Falling Back on Executable Specifications

Hesam Samimi, Ei Darli Aung, and Todd Millstein

Computer Science Department
University of California, Los Angeles
{hesam,eidarli,todd}@cs.ucla.edu

Abstract. We describe a new approach to employing specifications for software reliability. Rather than only using specifications to *validate* implementations, we additionally employ specifications as a *reliable alternative* to those implementations. Our approach, which we call *Plan B*, performs dynamic contract checking of methods. However, instead of halting the program upon a contract violation, we employ a constraint solver to automatically *execute* the specification in order to allow the program to continue properly. This paper describes Plan B as well as its instantiation in an extension to Java with executable specifications that we call PBnJ (Plan B in Java). We present the design of PBnJ by example and describe its implementation, which leverages the Kodkod relational constraint solver. We also describe our experience using the language to enhance the reliability and functionality of several existing Java applications.

1 Introduction

Many researchers have explored the use of *specifications*, for example pre- and postconditions on methods expressed in a variant of first-order logic, to gain confidence in the correctness of software. One approach employs specifications for static program verification, guaranteeing that each method meets its declared specification for all possible executions. In recent years this approach has been increasingly automatable via the use of constraint solvers (e.g., [11, 1, 4, 30, 31]). However, the limits of static verification make it difficult to scale this technology to complex programs and rich program properties. A complementary approach employs specifications for dynamic contract checking (e.g., [22, 10]). In this style pre- and postconditions are checked as a program is executed. Performing the checking is straightforward since specifications are only enforced on a single run-time program state at a time. However, if a specification violation is found, there is little recourse other than halting the program, which is unacceptable in many situations.

In this paper we explore a new approach to employ specifications for software reliability, which we call *Plan B*. The main idea is that specifications can be used not only to check an implementation's correctness but also as a *reliable alternative* to faulty or incomplete implementations. Like dynamic contract checking, our approach checks for violations of method postconditions at run time. However, rather than simply halting the program upon a violation, Plan B *falls back* on the specification itself, directly executing it in order to safely continue program execution. We observe that specifications can be executed using the same kinds of constraint solvers that are traditionally used for static verification. Rather than using the constraint solver to verify the correctness of a method for all possible executions, Plan B *ignores* the method implementation and lets the constraint solver search for a *model* that satisfies the method's postcondition given the dynamic program state on entry to the method.

Integrating executable specifications into a programming language in this fashion provides several benefits. As described above, Plan B can be used to safely recover from dynamic contract violations. Similarly, Plan B can safely recover from arbitrary errors that prematurely terminate a method's execution, for example a null pointer dereference or out-of-bounds array access. Finally, Plan B allows programmers to leverage executable specifications to simplify software development. For example, a programmer could implement the common cases of an algorithm efficiently but explicitly defer to the specification to handle the algorithm's complex but rare corner cases. While executing specifications can be significantly less efficient than executing an imperative implementation, current constraint-solving technology is acceptable in many situations, especially those for which the only safe alternative is to halt the program's execution. Furthermore, Plan B can take advantage of continual improvements in constraint-solving technology to broaden its scope of applicability over time.

Plan B builds on two existing strands of research. First, researchers have previously explored forms of executable specifications. For example, several prior projects propose executing specifications by translation to logic programs [29, 20]. The goal of this research is mainly to execute specifications in order to test and debug them, independent of the program that employs them. Morgan's specification statement [23] is closer to our work since it integrates specifications with implementations. However, the specifications in that setting are used as part of a manual process of program *refinement* to a correct-by-construction implementation. Plan B leverages similar technology but uses it for a different purpose, namely as a fallback mechanism that is tightly integrated with the execution of a traditional programming language. This new use of executable specifications requires constraint solving to be practical as an *online* tool during program execution. We address this challenge with several novel program annotations that allow the programmer to declaratively bound the search space in an application-specific manner.

Second, there have been several recent research efforts on dynamic *repair* to recover from program errors (e.g., [2, 3, 9, 8]). These tools use heuristic local search to find a "nearby" state to the faulty one that satisfies a class's integrity constraints. The goal is to allow execution to continue acceptably, even in the face of possible data loss or corruption due to the fault. Our Plan B approach is complementary: we ignore the faulty program state and "re-execute" the method using its postcondition (and the class's invariants) with the aid of general-purpose decision procedures. PlanB is in general less efficient than repair, which leverages the fact that many errors only break invariants in local ways. However, Plan B is useful when it is important to fully recover from a fault rather than simply repairing it. Plan B also enables safe recovery from an arbitrarily broken program state and can ensure rich program properties that relate the input state of a method to its output state, both of which are challenging for local-search-based approaches.

We have designed and implemented a concrete instantiation of the Plan B approach as an extension to Java called PBnJ (Plan B in Java). PBnJ augments Java with support for class invariants and method postconditions in a first-order relational logic similar to the Alloy modeling language [14]. These specifications are automatically translated to ordinary Java predicates and used for dynamic contract checking of methods. In addition, the PBnJ compiler automatically creates a translation of each specification for input to the Kodkod constraint solver for relational logic [27]. When a method fails its dynamic postcondition check, Kodkod is invoked to find a model for the postcondition, the resulting model is translated into appropriate updates to variables and fields, and execution continues.

After overviewing PBnJ by example (Section 2) and detailing its implementation strategy (Section 3), we describe our experience using PBnJ in several case studies (Section 4). First, we have created executable specifications for several common data structures. We use this case study as a stress test for our approach, employing complete specifications that ensure 100% correctness in the event of a fallback. Second, we show how executable specifications can enhance the reliability and functionality of several existing Java applications. The PBnJ compiler is available at `http://www.cs.ucla.edu/~hesam/planb`.

2 An Overview of PBnJ

This section overviews PBnJ and its benefits through a few motivating examples. After illustrating PBnJ's specification language, we describe how these specifications are used for dynamic contract checking and automatic fallback. Finally we discuss some novel language mechanisms we have introduced to make falling back on specifications practical.

2.1 Specifications

PBnJ includes fairly standard mechanisms for incorporating specifications into a Java program. Method postconditions are specified in an optional `ensures` clause on methods. Similarly, an optional `ensures` clause on a class declaration specifies any class invariants, which must hold at the end of the execution of each public method in the class. As a simple example, Figure 1 shows a square root function on integers and an associated specification. As is common, the keyword `result` refers to the value returned by the method.

