Full Functional Verification of Linked Data Structures

Karen Zee
MIT CSAIL, Cambridge, MA, USA
kkz@csail.mit.edu

Viktor Kuncak
EPFL, I&C, Lausanne, Switzerland
viktor.kuncak@epfl.ch

Martin C. Rinard
MIT CSAIL, Cambridge, MA, USA *
rinard@csail.mit.edu

Abstract

We present the first verification of *full functional correctness* for a range of linked data structure implementations, including mutable lists, trees, graphs, and hash tables. Specifically, we present the use of the Jahob verification system to verify formal specifications, written in classical higher-order logic, that completely capture the desired behavior of the Java data structure implementations (with the exception of properties involving execution time and/or memory consumption). Given that the desired correctness properties include intractable constructs such as quantifiers, transitive closure, and lambda abstraction, it is a challenge to successfully prove the generated verification conditions.

Our Jahob verification system uses *integrated reasoning* to split each verification condition into a conjunction of simpler subformulas, then apply a diverse collection of specialized decision procedures, first-order theorem provers, and, in the worst case, interactive theorem provers to prove each subformula. Techniques such as replacing complex subformulas with stronger but simpler alternatives, exploiting structure inherently present in the verification conditions, and, when necessary, inserting verified lemmas and proof hints into the imperative source code make it possible to seamlessly integrate all of the specialized decision procedures and theorem provers into a single powerful integrated reasoning system. By appropriately applying multiple proof techniques to discharge different subformulas, this reasoning system can effectively prove the complex and challenging verification conditions that arise in this context.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, Languages, Reliability, Verification

Keywords verification, Java, data structure, theorem prover, decision procedure

* This research was supported in part by the Singapore-MIT Alliance, DARPA Cooperative Agreement FA 8750-04-2-0254, NSF Grant CCR-0086154, NSF Grant CCR-0341620, NSF Grant CCF-0209075, and NSF Grant CCR-0325283.

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.

1. Introduction

Linked data structures such as lists, trees, graphs, and hash tables are pervasive in modern software systems. But because of phenomena such as aliasing and indirection, it has been a challenge to develop automated reasoning systems that are capable of proving important correctness properties of such data structures.

1.1 Background

In principle, standard specification and verification approaches should work for linked data structure implementations. But in practice, many of the desired correctness properties involve logical constructs such transitive closure and quantifiers that are known to be intractable for automated reasoning systems [36, 45]. Researchers have therefore focused on more tractable goals: verify some (but not all) of the desired correctness properties [6, 18, 28, 42, 47, 49, 50, 74, 88, 89], work with programs that do not manipulate recursive linked data structures [29, 81], or use finitization to check correctness properties within a bounded analysis scope [15, 20, 38, 70, 77, 78]. While systems exist that can specify, and in principle even potentially verify, the full range of desired data structure correctness properties [3, 7, 53, 82], to the best of our knowledge no previous system has actually done so (see Section 8).

1.2 The Result

This paper presents our experience using *integrated reasoning* in the Jahob verification system to verify a diverse set of challenging linked data structure implementations. Our proofs establish the *full functional correctness* of the data structure implementations — Jahob verifies formal specifications that completely capture the desired behavior of the data structure implementations (with the exception of properties involving execution time and/or memory consumption). The source code for the specifications, the data structures, and the Jahob system itself are all publicly available at `http://javaverification.org` [1]. To the best of our knowledge, this is the first verification of full functional correctness for our target class of linked data structures.

1.3 Basic Specification Approach

Our specifications use abstract sets and relations to characterize the abstract state of the data structure. A verified abstraction function establishes the correspondence between the concrete values that the implementation manipulates when it executes and the abstract sets and relations in the specification. Method preconditions and postconditions written in classical higher-order logic use these abstract sets and relations to express externally observable properties of the data structures. We find classical higher-order logic to be effective for specifying data structures because it naturally supports a number of constructs:

- quantifiers for invariants in programs that manipulate an unbounded number of objects,

- a notation for sets and relations, which we use to concisely specify data structure interfaces,
- transitive closure, which is essential for specifying important properties of recursive data structures,
- the cardinality operator, which is suitable for specifying numerical properties of data structures, and
- lambda abstraction, which can represent definitions of per-object specification fields and is useful for parameterized shorthands.

Sets and relations as specification variables enable developers to soundly hide data structure implementation details and provide intuitive method interfaces. Clients can use such interfaces to check that the data structure is used correctly and to reason about the effect of data structure operations.

1.4 Basic Verification Approach

Jahob proves the desired correctness properties by first generating verification condition formulas, then proving these formulas. The verification conditions are proof obligations that, together, ensure that the program respects method preconditions, postconditions, invariants, and preconditions of operations such as array accesses and pointer dereferences. The verification condition generator requires loop invariants. These can be supplied by the developer or, in some cases, by a shape analysis [88].

The verification condition formulas are expressed in an undecidable fragment of higher-order logic and are therefore beyond the reach of any automated decision procedure. Simple attempts to improve the tractability by limiting the expressive power of the logic fail because some of the correctness properties involve inherently intractable constructs such as quantifiers, transitive closure, and lambda abstraction.

1.5 Technical Insights

Upon examination, however, it becomes clear that while the verification conditions as a whole can be quite complex, they can also be represented as a conjunction of a large number of smaller subformulas, many of which are straightforward to prove. Moreover, the remaining subformulas, while containing a diverse group of powerful logical constructs, often have enough structure to enable the successful application of specialized decision procedures or theorem provers. Specifically, some subformulas can be proved with sufficient quantifier instantiations, congruence closure algorithms, and linear arithmetic solvers; precise reasoning about reachability is sufficient to discharge others; still others require complex quantifier reasoning but do not require arithmetic reasoning. In the worst case, it is always possible to use interactive theorem provers to discharge the remaining few complex subformulas.

Armed with this insight, we developed an integrated reasoning approach that enables the simultaneous application of a diverse group of interoperating reasoning systems to prove each verification condition. This approach is based on the following techniques:

- **Splitting:** Jahob splits verification conditions into equivalent conjunctions of subformulas and processes each subformula independently. It can therefore use different provers to establish different parts of proof obligations. Because it treats each prover as a black box, it is easy to incorporate new provers into the system. Moreover, each prover can run on a separate processor core, reducing the running time on modern workstations.
- **Formula Approximation:** Jahob uses a variety of new and existing external decision procedures, Nelson-Oppen provers, and first-order theorem provers, each with its own restrictions on the set of formulas that it will accept as input. Several formula approximation techniques make it possible to successfully deploy this diverse set of reasoning systems together within a single unified reasoning framework. These approximation techniques accept higher-order logic formulas and create equivalent or semantically stronger formulas accepted by the specialized decision procedures and provers.

  Our approximation techniques rewrite equalities over complex types such as functions, apply beta reduction, and express set operations using first-order quantification. They also soundly approximate constructs not directly supported by a given specialized reasoning system, typically by replacing problematic constructs with logically stronger and simpler approximations.

  Decision procedures such as MONA [30] perform reasoning under the assumption that the models of given formulas are trees. The Jahob interfaces to such decision procedures recognize subformulas that express the relevant structure (such as treeness or transitive closure). They then expose this structure to the decision procedure by applying techniques such as field constraint analysis [87] and encoding transitive closure using second-order quantifiers.

  Together, these techniques make it possible to productively apply arbitrary collections of specialized reasoning systems to complex higher-order logic formulas. Our implemented Jahob system, for example, contains a simple syntactic prover, interfaces to first-order provers (SPASS [84] and E [76]), an interface to SMT provers (CVC3 [26] and Z3 [19]), an interface to MONA [67], an interface to the BAPA decision procedure [44, 46], and interfaces to interactive theorem provers (Isabelle [63] and Coq [11]).
- **Proof Decomposition:** Jahob allows the developer to insert program-point-specific lemmas and proof hints into the imperative source code. Jahob proves these lemmas (using the full range of its reasoning techniques) then uses the lemmas as additional assumptions in verification conditions. The developer can also use interactive theorem provers to prove parts of verification conditions. These techniques enable the developer to guide the proof of any verification conditions that are beyond the reach of the fully automated techniques.

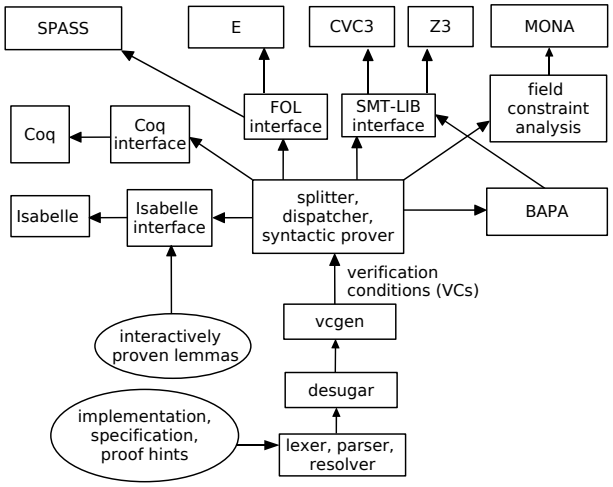

Figure 1. Integrated Reasoning in the Jahob System

Figure 1 graphically presents the resulting integrated reasoning architecture of the Jahob system. In practice, the syntactic prover quickly disposes of many of the conjuncts in each verification condition. A complex core of subformulas makes it through to the more powerful automated reasoning systems. Each of these reasoning systems proves the subset of subformulas for which it

is applicable; together, they prove the majority of the remaining conjuncts. When the automation does not succeed (typically due to conjuncts that contain large numbers of universally quantified assumptions), we manually guide the proof process by inserting verified lemmas and proof hints into the source code. Finally, we use interactive theorem provers to prove verification conditions that require complex inductive reasoning or nonlinear arithmetic, relying on a body of previously proved lemmas and the ability to fully control the proof process.

