Modular Pluggable Analyses for Data Structure Consistency

Viktor Kuncak, Patrick Lam, Karen Zee, and Martin C. Rinard
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

***Abstract*— Hob is a program analysis system that enables the focused application of multiple analyses to different modules in the same program. In our approach, each module encapsulates one or more data structures and uses membership in abstract sets to characterize how objects participate in data structures. Each analysis verifies that the implementation of the module 1) preserves important internal data structure consistency properties and 2) correctly implements a set algebra interface that characterizes the effects of operations on the data structure. Collectively, the analyses use the set algebra to 1) characterize how objects participate in multiple data structures and to 2) enable the inter-analysis communication required to verify properties that depend on multiple modules analyzed by different analyses.**

We implemented our system and deployed several pluggable analyses, including a flag analysis for modules in which abstract set membership is determined by a flag field in each object, a PALE shape analysis plugin, and a theorem proving plugin for analyzing arbitrarily complicated data structures. Our experience shows that our system 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.

***Index Terms*— Typestate, Data Structure, Invariant, Program Analysis, Program Verification, Shape Analysis, Formal Methods, Programming Language Design**

I. 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 the data structures of a program, the program can quickly exhibit unacceptable behavior and may crash. Motivated by the importance of this problem, researchers have developed algorithms for statically verifying that programs preserve important consistency properties [1]–[8].

However, two problems complicate the successful application of these kinds of analyses to practical programs: *scalability* and *diversity*. Because data structure consistency often involves quite detailed object referencing properties, many analyses fail to scale to the size of the entire program. Because of the vast diversity of data structures, each with its own specific consistency properties, it is difficult to imagine that any one algorithm will be able to successfully analyze all of the data structure manipulation code that may be present in a sizable program.

This paper presents a new perspective on the data structure consistency problem. Instead of attempting to develop a single new algorithm that can analyze some specific set of consistency properties, we propose a technique that developers can use to apply multiple pluggable analyses to the same program, with each analysis applied to the modules for which it is appropriate. Our system uses a range of static analyses to verify various classes of program properties. The analyses use a common abstraction based on sets of objects to communicate their analysis results. Our approach enables the verification of properties that involve multiple objects shared by multiple modules analyzed by different analyses.

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 discharged using an interactive theorem prover). The rest of the program uses these modules but does not directly manipulate the encapsulated data structures. The modules in the rest of the program can then be analyzed using more efficient analyses that operate at the level of the common set abstraction. These analyses can be viewed as generalizations of typestate analysis [9]–[14], with the typestate of an object given by the sets to which the object belongs. These analyses simultaneously 1) ensure that the rest of the program respects the preconditions of the data structure operations (that is, adheres to a protocol that guarantees the correct use of the data structure), and 2) verify high-level consistency properties between data structures, such as disjointness or containment of data structure contents.

We have implemented our analysis framework in the context of the Hob project [15], [16] and initially populated this framework with three analysis plugins: 1) the flag plugin, which is designed to analyze modules that use a flag field to indicate the typestate of the objects that they manipulate [17]; 2) the PALE plugin, which implements shape analysis for linked data structures using the PALE tool [1]; and 3) the theorem proving plugin, which generates verification conditions for consistency properties of arbitrarily complicated properties and discharges them using the Isabelle interactive theorem prover [18]. This paper discusses these three plugins. Thomas Wies has subsequently developed another shape analysis plugin, Bohne [19], [20], which offers more automation and a wider scope of applicability than the PALE plugin.

Our framework analyzes programs written in a memory-safe imperative language with Java-like syntax. We used our analysis framework to analyze several programs; our experience shows that our framework 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.

Structure of the paper and related Hob publications. In the rest of the paper, we present the basic concepts of our system. We use a running example to illustrate these concepts and their effectiveness in verifying complex data structure consistency properties in the context of an application. Additional details on earlier versions of the Hob system are presented in [15], [16]. The theorem proving plugin is presented in [21], the flag plugin in [17], and the field constraint analysis behind the (recently developed) Bohne plugin in [19], [22]. Novel specification-level constructs of Hob—scopes and defaults—are described in [23].

II. Minesweeper Example

To illustrate our technique, we present an example program that implements the popular minesweeper game. The central entity of the implementation is a `Cell` object, which stores the state of one cell in the field of the game. In terms of content, each `Cell` may or may not contain a mine; in terms of visibility, each cell can be exposed or unexposed; finally, the player can mark an unexposed cell if they believe that it contains a mine.

Our implementation uses the standard model-view-controller (MVC) design pattern [24]. The implementation has several modules (see Figure 1). The game board module (`Board`) represents the game state and plays the role of the "model" part of the MVC pattern; the controller module (`Controller`) responds to user input; the view module (`View`) produces the game's output; the exposed cell module (`ExposedSet`) uses an array to store the cells exposed by the player in the course of the current game; and the unexposed cell module (`UnexposedList`) uses an instantiated linked list to store the cells that have not yet been exposed. 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. (Full source code for the minesweeper example and other case studies, the interpreter for our language, and analysis engine are available from the authors' web pages.)

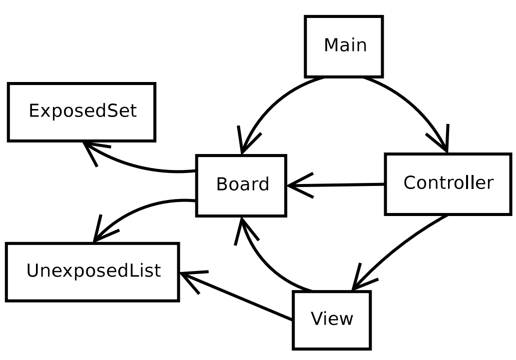

Fig. 1. Modules in Minesweeper implementation.

The minesweeper application exhibits a variety of data structures with a range of important consistency properties. Among the data structure consistency properties verified using our system are the following:

1) The set of unexposed cells in the `UnexposedList` module form an acyclic doubly-linked list with all `prev` references being the inverse of `next` references.
2) The iterator of the `UnexposedList` module is either null or points inside the list.
3) If the board is initialized, then the `ExposedSet` module storing the exposed cells is also initialized.
4) The set of unexposed cells maintained in the `Board` module (using flags) is identical to the set of unexposed cells maintained in the linked `UnexposedList` data structure.
5) The set of exposed cells maintained in the `Board` module (using flags) is identical to the set of exposed cells maintained in the `ExposedSet` array.
6) Unless the game is over, the set of mined cells is disjoint from the set of exposed cells.
7) The sets of exposed and unexposed cells are disjoint.
8) At the end of the game, all cells are revealed; *i.e.* the set of unexposed cells is empty.

Notice that this list contains two kinds of properties: i) data structure consistency properties that involve the implementation of a single data structure, such as Property 1, and ii) more abstract properties involving relationships between objects stored in multiple data structures, such as Properties 4, 5, 6, and 8. One somewhat unusual feature of these abstract properties is that they are outward looking: they capture important features of the system that are directly meaningful to the users of the system, and not just the implementors. To the best of our knowledge, the Hob system is the only currently existing system that supports and promotes the explicit identification and guaranteed checking of these kinds of outward-looking, application-oriented properties.

We next show how our system verifies these properties by combining multiple analyses with different strengths and different levels of automation. We start by describing the main elements of our language using an example of a doubly-linked list with an iterator, which corresponds to the `UnexposedList` module in the minesweeper example.

III. Modules in Hob

The basic unit of analysis in our system is a module. Because developers partition programs into multiple modules, our system can verify different parts of a program independently, using different analysis techniques. Each module in our system consists of an implementation section, a specification section, and an abstraction section. We illustrate different sections of a module using the example of a doubly-linked list with an iterator.

A. Implementation Section

Figure 2 contains a skeleton of the implementation section of a doubly-linked list with an iterator. Our implementation language is a standard memory-safe imperative language with

```
impl module DLLIter {
  format Node { next : Node; prev : Node; }
  var root, current : Node;

  proc isEmpty() returns e:bool {  return root==null; }
  proc add(n : Node) { ... }
  proc remove(n : Node) {
    if (n==current) {  current = current.next;  }
    if (n==root) { root = root.next; }
    Node prv = n.prev, nxt = n.next;
    if (prv!=null) { prv.next = nxt; }
    if (nxt!=null) { nxt.prev = prv; }
    n.next = null; n.prev = null;
  }

  proc openIter() { current = root; }
  proc nextIter() returns n : Node {
    Node n1 = current; current = current.next; return n1;
  }
  proc isLastIter() returns e: bool {return current==null;}
  proc closeIter() {current = null;}
}
```

Fig. 2. Implementation Section of a Doubly Linked List with an Iterator.

dynamically allocated objects. (See Section VI for a discussion of our choice of implementation language.) One interesting feature of our language is the ability to introduce new fields to an object in any module using *formats*. In our doubly-linked list example, the statement `format Node {next: Node; prev: Node;}` directs the compiler to add `next` and `prev` fields to all `Node` objects. These fields are encapsulated within the doubly-linked list module in which they are declared; no other module can access them because our typechecker only allows the use of fields introduced in the current module. The ability to encapsulate fields facilitates the modular analysis of the list by maintaining the encapsulation of the doubly-linked list data structure while still enabling objects in the list to be shared with other modules.

Our doubly-linked list implementation includes the standard `add` and `remove` procedures for a doubly-linked list, as well as an iterator interface, represented by the `openIter`, `nextIter`, `isLastIter`, and `closeIter` procedures. The `openIter` procedure initiates an iteration by setting the `current` reference to the root of the list. We shall see that our specification prevents an iteration from being initiated unless all previous iterations have completed. The `nextIter` procedure advances the `current` pointer to the next element of the list; it iterates through the contents of the linked list, returning each element in sequence. The `isLastIter` procedure indicates to the client when to stop iterating. The `closeIter` procedure terminates an iteration by skipping directly to the end of the list, which allows a new iteration to start. Note that the implementation of the `remove` operation must take into account the existence of an iterator by moving the iterator pointer when the corresponding element is removed.

