Verifying Linked Data Structure Implementations

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

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

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

Abstract

The Jahob program verification system leverages state of the art automated theorem provers, shape analysis, and decision procedures to check that programs conform to their specifications. By combining a rich specification language with a diverse collection of verification technologies, Jahob makes it possible to verify complex properties of programs that manipulate linked data structures. We present our results using Jahob to achieve full functional verification of a collection of linked data structures.

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.

Jahob is a program verification system that leverages a rich specification language and a diversity of automated theorem provers and decision procedures to verify complex properties of programs that manipulate linked data structures. Our specifications use abstract sets and relations to characterize the abstract state of the data structure. A verified abstraction function defines abstract sets and relations in terms of concrete objects and references that the implementation manipulates at run time. Specifications use abstract sets and relations to state externally visible properties of the data structure state and to specify method preconditions and postconditions. These specifications capture all of the semantic information that the developer needs to use the data structure.

Jahob establishes desired program properties using a number of external decision procedures, Nelson-Oppen provers, and first-order provers. It automatically generates verification condition formulas from the program and its specification, then splits each verification condition into an equivalent conjunction of properties. By processing each conjunct separately, Jahob can use different provers to establish different parts of the proof obligations. This is possible thanks to formula approximation techniques[10] that create equivalent or semantically stronger formulas accepted by the specialized decision procedures.

This paper presents our experience using the Jahob system to obtain full correctness proofs for a collection of linked data structure implementations. Unlike systems that are designed to verify partial correctness properties, Jahob verifies the full functional correctness of the data structure implementation. Not only do our results show that such full functional verification is feasible, the verified data structures and algorithms provide concrete examples that can help developers better understand how to achieve similar results in future efforts.

2 Example

In this section we use a verified association list to demonstrate how developers specify data structure implementations in Jahob. Figure 1 presents selected portions of the AssocList class. This class maintains a list of key, value pairs. Our system 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 1 identifies the abstract state content of the association list as a relation in the form of a set of pairs of objects (we present the abstraction function that defines this abstract state below).[1]

The put(k0,v0) method inserts the pair (k0,v0) into the association list, returning the previous association for k0 (if such an association exists). The requires clause indicates

[1]Our examples use mathematical notation for concepts such as set union and universal quantification. While our source code files encode this notation in text, Isabelle's version of the ProofGeneral Emacs mode renders the specifications within Java code in mathematical notation. The figures correspond to what the user sees on the screen in such mode. See the screen shots at `http://javaverification.org`.

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

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

Figure 1. Association List put method

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

Figure 2. Node Definition

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 states that the abstract state content after the method executes is the abstract state old content from before the method executed 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 exists.

Figure 2 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 visibility condition by a syntactic check.

Each Node object has a specification variable cnt. This variable holds a set of pairs representing all of the associations in the part of the association list starting at the given Node object. It is initialized to the empty set, is explicitly updated as the implementation manipulates the list of nodes, and is conceptually part of the state of the corresponding Node object. It is used only during the verification of the association list and does not exist when the program runs. The purpose of the cnt variable in the specification is to define the abstract state content of the association list.

Figure 3 presents the invariants that characterize the cnt specification variable and the corresponding abstract state content of the association list. The variable first holds a reference to the first Node in the linked list that implements the association list; the abstract state content is the value of the cnt specification variable of the first node in the list. The CntDef invariant recursively defines cnt for non-null nodes as the key, value pair in the node union the value of cnt for the next node in the list. The CntNull invariant defines cnt as empty for the null object.

The syntax of these invariants reflects the underlying semantic domain in which the verification takes place. The domain is an infinite set of objects. Classes correspond to sets of objects within this domain. Fields correspond to total functions from objects to values— each object has all of the fields from all of the classes. If the object is not a member of a given class, the values of all of the fields from that class are simply null. For example, x ∈ Node states that object x has class Node. The expression x..next is a shorthand for the application of function next to object x, with the next function modeling the Java field next.

```
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 3. Abstraction Function and Selected Invariants

```
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 4. Implementation of the get method

Figure 4 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 that the pair (k0, value) 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 the search does not skip the Node with key k0. Given the specification and the invariants, Jahob is capable of verifying that this method 1) correctly implements its specification, and 2) correctly preserves the invariants.