1.6 Consequences of Our Result

Full functional specifications decouple data structure interfaces (as expressed in formal logic) from implementations. The verification of these specifications fulfills, for the first time, the previously unrealized ideal of abstract data types and modular reasoning for data structures. We identify several ways in which this research may influence future efforts.

- **Verified Data Structure Libraries:** In retrospect, it is clear that data structures are a natural candidate for full functional verification. Because data structures have been studied for decades, there is a broad consensus on how they should behave. It is therefore relatively straightforward to develop the required complete formal specifications. Because the specification and verification effort can be profitably amortized over many data structure uses, data structures can support focused verification efforts that would be impractical for single systems.

  In the future, developers of data structure libraries may choose to deliver formally specified and fully verified implementations. Advantages of this development could include the elimination of ambiguity from data structure interfaces and increased confidence in the correctness of the implementation.

- **Integrated Reasoning Systems:** In recent years researchers have developed a range of decision procedures, theorem provers, and other reasoning tools [11, 16, 19, 26, 27, 30, 46, 47, 55, 63, 65, 76, 84]. Techniques that enable these reasoning tools to seamlessly interoperate within a unified reasoning framework (such as Nelson-Oppen combination [60] and our formula approximation) greatly increase the value of each individual tool. One potential result is a proliferation of specialized reasoning tools, a corresponding increase in the combined capabilities of automated reasoning systems in general, a move away from monolithic general-purpose reasoning systems, and an overall improvement in our ability to effectively reason about complex computer systems.

- **New Program Analysis Techniques:** Pointers and indirection, especially in the context of recursive data structures, are the bane of program analysis. Their presence often complicates, and sometimes even invalidates, many otherwise straightforward program analysis approaches.

  The field has responded by developing a variety of pointer and shape analyses [18, 28, 41, 50, 72–75, 83, 86]. These analyses reason directly about pointers across the entire analyzed range of the program. Despite intensive research efforts and many impressive technical results, scalability and/or precision issues continue to limit the utility of even the most advanced analyses.

  Upon reflection, it becomes clear that standard shape analysis approaches are at variance with modern software engineering practices. Decades ago all programmers used to reason directly about the pointers in their data structures. But the concept of abstract data types enabled a productive separation of reasoning concerns. With abstract data types, only the implementor of the data structure reasons directly about the pointers in the data structure implementation. Implementors of code that uses the data structure reason more productively at the more abstract level of the data structure interface.

  Until now, the informal and unverified nature of typical data structure interfaces has prevented sound automated reasoning systems from exploiting this kind of structure. But the availability of fully verified data structure implementations eliminates this problem, paving the way for new analysis approaches that use verified data structure specifications to reason soundly about the program at the higher level of abstract sets of objects and relations. We expect the resulting analyses 1) to be substantially more scalable than existing analyses and 2) to extract significantly more useful information. In particular, we have used this approach successfully in the Hob system to obtain analyses that deliver an unprecedented combination of scalability and precision [42, 49].

- **Commuting Operations:** If all operations in a computation commute, it is possible to generate code that executes the computation in parallel [69]. Applying this principle to computations that manipulate linked data structures can be challenging because commuting operations on linked data structures often produce different but semantically equivalent data structure states. Verified specifications that use sets and relations to soundly summarize the abstract state of the data structure can often eliminate this problem — different but semantically equivalent concrete data structure states (such as two lists that store the same set of objects) often have identical abstract states. If the abstract states also differ, it is often possible to use the specification and verification system to precisely state the desired equivalence condition, then prove that operations commute relative to this condition. The correctness proofs also ensure that properly synchronized parallel computations correctly preserve the data structure invariants (regardless of whether or not the operations commute). The availability of verified specifications may therefore substantially extend our ability to reason successfully about parallel computations that manipulate linked data structures.

1.7 Limitations

We identify several limitations of our verification system. First, we assume that each data structure operation executes atomically. For this assumption to hold in concurrent settings, some form of synchronization would be required. Our current system also does not support dynamic class loading, exceptions, or dynamic dispatch. Techniques exist, however, that should make it possible to extend our modular verification approach to support such constructs [8, 17, 32]. Two limitations could be eliminated by minor extensions. We currently model numbers as algebraic quantities with unbounded precision and assume that object allocation always successfully produces a new object. While these assumptions are often used in the verification field and are typically consistent with the execution of the program, they are at variance with the full semantics of the underlying programming language. Finally, we make no attempt to verify any property related to the running time or the memory consumption of the data structure implementation. In particular, we do not attempt to verify the absence of infinite loops or memory leaks.

2. Example

In this section we use a verified association list to demonstrate how developers use Jahob to specify data structure implementations. We also present a sized list example, which illustrates the coordinated application of multiple theorem provers and decision procedures to verify the specification of a single method.

2.1 Association List

Figure 2 presents selected portions of the AssocList class. This class maintains a list of key-value pairs. When presented with a given key it returns the corresponding value for that key. Jahob works with Java programs augmented with specifications. The specifications appear as special comments of the form /*: ... */ or //: ... , enabling the use of standard Java compilers and virtual machines. The first comment in Figure 2 identifies the abstract state content of the association list as a relation in the form of a set of pairs of objects. [1]

Method interfaces. The put(k0,v0) method inserts the pair (k0,v0) into the association list, returning the previous association for k0 (if such an association existed). The requires clause indicates that it is the client's responsibility to ensure that neither k0 nor v0 is null. The modifies clause indicates that the method observably changes nothing except the abstract state content of the association list. The ensures clause of the specification states that the abstract state content after the method executes is the abstract state old content from before the method executes augmented with the new association (k0,v0). Any previous association (k0,result) is removed from the association list, with result returned as the result of the put method. It returns null if no such previous association existed.

The get(k0) method returns the value v associated with k0 if such an association exists. Otherwise it returns null.

Concrete and abstract state. Figure 3 presents the definition of the Node class, which contains the key, value, and next fields that implement the linked list of key-value pairs in the association list. The assertion claimedby AssocList specifies that only the methods in the AssocList class can access these fields. Jahob enforces this specification by a simple syntactic check.

The Node class also has a specification variable cnt, whose purpose is to define the abstract state content of the association list. There is one cnt variable for each Node object. Specification variables exist only to support the specification and verification and do not exist when the program runs.

Figure 4 presents the CntDef and CntNull invariants, which together define the value of the cnt specification variable for each Node object x. The CntDef invariant recursively defines the value of cnt for an object x as the set of pairs stored in the part of the association list reachable from x by following next fields. The CntNull invariant defines the base case of the recursion: cnt is empty for the null object. Because cnt is a "ghost" specification variable, the implementation uses specification assignments to explicitly update cnt when changing the next, key, and/or value fields.

The specification variable content contains the set of key/value pairs that comprise the abstract state of the association list. The definition of content appears after the **vardefs** keyword and specifies that content is the value of cnt for the first node in the list. In contrast to cnt (which is a ghost specification variable), content is a defined specification variable: when one or more of the variables in the definition of content change, Jahob computes the corresponding changes to content automatically.

Figure 4 also illustrates how private defined specification variables such as edge can serve as useful shorthands. The edge variable denotes either the first or the next field of the corresponding object, making the InjInv invariant easier to write.

Semantic domain. The syntax of Jahob invariants reflects the underlying semantic domain in which the verification takes place. The domain (denoted obj) contains the infinite set of all objects that the program could use during any execution. Classes correspond to sets of objects within this domain. The notation x ∈ Node states that the object x is an element of the Node class). Fields correspond to functions from objects to values. The expression x..next denotes the application of the next function to the x object. It is often convenient for these functions to be total (i.e., always defined for every object) — if the object is not a member of a given class, the values of all of the fields from that class are simply null.

