Combining Theorem Proving with Static Analysis for Data Structure Consistency

Viktor Kuncak, Patrick Lam, Karen Zee, and Martin Rinard
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
{vkuncak,plam,kkz,rinard}@csail.mit.edu

Abstract

We describe an approach for combining theorem proving techniques with static analysis to analyze data structure consistency for programs that manipulate heterogenous data structures. Our system uses interactive theorem proving and shape analysis to verify that data structure implementations conform to set interfaces. A simpler static analysis uses the verified set interfaces to verify properties that characterize how shared objects participate in multiple data structures. We have succesfully applied this technique to several programs and found that we were able to effectively use theorem proving within circumscribed regions of the program to enable the verification, with an appropriate amount of effort, of large-scale program properties.

1 Introduction

A data structure is consistent if it satisfies the invariants necessary for the normal operation of the program. Data structure consistency is important for successful program execution—if an error corrupts a program's data structures, the program can quickly exhibit unacceptable behavior and may crash. Motivated by the importance of this problem, researchers have developed algorithms for verifying that programs preserve important consistency properties [3, 10, 12, 25, 31, 33].

Ensuring data structure consistency in general is a very difficult problem, because each class of data structures potentially requires reasoning techniques specific to that class. As a result, systems for reasoning about data structure consistency face a perpetual tradeoff between generality and automation.

On the one hand, static analysis techniques have proven to be successful in achieving high levels of automation for verifying the consistency of some important classes of data structures such as linked lists, trees, and graphs [12,14,25,31]. Unfortunately, while effective for data structures within their targeted class, these tools are necessarily either incomplete or unsound for many other data structures. Data structure diversity presents a real problem for verifying data structure consistency of non-trivial applications, which typically contain a range of different kinds of data structures.

On the other hand, theorem proving techniques can in principle verify arbitrarily complicated consistency properties; this statement especially applies to interactive theorem provers such as Isabelle [29] and Athena [2] that allow writing general mathematical statements about program state. The difficulty in using theorem proving tools is that their application may require manual effort and familiarity with their behavior.

The Hob project. Our Hob project [22,23] addresses the difficulties of verifying programs with heterogeneous data structures by applying different verification techniques to different regions of the program. Each verification technique is implemented in an analysis plugin; the Hob system applies a plugin to the appropriate region of the program to verify that it conforms to its interface. The verified interface information enables the different analyses to share information and effectively interoperate. This approach enables us to apply the most expensive analyses, such as interactive theorem proving and shape analysis, to only the most intricate sections of the program, with more automated techniques verifying the rest of the program.

The Hob system currently contains three analysis plugins: 1) the flags plugin, which analyzes modules that use object flag fields to indicate the typestate of the objects that they manipulate; 2) the PALE plugin, which analyzes linked data structures by interfacing to the PALE tool [25]; and 3) the theorem proving plugin, which verifies arbitrary data structures specified in higher-order logic using the Isabelle theorem prover [29]. We have used Hob to analyze several programs; our experience shows that Hob can effectively 1)

verify the consistency of data structures encapsulated within a module and 2) combine analysis results from different analysis plugins to verify properties involving objects shared by multiple modules analyzed by different analyses.

Theorem proving plugin. The focus of this paper is our experience with using the theorem proving plugin in conjunction with other plugins. We have identified data structures that implement dynamically changing sets as good candidates upon which to focus theorem proving effort. Using the ideas of data refinement [8,16], we can naturally specify the preconditions and postconditions on such data structures using formulas in the boolean algebra of sets. We can verify once and for all that the data structure implementation conforms to its interface; we then use the abstract characterization of data structure operations, represented by its set interface, to reason about data structures in the context of a larger program. In this way, our analysis system hides internal data structure complexity from data structure clients, and amortizes the verification effort over all programs that use the data structure.

Our theorem proving plugin generates verification conditions in classical higher-order logic and uses the Isabelle theorem prover [29] to discharge these verification conditions. Using this technique, we verified implementations of a set in a linear array, as well as a partial specification of a priority queue (heap) implemented as a binary search tree stored in an array. We found that many of the verification conditions arising in these examples are discharged automatically using Isabelle's built-in simplification tactics; other verification conditions require manual intervention. We found the underlying language of the theorem prover to be close to the mathematical language used in informal reasoning and therefore convenient for formalizing fundamental properties of data structures. In addition to verifying particular data structures, we found the theorem proving plugin to be useful in identifying formal reasoning patterns that can lead to the construction of future specialized and more automated plugins.

Paper structure. The rest of this paper is organized as follows. Section 2 introduces the example of a process scheduler that manipulates a priority queue data structure and a linked list data structure. Section 3 outlines the key components of the Hob system. Section 4 presents the theorem proving plugin, and Section 5 summarizes our static analysis plugins. Section 6 presents our experience in using the theorem proving plugins for verifying consistency of data structures using our system. Section 7 presents related work and Section 8 presents concluding remarks.

2 Example

We next present an example program that illustrates how we guarantee data structure consistency properties. Our example program is a process scheduler that maintains a list of running processes and a priority queue of suspended processes. The `wakeUpFirst` procedure selects a process from the priority queue and moves it to the running list; the `suspend` procedure removes the process from the running list and inserts it back into the queue. Our example contains three modules: the running list module, the priority queue module, and the scheduler module. The scheduler module invokes procedures in the running list and priority queue modules; its analysis therefore harnesses the power of the PALE and theorem prover plugins, while using a less powerful (and hence more scalable) analysis to verify scheduler properties.