In addition to this 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. For each of these methods, Jahob is able to statically verify full functional specifications.

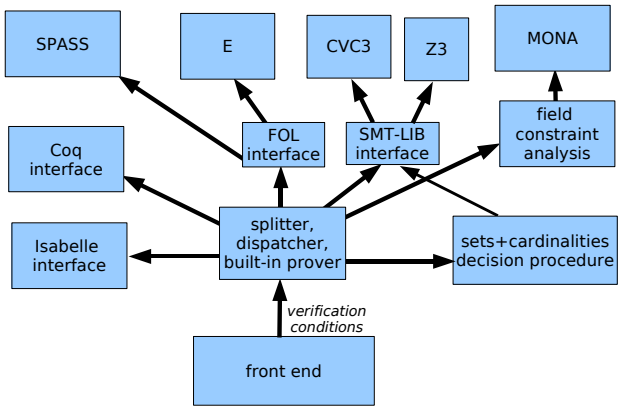

Figure 5. High-level structure of Jahob

3 System Overview

Figure 5 summarizes key aspects of the Jahob architecture, illustrating the provers that Jahob uses to establish data structure correctness. Jahob is a command-line tool, with command-line options that support verification at the granularity of methods or even individual assertions. Such modular verification enables interactive use that focuses on one part of the verification problem at a time. Developers specify Jahob programs using specification variable declarations, method contracts, class invariants, and annotations within method bodies. Many of these specification constructs contain formulas; the syntax and semantics of Jahob formulas follow Isabelle/HOL[16].

Jahob produces verification conditions by simplifying Java code and transforming it into extended guarded commands, desugaring extended guarded commands into simple guarded commands, and finally generating verification conditions from simple guarded commands in a standard way. For each method, Jahob produces a verification condition expressed in the Isabelle/HOL notation. Jahob exploits the property that each verification condition can typically be split into a conjunction of a large number of conjuncts. 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 5. Each implication must be valid for the data structure operation to be correct, and each proof can be performed entirely independently, opening up opportunities for parallelization.

A Jahob user specifies, for each 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. This is useful even on a single-core machine because it gives the appropriate prover the chance to quickly prove the fact instead of waiting for any inappropriate provers to finish.

Each Jahob prover typically accepts formulas in a proper subset of HOL. In practice, 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 HOL fragment. This integration is based on the concept of *formula approximation*, which maps an arbitrary HOL formula into semantically stronger formulas in a subset of HOL[10]. The fact that the formulas are stronger ensures that the approach is sound. For atomic formulas known to the target logic subset, the approximation produces the appropriate translation; for logical operations it proceeds recursively; and for unknown 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.

Jahob deploys the following provers: 1) an internal syntactic prover, which first tests whether the formula is trivially valid by checking for the presence of propositional constants and whether its conclusion appears in its assumption (modulo simple syntactic transformations that preserve validity); 2) first-order provers E[19] and SPASS[20]; 3) SMT provers CVC3[8] and Z3[7], based on Nelson-Oppen combination of decision procedures enhanced with quantifier instantiation[14]; 4) MONA, a decision procedure for monadic second-order logic over strings and trees[9] which Jahob uses for shape analysis; and 5) interactive theorem provers Isabelle[16][2], and Coq[5].

4 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 the spanning tree includes all nodes reachable from the root of the graph.
- **Hash Table:** A separately-chained hash table implementing a map from objects to objects.

[2] Jahob can also invoke Isabelle automatically on a given proof obligation using the general-purpose theorem proving tactic in Isabelle.