Verified method implementations. Figure 5 presents the implementation of the get(k0) method. This method searches the list to find the Node containing the key k0, then returns the corresponding value v (or null if no such value exists). The loop invariant states

[1] Our examples use mathematical notation for concepts such as set union (∪) and universal quantification (∀). Developers can use the ProofGeneral editor mode to view these symbols in either ASCII or mathematical notation [5]

```
class AssocList {
  //: public specvar content :: "(obj * obj) set"
  public Object put(Object k0, Object v0)
  /*: requires "k0 ≠ null ∧ v0 ≠ null"
      modifies content
      ensures
      "content = old content − {(k0, result)} ∪ {(k0, v0)} ∧
       (result = null → ¬ (∃ v. (k0, v) ∈ old content)) ∧
       (result ≠ null → (k0, result) ∈ old content)" */
  {...}
  public Object get(Object k0)
  /*: requires "k0 ≠ null"
      ensures "(result = null → ¬ (∃ v. (k0, v) ∈ content)) ∧
               (result ≠ null → (k0, result) ∈ content)" */
  {...}
}
```

Figure 2. Association List Operations

```
public /*: claimedby AssocList */ class Node {
    public Object key; public Object value; public Node next;
    //: public ghost specvar cnt :: "(obj * obj) set" = "{}"
}
```

Figure 3. Node Definition

```
private Node first;
vardefs "content == first..cnt";
invariant CntDef:
    "∀ x. x ∈ Node ∧ x ∈ alloc ∧ x ≠ null →
          x..cnt = {(x..key, x..value)} ∪ x..next..cnt ∧
          (∀ v. (x..key, v) ∉ x..next..cnt)";
invariant CntNull:
    "∀ x. x ∈ Node ∧ x ∈ alloc ∧ x = null → x..cnt = {}";
private static specvar edge :: "obj ⇒ obj ⇒ bool";
vardefs "edge == (λ x y. (x ∈ Node ∧ y = x..next) ∨
                         (x ∈ AssocList ∧ y = x..first))";
invariant InjInv:
    "∀ x1 x2 y. y ≠ null ∧ edge x1 y ∧ edge x2 y → x1=x2";
```

Figure 4. Abstraction Function and Invariants in AssocList

```
public Object get(Object k0)
/*: requires "k0 ≠ null"
    ensures "(result ≠ null → (k0, result) ∈ content) ∧
             (result = null → ¬(∃ v. (k0, v) ∈ content))" */
{
  Node current = first;
  while//: inv"∀v. ((k0,v) ∈content) = ((k0,v) ∈current..cnt)"
    (current != null) {
    if (current.key == k0) { return current.value; }
    current = current.next;
  }
  return null;
}
```

Figure 5. Implementation of the get method

```
class List {
  private List next;
  private Object data;
  private static List root;
  private static int size;
/*: private static ghost specvar nodes :: objset = "{}";
 public static ghost specvar content :: objset = "{}";
 invariant nodesDef:
   "nodes = {n. n ≠ null ∧ (root,n) ∈ {(u,v). u..next=v}*}";
 invariant contentDef:"content= {x. ∃n. x=n..data ∧ n∈nodes}";
 invariant sizeInv: "size = cardinality content";
 invariant treeInv: "tree [List.next]";
... */
  public static void addNew(Object x)
  /*: requires "comment ''xFresh'' (x ∉ content)"
      modifies content
      ensures "content = old content ∪ {x}" */
  {
    List n1 = new List();
    n1.next = root; n1.data = x;
    root = n1; size = size + 1;
    /*: nodes := "{n1} ∪ nodes";
        content := "{x} ∪ content";
        note "theinv sizeInv" by sizeInv, xFresh */
  }
}
```

Figure 6. Sized List

```
$jahob List.java -method List.add -usedp spass mona bapa
...
========================================================
Built-in checker proved 2 sequents during splitting.
SPASS proved 4 out of 8 sequents. Total time : 0.2 s
MONA proved 3 out of 4 sequents. Total time : 0.2 s
BAPA proved 1 out of 1 sequents. Total time : 0.0 s
========================================================
A total of 10 sequents out of 10 proved.
:List.add]
0=== Verification SUCCEEDED.
```

Figure 7. Command line and Jahob output in example that combines multiple decision procedures to verify one method

that the pair (k0, v) is in the association list if and only if it is in the part of the list remaining to be searched—in effect, that the search does not skip the Node with key k0. Given the specification and the invariants, Jahob is capable of verifying that this method both correctly implements its specification and correctly preserves the invariants.

In addition to the get method, the association list contains other methods that check membership of keys in the association list, add associations to the list, and remove associations from the list. The basic concepts are the same: statically verified full functional specifications, loop invariants where appropriate, and explicit updates of the appropriate specification variables.

2.2 Sized List

Figure 6 presents a simple example [40, Chapter 2] whose verification requires the combination of three Jahob provers. The add operation inserts a new element into the linked list. The invariants of the linked list maintain the set of linked list nodes, the set of objects stored in these nodes, and the size of the list. The sizeInv invariant requires the size of the data structure to be equal to the number of objects reachable from the root. Many natural logical fragments in which this invariant is expressible (such as monadic second-order logic with equicardinality constraints) are undecidable. However, thanks to the use of the nodes and content specification variables, the generated verification conditions can be split into a conjunction of implications, each of which is provable using one of the following three provers: 1) MONA [30], 2) the SPASS first-order prover [84], and 3) the BAPA decision procedure for sets with cardinality bounds [43, 46]. Our formula approximation technique eliminates assumptions not meaningful for a given prover; the values of the specification variables ensure that the resulting formula contains enough information to be provable. Thanks to such mechanisms, Jahob users can use specification variables and assertions to simulate the effect of Nelson-Oppen combination for complex logics to which the Nelson-Oppen procedure traditionally has not been applied [40, Section 4.4.5].

Figure 7 presents the Jahob verification report for the add method. This report indicates how many sequents (implications that represent parts of verification conditions) were proved by each theorem prover or decision procedure. The command line instructs Jahob to use SPASS, MONA, and the BAPA decision procedure (in that order) when attempting to discharge the proof obligations that arise during the verification.

3. Jahob Specification Constructs

Developers specify Jahob programs using specification variable declarations, method contracts, class invariants, and annotations within method bodies.

3.1 Jahob Formulas

Many Jahob specification constructs contain formulas. The syntax and semantics of Jahob formulas follow Isabelle/HOL [63]. Formulas are simply typed with ground types bool for boolean values, int for integers, and obj for objects, as well as type constructors $\Rightarrow$ for total functions, $*$ for tuples, and set for sets. The logic contains polymorphic equality, standard logical connectives $\wedge, \vee, \neg, \rightarrow, \forall, \exists$, as well as the λ binder, set comprehension $\{e.F\}$, and standard operations on sets and integers. It supports selected defined operations, most notably $(u, v) \in \{(x, y).G\}^*$ for transitive closure, tree$[f_1, \ldots, f_n]$ denoting that a data structure is a tree, and cardinality for the cardinality of finite sets.

3.2 Specification Variables

In addition to concrete Java variables, Jahob supports *specification variables* [24, Section 4], which do not exist during program execution but are useful to specify the behavior of methods without revealing the underlying data structure representation. The developer uses the specvar keyword to introduce a specification variable, indicate its type and an optional initial value, whether the variable is public or private, and whether it is a static or instance variable. If a variable is not static, Jahob lifts the variable's type from the specified type t to obj $\Rightarrow t$, converting it into a variable of function type. There are two kinds of specification variables in Jahob: *ghost variables* and *defined variables*. Ghost variables must be updated explicitly (but Jahob ensures soundness in the presence of such updates). Defined variables are simply ways to name the value of an expression (which appears in the declaration of the variable). The definitions of defined variables must be acyclic. For recursive definitions the developer can use transitive closure or a ghost variable with a class invariant that encodes the desired recursive relationship.

3.3 Method Contracts

A method contract in Jahob contains three parts: 1) a precondition, written as a requires clause, stating the properties of the program state and parameter value that must hold before a method is invoked; 2) a frame condition, written as a modifies clause, listing the components of the state that the method may modify (the remaining components remain unchanged); and 3) a postcondition, written

as an ensures clause, describing the state at the end of the method (possibly defined relative to the parameters and state at the entry of the method). Jahob uses method contracts for assume/guarantee reasoning in the standard way. When analyzing a method m, Jahob assumes m's precondition and checks that m satisfies its postcondition and the frame condition. Dually, when analyzing a call to m, Jahob checks that the precondition of m holds and assumes that the values of state components from the frame condition of m change subject only to the postcondition of m, and that the state components not in the frame condition of m remain unchanged. Public methods omit changes to the private state of their enclosing class and instead use public specification variables to describe how they change the state. Methods typically do not specify changes to newly allocated objects (the exception is that if a field f is changed for allocated objects and is otherwise not mentioned in the modifies clause, then the developer needs to add the special item NEW.f into the modifies clause).

3.4 Class Invariants

A *class invariant* can be thought of as a boolean-valued specification variable that Jahob implicitly conjoins with the preconditions and postconditions of public methods. The developer can declare an invariant as private or public (the default annotation is private). Typically, a class invariant is private and is visible only inside the implementation of the class. Jahob conjoins the private class invariants of a class C to the preconditions and postconditions of methods declared in C. To ensure soundness in the presence of callbacks, Jahob also conjoins each private class invariant of class C to each reentrant call to a method m declared in a different class C_1. This policy ensures that the invariant C will hold if $C_1.m$ (either directly or indirectly) invokes a method in C. To make the invariant F with label l hold less often than given by this policy, the developer can write F as $b \rightarrow I$ for some specification variable b. To make F hold more often, the developer can use assertions with the shorthand $(\mathsf{theinv}\, l)$ that expand into F.

3.5 Annotations Within Method Bodies

The developer can use several kinds of annotations inside a method to refine expectations about the behavior of the code, to guide the analysis by stating intermediate facts, or to debug the verification process.