B. Specification Section

In conventional programming languages, a module interface contains only type declarations that indicate the format of procedure parameters. Type declarations typically do not describe the behavior of procedures, which are usually left to the informal, unchecked documentation of the module. In contrast, the specification section of a module in our language contains a procedure contract for each public procedure of the module.

```
spec module DLLIter {
  format Node;
  specvar Content, Iter : Node set;
  invariant Iter in Content;

  proc isEmpty() returns e:bool
    ensures e' <=> (card(Content') = 0);
  proc add(n : Node)
    requires card(n)=1 & not (n in Content)
    modifies Content
    ensures (Content' = Content + n);
  proc remove(n : Node)
    requires card(n)=1 & (n in Content)
    modifies Content, Iter
    ensures (Content' = Content - n) &
            (Iter' = Iter - n);

  proc openIter()
    requires card(Iter) = 0
    modifies Iter
    ensures (Iter' = Content);
  proc nextIter() returns n : Node
    requires card(Iter)>=1
    modifies Iter
    ensures card(n')=1 & (n' in Iter) &
            (Iter' = Iter - n');
  proc isLastIter() returns e:bool
    ensures not e <=> (card(Iter') >= 1);
  proc closeIter()
    modifies Iter
    ensures card(Iter') = 0;
}
```

Fig. 3. Specification Section of a Doubly Linked List with an Iterator.

Figure 3 presents the specification section for our iterable doubly-linked list module. To describe the behavior of procedures without exposing implementation details, the specification module introduces abstract variables. The abstract variables in our example are the sets `Iter` and `Content`. The set `Iter` represents the set of objects still to be iterated over; this set is a subset of the `Content` set, as indicated by the formula following the `invariant` keyword in Figure 3. The `in` keyword denotes the subset relation on sets. Because our language represents individual elements as singleton sets, `in` also serves as the set membership operator.

Procedure contracts. A procedure contract consists of a `requires` clause that specifies a condition that must hold before calling a procedure, an `ensures` clause that specifies the postcondition that the procedure guarantees, and a `modifies` clause that states the sets that may change during the execution of the procedure. For example, the `add` operation has a precondition that the element being inserted is not in the list already, as given by the conjunct `not (n in Content)`. We represent references to objects as sets of cardinality at most one, with $\emptyset$ denoting a null reference, and a local variable name x denoting a singleton set containing the object referenced by x. In particular, the conjunct `card(n)=1` in the precondition of `add` indicates that the parameter `n` is not `null`.[1] The `ensures` clause can refer to initial variables at procedure entry, with unprimed variables indicating the values at procedure entry and primed variables indicating the values at the end of

[1]The idea of using singleton sets to represent elements appears in decision procedures for several logics for reasoning about sets [25], [26] and is used successfully in the Alloy modelling notation [27].

procedure execution. For example, the notation `Content' = Content + n` indicates that the final version of the `Content` set is equal to the union of the initial version of the set and the newly inserted element `n`. A modifies clause lists all sets that may change during the execution of the procedure and can be thought of as a shorthand for the condition $\bigwedge_{\mathcal{M}} S' = S$ which would otherwise appear in the `ensures` clause, where $\mathcal{M}$ ranges over all sets of the program that are not listed in the `modifies` clause (also known as the frame condition [28]). In our example, the modifies clause of the `add` procedure states that `add` does not change the `Iter` set.

Boolean algebra of sets. Procedure preconditions, postconditions, and invariants are first-order logic formulas in the language of the boolean algebra of sets, which is a decidable theory for reasoning about sets of uninterpreted elements [26], [29]. Formulas in the language of boolean algebra of sets contain set expressions built using set union, intersection, and difference. Atomic formulas in this language can state set inclusion, set equality, as well as cardinality constraints card(S) ρ `k` on sets with constant cardinality bound `k` and some ordering or equality relation $\rho \in \{=, <, \leq, \geq, >\}$. Such atomic formulas can then be combined using arbitrary propositional combinations, as well as quantification over sets.

Specifying an iterable list. Procedure contracts summarize the behavior of the doubly-linked list in terms of abstract sets, and impose constraints on both the clients and the implementation of the `DLLIter` module. Therefore, the contracts of procedures in Figure 3 present the intended use of these procedures. The developer initiates the iteration by calling `openIter`, which initializes `Iter` to contain all members of `Content`. `openIter` requires that `Iter` be empty upon entry, which requires the client to end each iteration before beginning a new iteration. Note that no analysis of the linked list implementation in isolation could ensure this particular precondition: it is the responsibility of the client to ensure that this precondition holds. Subsequent calls to `nextIter` remove an item from `Iter` and return that item, while preserving the underlying `Content` set. The precondition of this procedure requires that `Iter` contain at least one remaining item (`card(Iter)>=1`). The `isLastIter` procedure tests whether any such item exists. By calling the `isLastIter` procedure and testing the result before calling the `nextIter` procedure, the developer can determine if there are any remaining elements and therefore satisfy the precondition of the `nextIter` procedure. Furthermore, iterating until `isLastIter` becomes true ensures that the `Iter` set is empty, which enables the next iteration to begin. Another way to enable subsequent iterations is to call the `closeIter` procedure, which also ensures that the `Iter` set is empty upon exit.

Invariants between sets. The implementation of our iterable doubly-linked list preserves the abstract invariant `Iter in Content`. Hob ensures that this invariant holds throughout the entire program's execution by assuming that the invariant holds upon entry to the module and proving it upon exit. Because the sets `Iter` and `Content` are encapsulated within the `DLLIter` module, showing that the invariant always holds upon exit given that it holds in the initial program state is sufficient to show that the invariant always holds outside the `DLLIter` module. A module entry occurs when a procedure that does not belong to the module calls a procedure that does belong to the module. Conversely, module exit occurs when a procedure inside the module returns control to a procedure outside the module. Note that, together, the invariant `Iter in Content` and the `openIter` and `nextIter` specifications naturally express the essence of iteration over a set.

```
abst module DLLIter {
  use plugin "PALE";
  Content = { n : Node | "root<next*>n" };
  Iter = { n : Node | "current<next*>n" };

  invariant "type Node = {
               data next:Node;
               pointer prev:Node[this^Node.next = {prev}];
             }";
  invariant "data root : Node;";
  invariant "pointer current : Node;";
}
```

Fig. 4. Abstraction Section of a Doubly Linked List with an Iterator.

C. Abstraction Section

Previous sections described how developers write implementations of Hob programs in the implementation sections and how they specify interfaces of operations in the specification sections. Such separation into different sections enables Hob to perform modular analysis of data structure clients. To verify that the data structure implementation itself conforms to its interface, it is necessary to specify the connection between the implementation and the specification. For this purpose, each module in Hob has an *abstraction section*, specifying the abstraction function as well as any additional information needed to verify the implementation. Figure 4 shows the abstraction section for the doubly linked list with an iterator, which connects the specification in Figure 3 with the implementation in Figure 2. The abstraction section specifies the appropriate *analysis plugin* that will analyze the module, the *abstraction function* that gives the values of abstract variables, and the *representation invariants* that should hold inside the module. We next discuss each of these pieces of information in greater detail.

Analysis plugins. The `use plugin` keywords in the abstraction section of a module specify the specialized analysis, or *plugin*, that the system should invoke to verify the module. In the example in Figure 4, the Hob system will invoke the PALE plugin to analyze the iterable list module; this choice of plugin is appropriate because the PALE plugin is specialized for analyzing linked data structures.

All verification in Hob is ultimately performed by the analysis plugins. The Hob system currently contains four analysis plugins, of which we present three that illustrate different target properties and different tradeoffs between expressive power and automation.

- The flag plugin, described in Section IV-C, propagates constraints between sets and tracks the values of constant flags of fields of objects. The flag analysis is an example of a simple, automated analysis that infers loop invariants.

- The PALE shape analysis plugin, described in Section IV-A, verifies properties of tree-like linked data structures using monadic second-order logic. The use of monadic second-order logic enables PALE to verify specifications that contain reachability expressions that are not expressible in first-order logic. PALE requires the use of loop invariants, but the subsequently developed Bohne tool [19], [22] shows how such invariants can often be inferred.
- The theorem proving plugin, described in Section IV-B, illustrates that our approach can verify arbitrarily complex implementations of data structures as long as this complexity is encapsulated within the data structure itself and the (partial) interface can be expressed in the set specification language. The internal language of the theorem proving plugin is the higher-order logic of the Isabelle interactive theorem prover [18] which can express all data structure properties we have encountered. The developer can prove the generated proof obligations interactively using the Isabelle theorem prover, which, given enough manual effort, enables developers to verify arbitrarily complex properties.

Hob supports a loose interaction model between different plugins: each procedure is analyzed by a single plugin. The plugin attempts to establish that the procedure conforms to its specification and reports an error if this is not the case. The analysis plugins in Hob do not directly interact with each other; the interaction is solely through the fact that one plugin shows the correctness of explicitly provided set interfaces that can be used by another plugin. Thanks to this architecture, there are very few requirements on each Hob plugin: each plugin only needs to be able to extract the information from the set interfaces of public methods. The plugin can use arbitrarily expressive logics and data structures to accept from the developer or synthesize automatically properties that are not visible to other modules. Such properties can occur in the form of loop invariants, data structure representation invariants, and the specifications of private procedures; we illustrate such more complex properties below.

Abstraction functions. Analysis plugins establish that the behavior of the implementation is observationally equivalent to the behavior of its specification. To help the analysis plugins in this task and to serve as design documentation, we require developers to specify an abstraction function that maps the state of the implementation to the state of the specification. The abstraction section of a module specifies this abstraction function by defining the meaning of each specification variable in terms of concrete variables. The abstraction section of the iterable list module in Figure 4 defines the set `Content` as the set of all nodes reachable from the root of the doubly-linked list along the `next` field. Namely, we can view the `next` field as a binary relation, so `next*` denotes the transitive closure of `next`, and `root<next*>n` denotes the statement that the transitive closure of `next` holds for the pair (`root`,`n`), which means that `n` is reachable from `root` along `next`. Similarly, the set `Iter` is defined as the set of nodes reachable from the global reference variable `current` that denotes the next element to return from the iterator. Note that these definitions use reachability expressions, whose expressive power is beyond first-order logic. In general, the developer defines the meaning of set variables using a notation specific to one of Hob's analysis plugins; the expressive power of this notation is not limited by the set specification language. This abstraction function allows both the analysis and the developer to interpret procedure contracts in terms of the implementation: replacing the set variables with their definitions results in a contract that refers to the implementation state. Our design supports the development of analysis plugins for verifying arbitrarily complex data structure implementations, while ensuring that such plugins remain capable of communicating with other components of the system through the set specification language.