In addition to supporting side-effect-free primitive operations in Java, PBnJ's specification language includes a form of first-order relational logic based on Alloy [14]. In this style, Java classes are modeled as unary relations (i.e., sets

```
public static int intSqrt(int i)
  ensures result >= 0 &&
          result <= i / result &&
          (result + 1) > i / (result + 1) { ... }
```

Fig. 1. A specification for the integer square root function. We convert multiplications into divisions to avoid integer overflow [25].

```
SpecExpr ::= QuantifiedExpr | SetComprehension | SpecPrimary
QuantifiedExpr ::= ( all | no | some | one | lone ) QuantifiedPart
SetComprehension ::= { ( all | some ) QuantifiedPart }
QuantifiedPart ::= Type Identifier [: SpecPrimary ] | SpecExpr
SpecPrimary ::= Lit | Primary | FieldClosure | IntegerInterval
            | various Java primitive operations on integers and booleans
Lit ::= null | this | result | IntegerLiteral | BooleanLiteral
FieldClosure ::= Primary . (* |^ |> ) Identifier (+ Identifier )*
IntegerInterval ::= Primary .. Primary
```

Fig. 2. Specifications in PBnJ. The nonterminals Primary, IntegeralLiteral, and BooleanLiteral are defined as in the Java Language Specification [13].

of objects) and Java fields are modeled as binary relations between an object and its field value. The syntax of `ensures` specification expressions is shown in Figure 2 and includes forms of quantification as well as transitive closure on relations. We also provide procedural abstraction for specifications through a notion of *specification methods*, which additionally support side-effect-free statement forms including assignment to local variables and if-then-else statements.

Figure 3 uses these features to provide the specification for a linked list implementation. The `List` class includes a `spec` method `nodes`, defined as the reflexive, transitive closure of the `next` relation starting from `this.head`. The `List` class uses this method to specify that the list must be acyclic (specification appearing at class header) and to specify the postcondition for a sorting routine. Specification methods can invoke other specification methods, but not ordinary Java methods, and only specification methods can be invoked from an `ensures` clause. The `nodesOfValue` method uses PBnJ's facility for set comprehensions, and the `values` method uses the `.>` operator to map the `value` relation on each node in `nodes()`.

Each object in PBnJ has an implicit field named `old` that can be used in method specifications to refer to the state of that object on entry to the method. This simple mechanism is very powerful because `old` has a "deep copy" semantics. For example, the specification of `bubbleSort` uses the `old` field of `this` to ensure that the implementation of the method does not add or remove any nodes from the list. Because of the declared class invariant, the resulting

```
public class List ensures isAcyclic()  {
  Node head;

  spec public PBJSet<Node> nodes() { return head.*next; }
  spec public PBJSet<Integer> values() { return nodes().>value; }

  spec public boolean isAcyclic() {
    return head == null || some Node n : nodes() | n.next == null; }

  spec public boolean isSorted() {
    return all Node n : nodes() |
      (n.next == null || n.value <= n.next.value); }

  spec public PBJSet<Node> nodesOfValue(int i) {
    return { all Node n : nodes() | n.value == i  }; }
  spec int occurrencesOf(int i) { return nodesOfValue(i).size(); }

  spec boolean isPermutedSublistOf(List l) {
    return all int i : values() |
      occurrencesOf(i) <= l.occurrencesOf(i); }
  spec public boolean isPermutationOf(List l) {
    return this.isPermutedSublistOf(l) && l.isPermutedSublistOf(this); }

  public void bubbleSort()
    ensures this.isPermutationOf(this.old) && this.isSorted() {
    Node curr, tmp, prev = null, last = null;
    while (last != head) {
      curr = head;
      while (curr != last) {                   // A
        if (curr.value > curr.next.value) {
          if (curr == head)
            head = curr.next;
          else
            prev.next = curr.next;
          tmp = curr.next.next;
          curr.next.next = curr;
          prev = curr;                         // B
          curr.next = tmp;
        } else {
          prev = curr;
          curr = curr.next;
        }
      }
      last = curr;
} } }
```

Fig. 3. A linked list of integers in PBnJ. The `Node` class (not shown) includes an integer `value` field and a `next` field of type `Node`. The marked lines are discussed later.

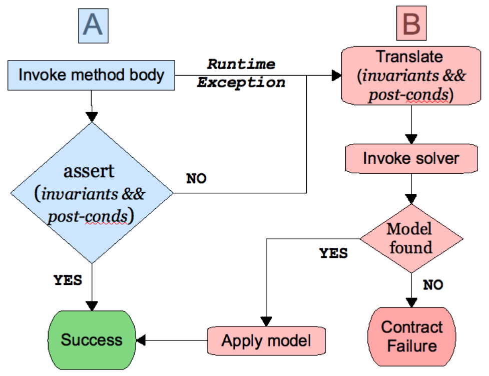

Fig. 4. Method invocation in PBNJ: falling back from Plan A (ordinary method execution) to Plan B (executing the method specification).

list is also required to be acyclic. Our implementation incurs a performance overhead because of these deep copies, but this mechanism has allowed us to easily experiment with a variety of expressive postconditions. In the future we will explore several approaches to optimize the implementation of `old`, and we may consider switching to a more standard, shallow, notion of `old` as found for example in JML [21].

A secondary benefit of specification methods in PBNJ is that they are directly executable. Specification methods can be invoked by ordinary Java methods and thereby used as part of the implementation of a class. For example, clients of our list can invoke the `nodes` method to get a set of all nodes which can then be manipulated as usual in Java code. In this way, specifications are useful not only as a fallback mechanism when an implementation fails (as described below), but also to make implementations more declarative and hopefully more reliable by construction. The PBNJ compiler automatically translates specification methods into ordinary Java methods (see Section 3). We represent sets using our own `PBJSet` interface which is similar to that in the Java standard library but also provides functional versions of several operations (e.g., adding an element, set union), since specifications need to be free of side effects.

PBNJ does not support user-defined preconditions on methods, but only postconditions. Our research focuses on the idea of falling back on executable specifications, which is more natural for postconditions (and class invariants) than preconditions. Nonetheless, it may be useful to incorporate preconditions in the future.

2.2 Contract Checking and Recovery

Figure 4 overviews the execution of a method in PBNJ. The method is executed as usual. Upon normal completion, we check that the method obeys the declared class invariant and method postcondition by executing (Java translations of) these predicates. If the method invocation violates the declared specification, we fall back to the specification. Execution also falls back to the specification if the method terminates with a Java `RuntimeException` (e.g., an `ArrayIndexOutOfBoundsException` or `NullPointerException`). Fallback involves translating the class invariant and method postcondition into the logic of the Kodkod relational constraint solver [27] and invoking the solver to search for a *model* satisfying the specification. If a model is found we use the model to transform the current program state and continue execution. If Kodkod reports unsatisfiability, then the specifications have no solution within the search bounds provided by the programmer (see Section 2.3). In such a case we throw a `ContractViolationException`, similar to what would happen with traditional contract checking.