2.1 Suspended Queue Module

The priority queue module implements a priority queue of suspended processes using a binary heap. Figure 1 presents the skeleton of the `SuspendedQueue` implementation module. This module introduces one field into the `Process` format, namely the `p` field indicating a process' priority.

The priority queue contains three public procedures and several private procedures. The `init` procedure creates the backing array for the queue. The `insert` procedure inserts its parameter `n` as a node into the binary heap with the given priority `p1` (the notation `InQueue'` denotes the new version of `InQueue` after the `add` procedure executes; the unprimed `InQueue` denotes the old version before it executes). The `extractMax` procedure removes the highest-priority element from the heap. Together with the private helper procedures, these procedures implement a standard priority queue; we omit the implementation details.

Figure 2 presents the specification of the `SuspendedQueue` module. The specification summarizes priority queue procedures in terms of the set `InQueue`, which is the set of all `Process` objects stored in the queue: `init` initializes the set to the empty set; `insert(n)` inserts object `n` into the set; and `extractMax` removes an object from the set and returns it as the result.

Figure 3 presents the abstraction module that establishes the connection between the implementation and the specification of the priority queue by defining the set `InQueue` as the set of all objects x contained in the array `c` between indices 1 and s. Our analysis verifies that the implementation procedures preserve the priority queue data structure consistency properties for the

```
impl module SuspendedQueue {
   reference c:Process[];
   var s:int;
   format Process {
      p:int; }

   proc init() { ... }

   proc insert(n:PQEntry;p1:int) { ... }
   proc extractMax() returns n:PQEntry { ... }
   private proc swap(a:int; b:int) { ... }
   private proc bubbleDown(i:int) { ... }
}
```

Figure 1: Skeleton of the Priority Queue Implementation Module

```
spec module SuspendedQueue {
   format Process;
   sets InQueue : Process;

  proc init()
    requires true
    modifies InQueue
    ensures InQueue' = {};

  proc insert(n:Process; p1:int)
    requires not (n in InQueue)
    modifies InQueue
    ensures InQueue' = InQueue + n;

  proc extractMax() returns n:Process
    requires InQueue != {}
    modifies InQueue
    ensures (card(n)=1) & (n in InQueue) & (InQueue' = InQueue - n);
}
```

Figure 2: Priority Queue Specification Module

```
abst module SuspendedQueue {
    use plugin "vcgen";
    InQueue = { x : Process |
                   "exists j. 1 <= j & j <= s & x = c[j]"};

    invariant "0 <= s";
    invariant "forall i. (forall j.
                ((1 <= i) & (i <= s) & (1 <= j) &
                (j <= s) & (c[i] = c[j])) --> (i = j))";
}
```

Figure 3: Priority Queue Abstraction Module

```
spec module RunningList {
   format Process;
   sets InList : Process;

   proc add(p : Process)
     requires not (p in InList)
     modifies InList
     ensures InList' = InList + p;

   proc remove(p : Process)
     requires p in InList
     modifies InList
     ensures InList' = InList - p;  }
```

Figure 4: Running List Specification Module

set `InQueue` defined in this manner.

The Hob analysis system enables the application of an appropriate analysis to verify that the priority queue implementation satisfies its specification. Other modules subsequently use the priority queue specification to reason about the effects of calls to that queue, because all specifications are written in a common specification language. We use the Isabelle theorem prover to show that the priority queue implementation conforms to its specification; Section 6.1 explains how our system constructs that proof.

2.2 Running List Module

Figure 4 presents the specification module for our scheduler's list of running processes. The specification has a single abstract set, `InList`, which contains all of the `Process` objects in the running list. The `requires` clause of the specification of the `add` procedure requires the parameter `p` to not already be in `InList`. The `ensures` clause states that effect of the `add` procedure is to add the parameter `p` to `InList`. The `modifies` clause indicates that the procedure modifies the `InList` set only.

2.3 Scheduler Module

Figure 5 presents the `Scheduler` implementation module. This module contributes a `status` field to `Process` objects; this field is 0 if the process is suspended and 1 if the processes is running, therefore encoding the conceptual state of each process (either running or suspended) and enabling the module to quickly determine the status of a process. The implementation of the `Scheduler` module uses the `RunningList` and `SuspendedQueue` modules to actually store sets of running and suspended processes. Section 2.4 explains how Hob uses scopes to verify that the set of processes stored in the `RunningList` and `SuspendedQueue` modules coincides with the sets of processes with `status` set to 1 or 0, respectively.

