to the path condition on a path that has already been explored can be used as a starting point for searching for a solution to the path condition on a similar path.

Our previous work on TestEra [26] presents an abstract treatment of arrays and commonly used classes, such as `java.lang.Integer`, and has a flavor similar to Dianju but uses specifications written in Alloy [18]—a first-order declarative language. Our more recent work on generalized symbolic execution [20] treats strings symbolically, and builds string constraints and implements a rudimentary algorithm for solving them.

5.1 Hybrid approaches to symbolic execution

We believe techniques based on symbolic execution hold a lot of promise for increasing software reliability. Dianju presents a hybrid approach that aims to scale symbolic execution using a combination of abstract level and representation level executions. Another hybrid approach that we are exploring is to perform symbolic execution *selectively* by treating a dynamic subset of all possible variables/fields as symbolic [14, 19, 30]. We have used this approach in data structure repair. The experimental results are encouraging: repair based on symbolic execution seems to scale linearly with the size of the corrupt structure (for a small fixed number of corrupt fields).

5.2 Unification of Verification and Repair

A key insight that our work provides is that scalable approaches to symbolic execution can unify the fields of software verification and resilient computing—two fields that have traditionally employed very different algorithms. The ability to efficiently solve constraints that are given as predicates in an imperative language enables not only approaches for bounded exhaustive testing [6, 20, 24] but also approaches for repairing structurally complex data [14, 19, 30].

5.3 Assert-first programming

Our envisioned unification has the potential to pave the way for an unprecedented increase in software reliability at a much lower cost. The use of assertions poses a minimal burden on practitioners: an assertion can be written using the underlying programming language and there is no need to learn a new annotation/specification language. Assertions, in fact, are already one of the most commonly used forms of program annotations. Fully automatic analyses based on assertions can make them a lot more attractive.

The act of writing an assertion makes the user consider what behavior is intended. This focus on intent is the key idea behind the success of test-first programming [5]. We believe *assert-first programming* can in fact realize the same benefits as test-first programming but at a lower cost: writing an assertion is often less work than manually writing a high quality test-suite. Having written appropriate assertions, the user can rely on our symbolic execution based analyses to fully automatically test the code before deployment as well as ensure that any executions that go awry once the code is deployed will be corrected at run-time.

In some cases, the code already contains assertions, as is strongly advocated by defensive programming [23]. For example, the SGLIB C library [1] comes with annotated class invariants. Abstract symbolic execution enables testing and repair in such a case for free.

The use of assertions in hardware verification is already immensely popular: assertion-based verification is an essential part of modern-day chip design [9,32]. We believe the time has also come to realize the potential benefits that assertions offer in software.

6. CONCLUSION

We have presented Dianju, a new technique that aims to address the scalability of symbolic execution. The fundamental idea in Dianju is to perform symbolic execution of commonly used library classes (such as strings, sets and maps) at the abstract level rather than the representation level. Dianju defines semantics of operations on symbolic objects of these classes, which allows Dianju to abstract away from the complexity that is normally inherent in library implementations, thus promising scalable analyses based on symbolic execution.

Dianju uses specialized generators, which enable it to enumerate objects in a highly efficient manner. Dianju further guides the generation based on the path condition constraints that it builds during symbolic execution.

We believe Dianju presents a promising new approach for making analyses based on symbolic execution feasible for a wide class of programs, including those that use advanced constructs of modern programming languages.

7. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and Darko Marinov for detailed comments on a previous draft. This work was funded in part by NSF ITR-SoD award #0438967.

8. REFERENCES

[1] SGLIB—A Simple Generic Library for C. `http://xref-tech.com/sglib/main.html`.

[2] T. Andrews, S. Qadeer, S.K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *16th International Conference on Computer Aided Verification (CAV)*, Boston, MA, July 2004.

[3] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Rosu, K. Sen, W. Visser, and R. Washington. Combining test case generation and runtime verification. *Theoretical Computer Science*, 336:209–234, May 2005.

[4] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16th International Conference On Computer Aided Verification*, Boston, MA, July 2004.

[5] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.

[6] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, July 2002.

[7] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, 1975.

[8] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience*, 30(7):775–802, 2000.

[9] Cadence. Components of a complete assertion-based verification solution, 2005. `http://www.cadence.com/whitepapers/abv_wp.pdf`.

[10] Shigeru Chiba. Javassist—a reflection-based programming wizard for Java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.

[11] Christoph Csallner and Yannis Smaragdakis. Check 'n' crash: Combining static checking and testing. In *Proc. 27th International Conference on Software Engineering (ICSE)*, St. Louis, MO, May 2005.

[12] Markus Dahm. Byte code engineering library. `http://bcel.sourceforge.net/`.

[13] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, 2002.

[14] Iván García. Enabling symbolic execution of Java programs using bytecode instrumentation. Master's thesis, Department of Electrical and Computer Engineering, The University of Texas at Austin, May 2005.

[15] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, pages 174–186, Paris, France, January 1997.

[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, 2005.

[17] Gerald Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[18] Daniel Jackson. Micromodels of software: Modelling and analysis with Alloy, 2001.

[19] Sarfraz Khurshid, Iván García, and Yuk Lai Suen. Repairing structurally complex data. In *12th International SPIN Workshop on Model Checking of Software*, San Francisco, CA, August 2005.

[20] Sarfraz Khurshid, Corina Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Warsaw, Poland, April 2003.

[21] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[22] Bogdan Korel. Automated test data generation for programs with procedures. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, San Diego, CA, 1996.

[23] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[24] Darko Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, December 2004.

[25] Darko Marinov, Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Martin Rinard. An evaluation of exhaustive testing for data structures. Technical Report MIT-LCS-TR-921, MIT CSAIL, Cambridge, MA, September 2003.

[26] Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, CA, November 2001.

[27] Corina S. Pasareanu and Willem Visser. Verification of java programs using symbolic execution and invariant generation. In *Proc. 11th International SPIN Workshop on Model Checking of Software*, Barcelona, Spain, April 2004.

[28] William Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 31(8), August 1992.

[29] Sanjit Seshia and Randal Bryant. Deciding quantifier-free presburger formulas using parameterized solution bounds. In *Nineteenth Annual IEEE Symposium on Logic in Computer Science (LICS)*, Turku, Finland, July 2004.

[30] Yuk Lai Suen. Automatically repairing structurally complex data. Master's thesis, Department of Electrical and Computer Engineering, The University of Texas at Austin, May 2005.

[31] Sun Microsystems. *Java 2 Platform, Standard Edition, v1.3.1 API Specification*.

[32] Synopsis. Assertion-based verification, March 2003. `http://www.synopsys.com/products/simulation/assertion_based_wp.pdf`.

[33] Margus Veanes, Colin Campbell, Wolfram Schulte, Pushmeet Kohli, N. Tillmann, and W. Grieskamp. On-the-fly testing of reactive systems. (Submitted for publication.).

[34] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000.

[35] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.

[36] Srinivas Visvanathan and Neelam Gupta. Generating test data for functions with pointer inputs. Edinburgh, Scotland, September 2002.

[37] Jesse Whittemore, Joonyoung Kim, and Karem Sakallah. SATIRE: A new incremental satisfiability engine. In *Proc. 38thConference on Design Automation (DAC)*, Las Vegas, NV, June 2001.

[38] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. 11th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Edinburgh, UK, April 2005.