Before a method is executed, we make a deep copy of the reachable state from any object whose `old` field is mentioned in the method's postcondition. This copy is then used to check the postcondition after the method's execution completes. If the postcondition is not satisfied, the old state on entry to the method is translated into a set of relations that are provided to the Kodkod solver for the purpose of model finding. Details on this translation are provided in the next section.

Accidental Fallback Our linked list example in Figure 3 illustrates how PBnJ helps ensure reliability in the face of program errors. The `bubbleSort` method implementation has two errors, which are marked in the figure. First, the guard in the `while` loop at line A should read `curr.next != last`. The error will cause the subsequent line to throw a `NullPointerException` when `curr.next` is null. Second, line B should read `prev = curr.next`. This error does not cause any exceptions to be thrown, but on some lists it will erroneously throw away elements. With the provided specification, PBnJ catches and successfully recovers from both errors via the external constraint solver. Aside from a slowdown in the application (depending on the size of the list being sorted), this recovery is completely transparent to the user and to the rest of the application.

Intentional Fallback Figure 5 illustrates an example where fallback may be useful for a programmer to rely upon explicitly, in order to handle complex corner cases that arise in rare circumstances. SweetHome3D [26] is a popular interior design application implemented in Java. Users can add and arrange pieces of furniture in a room and view the results in a 3D view. We enhanced the implementation of SweetHome3D to automatically rearrange furniture as new pieces are added in a room to ensure that two pieces never overlap in space. This is a good application for intentional fallback, since it would be cumbersome and error prone to implement manually, and it may be reasonable to expect such rearranging to be necessary only rarely.

To implement the enhancement we simply augmented the existing method for adding a piece of furniture to invoke the empty method `moveFurnitureIfNeeded` shown in Figure 5. PBnJ's contract checking dynamically checks the method's

```
public void moveFurnitureIfNeeded()
  ensures notOverlapped() && notTooFar() && keepRelativePosition() { }

spec public boolean notOverlapped() {
  return all HomePieceOfFurniture p1 : this.furniture |
           all HomePieceOfFurniture p2 : this.furniture |
             (p1 == p2 ||
              (abs(p1.getX() - p2.getX())
                  >= ((p1.getWidth() + p2.getWidth())/2)) ||
              (abs(p1.getY() - p2.getY())
                  >= ((p1.getDepth() + p2.getDepth())/2)));
}
spec private boolean notTooFar() {
  return all HomePieceOfFurniture p: this.furniture |
           ((abs(p.getX() - p.old.getX()) <= p.getWidth()/2) &&
            (abs(p.getY() - p.old.getY()) <= p.getDepth()/2));
}
spec private boolean keepRelativePosition() {
  return all HomePieceOfFurniture p1: this.furniture |
           all HomePieceOfFurniture p2: this.furniture |
             ((compare(p1.getX(), p2.getX()) ==
                        compare(p1.old.getX(),p2.old.getX())) &&
              (compare(p1.getY(), p2.getY()) ==
                        compare(p1.old.getY(),p2.old.getY())));
}
```

Fig. 5. Enhancing SweetHome3D to automatically rearrange overlapping pieces of furniture. The `getX` and `getY` methods return the coordinates of the center of a piece of furniture. The `compare` method returns -1 if the first argument is less than the second argument, 0 if the arguments are equal, and 1 otherwise.

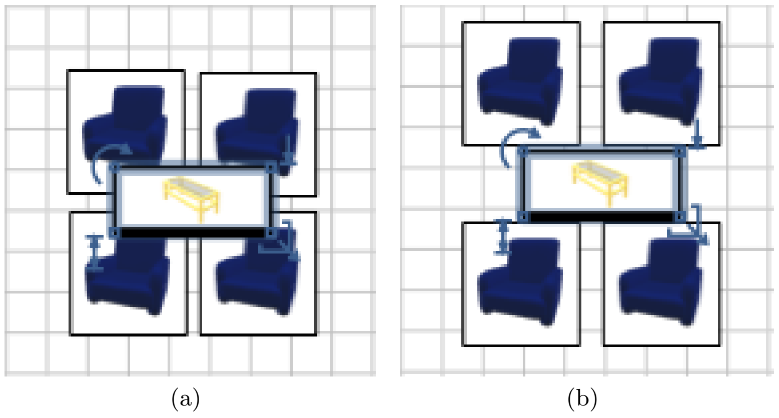

Fig. 6. (a) Four chairs and a coffee table overlapping the chairs. (b) PBnJ's fallback mechanism automatically rearranges the furniture.

postcondition and performs fallback if necessary, thereby allowing the programmer to safely ignore this special case. The declared postcondition ensures that there are no overlaps and that the new position of each piece of furniture is close to its old position and retains the same relative position to every other piece. Figure 6 shows "before" and "after" screenshots for a simple example.

2.3 Making Fallback Practical

We have developed two main techniques to allow PBnJ programmers to declaratively bound the search space for fallback in an application-specific manner.

Frame Conditions By default, the constraint solver can modify the values of any fields mentioned in a postcondition or class invariant in order to perform the fallback. However, it is useful to allow programmers to override this default by providing an explicit frame condition as a subset of fields that are intended to be modifiable. This is done with an optional `modifies fields` clause on a method. Traditionally such *frame conditions* have been used to aid static verification of methods (e.g., [11]). We instead use these annotations to improve the performance of constraint solving by limiting the search space. For example, annotating the `bubbleSort` method in Figure 3 with the following clause prevents the solver from attempting to change the integer values stored in each list node, but instead only their `next` pointers:

```
modifies fields List.head, Node.next
```

Frame conditions can also be used to simplify a specification by ensuring that certain nonsensical solutions are ruled out.

By default any object reachable from `this` and formal parameters on entry to the faulty method may be modified. We allow programmers to override this default through a novel `modifies objects` clause, which specifies a Java expression that evaluates to a collection of objects; PBnJ's fallback mechanism considers all other objects to be immutable for purposes of fallback. Consider again the `moveFurnitureIfNeeded` method in Figure 5. In a room with a lot of furniture, it may be useful to restrict which pieces to consider moving upon an overlap. Aside from the performance benefit, this restriction ensures that pieces are only moved when necessary. Rather than building this constraint into the postcondition, which would be tedious and complex, the programmer can provide the following clause, where `surroundingPieces` is a regular Java method that returns the set of pieces that are sufficiently close to the overlap:

```
modifies objects surroundingPieces(p)
```

PBnJ invokes the `modifies objects` expression dynamically when a fallback event is triggered, and the resulting set of objects is communicated to the solver as being modifiable; all other reachable objects are treated as immutable. (We evaluate the `modifies` clause on *old* copies of objects in order to retain its on-entry evaluation semantics.) This approach allows for significant flexibility. For example, we have created an alternate version of our SweetHome3D enhancement whereby the modifiable pieces of furniture are determined by the SweetHome3D user instead of being computed. In this style, the user explicitly clicks on pieces of furniture to specify which ones can be automatically rearranged if necessary.

