An Overview of the Jahob Analysis System
Project Goals and Current Status

Viktor Kuncak and Martin Rinard
MIT Computer Science and Artificial Intelligence Lab, Cambridge, USA
{vkuncak,rinard}@csail.mit.edu

Abstract

We present an overview of the Jahob system for modular analysis of data structure properties. Jahob uses a subset of Java as the implementation language and annotations with formulas in a subset of Isabelle as the specification language. It uses monadic second-order logic over trees to reason about reachability in linked data structures, the Isabelle theorem prover and Nelson-Oppen style theorem provers to reason about high-level properties and arrays, and a new technique to combine reasoning about constraints on uninterpreted function symbols with other decision procedures. It also incorporates new decision procedures for reasoning about sets with cardinality constraints. The system can infer loop invariants using new symbolic shape analysis. Initial results in the use of our system are promising; we are continuing to develop and evaluate it.

1 Introduction

Complex software systems currently play a crucial role in the management and operation of our society. Moreover, this role will only increase in importance as software becomes even more pervasively deployed across the activities, infrastructure, and devices of our society. Given this central role, software reliability is a critical and increasingly important issue.

The goal of the Jahob project is to increase software reliability by statically verifying that certain classes of errors can never occur. The Jahob system analyzes annotated programs written in a subset of Java. A basic idea behind Jahob is to model the state that the program manipulates (its data structures) as abstract sets of objects and relations between these objects. The program uses these sets and relations to state key data structure consistency constraints that must hold between the data structures. Each method also uses these sets and relations to state its specification, which consists of a precondition and a postcondition. Given the invariants and specifications, the Jahob verifier statically analyzes the program to ensure that 1) it preserves important data structure consistency properties, and 2) each method conforms to its specification. Method specifications connect the actions of the program to the data structures that it manipulates, enabling the verification of properties that relate actions and state.

There are several challenges associated with this effort. First, there must be a verified connection between the concrete data structures that the program manipulates and the sets and relations that the Jahob analyzer operates with. To establish this connection, Jahob program encapsulate their data structures in modules, with each module containing an abstraction function that maps the encapsulated concrete data structure to the corresponding sets and relations.

A second challenge is that the Jahob analyzer is designed to verify extremely precise, detailed properties that are significantly beyond the reach of traditional analyses. Moreover, the range of potential properties to verify is extremely large, making it implausible that any single analysis will be able to verify all properties of interest. The Jahob system is therefore structured to incorporate multiple specialized analyses, each of which is tailored to analyze a targeted class of properties. Together, these analyses are capable of verifying important properties of unprecedented sophistication and importance.

2 Example

In this section we use a simple list example to demonstrate how the Jahob system can verify data structure consistency properties. Figure 1 presents the specification for a standard List class. The running program uses a standard linked list data structure to implement instances of this class (we present and discuss the implementation below). Clients, however, should not be concerned with the details of any particular implementation. The specification of the List class therefore serves as an interface that abstracts away the particular implementation details of the class, leav-
class List
{
    /**: public static specvar content :: objset; */

    public List() /*:
    modifies content
    ensures "content = {}"
*/
    public void add(Object o) /*:
    requires "o ~: content & o ~= null"
    modifies content
    ensures "content = old content Un {o}"
*/
    public boolean empty() /*:
    ensures "result = (content = {})"
*/
    public Object getOne() /*:
    requires "content ~= {}"
    ensures "result : content"
*/
    public void remove(Object o) /*:
    requires "o : content"
    modifies content
    ensures "content = old content - {o}"
*/
}

Figure 1. List Specification

In this case, the List specification uses the abstract specification variable content to hold the set of objects present in the list. This set does not exist when the program runs — it is simply an abstraction that the Jahob program uses to express the specification and that the Jahob verifier uses as it verifies the program.

As Figure 1 shows, Jahob is structured as an annotation language for Java. Jahob annotations appear as comments to the standard Java compiler. It is possible to distinguish Jahob annotations from standard comments by the fact that Jahob annotations all start with either //: or /*: — in other words, they have a ":" after the initial comment token. For example, the first comment in Figure 1 declares the specification variable content (which, as mentioned above, abstracts the contents of the list).

2.1 Method Interfaces

After the declaration of the specvar specification variable, the List specification contains a sequence of method interface declarations. Each declaration may contain a requires clause, which states the precondition of the method; a modifies clause, which states the sets and relations that the method may modify; and an ensures clause, which states the properties that the method guarantees will hold when it returns, assuming that the precondition held when it was invoked.

The List constructor List(), for example, modifies the content specification variable — specifically, it ensures that the content specification variable is empty when it constructs the List.