Loop invariants. The developer states a loop invariant of a while loop immediately after the while keyword using the keyword invariant (or inv for short). Each loop invariant must hold before the loop condition and be preserved by each iteration of the loop. The developer can omit conditions that depend only on variables not modified in the loop — Jahob uses a simple syntactic analysis to conclude that the loop preserves such conditions.

Local specification variables. In addition to specification variables at the class level, the developer can introduce ghost or non-ghost specification variables that are local to a particular method and are stated syntactically as statements in the method body. Such variables can be helpful to simplify proof obligations or to state relationships between the values of variables at different program points.

Non-deterministic change. A specification statement of the form $\mathsf{havoc}\, x\ \mathsf{suchThat}\ G$, where x is a variable and G is a formula, changes the value of x subject only to the constraint G (for example, the statement $\mathsf{havoc}\, x\ \mathsf{suchThat}\ 0 \leq x$ sets x to an arbitrary non-negative value). To ensure soundness, Jahob emits an assertion that verifies that at least one such value of x exists. Consequently, a havoc statement can also be used to "pick a witness" for an existentially quantified assumption $\exists x.G$ and to make this witness available for subsequent specification. A specification assignment of the form $x := e$ (for x not occurring in e) is a special case of the havoc statement whose condition is $x = e$ (its feasibility condition is trivial). Jahob also supports field specification assignments of the form $x..f := e$, which is a shorthand for $f := f(x := e)$. Here $f(x := e)$ is the standard function update expression returning a function identical to f except at x where it has value e.

Assert. An $\mathsf{assert}\, G$ annotation at program point p in the body of the method requires the formula G to be true at the program point p. Like standard Java assertions, Jahob assertions identify conditions that should be true at a given program point. An important difference is that Jahob assertions are statically checked to hold for all executions rather than dynamically checked for only the current execution. In particular, Jahob assertions produce proof obligations that Jahob statically verifies to guarantee that G will be true in all program executions that satisfy the precondition of the method.

Assume. An $\mathsf{assume}\, G$ statement is dual to the assert statement. Whereas an assert requires Jahob to demonstrate that G holds, an assume statement allows Jahob to assume that G is true at a given program point. The developer-supplied use of assume statements may violate soundness and causes Jahob to emit a warning. The intended use of assume is debugging, because it allows Jahob to verify a method under the desired restricted conditions. For example, a specification statement assume False at the beginning of a branch of an if statement means that Jahob will effectively skip the verification of that branch. More generally, assume statements allow the developer to focus the analysis on a particular scenario of interest (e.g. a particular aliasing condition) and therefore understand better why a proof attempt is failing.

Specifying lemmas. A $\mathsf{note}\, G$ statement is a sequential composition of $\mathsf{assert}\, G$ followed by $\mathsf{assume}\, G$. It is always sound for the developer to introduce a note statement because Jahob first checks the condition G before assuming it. Therefore, $\mathsf{note}\, G$ is semantically equivalent to $\mathsf{assert}\, G$, but instructs Jahob to use G as a useful lemma in proving subsequent conditions. Such lemmas can often overcome limitations of automated provers by providing key intermediate steps in the proof.

Specifying which assumptions to use. $\mathsf{note}\, G$ and $\mathsf{assert}\, G$ statements can optionally contain a clause "$\mathsf{by}\ l_1, \ldots, l_n$" to identify the facts from which the formula G should follow. We found the "by" construct particularly helpful for guiding first-order and SMT provers to an appropriate set of facts to use when the number and the complexity of the invariants becomes large [14]. The identifiers l_i can refer to the labels of facts introduced by previous note and assume statements, preconditions, invariants, conditions encoding a path in the program, or parts of formulas explicitly labelled using the comment construct.

Case analysis and hypothetical reasoning. When establishing note G, the developer can help the provers by doing a case analysis on some condition F. One way to achieve the case analysis is to use a sequence

$$\mathsf{note}\ l_1{:}(F \rightarrow G); \quad \mathsf{note}\ l_2{:}((\neg F) \rightarrow G); \quad \mathsf{note}\ G\ \mathsf{by}\ l_1, l_2$$

However, proving an implication such as $F \rightarrow G$ may itself require further note statements, which are valid only under the assumption F. Jahob therefore supports hypothetical blocks of the form $\mathsf{assuming}\ l_1{:}F\ \mathsf{in}\ (c\,;\mathsf{note}\ G)$. Within such a block, Jahob inserts F as an assumption. The block can contain further proof statements that derive consequences of F and previously known facts. (The block may not contain executable Java statements.) If the verification of the block succeeds, Jahob inserts the formula $F \rightarrow G$ after the block. To ensure soundness, the assumption F does not persist outside the block.

Proving universally quantified statements. To show that an assertion $\forall x.G$ holds after a sequence of statements c, instead of us-

ing the sequence $(c; \mathsf{note}\ \forall x.G)$, a Jahob developer can instead use the statement $(\mathsf{pickAny}\ x\ \mathsf{in}\ (c\,;\mathsf{note}\ G))$. With the pickAny construct, the universally quantified variable x becomes visible inside the statements c as a specification variable with arbitrary value. The developer can therefore state lemmas that involve x as a fixed variable and therefore help in the proof of G. Note that x is not a quantified variable in such verification conditions, which simplifies the theorem proving task. Moreover (unlike the assuming block), the statements c may contain Java code (including loops). The use of the pickAny construct may therefore make it possible to eliminate universal quantifiers from loop invariants (and the resulting verification conditions).

4. Generating Verification Conditions

Jahob produces verification conditions by simplifying the Java code and transforming it into extended guarded commands (Figure 8), then desugaring extended guarded commands into simple guarded commands (Figure 9), and finally generating verification conditions from simple guarded commands in a standard way (Figure 10).

4.1 Representation of Program Memory

The state of a Jahob program is given by a finite number of concrete and specification variables. The types of specification variables appear in their declarations. Jahob maps the types of concrete Java variables as follows. Static reference variables become variables of type obj (obj is the type of all object identifiers). An instance variable f in a class declaration class C {D f} becomes a function $f :: \mathsf{obj} \Rightarrow \mathsf{obj}$ mapping object identifiers to object identifiers. The Java expression x.f becomes fx, that is, the function f applied to x. Jahob represents Java class information using a set of objects for each class. For example, Jahob generates the axiom $\forall x.x \in C \rightarrow fx \in D$ for the above field f. Note that the function f is total. When x is null or of a class that does not include the field f, Jahob assumes that $fx = \mathsf{null}$. (Jahob correctly checks for the absence of null dereferences by creating an explicit assertion before each dereference.) Jahob represents an object-valued array as a function of type $\mathsf{obj} \Rightarrow \mathsf{int} \Rightarrow \mathsf{obj}$, which accepts an array and an index and returns the value of the array at the index. Jahob also introduces a function of type $\mathsf{obj} \Rightarrow \mathsf{int}$ that indicates the array size, and uses it to generate array bounds check assertions. The type int represents the integer type, which Jahob models as the set of unbounded mathematical integers.

4.2 From Java to Guarded Commands

Jahob's transformation of Java into guarded commands resembles a compilation process. Jahob simplifies executable statements into three-address form to make the evaluation order in expressions explicit. It also inserts assertions that check for null dereferences, array bounds violations, and type cast errors. It converts field and array assignments into assignments of global variables whose right-hand side contains function update expressions. Having taken the side effects into account, it transforms Java expressions into mathematical expressions in higher-order logic.

Receiver parameters in specifications. Java makes most uses of the receiver "this" parameter implicit, with the compiler using scoping rules to resolve the reference. Jahob applies similar rules to disambiguate occurrences of variables in specification constructs. When a field f occurs in an expression that is not immediately of the form x..f and when f is not qualified with a class name, Jahob converts the occurrence of f into this..f. Jahob also transforms each definition x=f of a non-static specification variable x into the definition $\mathsf{x} = \lambda\ \mathsf{this.f}$. If, after transformation, a class invariant *Inv* in class C contains an occurrence of this, Jahob transforms *Inv* into the invariant $\forall \mathsf{this.this} \in C \wedge \mathsf{this} \in \mathsf{alloc} \rightarrow \mathit{Inv}$. An invariant stated for a given object is therefore implicitly interpreted as being required for all allocated objects of the class, and becomes a global invariant. This mechanism enables Jahob developers to not only concisely state invariants on a per-object basis but also to use global invariants that state relationships between different instances of the class.

4.3 From Extended to Simple Guarded Commands

We call the main internal representation of Jahob the "extended guarded command language" because it contains guarded command statements, proof commands, and simple control structures. Figure 8 presents the syntax of the extended guarded command language. We next describe how Jahob transforms such guarded commands into the simple guarded command language (for which verification condition generation is standard as presented in Figure 10).

Translating state changing statements. Figure 11 describes the translation of guarded commands that change the state. We can represent assignments using havoc followed by an equality constraint, which reduces all state changes to havoc statements. Conditional statements become non-deterministic choice with assume statements, as in control-flow graph representations. The Jahob encoding of loops with loop invariants is analogous to the sound version of the encoding in ESC/Java [25].

Encoding and semantics of proof constructs. One of our observations is that proof constructs have natural translations into the guarded command language (as presented in Figure 12). This translation can also be viewed as providing a semantics for our proof constructs. With very modest requirements on the underlying provers (completeness for propositional reasoning and the ability to perform unification), the constructs pickAny, havoc, and note form a complete proof system for first-order logic. We therefore believe that the expressive power of these constructs and the simplicity of their translation into guarded commands makes them an appealing candidate for annotations in any software verification system.

4.4 Accounting for Variable Dependences

The semantics of extended guarded commands assumes a set D of specification variable definitions (v, D_v) where v is a variable and D_v is a term representing the definition of v in terms of other variables. If $\vec{u}$ is a list of variables we write $\mathsf{deps}(\vec{u})$ for the set of all variables that depend on any of the variables in $\vec{u}$, that, is, variables whose value may change if one of the variables $\vec{u}$ changes. To define this set precisely, let $\mathsf{FV}(G)$ denote all free variables in G, let the dependence relation be $\rho = \{(v_1, v_2) \mid (v_2, D_2) \in D \wedge v_1 \in \mathsf{FV}(D_2)\}$ and let ρ^* denote the transitive closure of ρ. Then $\mathsf{deps}(u_1, \ldots, u_n) = \cup_{i=1}^{n}\{v \mid (u_i, v) \in \rho^*\}$. We write $\mathsf{defs}(\vec{u})$ for the set of constraints expressing these dependencies, with $\mathsf{defs}(\vec{u}) = \bigwedge\{v = D_v \mid v \in \mathsf{deps}(\vec{u}) \wedge (v, D_v) \in D\}$. To correctly take dependences into account during verification condition generation, it suffices to treat in Figures 11 and 12 each command of the form $\mathsf{havoc}\ \vec{x}$ as the command $(\mathsf{havoc}\ (\vec{x}, \mathsf{deps}(\vec{x}))\,;\mathsf{assume}\ \mathsf{defs}(\vec{x}))$.

Eliminating unnecessary assumptions. To simplify the generated verification conditions, some of the internally generated assume statements indicate a variable that the statement is intended to constrain. For example, an assumption generated from a variable definition $v = D_f$ is meant to constrain the variable v, as are assumptions of the form $v \in C$ where C is a set of objects of class C. Ignoring an assumption is always sound, and Jahob does so whenever the postcondition does not contain a variable that the assumption is intended to constrain. Moreover, in certain cases Jahob reorders consecutive assume statements to increase the number of assumptions that it can omit.

$$
\begin{array}{rcl}
c & ::= & \mathsf{assume}\ l{:}\ F \mid \mathsf{assert}\ l{:}\ F\ \mathsf{by}\ h \\
& \mid & \mathsf{havoc}\ \vec{x}\ \mathsf{suchThat}\ F \\
& \mid & x := F \\
& \mid & \mathsf{note}\ l{:}F\ \mathsf{by}\ h \\
& \mid & \mathsf{assuming}\ F\ \mathsf{in}\ (c_{\mathsf{pure}}\ ;\ \mathsf{note}\ G) \\
& \mid & \mathsf{pickAny}\ \vec{x}\ \mathsf{in}\ (c\ ;\ \mathsf{note}\ G) \\
& \mid & c_1 \ [] \ c_2 \mid c_1\ ;\ c_2 \\
& \mid & \mathsf{if}(F)\ c_1\ \mathsf{else}\ c_2 \\
& \mid & \mathsf{loop\ inv}(I)\ c_1\ \mathsf{while}(F)\ c_2
\end{array}
$$

Figure 8. Extended guarded commands

$$
c ::= \mathsf{assume}\ l{:}\ F \mid \mathsf{assert}\ l{:}\ F\ \mathsf{by}\ h \mid \mathsf{havoc}\ x \mid c_1 \ [] \ c_2 \mid c_1\ ;\ c_2
$$

Figure 9. Simple guarded commands

$$
\begin{array}{lcl}
\mathsf{wlp}((\mathsf{assume}\ l{:}\ F), G) & = & F^{[l]} \rightarrow G \\
\mathsf{wlp}((\mathsf{assert}\ l{:}\ F\ \mathsf{by}\ h), G) & = & F^{[l;h]} \wedge G \\
\mathsf{wlp}((\mathsf{havoc}\ \vec{x}), G) & = & \forall \vec{x}.\, G \\
\mathsf{wlp}((c_1 \ [] \ c_2), G) & = & \mathsf{wlp}(c_1, G) \wedge \mathsf{wlp}(c_2, G) \\
\mathsf{wlp}((c_1\ ;\ c_2), G) & = & \mathsf{wlp}(c_1, \mathsf{wlp}(c_2, G))
\end{array}
$$

Figure 10. Weakest preconditions for simple guarded commands

$$
\begin{array}{rl}
\text{for } v \text{ fresh variable, } [\![x := F]\!] = & \mathsf{assume}\ (v = F)\ ; \\
& \mathsf{havoc}\ x\ ;\ \mathsf{assume}\ (x = v) \\
[\![\mathsf{if}(F)\ c_1\ \mathsf{else}\ c_2]\!] = & (\mathsf{assume}\ F\ ;\ c_1)\ [] \\
& (\mathsf{assume}\ \neg F\ ;\ c_2) \\
[\![\mathsf{loop\ inv}(I)\ c_1\ \mathsf{while}(F)\ c_2]\!] = & \\
& \mathsf{assert}\ I\ ;\ \mathsf{havoc}\ \vec{r}\ ;\ \mathsf{assume}\ I\ ; \\
& c_1\ ; \\
& (\mathsf{assume}\ (\neg F)\ [] (\mathsf{assume}\ F\ ; \\
& \qquad c_2\ ;\ \mathsf{assert}\ I\ ; \\
& \qquad \mathsf{assume\ false}))
\end{array}
$$

Figure 11. Translating executable constructs into simple commands

$$
\begin{array}{rl}
[\![\mathsf{havoc}\ \vec{x}\ \mathsf{suchThat}\ F]\!] = & \mathsf{assert}\ \exists \vec{x}.F\ ; \\
& \mathsf{havoc}\ \vec{x}\ ;\ \mathsf{assume}\ F \\
[\![\mathsf{note}\ l{:}F\ \mathsf{by}\ h]\!] = & \mathsf{assert}\ l{:}\ F\ \mathsf{by}\ h\ ; \\
& \mathsf{assume}\ l{:}\ F \\
[\![\mathsf{assuming}\ F\ \mathsf{in}\ (c_{\mathsf{pure}}\ ;\ \mathsf{note}\ G)]\!] = & (\mathsf{skip}\ [] (\mathsf{assume}\ F\ ; \\
& \qquad c_{\mathsf{pure}}\ ;\ \mathsf{assert}\ G\ ; \\
& \qquad \mathsf{assume\ false}))\ ; \\
& \mathsf{assume}\ (F \rightarrow G) \\
[\![\mathsf{pickAny}\ \vec{x}\ \mathsf{in}\ (c\ ;\ \mathsf{note}\ G)]\!] = & (\mathsf{skip}\ [] (\mathsf{havoc}\ \vec{x}\ ; \\
& \qquad c\ ;\ \mathsf{assert}\ G\ ; \\
& \qquad \mathsf{assume\ false}))\ ; \\
& \mathsf{assume}\ \forall \vec{x}.G
\end{array}
$$

Figure 12. Translating proof constructs into simple commands

$$
\begin{array}{rcl}
\vec{A} \rightarrow G_1 \wedge G_2 & \leadsto & \vec{A} \rightarrow G_1,\ \vec{A} \rightarrow G_2 \\
\vec{A} \rightarrow (\vec{B} \rightarrow G^{[p]})^{[q]} & \leadsto & (\vec{A} \wedge \vec{B}^{[q]}) \rightarrow G^{[pq]} \\
\vec{A} \rightarrow \forall x.G & \leadsto & \vec{A} \rightarrow G[x := x_{\mathsf{fresh}}]
\end{array}
$$

Figure 13. Splitting rules converting a formula into implication list (notation $F^{[c]}$ denotes formula F annotated with string c)

5. Proving Verification Conditions

Jahob generates a proof obligation for each method it verifies. These verification conditions are expressed in a subset of the Isabelle/HOL notation. We next discuss how Jahob proves such verification conditions.

5.1 Splitting

Jahob follows the standard rules in Figure 10 to generate verification conditions. Verification conditions generated using these rules can typically be represented as a conjunction of a large number of conjuncts. Figure 13 describes Jahob's splitting process, which produces a list of implications whose conjunction is equivalent to the original formula. The individual implications correspond to different paths in the method, as well as different conjuncts of assert statements, operation preconditions, invariants, postconditions, and preconditions of invoked methods.

The splitting rules in Jahob preserve formula annotations, which are used for assumption selection and in error messages to indicate why a verification failed. Because Jahob splits only the goal of an implication, the number of generated implications is polynomial in the size of the original verification condition (the verification condition itself can be exponential in the size of the method). During splitting Jahob eliminates simple syntactically valid implications, such as those whose goal occurs as one of the assumptions.

5.2 Using Multiple Provers

A typical data structure operation generates a verification condition that splitting separates into a few hundred implications, each of which is a candidate for any of the provers in Figure 1. Each implication generated from a verification condition must be valid for the data structure operation to be correct. Each proof can be performed entirely independently.