Bounding the Universe Kodkod is a SAT-based reasoning tool, and it therefore expects a finite bound for the search space for each of the types, including primitives. This implies that if the tool does not find a model, we can only assume the problem is unsatisfiable within the given bounds. This incompleteness can cause PBnJ to signal a contract violation exception when a recovery might have been possible, but this is no worse than the situation with traditional contract checking.

When executing specifications involving search for integer values, the usual 32-bit integer range is often not a tractable space. By default Plan B assumes 8-bit integers when executing specifications. However, we allow the user to explicitly set bounds for integers. The number of objects of each class must also be bounded. By default we bound each class by the number of instances of that class that are reachable from the receiver and formal parameters at the point of the fallback. However, this bound is insufficient if the faulty method may need to instantiate new objects. To handle this situation, we allow each method to include an optional annotation specifying an upper bound on the number of new instances of each class that Kodkod may create in order to satisfy the method's specification. For instance, the specification for an `add` method in our `List` class, which adds one element to the list, states that one new `Node` object should be allowed:

```
adds 1 Node
```

3 Implementation

We have implemented a prototype compiler for PBnJ using the Polyglot extensible compiler framework [24]. The compiler employs *Kodkod* [27], an efficient SAT-based relational constraint-solving tool, which can in turn invoke any standard boolean satisfiability solver to find models. This section details our compilation scheme.

3.1 Translating Specifications To Java

The PBnJ compiler translates each specification method to regular Java code and creates a regular Java method for each declared method postcondition and class invariant. The translation from our specification language to Java is straightforward. For example, the transitive closure operation on a field f is implemented by a simple worklist algorithm that traverses f fields from the

```
Relation nodes_kk() {
  return This.join(List_head).join(Node_next.reflexiveClosure())
          .difference(Null); }
```

Fig. 7. Kodkod translation of the `nodes` specification method in Figure 3.

specified root object and adds each encountered object into the result set until reaching either the `null` value or an object that has already been encountered. The PBnJ compiler then instruments each public method in a class to perform dynamic contract checking. We wrap the method body in a `try` block and use the `finally` clause to invoke the Java translations of the method's postcondition and class invariant. We also use this `try` block to catch any run-time exceptions. If either contract checking fails or a run-time exception is thrown, we proceed to fall back to the Kodkod solver (described below).

3.2 Translating Specifications to Kodkod

The PBnJ compiler also creates versions of each specification method, postcondition, and class invariant for input to Kodkod. Each of these translations is placed in a new method inside the enclosing class. Kodkod is implemented as a Java library [27], so each method contains regular Java code that constructs a Kodkod-specific data structure representing the original specification formula. Dynamically these methods are invoked to build a data structure representing a Kodkod formula, which is passed to the solver for model finding.

Since our specification language is based on Kodkod's relational logic, these methods are quite simple. For example, Figure 7 shows the translation of the `nodes` specification method from Figure 3 to a method that constructs a corresponding Kodkod `Relation`. The compiler declares a unary relation corresponding to each class and a binary relation corresponding to each field. For example, Figure 7 refers to `List_head`, which is a binary relation between lists and nodes. Accessing the field value of a particular object corresponds to a join of the (singleton) relation denoting that object with the field's relation. As the example shows, we also declare singleton relations to represent `this`, `null`, and other values in scope. The `null` value is removed from the result set to properly account for the semantics of transitive closure in PBnJ. Finally, to handle specifications that refer to `old`, the compiler creates a second binary relation for each field to represent that relation's value on entry to a method for which we perform fallback.

3.3 Model Finding with Kodkod

Consider the `bubbleSort` method in Figure 3. Dynamically on entry to the method, we perform a deep copy of the receiver object, in order to be able to

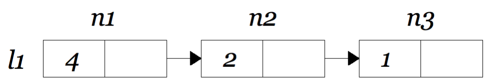

Fig. 8. An example of the reachable state from the receiver object on entry to some invocation of `bubbleSort` from Figure 3.

Table 1. A relationalized version of the program state in Figure 8.

Relation	Bound
List	{[L1]}
Node	{[N1], [N2], [N3]}
List_head_old	{[L1, N1]}
Node_value_old	{[N1, 4], [N2, 2], [N3, 1]}
Node_next_old	{[N1, N2], [N2, N3], [N3, Null]}

properly check the postcondition after the method finishes executing. We then proceed to execute the `bubbleSort` method as usual. As described earlier, the method is instrumented to perform contract checking and to fallback to Kodkod if necessary.

If fallback is required we invoke the method `bubbleSort_spec_kk`, which produces a Kodkod `Formula` representing the postcondition. We do the same for the declared class invariant and conjoin these formulas. In order to find a model for the resulting formula, we must provide Kodkod with *lower* and *upper* bounds for each relation. Upper bounds specify which values *may* appear in a relation, while lower bounds state which ones *must* appear. The bounds for the unary relations representing classes as well as the binary relations representing *old* field values are created by traversing the reachable state starting from *old* versions of the receiver and formal parameters. For example, suppose the receiver object on entry to an invocation of `bubbleSort` looks as in Figure 8. In that case, we will set both the lower and upper bounds for various relations as shown in Table 1, ensuring that Kodkod cannot change the values of these relations.

By default, the bounds on all other relations are trivial, as shown in Table 2. Each relation has an empty lower bound and an upper bound that simply indicates the type of the relation. The "function" keyword tells Kodkod that these binary relations are in fact functions from the first component to the second, which ensures that all solutions will be valid. For example, the upper bound for `Node_next` relation is requiring either a node or null value for each of the node objects shown in Table 1.

These default bounds can be narrowed by the programmer using `modifies` clauses. For example, suppose the `modifies fields` declaration for `bubbleSort` in Section 2.3 is provided by the programmer. In that case we know that the Node_value relation should be unchanged in the solution. Therefore we use the value of the Node_value_old relation from Table 1 as both the lower and upper

Table 2. Default bounds for the relations to be solved for in a fallback for `bubbleSort` from Figure 3.

Relation	Lower Bound	Upper Bound
List_head	{}	function List→(Node+Null)
Node_value	{}	function Node→Integer
Node_next	{}	function Node→(Node+Null)

bound for the Node_value relation. Further suppose a `modifies objects` clause is provided by the programmer. In that case, we execute the associated expression on the *old* receiver to obtain the set of modifiable objects. For any object o not in the result set, with relational counterpart atom O, for any pair (O, O') in some relation of the form C_f_old, we also place the pair (O, O') in the lower bound for the relation C_f. This ensures that while the relation C_f can be modified by Kodkod, the value of o's field cannot change in the solution. Finally, if the method contains an `adds` annotation indicating the number of new objects of each class that may be created, we update the associated unary relations with a corresponding number of fresh atoms.