Note that the program may invoke the List constructor multiple times to construct many different lists. According to the semantics of Jahob, each instantiation has its own specification variable content. It is therefore possible to write specifications that relate different instances of the specvar variable that come from different instantiations of the List module. One could, for example, state that one instantiation contains a set of objects that is a subset of another, or that two lists contain disjoint objects.

We next consider the interface for the add(o) method, which adds the object o to the list. Here the requires clause states that o must not already be in the list (o ~: content) and that o must not be null (o~=null). As this example illustrates, developers can use boolean combinations of clauses in the requires and ensures clauses.

The ensures clause of the add method uses the Un (union) operator to state that the effect of the add
method is to add the object $o$ to the set of objects $\text{content}$ already in the list. The remaining methods ($\text{empty}$, $\text{getOne}$, and $\text{remove}$) similarly use $\text{requires}$, $\text{modifies}$, and $\text{ensures}$ clauses to specify their interfaces.

2.2 List Client

We next show how a client can instantiate the $\text{List}$ class to obtain multiple $\text{List}$ instances, specify invariants involving these instances, manipulate the lists, and use the Jahob system to verify that the program correctly respects the invariants. The $\text{Client}$ class in Figure 2 creates two lists ($a$ and $b$), adds some objects to these lists, then moves all of the elements from $a$ into $b$.

The key invariant in this example is that the sets of elements in the two lists are disjoint and remain disjoint throughout all of the manipulations of the client. There is, however, a technical detail that somewhat complicates the expression of this invariant. Specifically, before the client is instantiated, the lists do not exist. It therefore does not make sense to express the invariant directly as holding whenever the program executes. Instead, the $\text{Client}$ uses the boolean $\text{init}$ specification variable to state that the invariant holds whenever the $\text{Client}$ exists. The invariant in Figure 2 also states that, once the $\text{Client}$ has been initialized, that $a$ and $b$ are not null.

2.3 List Implementation

We next discuss the implementation of the $\text{List}$ class. There are two key considerations: 1) implementing the $\text{List}$ methods in Java, and 2) establishing the connection between the $\text{List}$’s Java data structures and the abstract specification variables used to specify the $\text{List}$ interface. Figure 3 presents the state of the $\text{List}$.

The implementation uses the variable $\text{first}$ to refer to the first $\text{Node}$ in the list. Each $\text{Node}$ object has a field $\text{next}$ that contains a reference to the next object in the list and a field $\text{data}$ that contains a reference to the object in the list. The private specification variable $\text{nodes}$ is the set of all $\text{Nodes}$ that is reachable by following $\text{next}$ references starting from the $\text{first}$ variable.

The Jahob specification uses an abstraction function to define the contents of the $\text{nodes}$ set. This abstraction function consists of a set comprehension that states that $\text{nodes}$ is the set of all objects $n$ in the reflexive, transitive closure of the $\text{next}$ relation on $\text{Node}$ objects starting with $\text{first}$. The specification can then use the $\text{node}$ set to define the $\text{content}$ set as the set of all objects which objects in the $\text{node}$ set reference. This definition uses the existential quantifier $\text{EX}$ in its set comprehension.

Note that these abstraction functions directly reference implementation entities ($\text{first}$, $\text{next}$) to define the sets $\text{nodes}$ and $\text{contents}$ in terms of the state that the implementation uses to represent the list. The abstraction functions therefore establish a formal connection between the concrete implementation state and the abstract specification state. This connection allows the the Jahob verifier to start with facts that have been established by reasoning about the abstract state and conclude facts that are valid about the concrete state of the program as it is running.

In our example, the Jahob verification of the disjointness of the two $\text{List contents}$ sets in the client in Figure 2, in combination with the abstraction function, enables the Jahob verifier to conclude that the concrete lists are disjoint as well. Of course, this verification also depends on the verification that the list methods correctly implement their interfaces. For this verification to succeed, the concrete data structures must satisfy several additional invariants. Figure 3 presents these properties — specifically, the list must be acyclic with no sharing of sublists, no node in the list refers to the $\text{first}$ node, and the data references are not shared.

```java
public List() { }

public void add(Object o) {
    Node n = new Node();
    n.data = o;
    n.next = first;
    first = n;
}

public boolean empty() {
    return (first==null);
}

public Object getOne() {
    return first.data;
}

public void remove (Object o) {
    if (first!=null) {
        if (first.data==o) {
            first = first.next;
        } else {
            Node prev = first;
            Node current = first.next;
            boolean go = true;
            while (go && (current!=null)) {
                if (current.data==o) {
                    prev.next = current.next;
                    go = false;
                }
                current = current.next;
            }
        }
    }
}

Figure 4. List Implementation Methods
```
class List {
private Node first;
/*:
// representation nodes:
specvar nodes :: objset;
private vardefs "nodes == {n. n ~= null & rtrancl_pt (% x y. x..Node.next = y) first n}"