Representation invariants. Data structures often maintain private *representation invariants* that are true before and after each public data structure operation. Abstraction modules allow the developer to specify representation invariants using a language specific to the analysis plugin. The representation invariants in Figure 4 use the notation of graph types [1] to specify that the linked data structure has the shape of a doubly-linked list with a back pointer. For example, the invariant states that the `next` field is a `data` field, indicating that it is part of the tree backbone of the data structure, and that the `prev` field is an auxiliary `pointer` field that is the inverse of the `next` field.

In general, representation invariants [30], [31] allow the developer to specify data structure consistency properties that are internal to the data structure and are expressed directly in terms of the data structure implementation. In contrast, the `requires` clauses and specification module invariants indicate those preconditions of operations that are expressible solely in terms of abstract specification variables, such as the public invariant `Iter in Content` in Figure 3. Representation invariants are often essential for proving that procedures satisfy their set specifications: for example, the remove operation in Figure 2 would be incorrect without the property that `prev` is the inverse of `next`. However, incorporating such conditions into procedure contracts would violate the data structure abstraction boundary.

Because representation invariants mention concrete variables of a module, they are only visible while analyzing the implementation of a module. A plugin proves the invariant when control leaves the module and in the initial state of the module (the initial state is given by the initial values of variables according to the semantics of our implementation language). Because variables participating in representation invariants are private, outside actions cannot violate the representation invariants. In our example, our system ensures that, in the initial state with no objects, `prev` is the inverse of `next`, which holds trivially, and conjoins the representation invariant to both the precondition and postcondition when verifying that each procedure conforms to its specification.

D. Instantiating and Using Modules

We next illustrate the module instantiation mechanism in Hob, and then present an example of using a module in our system.

Module instantiation. To allow the reuse of modules, our language supports a static instantiation mechanism that introduces a new module into the system by copying an existing module and possibly renaming its types. In our minesweeper example, we use the declaration

```
spec module UnexposedList = List with Node <- Cell;
```

to instantiate `UnexposedList` as a `List` module with the `Node` type replaced by the `Cell` format. Modules generated using instantiation behave no differently from other modules. Our module instantiation mechanism is similar in spirit to SML's functor mechanism, which we discuss further in Section VII.

Verifying correct data structure use. Our minesweeper implementation uses iterators to process the list of unexposed cells in two contexts; both of these contexts are shown in Figure 5. One use of iteration is at the end of the game, at which point the implementation exposes all of the cells. The second use is in a "peek" command, which we added to our minesweeper implementation. The "peek" command allows the player to peek at all unexposed cells. This command is implemented by iterating twice over the set of unexposed cells, first exposing them, then hiding them.[2]

Benefits of set abstraction. Because the clients of the list data structure need not reason about pointers, only abstract sets, it is possible to build more scalable and more automated analyses of clients. In particular, the fragments of quoted text in Figure 5 make up part of the loop invariants for our loops. [3] Our flag analysis plugin can, in fact, infer these loop invariants [32], eliminating the additional annotation burden on the programmer. One reason for the success of our loop invariant inference technique is that it works at the level of abstract set variables.

Note that client code always uses the list through its interface; it cannot directly manipulate the list itself. In general, verifying consistent interface use is simpler than verifying consistency of data structure operations, and our Hob system therefore uses the simpler but more efficient *flag* plugin to verify the consistency of data structure uses. The flag plugin verifies that the precondition for `nextIter` (the `Iter` set is nonempty) is always satisfied before `nextIter` is called. This follows from the fact that `isLastIter` always returns `false` before `nextIter` is called. The `peek` example nondestructively iterates over the `UnexposedList` set without changing the backing `Content` set, whereas the `revealAllUnexposed` procedure removes all elements from the list during iteration. The `revealAllUnexposed` procedure guarantees that

```
// in Board specification

proc setExposed(c:Cell; v:bool) returns causedGameOver:bool
  ...
ensures (v => (ExposedCells' = ExposedCells + c)
         & (UnexposedCells' = UnexposedCells - c)
         & (UnexposedList.Iter' = UnexposedList.Iter - c))
     & ((not v) => ((ExposedCells' = ExposedCells - c)
               & (UnexposedCells' = UnexposedCells + c)))
     & ...

proc revealAllUnexposed()
  requires gameOver
  modifies ExposedCells, UnexposedCells
  ensures card(UnexposedCells') = 0;

// in Board implementation

proc peek() {
  peeking = true;
  Cell c;
  UnexposedList.openIter();
  bool b = UnexposedList.isLastIter();
  while "(b' <=> (UnexposedList.Iter' = {})) & peeking'"
  (!b) {
    c = UnexposedList.nextIter();
    View.drawCellEnd(c);
    b = UnexposedList.isLastIter();
  }
  // ... wait for key press ...
  UnexposedList.openIter();
  b = UnexposedList.isLastIter();
  while "(b' <=> (UnexposedList.Iter' = {})) & peeking'"
   (!b) {
    c = UnexposedList.nextIter();
    View.drawCell(c);
    b = UnexposedList.isLastIter();
  }
  peeking = false;
}

proc revealAllUnexposed() {
  UnexposedList.openIter();
  bool b = UnexposedList.isLastIter();
  // loop invariant in quotes below:
  while "... & (b' <=> (UnexposedList.Iter' = {})) &
   (UnexposedList.Iter' = UnexposedList.Content')" (!b) {
    Cell c = UnexposedList.nextIter();
    setExposed(c, true);
    b = UnexposedList.isLastIter();
  }
}
```

Fig. 5. Doubly-Linked List Client. An optional loop invariant appears in quotes after the `while` keyword.

the unexposed set is empty at the end of the procedure, as follows. The procedure maintains the invariant that the `Iter` set equals the `Content` set during every loop iteration because `nextIter` removes an element from the `Iter` set, and `setExposed` removes the same element from `Content`. The loop exit condition implies that the `Iter` set is empty upon completion, which, in turn, implies that `Content` is empty as well.

Addressing specification aggregation. When analyzing the start of an iteration, which contains a call to the `openIter` procedure, the analysis must show `openIter`'s precondition `card(Iter)=0`. The analysis should be able to use the preconditions of `peek` and `revealAllUnexposed` for this purpose, but because Hob analyzes each procedure in isolation, preconditions such as `card(Iter)=0` would need to propagate into the preconditions and postconditions of procedures that call `peek` and `revealAllUnexposed`. We call this phenomenon *specification aggregation* [23]. We address this prob-

[2] One of the authors successfully used the "peek" command to dramatically improve his minesweeper score.

[3] Our notation of primed and unprimed variables in loop invariants is similar to the convention in postconditions: unprimed variables denote values at procedure entry, whereas primed variables denote current values. In Figure 5, all variables refer to current values; in general, a loop invariant can relate the current state with the state at procedure entry (see Figure 7).

lem by factoring out such global invariants as `card(Iter)=0` that apply to many preconditions and postconditions, and specifying them only once. For this purpose, we introduce the notion of a *scope*, which groups several modules along with invariants on public specification variables of these modules. If the developer specifies `card(Iter)=0` as an invariant of a scope, Hob will implicitly conjoin it to the preconditions and postconditions of public procedures in the scope. This invariant, however, may be violated during iteration over the list, so it must not be conjoined to preconditions of procedures called during the iteration. Therefore, we introduce a guarded version of the invariant into a scope:

```
(not Board.peeking) => (card(UnexposedList.Iter) = 0)
```

As the procedure `peek` illustrates, the program can explicitly set `Board.peeking` to `true` to disable the invariant during the iteration over the list. This reduces the evaluation of `card(Iter)=0` to finding the truth value of the boolean variable `Board.peeking`. To provide fine-grained control over when `Board.peeking` holds, the developer can use the concept of *defaults* [23] to write an expression over program points specifying the preconditions and postconditions where `Board.peeking` holds. In our example, the developer specifies the default `not peeking` with an expression that syntactically identifies the preconditions of both `peek` and `revealAllUnexposed` as the points in the source code where the default applies. The analysis expands the default and conjoins the scope invariant, so both `not peeking` and `(not Board.peeking) => (card(UnexposedList.Iter) = 0)` are part of the precondition of `peek` and `revealAllUnexposed`, which allows the analysis to deduce `card(Iter)=0` and verify these procedures.

Separate verification of data structure use and data structure implementation. Hob's analysis of an implementation proves that each procedure conforms to its specification. This specification is expressed in terms of abstract sets; the concrete meaning of abstract sets is given by the abstraction module, as seen in the linked list example. On the other hand, data structure clients can use a module's specification, as expressed in Hob's set specification language, to reason about the effects of operations and to ensure that a module's preconditions are satisfied when calling into the module; in our example above, the minesweeper board guarantees that the iterator always has at least one iterable element before each call.

In our approach, clients of the linked-list data structure need not be analyzed using the shape analysis plugin. Hob provides another analysis plugin (the flag plugin) that performs a dataflow analysis over set algebra formulas. This plugin is more efficient than the shape analysis plugin, as it tracks less-detailed properties. The availability of additional analysis plugins is crucial in deploying shape analysis techniques in the context of larger programs: our technique for composing analysis plugins allows the focused application of shape analysis to only the relevant module in isolation, while other analysis plugins guarantee that the remainder of the program uses the module correctly.

Two sides of data structure consistency. To clarify the relevance of our approach we next emphasize two equally important components of the data structure consistency problem. Consider an application that manipulates an encapsulated data structure. To ensure that the data structure satisfies consistency properties at run time, we need to ensure both 1) that the data structure operations conform to their contracts, and 2) that the rest of the program invokes data structure operations in states where operation preconditions are satisfied. Without the first condition we cannot say anything about the preservation of our data structure consistency property, and without the second condition we cannot assume that procedure contracts apply. Writing procedure contracts without checking the implementation runs the risk of writing incorrect specifications of procedures that do not correspond to the actual data structure implementation. Conversely, writing contracts without checking their use in the context of a program runs the risk of writing too-strong preconditions that the make the operations impossible to use, or too-weak postconditions that make it impossible to satisfy preconditions of subsequent operation invocations. This is why Hob verifies both components of data structure consistency. It does so using potentially different analyses because these two components are likely to require different precision/scalability trade-offs. In the next section we give an overview of the PALE plugin and the theorem proving plugin as two precise analysis plugins suitable for verifying data structure implementations, and then present a more scalable typestate analysis plugin suitable for verifying data structure use.