Once the bounds are calculated, we invoke the Kodkod solver to find a model satisfying the specification formula. If a model is found, we iterate through the relations of the resulting model and use reflection to update the corresponding objects' fields with the specified values. Program execution then continues as usual.

4 Case Studies

This section describes our experience using PBnJ specifications as a reliable fallback mechanism and evaluates the expressiveness and run-time performance of this approach. First we discuss the use of executable specifications to make common data structures like lists and trees robust to implementation errors. Then we describe our experience providing a fallback mechanism for existing Java applications.

4.1 Fallback for Data Structures

Figure 3 in Section 2 showed a portion of a linked list written in PBnJ. In addition to a complete linked list, we also implemented a binary search tree as well as a red-black tree. Figure 9 shows a portion of our red-black tree. The class invariant ensures the various properties required of a red-black tree, which guarantee that the tree satisfies the usual binary search tree invariant and that the tree is balanced. The `nodes` specification method is similar to that from Figure 3, but we use the `+` operator to take the union of the `left` and `right` relations. The `isBinarySearchTree` method shows how nested quantifiers provide a declarative and powerful mechanism for expressing complex invariants.

```
public class RBTree ensures
  isBinarySearchTree() && rootBlack() && redsChildren() && eqBlacks() {

  class Node {
    Node left, right, parent;
    int value;
    boolean color;
    spec public PBJSet<Node> descendants() {
      return this.^(left+right);
    }
  }

  protected Node root;

  spec public PBJSet<Node> nodes() { return root.*(left+right); }
  spec public PBJSet<Integer> nodeValues() { return nodes().>value; }

  spec public boolean isBinarySearchTree() {
    return all Node n : nodes() |
      ((n.left == null  ||
        all Node lc : n.left.descendants() | lc.value < n.value) &&
       (n.right == null ||
        all Node rc : n.right.descendants() | rc.value > n.value));
  }

  public void insert(int value)
    modifies fields RBTree.root, Node.color, Node.left,
                    Node.right, Node.parent
    adds 1 Node
    ensures nodeValues().equals(old.nodeValues().plus(value)) { }

  public void delete(int value)
    modifies fields RBTree.root, Node.color, Node.left,
                    Node.right, Node.parent
    ensures nodeValues().equals(old.nodeValues().minus(value)) { }
}
```

Fig. 9. A portion of our red-black tree with executable specifications in PBnJ.

In order to evaluate the power and cost of our fallback mechanism, we have not provided any implementations of methods like `insert` and `delete`. Dynamically all invocations of these methods will fail to satisfy the declared postconditions, triggering a fallback to the specification. In this way, the program serves as a *self-describing, runnable interface* of a red-black tree, which can be used to ensure reliability for more efficient implementations. The code size is roughly five times smaller than a typical Java implementation of a red-black tree, mainly because of complex corner cases that imperative implementations of the insert and delete operations must handle.

The `insert` and `delete` operations include frame conditions which ensure that a node's value will remain unchanged in the face of a fallback. However, these conditions still allow the link structure of *every* node to be modified. Since the postconditions of these methods only ensure that the resulting tree has the correct values, fallback may alter the tree in a manner that differs from a typical implementation, but the resulting tree will still satisfy the red-black tree invariants and contain the proper values.

One way to preserve the structure of the original tree upon an `insert` or `delete` operation is to include a `modifies objects` clause. Doing so is fairly straightforward for a binary search tree. For example, there is always only a single node in the tree that is affected by an insertion operation. Therefore the programmer can include a clause on `insert` as follows, which invokes a function that produces the singleton set containing the affected node:

```
modifies objects getParentToBeFor(value)
```

The same thing can be done for the red-black tree, but in that case computing the set of nodes affected by an insertion or deletion is more complex due to the need to potentially rebalance the tree.

Performance We employed the data structures described above as a stress test for our fallback mechanism, using fallback to guarantee complex invariants with 100% functional recovery from an arbitrary failure. Table 3 shows the running times of a fallback event for an insertion into a binary search tree, an insertion into a red-black tree, and an invocation of `bubbleSort` for the linked list from Figure 3, for various sizes of the data structures. Without object frame conditions the Kodkod-based fallback mechanism is only practical for relatively small trees. However, when object frame conditions are provided our approach becomes feasible up to a 200-node tree. The object frame conditions keep the number of unknowns roughly the same as the problem size increases, so SAT solving time (*sat*) scales well. The main bottleneck is instead Kodkod's translation from a relational logic formula to a SAT formula (*tr*). In the future we would like to explore techniques for optimizing this encoding step.

Comparison with Data Structure Repair Techniques The Plan B approach is complementary to that of recent online data repair tools. Such tools

Table 3. Fallback pre- and post-processing overhead, including copying, contract checking, and conversion to Kodkod (*fb.*), Kodkod's translation to SAT (*tr.*) and SAT solving time (sec.) (*sat.*) using MiniSat [6] of a fallback event on an `insert` call in a binary-search tree (BST) or red-black tree (RBT) and a `bubbleSort` call on a linked list (List), with n nodes. We report timings without object frame conditions (*no frame*) and with them (*with frame*). Timeout $t/o = 600$.

	BST						**RBT**						**List**		
	insert												**bubbleSort**		
Size	**no frame**			**with frame**			**no frame**			**with frame**			**no frame**		
(n)	*fb*	*tr*	*sat*	*fb*	*tr*	*sat*	*fb*	*tr*	*sat*	*fb*	*tr*	*sat*	*fb*	*tr*	*sat*
10	.1	.58	.29	.1	.21	0	.1	.83	.60	.1	.59	0	.1	.58	.03
20	.1	1.3	37	.1	.39	0	.2	2.5	108	.1	1.1	.01	.1	1.2	.57
40	t/o			.1	.86	0	t/o			.1	1.9	.01	.1	5.5	18
60	t/o			.1	1.1	0	t/o			.1	1.8	.01	.2	18	278
80	t/o			.1	1.5	.01	t/o			.1	3.3	.05	.4	57	395
100	t/o			.1	1.7	.01	t/o			.1	3.2	.01	t/o		
200	t/o			.1	3.7	.04	t/o			.1	21	.09	t/o		

are very efficient and useful when data is broken in local ways and some data loss or corruption is acceptable. Our approach is more expensive but can recover the intended semantics of a faulty method and can properly recover from arbitrarily broken program states. To concretely illustrate these differences, we ran the Juzi repair tool [8] on our binary search tree using intentionally broken implementations.