To prove an implication, Jahob may attempt to use any of the available provers. In practice, a Jahob user specifies, for a given verification task, a sequence of provers and their parameters on the command line. Jahob tries the provers in sequence, so the user lists the provers starting from the ones that are most likely to succeed or, if possible, fail quickly when they do not succeed. Often different provers are appropriate for different proof obligations in the same method. For such cases Jahob provides a facility to spawn provers in parallel and succeed as soon as at least one of them succeeds. On multi-core machines the resulting parallel execution can reduce the overall proof time. On single-core machines it may enable an appropriate prover to quickly prove the fact without waiting for any inappropriate provers to finish.

5.3 Formula Approximation

Efficient provers are often specialized for a particular class of formulas. One of the distinguishing characteristics of Jahob is its ability to integrate such specialized provers into a system that uses an expressive fragment of higher-order logic. This integration is based on the concept of *formula approximation*, which maps an arbitrary formula into semantically stronger but simpler formulas in an appropriate subset of higher-order logic. Because the resulting formulas are stronger, the approach is sound.

Figure 14 presents the general idea of approximation: for atomic formulas representable in the target logic subset, the approximation produces the appropriate translation; for logical operations it proceeds recursively; for unsupported atomic formulas it produces true or false depending on the polarity of the formula. To improve the precision of this recursive approximation step, Jahob first applies rewrite rules that substitute definitions of values, perform beta reduction, and flatten expressions. The details of rewriting and approximation depend on the individual prover interface.

$$
\begin{aligned}
\alpha &: \{0,1\} \times F \to \mathrm{C} \\
\alpha^p(f_1 \wedge f_2) &\equiv \alpha^p(f_1) \wedge \alpha^p(f_2) \\
\alpha^p(f_1 \vee f_2) &\equiv \alpha^p(f_1) \vee \alpha^p(f_2) \\
\alpha^p(\neg f) &\equiv \neg \alpha^{\overline{p}}(f) \\
\alpha^p(\forall x.f) &\equiv \forall x.\alpha^p(f); \quad \alpha^p(\exists x.f) \equiv \exists x.\alpha^p(f) \\
\alpha^p(f) &\equiv e, \text{ for } f \text{ directly representable in } C \text{ as } e \\
\alpha^0(f) &\equiv \mathsf{false}, \text{ for } f \text{ not representable in } C \\
\alpha^1(f) &\equiv \mathsf{true}, \text{ for } f \text{ not representable in } C
\end{aligned}
$$

Figure 14. General formula approximation scheme

6. Provers Deployed in Jahob

We next describe how we integrated several provers into Jahob.

6.1 Syntactic Prover

Before invoking external provers, Jahob first tests whether the formula is trivially valid. Specifically, it checks for the presence of appropriately placed propositional constants false and true. It also checks whether or not the conclusion of an implication appears in the assumption (modulo simple syntactic transformations that preserve validity). In practice these techniques discharge many verification condition conjuncts. The first source of such conjuncts is checks such as null dereferences which occur (implicitly) many times in the source code. The second source is sequences of method calls, specifically when class invariants that hold after one method call need to be shown to hold for subsequent calls. For complex formulas the syntactic prover is very useful because more sophisticated provers often perform transformations that destroy the structure of the formula, converting it into a form for which the proof attempt fails.

6.2 First-order Provers

Decades of research into first-order theorem proving by resolution have produced carefully engineered systems capable of proving non-trivial first-order formulas [76, 80, 84]. Jahob leverages this development by translating higher-order logic into first-order logic [14]. This translation is very effective for formulas without transitive closure and arithmetic. Such formulas may contain set expressions, but those expressions are typically quantifier-free, which enables their translation into quantified first-order formulas. Using ghost variables and recursive axioms we are also able to use first-order provers to prove strong properties about reachability in data structures [14]. Our translation uses an incomplete set of axioms for ordering and addition to provide partial support for linear arithmetic. We found this axiomatization effective for reasoning about data structures such as hash tables.

6.3 SMT Provers

Provers based on Nelson-Oppen combination of decision procedures enhanced with quantifier instantiation have been among the core technologies of past verification systems [60]. Jahob incorporates state-of-the art solvers in this family using the SMT-LIB standard format [67]. Overall, the approximation for this format is similar to the approximation for first-order provers, but uses the SMT-LIB representation of linear arithmetic. We have primarily used two SMT provers in Jahob: CVC3 [26] and Z3 [19].

6.4 MONA

MONA is a decision procedure for monadic second-order logic over strings and trees [30]. Its expressive power stems from its ability to quantify over sets of objects. Quantification over sets can in turn encode transitive closure, which is extremely useful for reasoning about recursive data structures. Jahob contains a flexible interface that enables the use of MONA even for some non-tree data structures [87]. When proving an implication $A_1, \ldots, A_n \to G$ this interface identifies assumptions of the form $\mathsf{tree}[f_1, \ldots, f_n]$, then interprets the formula assuming that $f_1, \ldots, f_n$ form the tree backbone of the data structure. Furthermore, it identifies assumptions A_i of the form $\forall xy.f(x){=}y \to H(x,y)$ (for $f \notin \{f_1, \ldots, f_n\}$) and soundly approximates a goal of the form $G(f(t))$ with the stronger goal $\forall u.H(t,u) \to G(u)$. This enables the approximation to maintain information about non-tree fields and provides certain completeness guarantees [87, Theorems 2 and 3].

6.5 BAPA

Jahob also implements decision procedures for sets with symbolic cardinality bounds [43, 46]. This decision procedure can prove a class of verification conditions that use set algebra, symbolic cardinality constraints, and linear arithmetic (i.e., quantifier-free Presburger arithmetic). Such verification conditions arise when checking invariants on the size of allocated structures, as in the sized list example of Section 2.2 and other examples such as tracking the number of objects that a method allocates [46]. Previous theorem provers have limited effectiveness for such formulas because set algebra and linear arithmetic interact in non-trivial way through the cardinality operator.

6.6 Isabelle and Coq

Jahob provides interfaces to the Isabelle [63] and Coq [11] interactive theorem provers. Jahob can invoke Isabelle automatically on a given proof obligation using the general-purpose theorem proving tactic in Isabelle. In some cases (e.g., for relatively small proof obligations that involve complex set expressions) this approach succeeds even when other approaches fail. In general, Isabelle requires interaction, so the user can prove the implication interactively and save it into a file. Jahob loads this file in future verification attempts and treats such proven lemmas as true.

7. Verified Data Structures

We have specified and verified the following data structures:

- **Association List:** The association list data structure discussed in Section 2.
- **Space Subdivision Tree:** A three-dimensional space subdivision tree. Each internal node in the tree stores the pointers to its subtrees in an eight-element array.
- **Spanning Tree:** A spanning tree for a graph. Verified properties include that the produced data structure is, in fact, a tree and that this tree includes all nodes reachable from the root of the graph.
- **Hash Table:** A hash table implementing a map from objects to objects, implemented as an array of linked lists storing keys and values.
- **Binary Search Tree:** A binary search tree implementing a set, with tree operations verified to preserve tree shape, ordering, and changes to tree content.
- **Priority Queue:** A priority queue stored as a complete binary tree in a dense array, with parent and child relationships computed by arithmetic operations on array indices. Among the verified properties is that the findMax method returns the smallest element in the queue, a property that requires verifying that all operations preserve the heap ordering invariant.
- **Array List:** A list stored in an array implementing a map from integers to objects, optimized for storing maps from a dense subset of the integers starting at 0 (modelled after

Data Structure	Syntactic Prover	MONA	Z3	SPASS	E	CVC3	Isabelle Script	Interactive Proof	Total Time
Association List	227			120 (8.9s)					12.0s
Space Subdivision Tree	392		269 (46.9s)	9 (2.5s)				1	70.9s
Spanning Tree	368			80 (142.6s)		22 (2.0s)			172.2s
Hash Table	570			222 (58.3)			1(0.5)	6	73.6s
Binary Search Tree	469	665 (6232.1s)	170 (7.5s)		10 (0.5s)				6265.0s
Priority Queue	311		179 (4.9)					4	12.9s
Array List	400		306 (60.8s)	16 (66.7s)		2 (9.9s)			161.1s
Circular List	26	100 (183.6s)							184.4s
Singly-Linked List	74			94 (5.9s)					6.9s
Cursor List	193		218 (27.6s)	17 (2.3s)					41.2s

Figure 15. Number of Proved Sequents and Verification Times for Verified Data Structures

`java.util.ArrayList`). Method contracts in the list describe operations using an abstract relation $\{(0, v_0), \ldots, (k, v_k)\}$, where $k + 1$ is the number of stored elements.

- **Circular List:** A circular doubly-linked list implementing a set interface.
- **Singly-Linked List:** A null-terminated singly-linked list implementing a set interface.
- **Cursor List:** A list with a cursor that can be used to iterate over the elements in the list and, optionally, remove elements during the iteration. Method contracts include changes to the list content and to the position of the iterator.

Together, these data structures comprise a representative subset of the data structures found in a typical program.