IV. Modular Analysis in Hob

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. To analyze a module M, the analysis uses the implementation, specification, and abstraction sections of M, as well as the specification sections of all modules whose procedures M invokes. Apart from these requirements, the details of the analysis are entirely plugin-specific, which gives our system great flexibility in leveraging different analysis techniques.

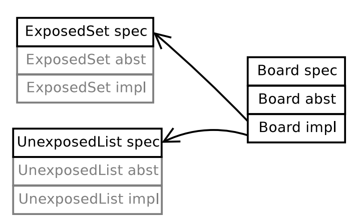

Fig. 6. Checking implementation of minesweeper board.

Figure 6 illustrates our analysis of the `Board` module from minesweeper: to ensure that `Board` meets its specification, we use the flag plugin (described in Section IV-C). Instead of using the implementation of all of the modules in Figure 1, the plugin only needs to read the the specification sections of the `ExposedSet` and `UnexposedList` module in addition to the implementation, abstraction and specification sections of the `Board` module. As a special case, if a module is in the leaf of the call graph, as is the case with the `ExposedSet` or `UnexposedList` modules in Figure 1, then it suffices to examine the implementation, specification, and abstraction sections of that module.

Soundness of modular reasoning. The modular approach of Hob is possible because different modules access disjoint regions of state: a module M directly accesses only the fields declared in M, and the invariants in M depend only on those fields that are declared in M. When the developer instantiates modules, the fields declared in each module instance are distinct from the fields of other instances of the same module. This semantics prevents sharing of fields, while allowing sharing of objects. Arrays in Hob are not objects and have a distinct type, and we impose the following simple rules to ensure the soundness of modular reasoning in their presence: if an abstraction function or a representation invariant depends on an array, then this array must be a global variable initialized by array allocation in the module, and module operations are not allowed to introduce additional references to the array (but they can copy its content into fresh arrays). When an array does not affect the value of a specification variable, then there are no restrictions on its use.

We next describe three of the analysis plugins that we have implemented in our analysis framework. These plugins enabled us to modularly verify data structure consistency properties in our example programs.

A. The PALE Analysis Plugin

Our PALE plugin uses a previously implemented tool, the Pointer Assertion Logic Engine [1] (PALE). We incorporated PALE into our framework with very few changes to the tool itself: we reported a few bugs in the underlying MONA tool and modified PALE to return different exit codes on success and failure.

The PALE analysis system takes as input a program written in the PALE imperative language [1]. This program includes preconditions, postconditions, loop invariants, and *graph type* declarations [33]. A graph type is a tree-like pointer-based (potentially recursive) data structure with a distinguished set of *data fields* (such as the `next` field in Figure 4), whose values form the spanning tree *backbone* of the data structure. In addition to data fields, a graph type may contain *routing fields* [33] (such as the `prev` field in Figure 4). These routing fields are functionally determined by the backbone; the `prev` field in Figure 4, for example, is uniquely determined as the inverse of the `next` field. By identifying data fields that form the spanning tree and by providing the definitions for the derived fields, graph type declarations allow the developer to specify the representation invariants that the data structures must satisfy.

PALE preconditions, postconditions, and loop invariants are formulas written in monadic second-order logic (MSOL), which is decidable [34]. Our use of MSOL allows the use of transitive closure over object reference fields to identify the set of all objects that participate in that data structure. This feature of MSOL makes it more appropriate for verifying linked data structures than first-order logic, which cannot express reachability properties. The PALE analysis system translates an input program into a collection of verification conditions whose validity guarantees that the procedures in the program satisfy their precondition/loop invariant/postcondition relationships. These verification conditions are formulas in MSOL, and the PALE system then uses the MONA decision procedure [35], [36] to determine the validity of these verification conditions. If all of these conditions are valid, the program satisfies its PALE specification.

Using the PALE plugin. As illustrated in Section III, the developer specifies the abstraction function for a data structure verified by PALE by defining the content of an abstract set using a formula in monadic second-order logic.

The developer specifies the representation invariants for the PALE plugin using `invariant` declarations in the abstraction section. An invariant for the PALE plugin can be a graph type definition, such as the definition of the `Node` graph type in Figure 4; the declaration of a routing restriction for the backbone of the data structure, such as the declaration `data root: Node`; or the declaration of a non-backbone routing restriction, such as the declaration `pointer current: Node`.

These representation invariants impose the following constraint on the heap: each object is either 1) a *member* of the data structure or 2) an object *external* to the data structure. Each member object is reachable from the data structure root along specially-marked *data* fields (denoted by the `data` keyword). In addition to data fields, a member object may have routing fields (denoted by the `pointer` keyword) whose value is given by the formula specified in the graph type definition. On the other hand, each external object is unreachable from the data structure root, and all of its fields that are declared in the analyzed module are `null`.

The member/external constraint applies to the projection of the heap onto the fields declared in the currently analyzed module. The constraint does not apply to fields declared in other modules, enabling objects to simultaneously participate in multiple data structures.

The Hob PALE plugin invokes the external PALE tool to enforce this constraint throughout the procedure, with the exception of points in the interior of a basic block. These interior points may violate the constraint, provided that they reestablish the constraint by the end of the basic block.

Translation to PALE Input Language. 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. (The Hob system accepts loop invariants in the form of quoted strings embedded into implementation sections). The loop invariants in implementation modules verified by the PALE plugin contain two parts. The first part contains concrete data structure properties, and is written directly in the PALE specification language. The second part contains abstract set properties, and the PALE plugin translates this part in the same way as `requires` and `ensures` clauses. Our translation also elides scalar variables (which are not supported by PALE) from the input program.

A sketch of the translation follows. For each set definition of the form

$$S = \{x : T \mid F(x)\}$$

that appears in the abstraction section, the translator produces a second-order predicate of the following form that takes a set as an argument:

```
isS(set S:T) = allpos x of T: x in S <=> F(x)
```

The `isS` predicate therefore selects `S` to be the set of objects that satisfy predicate `F`. Together, the `isS` predicates enable us to interpret a formula $B(S_1, \dots, S_n)$ in the boolean algebra of sets as the second-order formula

$$\exists S_1, \dots, S_n. \bigwedge_{i=1}^{n} \mathtt{isS}_i(S_i) \ \wedge\ B(S_1, \dots, S_n).$$

In this way, current values of abstract set variables in loop invariants and postconditions are effectively replaced by their definitions, using the fact that an expression $E(\{x \mid P(x)\})$ is equivalent to the expression $\exists S.\ S = \{x \mid P(x)\} \wedge E(S)$.

The predicates $\mathtt{isS}_i$ enable the PALE plugin to translate procedure specifications. For instance, the `remove` procedure whose implementation is in Figure 2 and whose contract is in Figure 3 gives PALE code of the form:

$\mathtt{set}$ Content : Node;
$\mathtt{set}$ Iter : Node;
$/ * \mathtt{precondition} * /$
$[\mathtt{isContent}(\mathsf{Content}) \ \wedge\ \mathtt{isIter}(\mathsf{Iter}) \ \wedge$
$n \in \mathsf{Content} \ \wedge\ \mathsf{Iter} \subseteq \mathsf{Content}]$

{stmts}

$/ * \mathtt{postcondition} * /$
$[\ \mathtt{existset}\ \mathsf{Content}'\ \mathtt{of}\ \mathsf{Node} : \mathtt{isContent}(\mathsf{Content}') \ \wedge$
$\mathtt{existset}\ \mathsf{Iter}'\ \mathtt{of}\ \mathsf{Node} : \mathtt{isIter}(\mathsf{Iter}') \ \wedge$
$\mathsf{Content}' = \mathsf{Content} \setminus \{n\} \ \wedge\ \mathsf{Iter}' = \mathsf{Iter} \setminus \{n\} \ \wedge$
$\mathsf{Iter}' \subseteq \mathsf{Content}']$

In the PALE code, the definitions `set Content : Node` and `set Iter : Node` introduce local set variables that are used to track the initial content of the data structure. The precondition establishes the relationship between these abstract variables and the concrete state of the program. Procedure postconditions and loop invariants can then use these variables to refer to the initial content of the data structure. These annotations can also refer to current values of abstract set variables, which are effectively substituted using $\mathtt{isS}_i$ predicates and existential quantification.

Our example uses two $\mathtt{isS}_i$ predicates, `isContent` and `isIter`. Combined with the quantification induced by the `set` declarations, these predicates bind the `Content` and `Iter` sets to their definitions, giving meaning to the conjuncts $n \in \mathsf{Content}$ and $\mathsf{Iter} \subseteq \mathsf{Content}$ in the precondition; a similar translation is done for the postcondition.

Note that the actual translation is slightly more complicated than what we have presented here because our PALE plugin introduces additional instrumentation fields that conceptually store the external objects (objects that are not part of the currently analyzed data structure).

Consequences. The PALE analysis tool implements a sophisticated analysis that can verify detailed properties of complex linked data structures. For scalability reasons, it is impractical to use PALE to analyze anything other than encapsulated data structure implementations: PALE invokes the MONA decision procedure [35], [36] for monadic second-order logic, which has non-elementary complexity [37]. (Other shape analyses also have high complexity.) But within this domain it can provide exceptional precision and verify important properties that are clearly beyond the reach of more scalable analyses. Our successful integration of the PALE analysis system demonstrates that it is possible to apply very precise analyses to focused parts of the program. Our results therefore show how to unlock the potential of these analyses to verify important data structure consistency properties in programs that would otherwise remain beyond reach of static analysis. In the next section we show how to verify potentially even more detailed properties using theorem proving.

B. *The Theorem Proving Plugin*

The theorem proving plugin [21] generates verification conditions using weakest liberal preconditions [38] and discharges them using the Isabelle theorem prover. We have chosen this technique for verifying arbitrarily complicated data structure implementations. The logic for specifying abstraction functions is based on typed set theory. Proof obligations can be discharged using either automated theorem proving or a proof checker for manually generated proofs. As a result, there is no *a priori* bound on the complexity of the data structures (and data structure consistency properties) that can be verified using this technique.