First we modified the `insert` method of the BST implementation to corrupt a single node and asked Juzi to restore the binary search tree invariant but not the postcondition of `insert`. This kind of local repair is ideally suited for Juzi, which satisfies the binary search tree invariant in 0.1 seconds for a tree with 10 nodes. In contrast, PBnJ reverts to the state before the faulty method was invoked, so it cannot leverage the locality of the error. Aside from increasing the cost of repair, this choice means that without including the method postcondition Plan B is likely to produce a trivial solution such as an empty tree. We then augmented the tree's class invariant to include the postcondition for `insert` by manually maintaining a field denoting the original set of nodes on entry to the method. In that case Juzi timed out after a minute. On the other hand, PBnJ recovers from the corruption and additionally ensures that the insertion happens properly in a second without object frame conditions and 0.3 seconds with them.

4.2 Fallback for Existing Java Applications

We ported several existing Java applications to PBnJ, allowing us to explore the expressiveness of PBnJ's specification language as well as the practicality

```
spec protected boolean arrangeGridLayoutValid() {
  // for any given component in the window:
  return all Component c1 : components |
    (boundsValid(c1) && sizeValid(c1) && positionValid(c1) &&
     all Component c2 : components |
       (c1 == c2  ||
        (noOverlaps(c1,c2) && relPositionsValid(c1,c2)))); }
```

Fig. 10. Our specification for the `arrangeGrid` method in `GridBagLayout`.

of fallback for various kinds of constraints. In addition to the SweetHome3D application described in Section 2, we ported Java's `GridBagLayout` class and an open-source implementation of chess. Since these applications rely on the collection classes in Java's `java.util` library, we also provided PBnJ versions of many of those classes (e.g., `ArrayList`). This entailed turning some existing methods into `spec` methods (e.g., `size()`) so they could be used in clients' specifications and adding new `spec` methods as necessary. In order to support quantifying over a collection, we also implemented a `toPBJSet` specification method for each collection class, which returns a set of the collection's elements.

`GridBagLayout` The layout task in GUI applications is often complex. While individual constraints are usually simple arithmetic restrictions, laying out a window with many different components with both individual constraints and dependencies among one another is non-trivial. The `java.awt.GridBagLayout` class from Java's widely used Abstract Window Toolkit (AWT) library is a case in point[1]. This class is perhaps the most flexible of Java's layout managers, allowing components of varying sizes to be laid out subject to a variety of constraints.

We augmented several methods in `GridBagLayout` with PBnJ specifications. The main layout (and most involved) method in `GridBagLayout` is `arrangeGrid`, which is invoked whenever a user makes any change to the window (e.g., resizing) and contains over 300 lines of code. We used the informal documentation provided by Java to provide a partial specification for this method. Our specification, which is shown in Figure 10, requires that each component is located within the bounds of the window, is resized appropriately with respect to the window size, satisfies various position constraints (e.g., each row in the grid is left- and right-justified), does not overlap any other component, and retains its position relative to other components. PBnJ supports quantification over arrays (such as the `components` field in the figure) in addition to `PBJSet`s. The complete specification including helper methods is 35 lines of code.

[1] See `http://www.youtube.com/watch?v=UuLaxbFKAcc` for a funny video about the difficulties of using `GridBagLayout`.

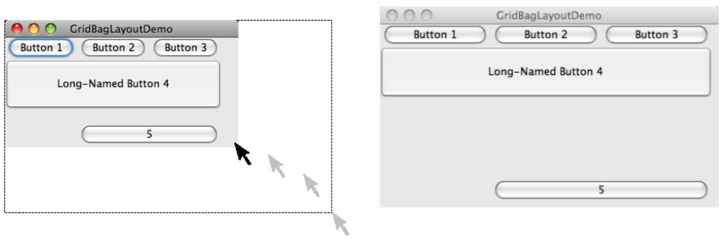

Fig. 11. Using executable specification for `arrangeGrid` (a) to layout a window originally and (b) after a resize event.

To execute its specification, we removed the original body of `arrangeGrid` so that fallback would occur on each invocation. Figure 11 shows a screenshot of the initial layout for a window with five buttons using our specification, as well as the layout after the user resizes the window. The fallback mechanism takes around 5 seconds on average in each case, and the result is indistinguishable from that of the original `arrangeGrid` implementation. Although not yet an acceptable performance overhead to use as a complete replacement for the original implementation, PBnJ provides a practical way to ensure reliability of an implementation in the face of a crash or incorrect layout.

We also experimented with a simple optimization that cuts down the time significantly with little impact on the results. The `arrangeGrid` method uses constraints that are described in terms of the entire screen's pixel coordinates (e.g., 1024x768), which constitutes a large search space. We experimented with a version of our specification in which we solve a scaled version of the problem by dividing all coordinates by a fixed constant (10 in our experiment), solve for a model, and then multiply the constant back to get the actual coordinate values to use. This approach reduced fallback time to under a second with little perceptible difference in the resulting layout. The fallback time for the Sweet-Home3D application shown in Figure 6 was also reduced to less than a second after incorporating the same optimization technique. In general, PBnJ allows programmers to provide such underspecified postconditions, which can be used to allow practical fallback at the expense of some degradation in the quality of the result.

JChessBoard JChessBoard [16] is an open-source chess implementation in Java. The application has a feature to highlight the valid moves for a piece clicked by the user. The same method is also called when generating candidate moves for the computer player. For this case study, we annotated that method, `getPossibleMoves`, with a PBnJ postcondition that itself computes the set of valid moves and compares the result to the result of `getPossibleMoves`'s implementation. This case study demonstrates the ability of our specification language to express complex properties and perform sophisticated computations.

```
spec PBJSet<Move> allValidMovesFrom(Piece p, int from) {
  return { all Move move : allMoves.toPBJSet() |
             (move.getFrom() == from
              && isValidMove(p, from, move.getTo())) };
}
spec boolean isValidMove(Piece p, int from, int to) {
  if (p == BISHOP)
      return isValidBishopMove(from, to);
  else if ...
}
```

Fig. 12. Computing valid chess moves as a PBnJ specification.

Figure 12 shows the specification method that computes the valid moves for a given piece on the board. JChessBoard uses a single-dimensional array `board` of size 64 to represent the board, and a given square (x, y) is indexed using the formula $8x + y$, where x and y are in the range [0,7]. In order to generate the set of valid moves, JChessBoard iterates over a statically calculated Java `Vector` named `allMoves` — the collection of all moves that can possibly originate from each square, assuming the square could hold any possible piece. Our specification quantifies over this vector in order to obtain only moves for the given square `from` that are valid for the given piece `p`.