Data Structure	Syntactic Prover	MONA	Z3	SPASS	E	CVC3	Isabelle Script	Interactive Proof	Total Time
Association List	227			120 (8.9s)					11.97s
Space Subdivision Tree	392		269 (46.9s)	9 (2.5s)				1	70.91s
Spanning Tree	368			80 (142.6s)		22 (2.0s)			172.17s
Hash Table	570			222 (58.3)			1(0.5)	6	73.65s
Binary Search Tree	469	665 (6232.1s)	170 (7.5s)		10 (0.5s)				6265s
Priority Queue	311		179 (4.9)					4	12.86s
Array List	400		306 (60.8s)	16 (66.7s)		2 (9.9s)			161.10s
Circular List	26	100 (183.6s)							184.37s
Singly-Linked List	74			94 (5.9s)					6.94s
Cursor List	193		218 (27.6s)	17 (2.3s)					41.24s

Figure 6. Number of Proved Sequents and Verification Times for Verified Data Structures

- **Binary Search Tree:** A binary search tree (with verified ordering and membership changes).
- **Priority Queue:** A priority queue stored as a complete binary tree in a dense array. One verified property, among others, is that the findMax method does, in fact, return the largest element in the queue.
- **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. Method contracts in the list describe operations using an abstract relation $\{(0, v_0), (1, v_1), \ldots, (k, v_k)\}$ 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 indicate changes to list content and to the position of the iterator.

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

4.1 Verification Statistics

Figure 6 contains, for each verified 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 table entry indicates that the corresponding theorem prover or decision procedure was not used during the verification. Figure 6 also presents, for each theorem prover or decision procedure, the time it took to prove its sequents. 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 to prove the 120 sequents that it successfully proved. The final column presents the total verification time for the data structure. In addition to the time spent in successful sequent proof attempts, these times include the time spent in theorem provers or decision procedures that did not manage to prove the sequent before they timed out. 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.

4.2 Discussion

Figure 6 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/or decision procedures are necessary to obtain the full functional 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. The result is a reusable data structure that is guaranteed to correctly implement its specification.

5 Related Work

We review related work in program verification systems, shape analysis, and interactive theorem provers.

Software verification tools. Software verification tools based on theorem proving include Spec#[3], ESC/Java2[6], Krakatoa[12], KIV[2], and KeY[1]. We are not aware of any effort to use these systems to verify that a significant collection of data structure implementations conforms to their specifications, nor of any other system that integrates the diversity of theorem provers and decision procedures that Jahob does.

Shape analysis. Many approaches in shape analysis focus on increasing automation by loop invariant inference, but verify only shape properties and not full functional correctness properties such as the change of data structure content[18] and certain correctness properties[11]. TVLA generally accepts the supplied predicate update formulas without verifying them and applies them automatically. Jahob, in contrast, requires manual updates of ghost variables, but verifies that these updates do not violate soundness. Approaches to automating separation logic have similarly focused primarily on shape properties as opposed to full correctness properties[4, 15]. Most approaches based on type systems[21] have so far also been applied only to partial correctness properties.

Interactive theorem proving systems. The notation for formulas in Jahob is based on Isabelle/HOL[16]. We use Isabelle for our interactive proofs, but other interactive provers could also be used for this purpose[5]. Integration of interactive proof with decision procedures is also used in PVS[17]. Ongoing efforts integrate Isabelle with first-order provers[13]. We believe that the Jahob approach is useful for proof obligations arising in data structure verification, whether these proof obligations arise within the context of a program verification system or an interactive theorem prover.

6 Conclusion

Full functional verification has long been viewed as an impractical or even unrealizable goal. The results in this paper demonstrate that this goal is within practical reach for linked data structures. Using the Jahob system, we have verified many of the data structures that programmers use in practice. Our results are especially compelling given the widespread reuse of data structure libraries and the central role that linked data structures play in computer science. In the near future it is not unreasonable to expect to see core data structure libraries shipped only after full functional specification and verification.

References

[1] 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.

[2] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS. Springer, 2000.

[3] 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.

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

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

[6] P. Chalin, C. Hurlin, and J. Kiniry. Integrating static checking and interactive verification: Supporting multiple theories and provers in verification. In *Proceedings of Verified Software: Tools, Technologies, and Experiences (VSTTE)*, 2005.

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

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

[9] J. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS '95, LNCS 1019*, 1995.

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

[11] 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.

[12] 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.

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

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

[15] 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.

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

[17] 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.

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

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

[20] 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.

[21] D. Zhu and H. Xi. Safe programming with pointers through stateful views. In *Proc. 7th Int. Symp. Practical Aspects of Declarative Languages*. Springer-Verlag LNCS vol. 3350, 2005.