For our minesweeper example, we have applied this plugin to the verification of the `ExposedSet` module, which implements a set by storing objects in a global array. The implementation of `ExposedSet` is shown in Figure 7. Note that the implementation contains explicit loop invariants (in quotes); our verification condition generator does not support loop invariant inference. The state of the `ExposedSet` module is represented by the global array `d` which stores the set elements, and the integer variable `s` which indicates the currently used part of the array.

One of the procedures in the `ExposedSet` module is the `add` procedure, which adds a `Node` to the set of `Nodes` representing the exposed cells. The specification section of the module states this contract more precisely in terms of the abstract set `Content` (see Figure 8), which corresponds to the set of `Node` objects in the `ExposedSet`.

The abstraction function in Figure 9 relates the abstract set `Content` to its concrete implementation; it simply states that `Content` corresponds to the set of `Node` objects contained within the array `d` with index between zero and `s - 1` inclusive.

To verify that the `add` procedure conforms to its specification, the analysis plugin first augments the `add` procedure's postcondition by conjoining it with a frame condition derived from the modifies clause. The resulting formula is `(Content' = Content + n) & (setInit' = setInit)`.

The next step is to apply the definition of `Content` from the abstraction section to the `add` procedure's preconditions,

```
impl module ExposedSet {
  format Node {}
  reference d : Node[];
  var s : int;

proc init() { ... }
proc add(n : Node) { d[s] = n; s = s + 1; }
proc remove(n : Node) {
  int i = 0;
  // loop invariant for removal
  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 ~= null} =
   {x. exists j. 0 <= j & j < i' &
                 x = d[j] & x ~= null} - {n}"
  (i < s) {
      if (d[i] == n) d[i] = null;
      i = i + 1
  }
}
proc contains(n : Node) returns b : bool {
  int i = 0;
  bool result = false;
  // loop invariant
  while "0 <= i' & i' <= s & d' = d &
    (result' <=> (n : {x. exists j. 0 <= j & j < i' &
                                x = d'[j] & x ~= null})"
    (!result && (i < s)) {
      if (d[i] == n) result = true;
      i = i + 1
  }
  return result;
}
}
```

Fig. 7. Implementation Section of the ExposedSet Module.

```
spec module ExposedSet {
  format Node;
  specvar setInit : bool;
  specvar Content : Node set;

  proc init()
    requires true
    modifies Content, setInit
    ensures setInit' & (Content' = {});

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

  proc remove(n : Node)
    requires setInit & card(n) = 1
    modifies Content
    ensures (Content' = Content - n);

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

Fig. 8. Specification Section of the ExposedSet Module.

```
abst module ExposedSet {
  use plugin vcgen;
  Content = { x : Node | "exists j. 0 <= j & j < s &
                          x = d[j] & x ~= null"};
  invariant "0 <= s";
}
```

Fig. 9. Abstraction Section of the ExposedSet Module.

```
assume setInit & card(n) = 1 & 0 <= s;
d[s] = n;
s = s + 1;
assert { x | exists j. 0 <= j & j < s' &
                       x = d'[j] & x ~= null} =
       { x | exists j. 0 <= j & j < s &
                       x = d[j] & x ~= null} + {n} &
       setInit' = setInit & 0 <= s;
```

Fig. 10. Translated Implementation of add in Loop-Free Guarded Command Language.

postconditions, loop invariants and assertions. The resulting conditions are expressed in terms of the concrete data structure state. For example, the formula "Content' = Content + n" translates into:

$$\{x \mid \exists j.\ 0 \leq j \wedge j < s' \wedge x = d'[j] \wedge x \neq \mathsf{null}\} =$$
$$\{x \mid \exists j.\ 0 \leq j \wedge j < s \wedge x = d[j] \wedge x \neq \mathsf{null}\} \cup \{n\}$$

The analysis then conjoins both the precondition and postcondition with the representation invariants specified in the abstraction section. Our example contains the representation invariant $0 \leq s$.

Next, the analysis translates the statements from the implementation of add into a loop-free guarded command language similar to that used in [39]. The result of the translation is given in Figure 10.

By computing weakest liberal preconditions, the analysis then creates a formula from the translated code; the validity of this formula implies the conformance of the procedure to its specification.

To simplify the task of discharging the resulting verification condition, the formula is split into as many conjuncts as possible by performing a simple non-backtracking walk through the connectives $\forall$, $\Rightarrow$, $\wedge$ in the formula syntax tree. The analysis then attempts to verify each conjunct in turn. It first searches a library of previously proven lemmas for a match to the current conjunct. If it does not find a match, the analysis invokes Isabelle's built-in simplifier and classical reasoner with array axioms, attempting to prove the formula automatically. In our example, this attempt succeeds for most of the generated verification-condition conjuncts. For the remaining conjuncts, the fully automated verification fails and the plugin saves them as "not known to be true". The user then interactively proves these difficult cases in Isabelle, and stores them in the library of verified lemmas. Subsequent verification attempts then execute without user assistance.

C. The Flag Plugin

Our flag analysis [40] verifies that modules implement set specifications in which integer or boolean flags indicate abstract set membership. The developer uses set definitions in the abstraction section of a module to specify the correspondence between concrete flag values and abstract sets from the specification.

Figure 11 presents the abstraction section of the Board module, which contains definitions of sets U, MarkedCells, ExposedCells, UnexposedCells, and MinedCells as well as several global boolean variables. The set U contains all initialized Cell objects in the program heap, that is, all Cell objects that have their init flag set to true. The other

```
abst module Board {
  use plugin "flags";
  U = { x : Cell | "x.init = true"};
  MarkedCells = U cap { x: Cell | "x.isMarked = true"};
  ExposedCells = U cap { x: Cell | "x.isExposed = true"};
  UnexposedCells = U cap { x: Cell | "x.isExposed = false"};
  MinedCells = U cap { x: Cell | "x.isMined = true"};
  predvar gameOver; predvar init; predvar peeking;
}
```

Fig. 11. Abstraction Section of `Board` Module.

sets are defined as intersections with `U`, *e.g.* using the syntax `MarkedCells = U cap { ... }`. This ensures that all sets defined in this abstraction section are initially empty and do not change when a different module allocates an object using the `new` statement.

The flag analysis performs abstract interpretation [41] with analysis domain elements represented by formulas. The transfer functions in the dataflow analysis update boolean formulas to reflect the effect of each statement, symbolically computing the relation composition of transition relations. When it encounters an assertion, procedure call, or procedure postcondition, our flag analysis generates a verification condition and discharges it using the MONA decision procedure for the monadic second-order logic of strings, which subsumes boolean algebras [36]. In our experience, applying several formula transformations drastically reduced the size of the formulas generated by the flag analysis, as well as the time that the MONA decision procedure spent verifying these formulas. These transformations greatly improved the performance of our analysis and allowed our analysis to verify larger programs. Complete treatments of the flag plugin appear in [17], [32].

In addition to tracking the values of sets introduced by flag values, the flag plugin also keeps track of the values of sets specified in client modules. In the `Board` module of the minesweeper example, this includes the sets `ExposedSet.Content` and `UnexposedList.Content`. This allows the analysis to verify invariants such as

```
Board.ExposedCells = ExposedSet.Content
Board.UnexposedCells = UnexposedList.Content
disjoint(MarkedCells, ExposedCells)
disjoint(ExposedCells.Content,
         UnexposedList.Content)
```

The last two properties are examples of high-level data structure invariants that correlate the values of sets that correspond to multiple data structures. Hob helps the developer and the analysis deal with such high-level invariants using the scope construct described in [23].

Flag example. Figure 12 presents a short procedure and its specification. This procedure either adds or removes an object from the `MarkedCells` set by mutating its `isMarked` boolean-valued field. We next present the formula that the flag plugin generates to verify this procedure, omitting irrelevant parts of the program state for the sake of brevity.

$$
\begin{aligned}
& (M = U \cap M_1) \wedge M' = U' \cap M_1' && (1)\\
\wedge\quad & ((M_1' = M_1 \cup c) \wedge v) \mid ((M_1' = M_1 \setminus c) \wedge \neg v) && (2)\\
\wedge\quad & c \subseteq U \wedge \mathsf{card}(c) = 1 && (3)\\
\wedge\quad & U' = U \wedge C' = C \wedge p \Leftrightarrow p' \wedge \ldots && (4)\\
\Longrightarrow\quad & && (5)\\
& C' = C \wedge p \Leftrightarrow p' && (6)\\
\wedge\quad & ((v \Leftrightarrow c \subseteq M') \wedge M' \subseteq M \cup c) && (7)
\end{aligned}
$$

```
impl module Board {
    proc setMarked(c:Cell; v:bool) {
      c.isMarked = v;
    }
}