7.1 Verification Statistics

Figure 15 contains, for each data structure, a line summarizing the verification process for that data structure. Each line contains a breakdown of the number of sequents proved by each theorem prover or decision procedure when verifying the corresponding data structure. The theorem provers or decision procedures are applied in the order in which they appear in the table. A blank entry indicates that the corresponding theorem prover or decision procedure was not used during the verification. Figure 15 also presents, for each theorem prover or decision procedure, the time it took trying to prove the sequents it attempted to prove. Consider, for example, the SPASS entry for the Association List. This entry is 120 (8.9s), indicating that, for the Association List data structure, the SPASS theorem prover took 8.9 seconds trying to prove sequents, and succeeded in proving 120 of them (the 8.9 seconds includes time spent on unsuccessful proof attempts for sequents that were later proved by another prover). The final column presents the total verification time, which includes the time spent in the verification condition generator, splitter, syntactic prover, and any applied decision procedures or theorem provers. Most of the data structures verify within several minutes. The outlier is the binary search tree with a total verification time of an hour and forty-five minutes (primarily due to the amount of time spent in the MONA decision procedure).

7.2 Discussion

Figure 15 illustrates how Jahob effectively combines the capabilities of multiple theorem provers and decision procedures to verify sophisticated data structure correctness properties. It also illustrates how the different capabilities of these theorem provers and decision procedures are necessary to obtain the correctness proofs. For example, although the vast majority of the sequents are proved by fully automated means, the occasional use of interactive proofs is critical for enabling the verification of our set of data structures.

In our experience specifying and verifying a new data structure requires insight into why the data structure implementation is correct combined with familiarity with the verification system. At this point we are able to implement, specify, and fully verify a new, relatively simple data structure (such as a list implementation of a set) in several hours. More complicated data structures (such as a space subdivision tree) can take days or even, in extreme cases, a week or more.

Our current design emphasizes the simplicity of the underlying semantic model. Almost all required correctness properties therefore appear explicitly in both the specifications and the generated verification conditions. Different designs are possible. For example, Jahob could support a system of defaults that would enable developers to work with simpler specifications. Similarly, an enhanced (but more complex) verification condition generator could use simple checks to eliminate many properties before verification condition generation. We anticipate that an optimal distribution of the verification responsibility across the different components of the verification system will become clearer as researchers gain more experience with the problem.

8. Related Work

We discuss the Hob analysis system and related work in shape analysis, software verification systems, interactive theorem provers, and finitization and automated testing.

Hob. The Hob system supports verified data structure interfaces that use sets of objects to summarize the state of the data structure and the effect of data structure operations [42,49]. Because the Hob specification language is based on sets, it is powerful enough to specify full functional correctness only for data structures that export a set interface. For data structures with richer interfaces (such as hash tables and search trees) it can specify and verify some, but not all, correctness properties. Like Jahob, Hob integrates a variety of reasoning techniques to successfully discharge generated verification conditions, including the use of arbitrarily precise reasoning techniques within data structure implementations.

Hob emphasizes the use of the verified specifications in the analysis of data structure clients. We have successfully used Hob to develop new analyses with an unprecedented combination of scalability and precision. Indeed, our results show that Hob is capable of specifying and verifying deep properties that capture important concepts from the underlying domain of the program. These properties are directly meaningful not just to the developers of the program, but (perhaps more importantly) also to its users. To the best of our knowledge, Hob is the first system capable of specifying and verifying these kinds of outward-looking, application- and domain-oriented correctness properties.

Shape analysis. The goal of shape analysis is typically to verify only data structure shape properties (and not full functional correctness properties such as the change of data structure content) [18, 28, 28, 41, 50, 74]. Parameterized shape analyses such as TVLA have been extended to prove properties beyond shape, such as ordering of list elements [52] and the correctness of a binary search tree with a set interface [68] (using manually devised and unverified rules for updating instrumentation predicates). Approaches to automating separation logic have similarly focused primarily on shape properties as opposed to full correctness properties [10]. These approaches have recently been extended to verify bag and size properties (although the system does not support arrays or loops) [62]. Advanced type systems similarly use recursive data structure specifications with fold and unfold as proof rules [22, 89]. Although these type systems could, in principle, be extended to prove specifications that use relations as specification variables, we are not aware of any system that has done so.

Decision procedures based on finite quantifier instantiation [55] are effective for reasoning about local properties of data structures but are not complete for reachability properties. However, decision procedures that support reachability exist that can verify programs that manipulate only linked lists [47]. In some cases, it is also possible to express reachability properties in first-order logic [14, 41, 48, 51, 55, 61]. Approaches based on MONA [58, 87] guarantee completeness for reachability properties. By themselves, these approaches are not sufficient for the verification of many important data structure properties. Our experience indicates, however, that they become very useful in combination with other techniques. The applicability of decision procedures such as MONA can be extended using structure simulation [37], provided that certain conditions are met. A Jahob user can use a tree declaration over manually updated ghost fields to obtain some of the benefits of structure simulation. Automated first-order or SMT provers can inductively prove the conditions required for the soundness of this approach as a part of the standard verification process, without requiring any special support or methodology.

Software model checkers based on predicate abstraction use theorem provers to over-approximate reachable program states [6, 31]. A recent combination with shape analysis can verify shape properties, yielding performance better than when using shape analysis alone [12]. Jahob contains an implementation of an alternative approach, symbolic shape analysis [66, 87, 88], which generalizes the predicate abstraction domain to perform shape analysis. We have not used symbolic shape analysis for the examples in this paper. However, we have applied symbolic shape analysis to somewhat simpler data structures, including lists, trees, and arrays [88]. The analysis successfully inferred loop invariants and proved the full functional correctness of operations that insert elements into data structures that implement a set interface.

Shape analyses occupy an uneasy position in a world with verified data structure specifications. Analyses involving data structure clients are invariably better off working with the higher-level abstractions present in the specifications rather than directly with the pointers in the implementations. Although shape analyses can be useful for automating parts of the analysis of data structure implementations (indeed, we have used this technology for this purpose ourselves), their value is undercut by the fact that less automated techniques are perfectly adequate for this purpose.

Software verification tools. Software verification tools that can prove properties of linked data structures include Spec# [8], ESC/Modula-3 [21], ESC/Java [24], ESC/Java2 [17], Krakatoa [23, 53], KIV [7], KeY [3], and LOOP [82]. To the best of our knowledge, none of these systems have been used to verify the full functional correctness of a collection of linked data structures. For example, the LOOP system has been used to prove the correctness of the Java Vector class implementation [33, 34], which is not a recursive linked data structure. LOOP, KIV, Jive, and Krakatoa have been used to verify smartcard applications (an electronic purse and the Mondex case study [29, 79, 81]), which do not contain complex linked data structures. KeY has also been used to prove the correctness of an insertion operation into a TreeMap [71]. While these efforts suggest that the verification of linked data structures is possible in principle, the scope of these previous results does not establish the extent to which this verification is feasible in practice using these systems.

Jahob's integrated reasoning approach (specifically the integration of a wide range of theorem provers and decision procedures via a new combination technique in conjunction with features that enable the developer to guide the verification process when necessary) makes it feasible to verify a range of linked data structure implementations. Moreover, previous systems do not, to the best of our knowledge, use decision procedures (such as MONA [30]) that enable complete reasoning over list and tree data structures, nor do they use decision procedures (such as the BAPA decision procedure [43, 46]) to reason about sets with cardinality constraints.

Proof methods based on natural deduction combined with automated provers have recently been shown to be effective for obtaining complex proofs in interactive provers [4, 85]. Although Jahob supports the use of interactive provers, its proof commands provide an alternative way of decomposing proof obligations without ever leaving the world of the original Java program. The fact that these proof constructs naturally translate into guarded commands suggests that they are intuitive for the verification of imperative programs. For example, the havoc ... suchThat statement works both as a proof construct of systems such as Isabelle, and as a specification statement in wide-spectrum languages [59].

Interactive theorem proving systems. The notation for formulas in Jahob is based on Isabelle/HOL [63]. Provers such as Isabelle/HOL support inductively defined data types and have been used to verify the correctness of purely functional data structures such as a binary search tree with a map interface [39], an AVL tree with a set interface [64], and garbage collection algorithms [56]. In the Verisoft project researchers have developed Isabelle proofs of correctness for doubly-linked list implementations [2]. It is natural to consider combinations of automated techniques to increase the granularity of interactive proof steps in interactive provers. This kind of integration is used in PVS [65], Boyer-Moore provers [16], and higher-order logic systems [35, 54]. There are recent and ongoing efforts to integrate Isabelle with monadic second-order logic over strings [9] and with first-order provers [57]. We believe that the Jahob approach is useful for proof obligations that arise in data structure verification (and potentially for other kinds of proof obligations as well), whether these proof obligations arise within the context of a program verification system or entirely within an interactive theorem prover.

Finitization and automated testing. Jahob and other systems based on theorem proving verify that data structures are correct for *all* executions. In contrast, testing and software model checking approaches based on finitization [13, 15, 20, 38, 70, 77, 78] check the correctness of only finitely many executions (and not the correctness of the remaining infinitely many executions). Systems such as Bogor [70] and JACK [13] integrate several techniques for checking finite models of software systems. We consider such approaches to be complementary to ours. With the current state of verification technology, a cost-effective approach to develop correct data structures might be to first develop the implementation and perform manual testing, then develop specifications and check them using finite state exploration techniques, and finally use a system such as Jahob to prove the implementations correct.