Figure 13 shows the specification method `isValidBishopMove`. The method checks that the `from` and `to` squares are on the same diagonal and that all intervening squares on that diagonal are empty. This specification makes use of many of the features of our specification language, including assignments to local variables, nested quantification, integer interval ranges for quantification, array accesses, and bitwise operations on integers. Except for the quantification expressions, the rest of the code comes from the original JChessBoard implementation; we simply added `spec` annotations. Nonetheless, this code can be automatically translated to Kodkod for the purposes of automatic fallback.

A fallback event for `getPossibleMoves`, which entails "executing" the above method `allValidMovesFrom` in Kodkod, takes 2-3 seconds on average. We have also explored an optimization which replaces the `allMoves` field with a separate statically computed vector per type of piece (`allBishopMoves`, `allQueenMoves`, etc.). Using the appropriate vector for the given piece during a fallback event rather than the generic `allMoves` vector reduces the execution time to about 0.5 seconds.

5 Related Work

Our work builds upon several lines of research on executable specifications and software reliability.

```
spec boolean isValidBishopMove(int from, int to) {
  int fromRow = getRow(from), fromColumn = getColumn(from);
  int toRow = getRow(to), toColumn = getColumn(to);

  return Math.abs(toRow - fromRow) == Math.abs(toColumn - fromColumn)
     && checkDiagonalLineOfSight(fromRow, fromColumn, toRow, toColumn);
}
spec boolean checkDiagonalLineOfSight
   (int fromRow, int fromColumn, int toRow, int toColumn) {
  int minRow = Math.min(fromRow, toRow);
  int maxRow = Math.max(fromRow, toRow);
  int minColumn = Math.min(fromColumn, toColumn);
  int maxColumn = Math.max(fromColumn, toColumn);
  return all int r : minRow + 1 .. maxRow - 1 |
           all int f : minColumn + 1 .. maxColumn - 1 |
             (Math.abs(r - fromRow) != Math.abs(f - fromColumn)  ||
              board[getSquare(r,f)] == EMPTY);
}
spec int getRow(int square) { return square >> 3; }
spec int getColumn(int square) { return square & 7; }
spec int getSquare(int row, int column) { return (row << 3) + column; }
```

Fig. 13. Specifying the valid bishop moves from a square.

5.1 Executing Specifications via Constraint Solving

The idea to execute specifications has been explored in a variety of contexts. Two recent examples include work on executable specifications for C++ [29] as well as for the JML modeling language for Java [20]. These works allow specifications to be executed on their own, for the purpose of gaining confidence in their correctness. Our work also executes specifications but in a manner that is tightly integrated with the host programming language. Specifications in PBnJ are directly executable as part of a Java program's execution, and our notion of fallback requires constraint solving to happen online and in collaboration with ordinary program execution.

The idea of a *mixed interpreter*, which can execute programs that consist of both specifications and implementations, is more closely related to our work. Morgan laid the formal foundations for this approach with his notion of a *specification statement* [23]. However, his goal was not to automate the execution of specifications but rather to support program reasoning uniformly during the process of manually *refining* specifications to implementations. Freeman-Benson and Borning introduced the notion of *constraint imperative* programming as embodied in their Kaleidoscope language [12], which can be viewed as an instantiation of the idea of a mixed interpreter. This language allows a class to declare constraints that are automatically enforced on instances of the class using a constraint solver that integrates decision procedures for several domains. Rayside *et al.* have recently described a vision of "agile specifications," which employs the notion of a mixed interpreter to unify the benefits of formal methods with those of the agile software development methodology [25].

Plan B can be seen as a special case of a mixed interpreter, where constraint solving is used only as a fallback mechanism rather than as a "first-class" part of the language. We believe that fallback is a compelling use of executable specifications that may be more practical to support than a full-fledged mixed interpreter. For example, it is reasonable for PBnJ to employ bounded constraint solving. Although this may result in missed opportunities to recover a program, in the worst case we can simply throw an exception as traditional contract checking would do. In contrast, it would be unreasonable for a general mixed interpreter to sometimes fail to execute a specification that is used as part of a program's implementation.

5.2 Data Structure Repair

Our work is inspired in part by recent work on online repair of data structures. As mentioned in earlier comparisons, we view the approaches as useful in different scenarios. Repair is useful when it is important for an application to continue executing despite the presence of errors, while Plan B is useful when it is important for an application to achieve the intended functionality of a faulty method before continuing.

The repair approach of Demsky and Rinard [2, 3] allows programmers to provide high-level specifications in terms of an abstract model of objects, along with a translation from the concrete to the abstract worlds. These specifications are checked dynamically and violations invoke a repair algorithm that employs a specialized set of repair actions (e.g., add or remove an element from a set). The authors provide several case studies illustrating the practicality of the approach for surviving errors in a variety of existing applications. The use of abstract models allows for a high-level approach to repair but also places additional burdens on programmers. It may be useful for us to consider incorporating user-defined abstractions in PBnJ, which could allow for higher-level encodings into Kodkod that are more efficient than our current concrete encoding of a program state.

Elkarablieh *et al.* describe *assertion-based* repair of data structures [7, 9, 8]. Their approach takes the broken program state along with a Java method representing the violated assertion predicate and performs a heuristically guided and bounded state-space exploration to search for a nearby state that satisfies the predicate. This approach is in some ways analogous to our use of SAT-solving technology to perform a bounded search. However, our search begins from the *pre-state* of the faulty method and strives to achieve the intended functionality of the method, while their search begins from the *post-state* of the faulty method and strives to perform local repairs to satisfy the class's integrity constraints. Our experiments in Section 4 compared against the Juzi tool directly. These repair tools employ symbolic execution [19] along with automatic decision procedures to solve for the values of integers and other primitives and employ static analysis to optimize the repair. Both ideas would be useful in our context as well in order to speed up fallback.

5.3 Alloy

Our use of a specification language adapted from Alloy [14] and of Alloy's underlying solver Kodkod [27] are no accident. Alloy's relational style of modeling programs has proven to be quite natural and powerful, and Alloy and Kodkod have been used in a variety of ways to gain confidence in the correctness of imperative programs. One line of work employs these tools for *bounded verification* of implementations [15, 18, 28, 4, 5]. In this approach, both the body of a Java method as well as its specification are translated to Alloy/Kodkod, and a constraint solver searches for executions (up to some bound on the length of an execution trace and the size of the heap) of the method that violate its specification. The Analyzable Annotation Language (AAL) [18] additionally employs Alloy to reason about the specifications themselves, for example to ensure that the specification of an `equals` method in Java is in fact an equivalence relation. Finally, TestEra [17] uses Alloy as a test generation tool for Java programs. Alloy generates nonisomorphic inputs up to some bound that satisfy a method's precondition, and Alloy is also used as the test oracle to determine whether the result of executing the test satisfies the method's postcondition.