spec module Board {
    proc setMarked(c:Cell; v:bool)
      requires (c in U) & (card(c)=1)
      modifies MarkedCells
      ensures (v <=> (c in MarkedCells')) &
              (MarkedCells' <= MarkedCells + c);
}
```

Fig. 12. Specification and implementation of procedures in `Board` Module.

The formula ranges over the set variables and boolean predicates in the program; procedure parameters occur as free variables of the formula, while the program's abstract state is given in terms of universally quantified variables. The formula contains two parts. Lines 1 through 4 specify the program state after symbolic execution of the procedure, while lines 6 and 7 state the requirements on the program state needed by the procedure's postcondition. To verify that the procedure satisfies its specification, MONA must deduce that lines 1 through 4 imply lines 6 and 7.

We first discuss the conditions known by Hob to hold upon procedure exit. For brevity, we have replaced `MarkedCells` by M and `peeking` by p. Line 1 states definitions for derived formulas. These definitions are repeated twice, once for unprimed variables and once for primed variables. Line 2 gives the result of the transfer function as computed over the procedure. It captures the effect of the assignment to the `isMarked` field, which defines the set M. Line 3 states the procedure precondition. Finally, line 4 constrains sets and variables that are unmodified by the procedure. Initially, all sets and variables are unmodified; each transfer function that modifies state also removes variables from this line.

Next, we discuss the required postconditions. Procedures must guarantee that, upon exit, sets that are not declared to be modified keep their initial values; line 6 states this requirement. Also, procedures must guarantee that their postconditions hold; in this case, line 7 states the needed postcondition.

Once the flag plugin generates the appropriate formula, it passes the formula on to the MONA tool. In this case, the verification succeeds because the antecedent is sufficiently strong: the procedure does implement its specification.

Flag analysis and generalized typestate. We have just observed that the flag plugin can establish high-level data structure properties such as equality and disjointness of sets. We have also seen (in Section III-D) that the flag plugin can be used to verify the correct use of the iterator interface. Here we present another perspective on an analysis that verifies contracts (interfaces) based on sets, by viewing sets as a generalization of typestate [9], [42].

Instead of associating a single state with each object, our system models each typestate as an abstract set of objects. If an object is in a given typestate, it is a member of the set that corresponds to that typestate, which leads to the following generalizations of the standard typestate approach:

- **Abstract Data Types:** For typestate purposes, abstract data types can be viewed as maintaining several abstract sets of objects. For example, an iterator contains one set for all objects in the list, and one set of objects that remain to be iterated over. In this way, the iterator indicates whether an object has already been iterated over. With this perspective, the typestate of an object is a function of its participation in the abstract data type as reflected in its membership in the data type's abstract sets of objects.
- **Orthogonal Composition:** In our formulation of typestate, an object can be a member of multiple sets simultaneously. This promotes composite typestate structures in which the developer endows each component with a collection of abstract sets, with each set corresponding to an aspect of the typestate relevant to the component. With this kind of structure, each object's typestate is an orthogonal composition of the typestate aspects from each of the components in which it participates. Examples include composite typestates for objects that participate in multiple data structures and objects that play multiple roles within a single component.
  The advantages of using multiple orthogonal sets include better modularity (because each component deals only with those aspects of the typestate that are relevant for its operation) and support for polymorphism (because each component can operate successfully on multiple objects that participate in different ways in other components).
- **Hierarchical Typestates:** Hierarchical classification via inheritance is a key element of the type systems in most object-oriented languages, but is absent in historical flat typestate systems [9]. Our formulation cleanly supports typestate hierarchies—a collection of sets can partition a more general set, with the subset inclusion ordering capturing the hierarchy. Typestate hierarchies also appear in [40], [43].
- **Sharing and Typestates:** Sharing via aliased object references has caused problems for standard typestate systems—it has been difficult to ensure that if the program uses one reference to access the object and change its typestate, the declared type of other references is appropriately adjusted. Our typestate formulation supports a new, more abstract form of sharing, which integrates aliasing information directly into the analysis domain, in the form of constraints expressed using sets (*e.g.* set disjointness for unaliased variables). This integration enables the flags plugin to update set membership in a manner consistent with known aliasing relationships between objects. Furthermore, if an object participates in multiple data structures, its typestate characterizes that kind of sharing by indicating its membership in multiple typestate sets, one for each data structure. This formulation supports non-monotonic changes—the set of objects that contain an element may change arbitrarily throughout the computation.

V. Experience

In this section, we describe our experience using the Hob system to implement and specify design information for other benchmark programs. In addition to the minesweeper example presented in Section II, we ran our analysis on a complete webserver, as well as short programs inspired by computational patterns from scientific computations, operating-system schedulers, and program transformation passes. Note in particular that the Hob webserver serves the Hob project webpage, at `http://hob.csail.mit.edu`. These benchmarks use a variety of data structures, and we have therefore implemented and verified sets, set iterators, queues, stacks, and priority queues. Table I illustrates the benchmarks we ran through our system. Our data structure implementations range from singly-linked and doubly-linked lists (with and without iterators) and tree insertion (all verified using the PALE plugin) through array data structures (verified using the theorem proving plugin). Our modular approach allowed us to reuse data structure implementations across multiple benchmarks. We expect such reuse to be possible in general, thus amortizing the cost of precise and potentially expensive data structure analyses across many clients of these data structures.

Section II gave the flavor of high-level properties that we verified in our examples through the example of minesweeper, using in particular sets of exposed and unexposed cells. Note that all Hob properties are expressed in terms of sets, but that the interpretations of sets vary on a per-application, domain-specific basis. A second example where Hob enforces design constraints arises in our web server example. A design decision for the web server was to have the server cache the content before sending it in response to a request. Our web server implementation contains a procedure, `sendEntry(c)`, which emits the contents of its parameter `c` to a socket. The interface of `sendEntry(c)` imposes the precondition that the entry `c` to be sent is either 1) stored in the cache or 2) in the set of "blacklisted" objects that are too large to be stored. Hob's analysis tracks the dynamically changing values of sets representing the cached content and the blacklisted content, and checks that the parameter `c` of `sendEntry` is in the union of these two sets. Note that the tracked property is simple to state and understand, yet goes beyond static type systems. The use of sets as opposed to flags associated with objects naturally captures the idea that being in the cache (or being blacklisted) is not the property of the content itself, but rather a property of the way in which the content is used in the web server, a fact that is reflected in the data structures in which the content is stored. In the `scheduler` example, we have verified disjointness of sets of suspended and running processes, the equality of sets represented by flags (allowing a constant-time membership test) and lists (for iteration), as well as typestate preconditions on scheduler operations. The `prodcons` example coordinates the actions of a producer and a consumer module that communicate via a shared stack (implemented as a list), with specifications ensuring that the elements are produced and consumed by the operations and that stack underflow never occurs. Water is a numerical simulation of water molecules that proceeds in several phases; we verify that these phases are performed in the desired order by encoding them using global boolean flags, specifying these flags in preconditions and specifying their changes in postconditions. The compiler example transforms nodes of an abstract syntax tree. We used sets to encode the typestates of the nodes. We provided specifications ensuring that the nodes are processed in the correct order and that the operations have the expected effect on the typestates.

System totals	# modules	# lines spec	# lines impl
compiler	3	113	211
water	10	542	1921
prodcons	3	54	78
scheduler	3	77	128
minesweeper	7	236	750
httpd	14	335	1229

TABLE I
BENCHMARK CHARACTERISTICS.

VI. SCOPE OF OUR TECHNIQUE

Our Hob system verifies data structure consistency properties for programs written in a memory-safe imperative implementation language and specified using a specification language based on the boolean algebra of sets. We now discuss some of the design decisions that we made while building Hob.

a) Choice of Implementation and Specification Languages: We designed the Hob implementation language to be syntactically similar to Java at a statement level. We decided to use a custom procedural implementation language as a convenient way to explore the automatic verification of data structure consistency properties while avoiding inessential complexities of a full-fledged programming language. In particular, we omitted common object-oriented features such as inheritance, dynamic dispatch, and object-based encapsulation. In our experience, it was relatively simple to port Java code to our implementation language. When comparing Java and Hob it is important to keep in mind that Hob has two constructs that approximately correspond to Java's classes: 1) formats are used to represent memory cells, and 2) modules are used to structure a program into its main constituent parts. The static module instantiation in Hob is less general than the dynamic instantiation of classes with methods in Java, but it encourages developers to express the static architecture of an application, and aids verifiability. We expect that structuring Java applications along similar principles would help the analysis of Java programs as well.

We believe that Hob's set specifications are natural for developers to use because they enable developers to state object membership properties and relationships between data structures [17]. After all, many data structures are simply implementations of sets. Set specifications can express many key data structure properties and, in particular, consistency properties which relate the contents of different data structures. Such consistency properties are often crucial design properties for a system which ought to hold throughout its life cycle; set specifications provide a concise and easy-to-understand way for developers to express and verify these properties.

To understand the scope of Hob's set specification language, consider the example of a map implementation. The set specification language can express, for example, that the set of keys and the set of values are disjoint, but cannot express that a particular key is related to a particular value, because the boolean algebra of uninterpreted elements does not contain any relation symbols [4]. Nevertheless, our experience shows that the boolean algebra of sets can express many interesting data structure properties. Such descriptions may not be full specifications of the behavior of operations, but they do indicate important partial correctness properties. We therefore believe that the set specification language makes a useful trade-off between the expressive power and tractability of the analyses. We chose to explicitly omit integer and floating-point arithmetic from our specification language.[5] Indeed, many data structure consistency properties do depend on general integer and floating-point arithmetic. Note that the set specification language does not support sets of pairs or sets of sets, only sets of uninterpreted elements. This is why it can be characterized using the boolean algebra of sets and decided in elementary time [29] and in practice often belongs to the quantifier-free fragment that can be decided in non-deterministic polynomial time.

It is important to distinguish between Hob's interface language, which was designed to be less expressive and more tractable, and the specification languages inside the abstraction modules, which express data structure representation invariants and abstraction functions. Inside abstraction modules, plugins may use arbitrarily powerful specification languages. For example, the monadic second-order logic used in the PALE plugin can express reachability properties that are not even expressible in first-order logic [46]. In general, Hob can analyze arbitrarily complex internal data structure properties given the presence of appropriate plugins.