The specification module in Figure 6 declares two abstract sets: the `Running` set of running processes and

```
impl module Scheduler {
   format Process { status : int; }

   proc suspend(p : Process) {
     p.status := 0;
     RunningList.remove(p);
     SuspendedQueue.add(p); }

   proc hasSuspended() returns b : boolean {
     b := not SuspendedQueue.isEmpty(); }

   proc wakeUpFirst() {
     p := SuspendedQueue.removeFirst();
     p.status := 1;
     RunningList.add(p); }
}
```

Figure 5: Scheduler Implementation Module

```
spec module Scheduler {
   format Process;
   sets Running, Suspended;

   proc suspend(p : Process)
     requires p in Running
     modifies Running, Suspended
     calls RunningList.remove, SuspendedQueue.add
     ensures Suspended' = Suspended + p and
             Running' = Running - p;

   proc hasSuspended() returns b : boolean
     calls SuspendedQueue.isEmpty
     guarantees b <=> Suspended!={};

   proc wakeUpFirst()
     requires Suspended != {}
     modifies Running, Suspended
     calls SuspendedQueue.removeFirst, RunningList.add
     ensures exists p in Suspended.
               Suspended' = Suspended - p and
               Running' = Running + p;
}
```

Figure 6: Scheduler Specification Module

```
abst module Scheduler {
   use plugin "flags";
   Running   = {x : Process | x.status=1};
   Suspended = {x : Process | x.status=0};
}
```

Figure 7: Scheduler Abstraction Module

the `Suspended` set of suspended processes. These sets correspond to the conceptual states that `Process` objects can be in. The specifications of the procedures (`suspend`, `hasSuspended`, and `wakeUpFirst`) therefore reflect the movement of objects between the various states.

The abstraction module in Figure 7 uses the `status` flag to define the `Running` and `Suspended` sets. The flag plugin [22] uses this abstraction function to verify that the scheduler implementation correctly implements its specification. We designed the flag plugin to be scalable and to operate fully automatically; we target it towards modules which coordinate the actions of worker modules.

When verifying the conformance of the `suspend` procedure, the flag plugin must take into account the effects of the `RunningList.remove` and `SuspendedQueue.add` procedures, which are located in modules outside the `Scheduler` module. It turns out that the relevant modules, `RunningList` and `SuspendedQueue`, are analyzed using entirely different plugins (the PALE plugin and the theorem proving plugin, respectively.) Nevertheless, our flag plugin can take into account the effect of these procedures using their specifications, because these specifications are expressed in the common specification language based on sets.

2.4 Scope Invariants

The process scheduler should satisfy several properties that involve data structures in multiple modules. Specifically, the `Running` set from the scheduler module should contain the same objects as the running list, the `Suspended` set should contain the same objects as the priority queue, and the `Running` and `Suspended` sets should be disjoint.

Note that these properties are legitimately (but temporarily) violated when the scheduler is running as it assigns the `status` flag and calls procedures in the running list and priority queue modules. Note also that there must be some mechanism to prevent external modules from calling running list and priority queue procedures directly without going through the scheduler module — such uncoordinated calls could cause the scheduler data structures to fall out of sync with each other, violating the three properties listed above.

We address these issues with *analysis scopes*; Figure 8 presents the analysis scope for our example. In general, scopes are a collection of modules and invariants; each scope may have private modules and exported modules. The purpose of scopes is to specify invariants that involve multiple modules, specify a policy on when the invariants should hold, and control access to module procedures from outside the scope.

```
scope ProcessScheduler {
   modules Scheduler, RunningList, SuspendedQueue;
   exports Scheduler;
   invariant disjoint(Scheduler.Running, Scheduler.Suspended) &
             (Scheduler.Running = RunningList.InList) &
             (Scheduler.Suspended = SuspendedQueue.InQueue);
}
```

Figure 8: Scope Declarations

Our example scope contains an invariant with three clauses that together express the set equality and disjointness properties discussed above. It also identifies a list of modules within which the invariant may be violated (these include the `Scheduler`, `RunningList`, and `SuspendedQueue` modules). Finally, it exports the `Scheduler` module, indicating that the `RunningList` and `SuspendedQueue` modules can be called only from within the other modules in the scope. `Scheduler` procedures, on the other hand, can be invoked from outside the scope. The flag analysis of the `Scheduler` module assumes the invariant holds at the start of each procedure and must show that it holds at the exit of the procedure.

Note that the flag analysis uses the specifications of the `SuspendedQueue` and `RunningList` modules (which are expressed in terms of abstract sets) to verify the invariant. By encapsulating the complexity of the internal data structure properties inside the relevant modules, our technique enables the use of expensive analyses in those modules that require them, while allowing the use of simpler and faster analyses in the remainder of the program.

3 The Hob Analysis System

We next discuss the basic strategy that we expect analysis plugins to implement, and discuss the tasks that they must perform to verify that each implementation section correctly implements its specification. In general, an analysis plugin must ensure that the implementation of a module conforms to its specification, and that any calls originating in the module it is analyzing satisfy their preconditions.

3.1 Implementation Language

Implementation sections for modules in our system are written in a standard memory-safe imperative language supporting arrays and the dynamic allocation of objects.[1] Analysis plugins use our system's core libraries to easily manipulate abstract syntax trees for this imperative language. Using these libraries, we have implemented an interpreter for our language; it would also be straightforward to write a compiler.

We point out one special feature of our imperative language, which we call *formats*. Formats aid modular reasoning about shared objects by encapsulating fields while allowing modules to share objects. When the program creates an object with format T, the newly-created object contains the fields contributed to format T by all modules in the program [6]. A simple type checker for the implementation language statically ensures that each module accesses only fields that it has contributed to an object. Note that no analysis plugin needs the full layout of an object; it will only need the fields which the module under analysis has contributed to that object.