Our work borrows the basic relational approach to modeling objects from these prior works, along with the approach to translating a program state into relations for input to Kodkod. However, rather than using this technology to gain confidence in an implementation, our approach ignores the implementation and instead employs Kodkod to "execute" Alloy-style specifications. Our approach avoids some complications of verification, for example the need to translate arbitrary Java code to relational logic. On the other hand, our use of online constraint solving poses a performance challenge, which we have addressed in part through novel program annotations (e.g., `modifies objects`).

6 Discussion and Future Work

The approach proposed in this paper is just a first step, and much more needs to be done to make Plan B an expressive and practical technique for software reliability. There are several avenues for future work. First, the PBnJ language can be improved in a few important ways. The specification language lacks support for floats and strings and associated operations. There is also no special support for class inheritance currently. For example, the specification of a method is not automatically inherited by an overriding method and need not have any relationship to the specification of that method. Finally, specification methods are currently forbidden from being recursive, which limits their expressiveness. Others have shown how to translate recursive definitions into Alloy [18], so it would be natural for us to adopt their approach.

Second, the PBnJ implementation can be optimized to reduce the overhead of fallback. We plan to explore ways to avoid the deep copying that we currently perform due to uses of the `old` field. One possibility is to use static analysis to determine the parts of an object's state that cannot change and therefore need not be copied. Another approach is to use a copy-on-write strategy, where state is only copied just before it is overwritten. Finally, with more experience we may find that an alternative semantics for `old` is more practical while still providing the desired expressiveness.

Third, we are interested in exploring other uses of online constraint solving for software robustness. One idea is to pursue the use of executable specifications to declaratively specify *background* tasks that can be performed asynchronously with the main program. For example, one could imagine refactoring a red-black tree implementation to perform rebalancing in the background or during idle periods, analogous to the way in which garbage collection runs as a background thread. This style could provide significant benefits while alleviating some of the current performance issues with online constraint solving. Another natural question is whether the results of automatic fallback can be used to help developers localize and correct errors in their implementations. Lastly, we are interested to explore the online use of other kinds of constraint solvers for software reliability, for example solvers that can maximize a user-provided objective function and those that can employ local-search heuristics.

7 Conclusion

We have presented the Plan B approach to software reliability. Our main contribution is the notion that formal specifications, when made *executable* by means of a *SAT-based constraint solver*, can act as *reliable alternatives* for incomplete or faulty method implementations. As a secondary benefit, such specifications can also be used directly to make implementations more declarative and reliable by construction. We have demonstrated both use cases via example in the PBnJ extension to Java and presented our experience using the language to guarantee rich properties on existing Java applications. We are excited about the possibilities of leveraging modern constraint solving technology as an online tool for software reliability. While many challenges remain, we are encouraged by our initial results and believe that there are several fruitful avenues for future research.

8 Acknowledgments

Thanks to Ted Kaehler for discussions about this work and to Dan Marino, Alex Warth, and the anonymous reviewers for valuable feedback on this paper. This work is supported by the National Science Foundation under award CCF-0545850 and by the Defense Advanced Research Projects Agency under award HR0011-09-1-0037.

References

1. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: an overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005.
2. B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 78–95, New York, NY, USA, 2003. ACM.
3. B. Demsky and M. C. Rinard. Data structure repair using goal-directed reasoning. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *ICSE*, pages 176–185. ACM, 2005.
4. G. Dennis, F. S.-H. Chang, and D. Jackson. Modular verification of code with sat. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 109–120, New York, NY, USA, 2006. ACM.
5. G. Dennis, K. Yessenov, and D. Jackson. Bounded verification of voting software. In *VSTTE '08: Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, pages 130–145, Berlin, Heidelberg, 2008. Springer-Verlag.
6. N. Een and N. Sorensson. MiniSat. http://minisat.se.
7. B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based repair of complex data structures. In R. E. K. Stirewalt, A. Egyed, and B. F. 0002, editors, *ASE*, pages 64–73. ACM, 2007.
8. B. Elkarablieh and S. Khurshid. Juzi: a tool for repairing complex data structures. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 855–858, New York, NY, USA, 2008. ACM.
9. B. Elkarablieh, S. Khurshid, D. Vu, and K. S. McKinley. Starc: static analysis for efficient repair of complex data. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 387–404, New York, NY, USA, 2007. ACM.
10. R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–15, New York, NY, USA, 2001. ACM.
11. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.
12. B. N. Freeman-Benson and A. Borning. Integrating constraints with an object-oriented language. In O. L. Madsen, editor, *ECOOP*, volume 615 of *Lecture Notes in Computer Science*, pages 268–286. Springer, 1992.
13. J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, Third Edition*. Addison-Wesley Professional, 2005.
14. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
15. D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 14–25, New York, NY, USA, 2000. ACM.

16. JChessBoard. http://jchessboard.sourceforge.net.
17. S. Khurshid and D. Marinov. Testera: Specification-based testing of java programs using sat. *Autom. Softw. Eng.*, 11(4):403–434, 2004.
18. S. Khurshid, D. Marinov, and D. Jackson. An analyzable annotation language. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 231–245, New York, NY, USA, 2002. ACM.
19. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7), 1976.
20. B. Krause and T. Wahls. jmle: A tool for executing jml specifications via constraint programming. In *FMICS/PDMC*, volume 4346 of *Lecture Notes in Computer Science*, pages 293–296. Springer, 2006.
21. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.
22. B. Meyer. Design by contract: Making object-oriented programs that work. In *TOOLS (25)*, page 360. IEEE Computer Society, 1997.
23. C. Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, 1988.
24. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *In 12th International Conference on Compiler Construction*, pages 138–152. Springer-Verlag, 2003.
25. D. Rayside, A. Milicevic, K. Yessenov, G. Dennis, and D. Jackson. Agile specifications. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 999–1006, New York, NY, USA, 2009. ACM.
26. SweetHome3D. http://www.sweethome3d.eu.
27. E. Torlak. *A constraint solver for software engineering: Finding models and cores of large relational specifications.* Ph.D. dissertation, Massachusetts Institute of Technology, 2009.
28. M. Vaziri and D. Jackson. Checking properties of heap-manipulating procedures with a constraint solver. In H. Garavel and J. Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 505–520. Springer, 2003.
29. T. Wahls, G. T. Leavens, and A. L. Baker. Executing formal specifications with concurrent constraint programming. *Automated Software Engg.*, 7(4):315–343, 2000.
30. K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 349–361. ACM, 2008.
31. K. Zee, V. Kuncak, and M. C. Rinard. An integrated proof language for imperative programs. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 338–351. ACM, 2009.