b) Developers' Responsibilities: Our technique requires developers to specify the data structure consistency properties that they would like the Hob system to verify. Developers express these properties 1) in terms of procedure preconditions and postconditions; and 2) in terms of high-level invariants for global data structure consistency properties. These properties are expressed using the set specification language. Since there is a gap between the abstract set specification language and the concrete implementation language, developers must also specify an interpretation for the program's abstract sets—in other words, abstraction functions. A key feature of the Hob approach is that it supports a variety of different abstraction function syntaxes, by delegating the core analysis task to a set of analysis plugins. In short, each plugin is required to verify that a procedure's implementation conforms to its specification, where the set interpretations are given by the abstraction function. While it would be possible to use analyses to infer specifications or abstraction functions, we found it very

[4]The Jahob system [44] supports a module specification language that permits the use of relations, and has successfully verified detailed specifications of map implementations such as hash tables.

[5]In [45], we describe how to decide Boolean Algebra with Presburger Arithmetic (BAPA); the Hob system's core specification language could be extended to support BAPA.

useful to explicitly state this information as part of software documentation, and decided to focus the inference effort of our analyses on loop invariants that are more program-point specific and therefore less interesting for documenting the key properties of the system.

c) Hob and the Development Life Cycle: The abstractness of Hob's set specification language encourages developers to think at a higher level of abstraction and enables them to express deeper properties of programs. Such properties can easily be obscured in a program's implementation. At the implementation level, design information is hidden behind a mass of details, which are necessary for implementing, but not useful for understanding, the underlying design. We believe that the set specification language exposes design information more effectively than imperative implementation languages, since set specification languages abstract away from the details of *how* the program carries out its tasks to express *what* the program does.

While a program's set specifications are useful from the earliest prototype stages of development, its specifications become especially valuable as a program moves through its development life cycle into the maintenance phase, when the design information may become outdated, and the original developers may have moved on to other projects. The Hob system enables developers to use data structure consistency properties as verified documentation. Our analysis tool verifies that these properties hold, not just at any one point in the program's life, but throughout changes by successive developers, who may not understand the program's original design at all. Our experience with Hob suggests that it is capable of recording design decisions taken by the original developers and ensuring that this design information remains up-to-date.

d) Scalability of the Hob Approach: Because the Hob system uses a modular verification approach, we believe that it should scale quite well. However, we have not yet evaluated its efficacy on programs larger than 2000 lines. The scalability of our system does not depend on asymptotic complexity arguments; we instead observe that researchers have not successfully performed shape analysis on programs that exceed a couple of hundreds of lines, and that our modular verification approach sidesteps such ceilings on analysis applicability.

We believe that Hob's use of a set specification language is especially productive for several reasons. First, our experience with the flag plugin suggests that set-based specification languages make symbolic loop invariant inference feasible because the space of possible invariants is relatively small (especially compared to a specification language with full integer arithmetic constraints). Second, the use of a simple set-based specification language imposes fewer restrictions on plugins, which need to understand specifications in this language. A simpler specification language therefore makes it easier to develop new analysis plugins. Finally, the use of a restricted specification language helps control the "specification creep" observed in ESC/Java [8], because the restrictions of the specification language force developers to avoid overly detailed specifications that would require unscalable techniques to check.

While our use of a custom implementation language has the advantage of being especially well-suited to our verification approach, it has the disadvantage of requiring us to translate benchmarks from other programming languages into the Hob language. The related Jahob project [47] explores the use of a Java subset as an implementation language for verifying data structure consistency properties.

VII. Related Work

We survey research in verification technology, existing verification tools, and other approaches to combining different verification techniques.

A. Verification technology

We first discuss the verification techniques used in the Hob system. Hob builds on ideas from shape analysis and typestate systems to verify data structure consistency properties. Since Hob analysis plugins use a decision procedure for boolean algebras for evaluating formulas in the set specification language, we discuss the decidability of this question and examine tools to decide this language.

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 [1], [2], [14]. These analyses focus on extracting or verifying detailed consistency properties of individual data structures. While these analyses are very precise, the level of detail of the properties that they must track have limited their scalability—many extant shape analysis algorithms have super-exponential complexity. One of our research goals is to enable the application of such sophisticated but expensive analyses in small regions of code (where their precision is needed), while taking advantage of modularity to analyze other parts of code using more scalable analyses.

Typestate systems. Typestate systems track the conceptual states that each object goes through during its lifetime in the computation [9], [48]. Typestate systems generalize standard type systems in that the typestate of an object may change during the computation. Our approach enables the checking of properties that generalize typestate properties [17], [40]. The developer can simply use sets to model typestates: if an object should be in a given typestate in the typestate system, it is a member of the corresponding set in our system. Aliasing (or more generally, any kind of sharing) is the key problem for typestate systems—if the program uses one reference to change the typestate of an object, the typestate system must ensure that either the declared typestate of the other references is updated to reflect the new typestate or that the new typestate is compatible with the old declared typestate at the other references. Role analysis [14] identifies this problem and suggests a solution based on a precise abstraction of the heap and user-specified procedure specifications. Fink, Yahav, Dor, Ramalingam, and Geay integrate pointer analysis with typestate property verification for Java programs [49]. Their approach scales due to the use of a series of related abstractions: the simpler abstractions quickly rule out many simple property violations, leaving the more sophisticated cases to the more expensive analyses. Our system uses a looser integration model for the constituent analyses, which makes integration of diverse program analyses easier and allows us to verify properties that go beyond traditional typestate properties. Bierhoff and Aldrich describe a dynamic analysis system for verifying typestate properties in Java programs that correctly handles typestates in the context of subclassing [43]. Like Hob, [43] also supports multiple orthogonal typestates. While a dynamic analysis can prevent programs from executing undesirable actions, typically by terminating a program when it attempts to execute such actions, the advantage of our static approach is that it provides stronger guarantees that programs never violate typestate constraints on any possible execution.

Decision procedures for boolean algebras. We use first-order logic formulas in the language of boolean algebras as the basis of our module specification language. The decidability of the satisfiability problem for the first-order theory of boolean algebras dates back to [26], [50]. To our knowledge, the only tool that can decide the first-order theory of boolean algebras is MONA [35]; it implements the more general decision procedure for monadic second-order logic over trees, and has non-elementary complexity in general but adequate performance in practice for the problems that arise in our program analysis framework. A decision procedure for an extension of boolean algebra with Presburger arithmetic operations is presented in [45], [51]; this extension allows reasoning about sizes of data structures.

Modularity mechanisms. Hob's ability [23] to encapsulate individual object fields in separate modules appeared in [52], is present in aspect-oriented programming implementations [53], and is used in intermediate languages for static checking tools [8], [54]. Hob's approach of allowing modules to share objects and not fields enables it to solve some of the information-hiding problems mentioned by O'Hearn, Yang and Reynolds in [55]. In particular, the change in perspective from encapsulating objects to encapsulating fields frees Hob plugins from the obligation to reason about which module owns which object: we have set up the system so that no module can affect the contents of any other module's sets. Note that, as in [55], the Hob system supports purely static modules and dynamically created objects.

Our module instantiation mechanism is similar in spirit to the functor mechanism of Standard ML [56]. One difference is that functors in Standard ML can take arbitrary declarations as parameters, whereas parameters in Hob are simply names of format types. Nevertheless, Hob's mechanism is sufficient for statically instantiating an arbitrary number of data structures.

B. Program verification and checking tools.

Methodologies for using formal specifications in software development include Gypsy [57], the B method [58], VDM [59], Z [60], Larch [61], and RAISE [62]. Some of these methodologies (for instance, Z) provide a general notation which developers may use to express program properties, and expect developers to carry out all of the proofs by hand. Most of these methodologies include some tool support in the form of verification condition generators and proof assistants. However, unlike Hob, these methodologies do not leverage current static analysis technologies, such as shape analysis, to automatically verify program properties. Tools based on verification condition generation and theorem proving include [61], [63], [64], and, more recently, [65]–[67].

ESC/Java [8] is a program checking tool which aims to identify common errors in programs using program specifications in a subset of the Java Modelling Language [68]. An explicit design goal of ESC/Java is to statically identify potential run-time errors, *e.g.* null-pointer exceptions, although ESC/Java does attempt to establish that preconditions hold at call sites. The Hob system was principally designed to verify program-specific properties, which include preconditions and postconditions, but also global data structure consistency properties. Hob's support for abstraction functions and scopes make data structure consistency properties much easier to express. ESC/Java also sacrifices soundness in that it 1) does not model all of the details of the program heap and 2) unrolls loops a finite number of times rather than using loop invariants. However, ESC/Java does detect some common programming errors.

The ESC/Java2 project extends the original ESC/Java work by supporting current versions of Java, and verifying more JML constructs. ESC/Java2 (as well as ESC/Modula-3 [69]) allows the use of heap abstractions via its support for model fields. Model fields use developer-provided representations. These representations are similar in spirit to the set definitions which appear in Hob's abstraction modules. However, not all model fields have representations; for instance, the library annotations provided with ESC/Java2 do not contain any definition of a list's contents. The first-order logic used by the underlying Simplify theorem prover [70] does not support transitive closure; effective (and necessarily partial) first-order axiomatizations of transitive closure are still active areas of research [71]–[73]. ESC/Java2 has therefore, to our knowledge, not been used to verify the data structure consistency properties that Hob verifies for linked lists. Another difference is that ESC/Java2 representations are expressed in terms of Java expressions or predicates, rather than the analysis-plugin-specific set definitions supported by Hob. Cok explains how ESC/Java2 handles model fields in [74]; essentially, it treats them as method calls and includes the postconditions of the model fields' representations.