9. Conclusion

This paper demonstrates the use of integrated reasoning to obtain the first verification of full functional correctness for a substantial collection of linked data structures. We have already verified many of the data structures that programmers use in practice. In the near future it is not unreasonable to expect to see data structure libraries shipped only after full functional specification and verification.

Full functional verification has long been viewed as an impractical or even unrealizable goal. The results in this paper demonstrate, for the first time, that this goal is within practical reach for linked data structure implementations. These results are especially compelling given the widespread reuse of data structure libraries and the central role that linked data structures play in computer science.

Acknowledgements. We thank Thomas Wies for his contributions to Jahob, which include the implementation of field constraint analysis, the MONA interface [87], the syntactic prover, and contributions to the SMT-LIB interface. We thank Charles Bouillaguet for developing the interface to first-order provers [14]. We also thank the anonymous reviewers and our shepherd, Rajeev Alur, for their useful feedback on the paper.

References

[1] The Jahob project web page. `http://javaverification.org`. last accessed: March 2008.

[2] Verisoft project. `http://www.verisoft.de`, Last accessed March 2008.

[3] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[4] K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard. Verifying a file system implementation. In *ICFEM*, volume 3308 of *LNCS*, 2004.

[5] D. Aspinall. Proof general: A generic tool for proof development. In *TACAS*, 2000.

[6] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, 2001.

[7] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In *FASE*, number 1783 in LNCS, 2000.

[8] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[9] D. Basin and S. Friedrich. Combining WS1S and HOL. In *Frontiers of Combining Systems 2*, 2000.

[10] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2005.

[11] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development–Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[12] D. Beyer, T. A. Henzinger, and G. Théoduloz. Lazy shape analysis. In *CAV*, 2006.

[13] A. Bouali, S. Gnesi, and S. Larosa. The integration project for the JACK environment. *Bulletin of the EATCS*, (54):207–223, 1994.

[14] C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, and M. Rinard. Using first-order theorem provers in a data structure verification system. In *VMCAI'07*, November 2007.

[15] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, 2002.

[16] R. S. Boyer and J. S. Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In *Machine Intelligence*, volume 11, pages 83–124. OUP, 1988.

[17] P. Chalin, C. Hurlin, and J. Kiniry. Integrating static checking and interactive verification: Supporting multiple theories and provers in verification. In *VSTTE*, 2005.

[18] S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In *Proc. 10th SAS*, volume 2694 of *LNCS*. Springer, 2003.

[19] L. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *CADE*, 2007.

[20] G. Dennis, F. Chang, and D. Jackson. Modular verification of code with SAT. In *ISSTA*, 2006.

[21] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report 159, COMPAQ Systems Research Center, 1998.

[22] J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, 2007. CMU-CS-07-129.

[23] J.-C. Filliatre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, 2003.

[24] C. Flanagan, K. R. M. Leino, M. Lilibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2002.

[25] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proc. 28th ACM POPL*, 2001.

[26] Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In *CADE*, 2007.

[27] M. J. C. Gordon and T. F. Melham. *Introduction to HOL, a theorem proving environment for higher-order logic*. Cambridge University Press, Cambridge, England, 1993.

[28] B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In *PLDI*, 2007.

[29] D. Haneberg, G. Schellhorn, H. Grandy, and W. Reif. Verification of Mondex electronic purses with KIV: from transactions to a security protocol. *Formal Asp. Comput.*, 20(1):41–59, 2008.

[30] J. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS*, 1995.

[31] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *31st POPL*, 2004.

[32] M. Hirzel, A. Diwan, and M. Hind. Pointer analysis in the presence of dynamic class loading. In *ECOOP*, 2004.

[33] M. Huisman. *Java program verification in higher order logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.

[34] M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's vector class. *STTT*, 3(3):332–352, 2001.

[35] J. Hurd. An LCF-style interface between HOL and first-order logic. In *CADE-18*, 2002.

[36] N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *Computer Science Logic (CSL)*, pages 160–174, 2004.

[37] N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. Verification via structure simulation. In *CAV*, pages 281–294, 2004.

[38] S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. *Autom. Softw. Eng.*, 11(4):403–434, 2004.

[39] V. Kuncak. Binary search trees. The Archive of Formal Proofs, `http://afp.sourceforge.net/`, April 2004.

[40] V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.

[41] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Annual ACM Symp. on Principles of Programming Languages (POPL)*, 2002.

[42] V. Kuncak, P. Lam, K. Zee, and M. Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 32(12), December 2006.

[43] V. Kuncak, H. H. Nguyen, and M. Rinard. An algorithm for deciding BAPA: Boolean Algebra with Presburger Arithmetic. In *CADE-20*, 2005.

[44] V. Kuncak, H. H. Nguyen, and M. Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *J. of Automated Reasoning*, 2006. `http://dx.doi.org/10.1007/s10817-006-9042-1`.

[45] V. Kuncak and M. Rinard. Existential heap abstraction entailment is undecidable. In *SAS*, 2003.

[46] V. Kuncak and M. Rinard. Towards efficient satisfiability checking for Boolean Algebra with Presburger Arithmetic. In *CADE-21*, 2007.

[47] S. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using smt solvers. In *POPL*, 2008.

[48] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL*, 2006.

[49] P. Lam. *The Hob System for Verifying Software Design Properties*. PhD thesis, Massachusetts Institute of Technology, February 2007.

[50] O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*, 2005.

[51] T. Lev-Ami, N. Immerman, T. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *CADE-20*, 2005.

[52] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Int. Symp. Software Testing and Analysis*, 2000.

[53] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 2003.

[54] S. McLaughlin, C. Barrett, and Y. Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. In *PDPAR*, volume 144(2) of *ENTCS*, pages 43–51, Jan. 2006.

[55] S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *CAV*, pages 476–490, 2005.

[56] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In *CADE-19*, 2003.

[57] J. Meng and L. C. Paulson. Translating higher-order problems to first-order clauses. In *ESCoR: Empir. Successful Comp. Reasoning*, pages 70–80, 2006.

[58] A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *Programming Language Design and Implementation*, 2001.

[59] C. Morgan. *Programming from Specifications (2nd ed.)*. Prentice-Hall, Inc., 1994.

[60] G. Nelson. Techniques for program verification. Technical report, XEROX Palo Alto Research Center, 1981.

[61] G. Nelson. Verifying reachability invariants of linked structures. In *POPL*, 1983.

[62] H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape, size and bag properties via separation logic. In *VMCAI*, 2007.

[63] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[64] T. Nipkow and C. Pusch. AVL trees. The Archive of Formal Proofs, `http://afp.sourceforge.net/`, March 2004.

[65] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th CADE*, volume 607 of *LNAI*, pages 748–752, jun 1992.

[66] A. Podelski and T. Wies. Boolean heaps. In *Proc. Int. Static Analysis Symposium*, 2005.

[67] S. Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.

[68] J. Reineke. Shape analysis of sets. Master's thesis, Universität des Saarlandes, Germany, June 2005.

[69] M. Rinard and P. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *TOPLAS*, 19(6), Nov. 1997.

[70] Robby, E. Rodríguez, M. B. Dwyer, and J. Hatcliff. Checking JML specifications using an extensible software model checking framework. *STTT*, 8(3), 2006.

[71] A. Roth. Deduktiver Softwareentwurf am Beispiel des Java Collections Frameworks. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, June 2002.

[72] R. Rugina and M. C. Rinard. Pointer analysis for structured parallel programs. *ACM Trans. Program. Lang. Syst.*, 25(1), 2003.

[73] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Trans. Program. Lang. Syst.*, 27(2), 2005.

[74] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.

[75] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *PPoPP*, 2001.

[76] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[77] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[78] A. Sobeih, V. Mahesh, D. Marinov, and J. Hou. J-Sim: An integrated environment for simulation and model checking of network protocols. In *IPDPS*, 2007.

[79] S. Stepney, D. Cooper, and J. Woodcock. An electronic purse: Specification, refinement, and proof. Technical monograph PRG-126, Oxford University Computing Laboratory, 2000.

[80] G. Sutcliffe and C. B. Suttner. The TPTP problem library: CNF release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[81] I. Tonin. Verifying the Mondex case study: The KeY approach. Technical Report 2007-4, Uni. Karlsruhe, July 2007.

[82] J. van der Berg and B. Jacobs. The LOOP compiler for Java and UML. Technical Report CSI-R0019, Computing Science Institute, Univ. of Nijmegen, Dec. 2000.

[83] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Proc. ACM PLDI*, June 2001.

[84] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.

[85] M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universitaet Muenchen, 2002.

[86] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA*, Denver, Nov. 1999.

[87] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field constraint analysis. In *VMCAI*, 2006.

[88] T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. Rinard. Verifying complex properties using symbolic shape analysis. In *Heap Abstraction and Verification*, 2007.

[89] D. Zhu and H. Xi. Safe programming with pointers through stateful views. In *PADL*, 2005.