The implementation language supports (but does not require) assertions and loop invariants, which enable fine-grained communication with the analysis plugin. The syntax of assertions is specific to the analysis plugin used to analyze the module. Assertions are ignored by the implementation language interpreter; once statically verified, they do not affect the run-time behavior of the program.

3.2 Specification Language

Figure 9 presents the syntax for the module specification language. A specification section contains a list of set definitions and procedure specifications, and lists the names of formats used in set definitions and procedure specifications. Set declarations identify the module's abstract sets, while boolean variable declarations identify the module's abstract boolean variables. Each procedure specification contains a `requires`, `modifies`, and `ensures` clause. The `modifies` clause identifies sets whose membership may change as a result of executing the procedure. The `requires` clause identifies the precondition that the procedure requires to execute correctly; the `ensures` clauses identifies the postcondition that the procedure ensures when called in program states that satisfy the `requires` condition. Both `requires` and `ensures` clauses use arbitrary first-order formulas B in the language of boolean algebras extended with cardinality constraints. Specification sections may also contain invariants in the same language; these invariants are automatically conjoined with `requires` and `ensures` clauses of procedures in that module. Free variables of these formulas denote abstract sets declared in specification sections. The expressive power of such formulas is the first-order theory of boolean algebras, which is decidable [19, 24]. The decidability of the specification language ensures that

[1] A formal context-free grammar for our language can be downloaded from our publicly-readable Subversion source code repository at `http://plam.csail.mit.edu/svn/repos/trunk/module-language/formatlanguage.sablecc`.

$$
\begin{array}{rcl}
M & ::= & \textsf{spec module } m\ \{F^*D^*I^*PV^*P^*\} \\
F & ::= & \textsf{format } t^*; \\
PV & ::= & \textsf{predvar } b^*; \\
I & ::= & \textsf{invariant } B; \\
D & ::= & \textsf{sets } S^* : t; \\
P & ::= & \textsf{proc } pn(p_1 : t_1, \ldots, p_n : t_n)[\textsf{returns } r : t] \\
 & & [\textsf{requires } B]\ \ [\textsf{modifies } S^*]\ \ \textsf{ensures } B \\
B & ::= & SE_1 = SE_2 \mid SE_1 \subseteq SE_2 \mid p \text{ in } SE \\
 & \mid & B \wedge B \mid B \vee B \mid \neg B \mid \exists S.B \mid \textsf{card}(SE){=}k \\
SE & ::= & \emptyset \mid [m.]\, S \mid [m.]\, S' \\
 & \mid & SE_1 \cup SE_2 \mid SE_1 \cap SE_2 \mid SE_1 \setminus SE_2 \\
 & \mid & \textsf{disjoint } (S_1, S_2)
\end{array}
$$

Figure 9: Syntax of Module Specification Language

analysis plugins can precisely propagate the specified relations between the abstract sets.

3.3 Analysis Overview

The analysis of a module M is performed by the analysis plugin specified in the abstraction section of module M. The abstraction section of module M establishes the connection between the specification and implementation sections of module M. Each analysis plugin augments the generic syntax of abstraction sections with a plugin-specific *plugin annotation language*. The plugin annotation language is used to define the mapping between the concrete and abstract representations of sets. The abstraction section of module M may additionally state representation invariants for the data structure implementing the abstract sets. The responsibility of each plugin is to guarantee that each procedure satisfies its specification; it may do so by any means practical. The specification of a procedure is derived from the abstract `requires`, `modifies`, and `ensures` clauses using the definitions of abstract sets as well as the representation invariants [21]. We also require that a procedure never violate the preconditions of its callees.

Figure 10 illustrates our analysis of the `Scheduler` module from our example: to ensure that `Scheduler` meets its specification, the flag plugin needs to read the implementation, abstraction and specification sections of the `Scheduler` module and only the specifications from the `SuspendedQueue` and `RunningList` modules.

We have implemented three plugins in our analysis framework: a flags plugin, which assigns set membership based on field values, a PALE plugin, which assigns set membership based on heap reachability, and a theorem proving plugin, which can use theorem proving techniques to verify arbitrary implementations of sets. We describe the theorem proving plugin at the core of this paper in Section 4; in Section 5, we briefly discuss the other two plugins in our system.

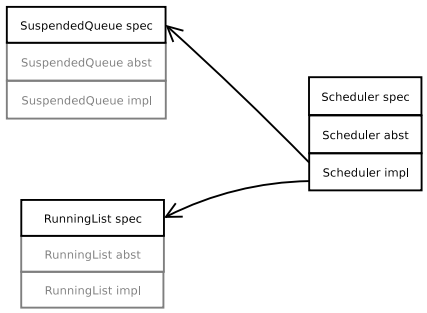

Figure 10: Checking process scheduler implementation

4 The Theorem Proving Plugin

The theorem proving plugin generates verification conditions using weakest preconditions and discharges them using the Isabelle theorem prover. This technique is intended to be used for verifying arbitrarily complicated data structure implementations. The logic for specifying abstraction functions is based on typed set theory and proof obligations can be discharged using automated theorem proving or a proof checker for manually generated proofs, which means that there is no *a priori* bound on the complexity of the data structures (and data structure consistency properties) that can be verified. In our current implementation we have explored this technique for data structures that implement sets by storing objects in global arrays. For example, we have verified the operations on abstract set Content given by an abstraction function

$$Content = \{x | \exists j.\ 0 \leq j \wedge j < s \wedge x \neq \textsf{null} \wedge x = d[j]\}$$

where d is a global array of objects and s is an integer variable indicating the currently used part of the array (see Section 6). We are currently using the Isabelle theorem prover to discharge the generated verification conditions.

Our theorem proving plugin analyzes each procedure independently, showing that it conforms to its specification using the following phases:

1. Concretization: Implicitly conjoin each postcondition with the frame condition derived from modifies clauses. Apply the definitions of sets from the abstraction section to preconditions and postconditions in specification sections, as well as loop invariants and assertions. The result are conditions expressed in terms of the concrete data structure state. For example, the postcondition $\textsf{Content}' = \textsf{Content} - e$ translates into the formula

$$\{x \mid \exists j.\ 0 \leq j \wedge j < s' \ \wedge\ x \neq \textsf{null} \wedge x = d'[j]\} = \{x \mid \exists j.\ 0 \leq j \wedge j < s \ \wedge\ x \neq \textsf{null} \wedge x = d[j]\} - \{e\}$$

2. Representation invariants: Conjoin both precondition and postcondition with representation invariants specified in the abstraction section. In our example we need a representation invariant $0 \leq s$.

3. Statement desugaring: translate statements into a loop-free guarded command language (*e.g.* [13]).

4. Verification condition generation: using weakest precondition semantics, create the formula whose validity implies the conformance of the procedure with respect to its specification.

5. Separation: Separate the verification condition into as many conjuncts as possible by performing a simple non-backtracking natural-deduction search through connectives $\forall$, $\Rightarrow$, $\wedge$.

6. Verification: Attempt to verify each conjunct in turn. Verify if the conjunct is in the library of proved lemmas; if not, attempt to discharge it using the proof hint supplied in procedure code; if no hint is supplied, invoke the Isabelle's built-in simplifier and classical reasoner with array axioms.

In our example, most of the generated verification-condition conjuncts are discharged automatically using array axioms. For the remaining ones, the fully automated verification fails and they are printed as "not known to be true". After interactively proving these difficult cases in Isabelle, they are stored in the library of verified lemmas and subsequent verification attempts pass successfully without assistance.

5 Static Analysis Plugins

This section summarizes two static analysis plugins currently available in the Hob analysis system: the *flag analysis plugin* and the *PALE plugin* (see the report [22] for additional details). These plugins are more automated than the theorem proving plugin, but are limited in the class of modules that they can verify. The flag analysis plugin verifies typestate properties determined by object field values and coordinates the work done by other modules, whereas the PALE plugin uses the PALE off-the-shelf shape analysis tool to verify consistency properties of linked heap data structures.

The flag analysis plugin verifies that modules implement set specifications in which integer or boolean flags indicate abstract set membership. The developer uses abstraction functions to specify the correspondence between the concrete flag values and the abstract sets from the specification, as well as the correspondence between the concrete and abstract boolean variables. The flag analysis plugin then carries out a dataflow analysis over boolean formulas for each procedure, updating set contents after procedure calls and flag field mutations. The flag plugin uses the MONA decision procedure [18] as well as a range of formula simplifications [22] to perform operations on these boolean formulas.

The PALE analysis plugin [25] implements a shape analysis that can verify detailed properties of complex linked data structures. We incorporated the PALE analysis system into our pluggable analysis framework by 1) using abstraction sections to translate our common set-based specifications into PALE specifications, 2) translating statements into the imperative language accepted by PALE, and 3) translating loop invariants into PALE loop invariants.

By combining the PALE plugin with the scalable flag analysis plugin and the powerful theorem proving plugin, the Hob system can analyze programs that simultaneously use heterogenous data structures; we report on our experience with Hob in Section 6.

6 Experience

This section presents our experience in using the Hob system to verify non-trivial data structure consistency in two examples. Section 6.1 presents our experience with the process scheduler example, which we have already introduced in Section 2. Section 6.2 presents our experience with the minesweeper game implementation. In addition to these examples, we ran our analysis on computational patterns from scientific computations, air-traffic control, and program transformation passes with data structures such as singly and doubly linked lists, trees, set iterators, queues, stacks, and priority queues.[2]

The examples we present below illustrate the strengths and the typical usage scenarios of the currently available Hob plugins. The theorem proving plugin successfully analyzed sets defined using membership in an array, which requires abstraction functions that are not expressible in either of the other two plugins. The PALE plugin successfully analyzed linked data structures, which are beyond the reach of the flag analysis module, and which contain heap reachability properties that would likely require many inductive proofs in the theorem proving plugin. Finally, the flag analysis plugin effectively discharged many verification conditions about abstract data structure inclusion, disjointness and equality properties on shared objects, which fall outside the restrictions on the memory model placed by the PALE plugin.

[2]Full source code for Hob and example programs is available at `http://cag.csail.mit.edu/~plam/mpa`. Our Subversion source code repository is also publicly accessible at `http://plam.csail.mit.edu/svn/repos/trunk/module-language`.

6.1 Processs Scheduler and the Priority Queue

Our process scheduler example introduced in Section 2 uses a priority queue implemented as a binary heap [7]. Recall the implementation (Figure 1), specification (Figure 2), and abstraction (Figure 3) modules of the priority queue data structure.

We illustrate the verification of the priority queue through the example of the insert operation in Figure 11. Our theorem proving plugin uses the abstraction function in Figure 3 to concretize the specifications in Figure 2, which are expressed in terms of sets. The plugin conjoins the resulting concrete preconditions and postconditions with the data structure representation invariants in Figure 3, and uses them along with the loop invariant to generate a guarded command language statement.

Computing the weakest precondition of the resulting guarded command statement yields a verification condition, which our system splits into 11 conjuncts. Out of these conjuncts, Isabelle discharges 5 automatically. We proved the remaining 6 conjuncts by hand. We found the proofs to be mostly straightforward, given an intuitive understanding of how the insert operation works. Most of the manual intervention involved specifying the appropriate case splits. Several verification condition conjuncts required showing the equality of set expressions containing set comprehensions. We used the built-in Isabelle tactic (auto) to reduce such expressions to quantifier-free formulas involving array accesses that are discharged using simplification with array axioms. In some cases, it was necessary to indicate instantiations of existential quantifiers in the goal. The proofs also required some basic properties of integer arithmetic such as montonicity of integer division.

In the verification process we discovered a borderline behavior of the `bubbleDown` procedure on queues containing one element, where the procedure is called on a non-existing element. By ruling out such boundary cases by an appropriate precondition, we simplified the reasoning about the correctnes of `bubbleDown` and `extractMax` operations.

We similarly verified several other procedures in the priority queue implementation; we found that the fraction of verification condition conjuncts that must be manually discharged is generally much lower than for the insert procedure.

While the most time-consuming part of the verification was discharging the verification conditions, the most challenging aspect was coming up with appropriate loop invariants. (In that respect, it would be useful to consider techniques for automated loop invariant inference.) On the other hand, we found it relatively straightforward to identify data structure representation invariants. Nevertheless, to control the scope of the verification task, it was important to identify which invariants were necessary for proving the conformance of the implementation with respect to the set interface. For example, the correctness of the `insert` procedure by itself only requires nothing but the simple representation invariant $0 \leq s$. However, verifying the partial correctness of `extractMax` requires the array injectivity invariant stating that all array elements are distinct; this invariant must then be preserved by `insert` as well. Note that the heap ordering condition is not necessary for verifying conformance with respect to the set interface. Although there is no reason why the heap ordering property could not be verified in the theorem proving plugin, it would require more effort to do so. This example illustrates how partial specifications can be useful in showing important data structure consistency properties while reducing the necessary verification effort.

Another way in which our approach simplifies reasoning about data structures is the use of strong enough preconditions. For example, our implementation maintains an invariant that the set of all elements in the array is distinct, which allows the `extractMax` operation to ensure that the removed element is not contained in the resulting set. To preserve this representation invariant, our implementation requires the argument of the `insert` procedure not to be contained in the array. While it may appear difficult to ensure such precondition, our scheduler example in Section 2 successfully ensures precisely this precondition by using a flag field that indicates the membership of an object in the priority queue data structure, and maintaining a global invariant that implies the equality of the set of objects with the particular value of the flag and the set of objects stored in the array. Note that the global invariant is maintained outside the theorem proving plugin, using a more automated flag analysis plugin based on the first-order theory of boolean algebras of sets [22], which illustrates another benefit of the approach of combining analyses in the Hob system.

6.2 Minesweeper

As another example, we implemented and verified the popular minesweeper game in our system. Our minesweeper implementation has several modules (see Figure 12): a game board module (which represents the game state), a controller module (which responds to user input), a view module (which produces the game's output), an exposed cell module (which stores the exposed cells in an array), and an unexposed cell module (which stores the unexposed cells in an instantiated linked list). We created a `Model` scope containing the game board and the exposed and unexposed cell

```
format PQEntry { p:int; }
reference c:PQEntry[];
reference s:int;

proc insert(n:PQEntry;p1:int) {
  n.p = p1;
  s = s + 1;
  int i = s;
  while
    "1 <= i' & i' <= s' &
     PQ = { x. exists j.
               1 <= j & j <= s' & ~(j = i') & x = c'[j] } &
     (forall j. forall k.
       ((1 <= j & j < k & k <= s' & j ~= i' &
        (k ~= i' | i' ~= s')) --> c'[j] ~= c'[k]))"
  (i > 1 && c[i/2].p < p1)
  {
    c[i] = c[i/2];
    i = i / 2;
  }
  c[i] = n;
}
```

Figure 11: Implementation and Loop Invariant for Priority Queue Insertion

modules; this scope encapsulates both the concrete and abstract states of the game board. There are 750 non-blank lines of implementation code in the 6 implementation sections of minesweeper, and 236 non-blank lines in its specification and abstraction sections; the array set accounts for 77 lines of implementation and 35 lines of specification.

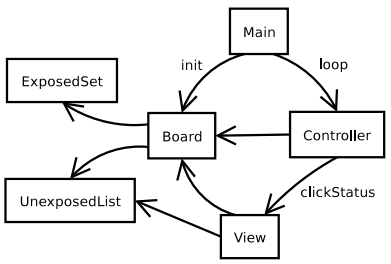

Figure 12: Modules in Minesweeper implementation

The Hob system verifies that our implementation has the following properties (among others):

- Unless the game is over, the set of mined cells is disjoint from the set of exposed cells.
- The sets of exposed and unexposed cells are disjoint.
- At the end of the game, all cells are revealed; *i.e.* the set of unexposed cells is empty.
- The set of unexposed cells maintained in the `Board` module is identical to the set of unexposed cells maintained in the `UnexposedList` list.
- The set of exposed cells maintained in the `Board` module is identical to the set of exposed cells maintained in the `ExposedSet` array.

Note that the truth of several of these properties depends on information obtained from multiple analysis plugins. Here we focus on the `ExposedSet` module, which implements a set of exposed cells using an array; we check the validity of this module using the theorem prover plugin.

Array Set Specification and Implementation. Figure 13 is our specification module for the set encapsulated in the `Arrayset` module. The set interface includes procedures `init`, `add` and `remove`. The `init` procedure guarantees that the set is initialized (represented by the `setInit` boolean predicate) and that the `Content` set is empty upon successful completion. The `add` procedure ensures that the `Content` set contains the element `e` after its execution, while the `remove` procedure ensures that the `Content` set does not contain `e` in the post-state. These specifications allow the clients of the array set to reason about the content of the array in terms of the stored elements, without worrying about the complexity of array index dereferencing.

Figure 15 presents the implementation of the array set data structures in terms of a set, using one global array and an integer indicating the used part of the array. Our implementation of the remove procedure removes all occurences of the element from the array by replacing them with null values. Figure 14 presents the abstraction module that specifies the connection between the specification and the implementation: the content of the set is given by all non-null elements stored in the array.

The theorem proving plugin applies the abstraction function to map the abstract pre and postconditions into concrete pre and postconditions, and conjoins the precondition and the postcondition with the representation invariant, which states that the size of the array is non-negative. These preconditions and postconditions allow the plugin to generate a verification condition that is split into 18 conjuncts that correspond to showing the validity of the loop invariant and the postcondition. Out of these 18 conjuncts, Isabelle discharges 15 fully automatically using the built-in simplifier. We discharged the three remaining conjuncts using a manually constructed sequence of invocations of Isabelle tactics. We similarly proved the correctness of the remaining procedures.

```
spec module Arrayset {
  format Node;

  predvar setInit;

  sets Content : Node;

  proc init()
    requires true
    modifies Content, setInit
    ensures setInit' & card(Content') = 0;

  proc add(e:Node)
    requires setInit & card(e) = 1
    modifies Content
    ensures (Content' = Content + e);

  proc remove(e:Node)
    requires setInit
    modifies Content
    ensures (Content' = Content - e);

  proc contains(e:Node) returns b:bool
    requires setInit & card(e)=1
    ensures b <=> (e in Content);
}
```

Figure 13: Array Set Specification Module

```
abst module Arrayset {
    use plugin "vcgen";
    Content = { x : Node | "x ~= nullObj &
                  (exists j. (0 <= j) & (j < s) & x = d[j])"};
    predvar setInit;

    invariant "0 <= s";
}
```

Figure 14: Array Set Abstraction Module

```
impl module Arrayset {
    format Node {}

    reference d : Node[];  /* array */
    var s : int;  /* array size */

    reference setInit : bool;

    ...

    proc remove(e:Node) {
      int i = 0;
      while "0 <= i' & i' <= s &
             (forall j. (i' <= j & j < s) --> d'[j]=d[j]) &
             ({x. (exists j. 0 <= j & j < i' & x = d'[j])} =
              {x. (exists j. 0 <= j & j < i' & x = d [j])} - e)"
      (i < s) {
        if (d[i] == e) d[i] = null;
        i = i + 1;
      }
    }
    ...
}
```

Figure 15: Array Set Abstraction Module

7 Related Work

We survey related work in shape analysis, program checking tools, theorem provers, and combining decision procedures.

Shape Analysis. The goal of shape analysis is to verify that programs preserve consistency properties of (potentially-recursive) linked data structures. Researchers have developed many shape analyses and the field remains one of the most active areas in program analysis today [20, 25, 31]. These analyses focus on extracting or verifying detailed consistency properties of individual data structures. These analyses are very precise on their domain of applicability, but are forced to make conservative assumptions on programs outside their domain. The Hob framework enables the use of shape analysis techniques in conjunction with theorem proving techniques: using our framework, the developer can select the most appropriate technique for guaranteeing data structure consistency on a per-module basis.

Program Checking Tools. ESC/Java [12] is a program checking tool whose purpose is to identify common errors in programs using program specifications in a subset of the Java Modelling Language [4]. ESC/Java sacrifices soundness in that it does not model all details of the program heap, but can detect some common programming errors. Other tools focus on verifying properties of concurrent programs [1,5] or device drivers [3,15]. One important difference between this research and our research is that our research is designed not to develop a single new analysis algorithm or technique, but rather to enable the application of multiple analyses that check arbitrarily complicated data structure consistency properties within a single program.

Theorem Provers. We use the Isabelle interactive theorem prover [29] to discharge the verification conditions generated by our analysis plugin. Other interactive theorem provers include Athena [2] and HOL [28]. The ACL2 [17] system can apply theorem-proving and term rewriting techniques to verify properties of large-scale systems, among them software systems [26].

Combinations of Decidable Theories. One possible alternative to combining analyses is to use a single analysis engine and combine the decision procedures for different properties using Nelson-Oppen techniques [27,30] and their generalizations such as [34–36]. Theorem provers based on these principles include Simplify [9], Verifun [11], and CVC [32]. Our system can take advantage of combined decision procedures, but also allows specialized analyses that use customized internal representations of dataflow facts.

8 Concluding Remarks

In this paper, we have presented our Hob analysis system, focussing on how Hob combines theorem proving techniques with static analysis. Hob guarantees data structure consistency in programs that manipulate heterogeneous data structures by applying different verification techniques to different regions of the program. We have described our experience in using the theorem proving plugin for verifying array-based data structures. The Hob system enabled us to prove meaningful data structure properties in the context of a larger program. Note that we did not have to use theorem proving techniques for the entire program: Hob's static analysis plugins were able to step in and guarantee data structure consistency for most parts of our example programs.

In verifying data structures using the theorem proving plugin we observed that partial specifications of insert operations require few data structure invariants and can be verified without assuming the usual ordering properties inherent to data structures. In contrast, removal from a data structure often requires the invariant that the data structure elements are distinct (if not ordered), and therefore complicates the verification effort. In some cases, object flags can be attached to objects to indicate the membership in data structures, provided that the appropriate global invariants are verified. In other cases, detecting membership requires knowledge of ordering properties. Even so, many data structure invariants such as balancing affect only the performance, but not the correctness, of data structure operations, which simplifies the verification task.

In the future we expect to add more automation to the theorem proving plugin using Nelson-Oppen decision procedures, [9, 32] while exploiting the special structure of specifications that arise by applying abstraction functions for sets to eliminate explicit reasoning about sets in verification conditions. We are also looking into incorporating additional analysis plugins into Hob, and use further examples to evaluate the effectiveness of our techniques. Furthermore, we are investigating the applicability of our system towards more-automatic verification of file system models; we have previously proved file system model correctness exclusively using theorem provers, [2] and our experience with Hob suggests that its use would greatly simplify the file system correctness proof.

References

[1] Z. M. and. Step: Deductive-algorithmic verification of reactive and real-time systems. In *8th CAV*, volume 1102, pages 415–418, 1996.

[2] K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard. Verifying a file system implementation. In *Sixth International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, Seattle, Nov 8-12, 2004 2004.

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

[4] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. Technical Report NII-R0309, Computing Science Institute, Univ. of Nijmegen, March 2003.

[5] S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *29th ACM SIGPLAN-SIGACT POPL*, pages 45–57. ACM Press, 2002.

[6] D. R. Cheriton and M. E. Wolf. Extensions for multi-module records in conventional programming languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 296–306. ACM Press, 1987.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (Second Edition)*. MIT Press and McGraw-Hill, 2001.

[8] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-oriented proof methods and their comparison*. Cambridge University Press, 1998.

[9] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories Palo Alto, 2003.

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

[11] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explication. In *CAV*, pages 355–367, 2003.

[12] C. Flanagan, K. R. M. Leino, M. Lilibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proc. ACM PLDI*, 2002.

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

[14] P. Fradet and D. L. Métayer. Shape types. In *Proc. 24th ACM POPL*, 1997.

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

[16] H. Jifeng, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *ESOP'86*, volume 213 of *LNCS*, 1986.

[17] M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[18] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.

[19] D. Kozen. Complexity of boolean algebras. *Theoretical Computer Science*, 10:221–247, 1980.

[20] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proc. 29th POPL*, 2002.

[21] P. Lam, V. Kuncak, and M. Rinard. On modular pluggable analyses using set interfaces. Technical Report 933, MIT CSAIL, December 2003.

[22] P. Lam, V. Kuncak, and M. Rinard. On our experience with modular pluggable analyses. Technical Report 965, MIT CSAIL, September 2004.

[23] P. Lam, V. Kuncak, K. Zee, and M. Rinard. The hob project web page. http://catfish.csail.mit.edu/∼plam/mpa/, 2004.

[24] L. Loewenheim. Über mögligkeiten im relativkalkül. *Math. Annalen*, 76:228–251, 1915.

[25] A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. ACM PLDI*, 2001.

[26] J. S. Moore. Proving theorems about Java and the JVM with ACL2, 2002.

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

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

[29] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in LNCS. Springer-Verlag, 1994.

[30] H. Ruess and N. Shankar. Deconstructing shostak. In *Proc. 16th IEEE LICS*, 2001.

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

[32] A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verification*, 2002.

[33] G. Yorsh. Logical characterizations of heap abstractions. Master's thesis, Tel-Aviv University, March 2003.

[34] C. G. Zarba. *The Combination Problem in Automated Reasoning*. PhD thesis, Stanford University, 2004.

[35] C. G. Zarba. Combining sets with elements. In N. Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 762–782. Springer, 2004.

[36] C. G. Zarba. A quantifier elimination algorithm for a fragment of set theory involving the cardinality operator. In *18th International Workshop on Unification*, 2004.