The Hob approach uses sets and abstraction modules in place of JML's model fields. Abstraction modules enable Hob to check that implementations satisfy both local properties, *i.e.* that representation invariants continue to hold, and global properties (whose meaning is made explicit by abstraction functions). Furthermore, abstraction modules allow Hob analysis plugins to use arbitrarily powerful logics for establishing local set properties. Despite this, the simplicity of Hob's set specification language allows even simple and scalable plugins to take advantage of detailed results produced by complex analyses.

A more recent effort is a sound static analysis tool for an object-oriented language, Spec#, which extends $C\#$ and comes with a particular methodology for the modular treatment of invariants [75]. Spec# has recently been extended with a treatment of model fields [76]. The Spec# static verifier currently uses the theorem provers Simplify and Zap [77] that do not directly support transitive closure; however, the overall Spec# methodology is largely independent of the underlying reasoning engine.

Other tools focus on verifying properties of concurrent programs [6], [78] or device drivers [3], [4] as opposed to properties of linked data structures.

Higher-order contracts. Modular analysis has also been proposed for functional programs. Verifying contracts in the context of higher-order programming languages is quite difficult; existing approaches for improving programmer productivity in this context typically focus on dynamic checking of contracts or static checking based on type system extensions. However, Meunier, Findler and Felleisen do propose a modular analysis based on contracts in [79]. Note that their use of the term "set-based" is not the same as our use of a set specification language; in their work, set-based refers to the static analysis technique used to verify contracts (in combination with 0CFA). Their approach uses the base programming language to specify predicates for use in contracts; contrast this with the Hob approach of using set specifications and abstraction functions. In principle, one could use arbitrary analyses to establish procedure contracts. In Hob, we have successfully used multiple cooperating analyses to establish data structure consistency properties. The analysis in [79] is modular in the same sense that Hob is modular: they both analyze the program one module at a time. In the context of higher-order functions, contracts become more complicated to analyze, since the verification of a procedure's contract must be delayed until higher-order function parameters are evaluated. A significant amount of effort goes towards handling this complication, which Hob avoids by using a first-order implementation language.

C. Hob analysis approach.

Our research aims to enable the application of multiple analyses that check arbitrarily complicated properties within a single program. Most existing approaches, in contrast, attempt to develop a single new analysis algorithm or technique. Our system supports the loose integration of analyses where each analysis applies to one procedure or module. This design decision makes the incorporation of external tools easy. In [80], Chang and Leino explore an approach that proposes a tighter combination of a particular domain (uninterpreted function symbols) with an arbitrary base domain. Their approach would enable the application of static analysis techniques which could reason about the program state using a number of different abstract domains. Briefly, our approach works well for combining analyses at granularities above the procedure level, while their approach is targeted towards combining analyses below the procedure level. The fine-grained combination of analysis techniques also appears in the Jahob verification system [47]. Note also that these techniques are not mutually exclusive: finer-grained combinations of analyses could be implemented and deployed as individual Hob analysis plugins.

VIII. 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.

This paper shows how to apply the full range of analyses to programs composed of multiple modules. The key elements of our 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.

Acknowledgements. We thank Thomas Wies for developing the Bohne symbolic shape analysis plugin that was incorporated into the Hob system. We thank Anders Møller for help with the PALE and MONA packages. We thank the anonymous reviewers for their insightful comments that helped improve the paper.

References

[1] A. Møller and M. I. Schwartzbach, "The Pointer Assertion Logic Engine," in *Programming Language Design and Implementation*, 2001.
[2] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," *ACM TOPLAS*, vol. 24, no. 3, pp. 217–298, 2002.
[3] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *Proc. ACM PLDI*, 2001.
[4] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," in *31st POPL*, 2004.
[5] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith, "Modular verification of software components in c," in *ICSE 2003*, 2003.
[6] N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe, "STeP: Deductive-algorithmic verification of reactive and real-time systems," in *8th CAV*, vol. 1102, 1996, pp. 415–418.
[7] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill, "CMC: A pragmatic approach to model checking real code," in *OSDI'02*, 2002.
[8] C. Flanagan, K. R. M. Leino, M. Lilibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended Static Checking for Java," in *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2002.
[9] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE TSE*, January 1986.
[10] R. DeLine and M. Fähndrich, "Enforcing high-level protocols in low-level software," in *Proc. ACM PLDI*, 2001.
[11] M. Fähndrich and K. R. M. Leino, "Heap monotonic typestates," in *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2003.
[12] J. Field, D. Goyal, G. Ramalingam, and E. Yahav, "Typestate verification: Abstraction techniques and complexity results," in *Int. Symp. Static Analysis*, ser. LNCS, vol. 2694. Springer, 2003.
[13] M. Fahndrich and R. DeLine, "Adoption and focus: Practical linear types for imperative programming," in *Proc. ACM PLDI*, 2002.
[14] V. Kuncak, P. Lam, and M. Rinard, "Role analysis," in *Annual ACM Symp. on Principles of Programming Languages (POPL)*, 2002.
[15] P. Lam, V. Kuncak, and M. Rinard, "On our experience with modular pluggable analyses," MIT CSAIL, Tech. Rep. 965, September 2004.
[16] ——, "Hob: A tool for verifying data structure consistency," in *14th International Conference on Compiler Construction (tool demo)*, April 2005.

[17] ——, "Generalized typestate checking for data structure consistency," in *6th Int. Conf. Verification, Model Checking and Abstract Interpretation*, 2005.

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

[19] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard, "Field constraint analysis," in *Proc. Int. Conf. Verification, Model Checking, and Abstract Interpratation*, 2006.

[20] A. Podelski and T. Wies, "Boolean heaps," in *Proc. Int. Static Analysis Symposium*, 2005.

[21] 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.

[22] T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. Rinard, "On verifying complex properties using symbolic shape analysis," Max-Planck Institute for Computer Science, Tech. Rep. MPI-I-2006-2-1, 2006.

[23] P. Lam, V. Kuncak, and M. Rinard, "Cross-cutting techniques in program specification and analysis," in *4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, 2005.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlisside, *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1994.

[25] J. R. Büchi, "Weak second-order arithmetic and finite automata," *Z. Math. Logik Grundl. Math.*, vol. 6, pp. 66–92, 1960.

[26] L. Loewenheim, "Über Mögligkeiten im Relativkalkül," *Math. Annalen*, vol. 76, pp. 228–251, 1915.

[27] D. Jackson, I. Shlyakhter, and M. Sridharan, "A micromodularity mechanism," in *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering / European Software Engineering Conference (FSE/ESEC '01)*, 2001.

[28] A. Borgida, J. Mylopoulos, and R. Reiter, "On the frame problem in procedure specifications," *TSE*, vol. 21, no. 10, pp. 785–798, Oct. 1995.

[29] D. Kozen, "Complexity of boolean algebras," *Theoretical Computer Science*, vol. 10, pp. 221–247, 1980.

[30] B. Liskov and J. Guttag, *Program Development in Java*. Addison-Wesley, 2001.

[31] H. Jifeng, C. A. R. Hoare, and J. W. Sanders, "Data refinement refined," in *ESOP'86*, ser. LNCS, vol. 213, 1986.

[32] P. Lam, V. Kuncak, K. Zee, and M. Rinard, "Set interfaces for generalized typestate and data structure consistency verification," *Theoretical Computer Science*, submitted.

[33] N. Klarlund and M. I. Schwartzbach, "Graph types," in *Proc. 20th ACM POPL*, Charleston, SC, 1993.

[34] J. W. Thatcher and J. B. Wright, "Generalized finite automata theory with an application to a decision problem of second-order logic," *Mathematical Systems Theory*, vol. 2, no. 1, pp. 57–81, August 1968.

[35] N. Klarlund, A. Møller, and M. I. Schwartzbach, "MONA implementation secrets," in *Proc. 5th International Conference on Implementation and Application of Automata*. LNCS, 2000.

[36] 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.

[37] L. Stockmeyer and A. R. Meyer, "Cosmological lower bound on the circuit complexity of a small problem in logic," *J. ACM*, vol. 49, no. 6, pp. 753–784, 2002.

[38] E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, Inc., 1976.

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

[40] P. Lam, V. Kuncak, and M. Rinard, "Generalized typestate checking using set interfaces and pluggable analyses," *SIGPLAN Notices*, vol. 39, pp. 46–55, March 2004.

[41] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *Proc. 6th POPL*. San Antonio, Texas: ACM Press, New York, NY, 1979, pp. 269–282.

[42] R. E. Strom and D. M. Yellin, "Extending typestate checking using conditional liveness analysis," *IEEE Transactions on Software Engineering*, May 1993.

[43] K. Bierhoff and J. Aldrich, "Lightweight object specification with typestates," in *Proceedings of ESEC-FSE '05*, H. C. Gall, Ed., September 2005, pp. 217–226.

[44] V. Kuncak, "Modular data structure verification," Ph.D. dissertation, Massachusetts Institute of Technology, 2007, to appear.

[45] V. Kuncak, H. H. Nguyen, and M. Rinard, "An algorithm for deciding BAPA: Boolean Algebra with Presburger Arithmetic," in *20th International Conference on Automated Deduction, CADE-20*, Tallinn, Estonia, July 2005.

[46] C. H. Papadimitriou, *Computational Complexity*. Addison-Wesley, Reading, Mass., 1994.

[47] V. Kuncak and M. Rinard, "An overview of the jahob analysis system: Project goals and current status," in *NSF Next Generation Software Workshop*, 2006.

[48] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini, "Fickle: Dynamic object re-classification," in *Proc. 15th ECOOP*, ser. LNCS 2072. Springer, 2001, pp. 130–149.

[49] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay, "Effective typestate verification in the presence of aliasing," in *ISSTA'06*, 2006.

[50] T. Skolem, "Untersuchungen über die Axiome des Klassenkalküls and über "Produktations- und Summationsprobleme", welche gewisse Klassen von Aussagen betreffen," Skrifter utgit av Vidnskapsselskapet i Kristiania, I. klasse, no. 3, Oslo, 1919.

[51] V. Kuncak, H. H. Nguyen, and M. Rinard, "Deciding Boolean Algebra with Presburger Arithmetic," *Journal of Automated Reasoning*, 2006, accepted for publication.

[52] D. R. Cheriton and M. E. Wolf, "Extensions for multi-module records in conventional programming languages," in *ACM PLDI*. ACM Press, 1987, pp. 296–306.

[53] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proc. 11th ECOOP*, Jyvaskyla, Finland, June 1997.

[54] V. Kuncak and K. R. M. Leino, "In-place refinement for effect checking," in *Second International Workshop on Automated Verification of Infinite-State Systems (AVIS'03), Warsaw, Poland*, April 2003.

[55] P. O'Hearn, H. Yang, and J. Reynolds, "Separation and information hiding," in *Proc. 31st ACM POPL*, 2004, pp. 268–280.

[56] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Mass., 1997.

[57] D. I. Good, R. L. Akers, and L. M. Smith, "Report on Gypsy 2.05," University of Texas at Austin, Tech. Rep., February 1986.

[58] J.-R. Abrial, M. K. O. Lee, D. Neilson, P. N. Scharbach, and I. Sørensen, "The B-method," in *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume 2*. Springer-Verlag, 1991, pp. 398–405.

[59] C. B. Jones, *Systematic Software Development using VDM*. Prentice Hall International (UK) Ltd., 1986.

[60] J. Woodcock and J. Davies, *Using Z*. Prentice-Hall, Inc., 1996.

[61] J. Guttag and J. Horning, *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[62] B. Dandanell, "Rigorous development using RAISE," in *Proceedings of the conference on Software for citical systems*. ACM Press, 1991, pp. 29–43.

[63] J. C. King, "A program verifier," Ph.D. dissertation, CMU, 1970.

[64] G. Nelson, "Techniques for program verification," XEROX Palo Alto Research Center, Tech. Rep., 1981.

[65] J.-C. Filliatre, "Verification of non-functional programs using interpretations in type theory," *Journal of Functional Programming*, vol. 13, no. 4, pp. 709–745, 2003.

[66] J.-C. Filliatre and C. Marché, "Multi-prover verification of c programs," in *ICFEM'04*, 2004.

[67] 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.

[68] 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," Computing Science Institute, Univ. of Nijmegen, Tech. Rep. NII-R0309, March 2003.

[69] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe, "Extended static checking," COMPAQ Systems Research Center, Tech. Rep. 159, 1998.

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

[71] G. Nelson, "Verifying reachability invariants of linked structures," in *POPL*, 1983.

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

[73] S. K. Lahiri and S. