Hob: A Tool for Verifying Data Structure Consistency

Patrick Lam, Viktor Kuncak, and Martin Rinard

Computer Science and Artificial Intelligence Laboratory,
Massachusetts Institute of Technology
{plam, vkuncak, rinard}@csail.mit.edu

Abstract. This tool demonstration presents Hob, a system for verifying data structure consistency for programs written in a general-purpose programming language. Our tool enables the focused application of multiple communicating static analyses to different modules in the same program. Using our tool throughout the program development process, we have successfully identified several bugs in both specifications and implementations of programs.

1 Introduction

Hob is a static analysis framework that verifies that program implementations satisfy their specifications. Using Hob, developers can apply multiple pluggable analyses to different parts of a program, applying each analysis to the modules for which it is most appropriate. Each Hob analysis plugin verifies that program modules 1) properly implement their specifications; and 2) respect the preconditions of the procedures that they call. Program modules often encapsulate data structures, and many data structures maintain a dynamically changing set of objects as their primary purpose; we have therefore found that set specifications allow developers to express crucial data structure interface properties, including in particular, the preconditions needed by typical data structure operations to successfully execute. Hob's common set specification language therefore enables different analyses to effectively communicate with each other.

The Hob project addresses the program verification problem [1, 5]. Our tool supports assume/guarantee reasoning and data refinement. The techniques embodied in the Hob tool are particularly suited for expressing and verifying data structure consistency properties: Hob allows static analysis plugins to verify that data structure preconditions hold upon entry to a data structure, that data structure operations preserve data structure invariants, and that data structure operations conform to their specifications.

Our technique is designed to support programs that encapsulate the implementations of complex data structures in instantiatable leaf modules, with these modules analyzed once by very precise, potentially expensive analyses (such as shape analyses or even analyses that generate verification conditions that must be manually discharged using a theorem prover or proof checker). The rest of the program uses these modules but does not directly manipulate the encapsulated data structures. These modules can then be analyzed by more efficient analyses that operate primarily at the level of the common set abstraction. Given the scalability issues associated with precise data structure verification techniques, this kind of approach is the only way to make these analyses viable in practice.

R. Bodik (Ed.): CC 2005, LNCS 3443, pp. 237–241, 2005.

We have implemented our analysis framework and populated this framework with three analysis plugins: 1) the flags plugin, which is designed to analyze modules that use a flag field to indicate the typestate of the objects that they manipulate [3]; 2) the PALE plugin, which implements a shape analysis for linked data structures (we integrated the Pointer Analysis Logic Engine analysis tool [4] into our system); and 3) the theorem proving plugin, which generates verification conditions designed to be discharged manually using the Isabelle interactive theorem prover [6]. We have used our analysis framework to analyze several programs; our experience shows that it can effectively 1) verify the consistency of data structures encapsulated within a single module and 2) combine analysis results from different analysis plugins to verify properties involving objects shared by multiple modules analyzed by different analyses. We have observed that our approach reduces the program annotation effort, improves the performance of the resulting analysis, and extends the range of programs to which each component analysis is applicable in isolation.

2 The Hob Approach

We next describe how developers write implementations and specifications for Hob. A program to be analyzed contains a number of modules. Each module is analyzed by an analysis plugin; plugins ensure that the module's implementation conforms to its specification and that the module satisfies all preconditions for the calls that it makes.

2.1 How Analysis Plugins Work

The basic task of an analysis plugin is to certify that the implementation for a module conforms to its specification and that the module meets all preconditions for calls that it makes. Implementation sections for modules in our system are written in a standard Java-like memory-safe imperative language supporting arrays and dynamic object allocation. Module specification sections give preconditions and postconditions for procedures in the boolean algebra of sets; these conditions are augmented with a `modifies` clause, which states the frame condition for the procedure. Specification modules may also name global boolean predicates to be tracked by the analysis. Finally, since modules may implement their specifications in a variety of ways, the abstraction section of a module describes the relationship between the module's implementation and its specification; each analysis plugin has a specialized syntax for abstraction settings, suitable for the type of properties checked by that plugin. An abstraction section may additionally state representation invariants applicable to the data structure implemented in that module.

In general, the Hob system analyzes individual modules as follows. For each module, Hob examines the implementation, specification, and abstraction sections of that module, as well as the specifications of all procedures that the module invokes. Hob first uses the abstraction function (from the abstraction section) to translate the `requires` and `ensures` clauses into the internal representation of the specialized analysis that will analyze the module (as specified in the abstraction section). Hob then conjoins the representation invariant to the translated `requires` and `ensures` clauses. Finally,

Hob invokes the specified analysis plugin to verify that each procedure conforms to its translated `requires` and `ensures` clauses.

2.2 Verifying Cross-Module Properties and Simplifying Specifications

Modules may belong to analysis scopes [2]. A scope encloses a number of program modules and designates a subset of these modules as exported modules; it also states scope invariants that always hold outside the scope. Scopes serve two purposes: they enable the specification and verification of cross-module invariants by identifying the subset of a program in which an invariant is expected to hold, and they combat annotation aggregation by hiding irrelevant sets from callers. Scopes are key to our system's verification of invariants containing sets from different modules: by designating the exported modules as external access points, and because scope invariants are preserved outside a scope, it is sufficient to check the scope invariants upon exit from a scope, therefore reducing the annotation and analysis burden which would otherwise be associated with scope invariants. Scopes also shield callers from irrelevant detail: only sets from exported modules may occur as free variables in specifications for modules in different scopes. This serves to bound the detail required in procedure specifications: the specification of procedure p belonging to scope $\mathcal{C}$ need only contain the effects of procedures on sets in $\mathcal{C}$ and sets belonging to exported modules outside $\mathcal{C}$.

Hob specification sections may also use defaults to simplify procedure preconditions and postconditions. A default is a clause that is automatically conjoined to procedure preconditions and postconditions across a specified program pointcut, unless explicitly suspended. In our example applications, we use defaults for ensuring that initialization predicates hold everywhere in a program except in the initial state; these defaults free the developer from the burden of manually conjoining the initialization predicate to a substantial portion of the program's specifications.

3 Hob in Practice

We have coded up several benchmark programs, using our system during the development of the programs. Our benchmarks include the water scientific computation benchmark, a minesweeper game, and programs with computational patterns from operating-system schedulers, air-traffic control, and program transformation passes. These benchmarks use a variety of data structures, and we have therefore implemented and verified sets, set iterators, queues, stacks, and priority queues. Our implementations range from singly-linked and doubly-linked lists and tree insertion (all verified using the PALE plugin) through array data structures (verified using the theorem proving plugin with the Isabelle theorem prover used to discharge verification conditions); our largest benchmark (water) contains approximately 2000 lines of implementation and 500 lines of specification. The Hob project homepage is

`http://cag.csail.mit.edu/~plam/hob/`

This homepage links to the O'Caml source tarball and publicly readable Subversion repository, further explains our example applications, and includes past presentations about Hob. Hob is distributed under the GNU General Public License.

The Hob infrastructure contains several general components that perform tasks required by all analyses. The implementation language component can parse and typecheck implementation sections. It produces an abstract syntax tree and methods that allow analyses to conveniently access this representation. The specification component can parse and type check specification sections and provides access to the resulting abstract syntax tree. Large parts of abstraction sections are expressed in a language that is specific to each analysis. The abstraction section component parses those parts of the abstraction section syntax that are common to all analyses and uses uninterpreted strings to pass along the analysis-specific parts. Using these components, it is fairly simple to create new analysis plugins and apply them to analyze more types of data structures. Our implementation consists of approximately 10,000 lines of O'Caml code, to which the flag plugin contributes 2000 lines, the PALE plugin another 700 lines, and the theorem proving plugin 1000 lines; the rest of the code is shared analysis infrastructure.

We next present an example of a client code that Hob successfully verifies.

```
impl module UseList {
  format Node {}
  proc use() {
    Node n1;
    Node n2;
    n1 = new Node();
    n2 = new Node();
    List.add(n1);
    List.add(n2);
    List.remove(n2);
    List.remove(n1); } }
```

```
spec module UseList {
  proc use1()
    requires List.Content = {}
    modifies List.Content
    calls List
    ensures List.Content' = {}; }

abst module UseList {
  use plugin "flags"; }
```

This `UseList` example is analyzed by the flags plugin; it uses a `List` module, which is verified by the PALE plugin. Note that the `UseList` module does not define any sets itself; it relies on the `List` module to store its `Node` objects in a linked list. The flags plugin verifies the `use` procedure by propagating boolean formulas; upon procedure entry, the `Content` set from list is assumed to be empty (this condition is verified in all callers of `use`.) After the pair of `List.add` operations completes, the `Content` set is known to contain the elements {`n1, n2`} (by incorporating the postcondition of `List.add`). Finally, the pair of `List.remove` operations ensures that `Content` is empty at the end of the procedure, ensuring the stated procedure postcondition.

4 Conclusion

The program analysis community has produced many precise analyses that are capable of extracting or verifying quite sophisticated data structure properties. Issues associated with using these analyses include scalability limitations and the diversity of important data structure properties, some of which will inevitably elude any single analysis.

The Hob tool can apply a full range of analyses to programs composed of multiple modules. The key elements of the Hob approach include modules that encapsulate object fields and data structure implementations, specifications based on membership in abstract

sets, and invariants that use these sets to express (and enable the verification of) properties that involve multiple data structures in multiple modules analyzed by different analyses. We anticipate that our techniques will enable the productive application of a variety of precise analyses to verify important data structure consistency properties and check important typestate properties in programs built out of multiple modules.

References

1. C. A. R. Hoare. The verifying compiler: still a Grand Challenge for computing research. ETAPS Invited Lecture, April 2003.
2. P. Lam, V. Kuncak, and M. Rinard. Crosscutting techniques in program specification and analysis. In P. Tarr, editor, *Proceedings of the Fourth Conference on Aspect-Oriented Software Development*, 2005.
3. P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking for data structure consistency. In *6th International Conference on Verification, Model Checking and Abstract Interpretation*, 2005.
4. A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. PLDI*, 2001.
5. G. Nelson. Techniques for program verification. Technical report, XEROX Palo Alto Research Center, 1981.
6. K. Zee, P. Lam, V. Kuncak, and M. Rinard. Combining theorem proving with static analysis for data structure consistency. In *International Workshop on Software Verification and Validation (SVV 2004)*, Seattle, November 2004.