// list content:
public specvar content :: objset;
private vardefs "content == {x. EX n. x = n..Node.data & n : nodes}"

// next is acyclic and unshared:
invariant "tree [List.first, Node.next]"

// 'first' is the beginning of the list:
invariant "first = null |
  (first : Object.alloc &
  (ALL n. n..Node.next ~= first &
   (n ~= this --> n..List.first ~= first)))"

// no sharing of data:
invariant "ALL n1 n2. n1 : nodes & n2 : nodes & n1..Node.data = n2..Node.data --> n1=n2";
*/
}
class Node {
  public /*: claimedby List */ Object data;
  public /*: claimedby List */ Node next;
}

Figure 3. List Implementation State and Invariants

Figure 4 presents the implementation of the List methods. These methods provide a standard list implementation. They manipulate only the concrete data structures that make up the list. The Jahob verifier must check that, given the definition of the abstract content set in Figure 3 and the method interfaces in Figure 1 (which provide the interfaces in terms of the abstract content set), that the method implementations in Figure 4 correctly implement the abstract method interfaces in Figure 1.

2.4 Verification

A key verification challenge is that there are an enormous number of possible data structures, many of which may require specialized verification strategies. It is therefore difficult to imagine that any single verification algorithm could successfully verify all data structure implementations. In our example, the verification of the List implementation involves detailed reasoning about the references in the implementation. Other programs may use array-based data structures such as hash tables that produce very different verification conditions. The Jahob framework is therefore set up as a verification condition generator that can invoke any one of a number of decision procedures to discharge the proof obligations provided by the verification condition generator. By populating Jahob with a variety of decision procedures, each of which may be specialized to the verification conditions that arise in the analysis of different data structures or clients, Jahob can effectively deploy very specialized, even unscalable, techniques to verify the full range of data structure implementations and clients.

One issue that arises in the generation of the verification conditions is loop invariants. The current verification condition generator is able to exploit the availability of explicitly-provided loop invariants for complex code. It is also able to leverage loop invariant inference engines, including speculative engines that may generate incorrect loop invariants. Any incorrect loop invariants would be detected and rejected during the verification condition analysis.

In our example, the verification condition generator analyzes each method in turn. It appropriately augments the requires and ensures clauses with the specified invariants to ensure that the methods preserve them. The verification conditions for the data structure implementation could be verified, for example, by a combination of field constraint analysis [80] and the MONA decision procedure [40]. Loop invariants could be provided explicitly or inferred by symbolic shape analysis [80, 65, 79].

The verification conditions for the client could be discharged by a decision procedure specialized for reasoning about membership changes in abstract sets of
objects. It is also possible in many cases to use off-the-shelf automated theorem provers [78] to discharge these kinds of verification conditions.

3 Status

We have implemented the Jahob framework, populated it with interfaces to the Isabelle interactive theorem prover [63], the SMT-LIB interface [67] to Nelson-Oppen style [62] theorem provers, the MONA decision procedure [40], and a decision procedure for Boolean Algebra with Presburger Arithmetic [43] based on reduction to the Omega decision procedure [66] for Presburger arithmetic. We are using a simple goal decomposition technique to prove different conjuncts in the goal using different decision procedures. In addition, we are using field constraint analysis [80] to combine reasoning about uninterpreted function symbols with reasoning using other decision procedures.

We have verified implementations and uses of global data structures. By providing intermediate assertions we have verified implementations of operations on association lists. We have also annotated and partially verified high-level properties in an implementation of a turn-based strategy game. We have also implemented a mechanism for reasoning about data structure representation in the presence of dynamic data structure instantiation, combining the ideas from the Hob project [47] with approaches from systems such as Spec# [6]. We are currently evaluating the practicality of our approach.

4 Related Work

Key features of Jahob system are modular reasoning with expressive procedure contracts and support for data abstraction, and automated support for reasoning about linked data structure implementation and usage. Jahob therefore builds on program verification research to provide a framework for modular analysis, and builds on new analyses for data structure implementation and data structure use to provide a higher degree of automation than verification frameworks based on general-purpose reasoning.

Verification systems with modular reasoning. Systems based on verification-condition generation and theorem proving include the program verifier [39], the interactive program verifier [17], the Stanford Pascal Verifier [74, 60], the Gypsy environment [28], Larch [30], ESC/Modula-3 [16], ESC/Java [22], ESC/Java2 [12], Boogie [6], Krakatoa [55], KeY [3], as well as more general frameworks such as ACL2 [38, 59], and SteP [8], and PVS [64]. Traditionally, these systems are based on verification condition generation combined with theorem provers. They typically require loop invariants, and additionally either require either interaction with the theorem prover or lemmas specific for the program being verified. Specification frameworks include Z [81], VDM [36], B [2], RAISE [13]. Many of these frameworks recognize the importance of data abstraction [36], which is an important component of Jahob. Some of these frameworks provide no automation for performing formal proofs, and some provide support in terms of verification condition generators and interactive theorem provers [1]. Jahob, on the other hand, aims at providing automated proofs that data structures conform to their abstraction; previous approaches have been less ambitious either in terms of automation [36] or in terms of using lighter-weight substitute of specification variables [51].

Recently, verification systems have incorporated techniques for inferring loop invariants [23, 21, 11, 50]. Like more specialized analyses [75, 82, 19, 70, 24], such techniques for loop invariant inference are effective for analyzing simple array data structures and basic memory safety properties, but have so far been limited in the range of properties that they can prove about linked data structures. These systems are compatible with our methodology of combining specialized analyses based on abstract interpretation to increase the automation in the context of a verification framework; one of the properties that makes Jahob different is the ability to utilize recently developed precise data structure analyses such as shape analysis.

Shape analysis. Shape analyses are precise analyses for linked data structures. They were originally used for compiler optimizations [37, 27, 26], but subsequently evolved into more precise analyses that have been successfully used to analyze invariants of data structures that are of interest for verification [42, 25, 41, 49, 58, 71]. Most shape analyses that synthesize loop invariants are based on precomputed transfer functions and a fixed set of properties to be tracked; recent approaches enable automation of such computation using decision procedures [86, 84, 85, 65, 80] or finite differencing [69].

Recently there has been a resurgence of decision procedures and analyses for linked list data structures [4, 18, 54, 7, 68], where the emphasis is on predictability (decision procedures for well-defined classes of properties of linked lists), efficiency (membership in NP), the ability to interoperate with other reasoning procedures, and modularity.

Shape analyses are among the most sophisticated analyses for structural properties of programs; they have also been applied to verify properties such as sorting, by abstracting the ordering relation [53, 58]. Analyses and decision procedures have also been constructed that combine reasoning about reachability and reasoning about quantitative properties such as length of lists and height and balancing of trees [32, 31, 45, 49, 58, 71].
New logics were recently proposed for reasoning about reachability, such as the logic of reachable shapes [83]. Existing logics, such as guarded fixpoint logic [29] and description logics with reachability [10] are attractive because of their expressive power, but so far no decision procedures for these logics have been implemented. Automated theorem provers such as Vampire [78] can be used to reason about properties of linked data structures, but axiomatizing reachability in first-order logic is non-trivial in practice [61, 52] and not possible in general.

Software Model Checking. Recent trends indicate the convergence of shape analysis with predicate abstraction [5, 33], with a spectrum of increasingly complex domains ranging from propositional combinations of predicates [5], through quantified propositional combinations [23], indexed predicates [44], to symbolic shape analysis [80, 65, 79]. The field remains an active area of research, with different approaches demonstrating different precision/efficiency/automation tradeoffs.

Typstate systems. Because many precise analysis approaches are difficult to scale, it is important to be able to combine them with more scalable analyses. Jahob uses expressive procedure interfaces to achieve such a combination, which means that scalable analyses must be able to communicate using procedure interfaces. Typstate analyses have emerged as data-flow analyses that take into account user-supplied interfaces [73, 14, 15]. In the Hob project [48, 87, 46] we have demonstrated that a combination of typstate analysis with shape analysis is feasible when interfaces use abstract sets to abstract global data structures. One of the goals of Jahob is to demonstrate that such an approach is feasible for a more general class of procedure interfaces that involve not only sets, but also relations.

Bug finding tools for complex properties. Given that many verification attempts demonstrate bugs in specifications or code, it is useful to supplement verification tools with bug finding tools. Finite model checkers such as the Alloy Analyzer [34] can be used to find bugs in code that manipulates linked data structures [35, 76]. Explicit state model checking and testing approaches can also be effective for this purpose [20, 56, 72, 77]. Although somewhat orthogonal to verification, bug finding can be combined with verification in productive ways, and we may consider such combinations in the future.

5 Conclusion

Software reliability is an increasingly important concern for our society. The automatic verification of program properties promises to address this concern by eliminating potential sources of software errors. The Jahob project focuses on data structure consistency properties and connections between the actions of the program and the effect that these actions have on the state. The combination of a general verification condition generator and an architecture that supports the integration of multiple specialized analyses is designed to enable the verification of properties of unprecedented precision.

Acknowledgements. We thank Thomas Wies, Karen Zee, Peter Schmitt, and Hai Huu Nguyen for contributions to the Jahob project.

References
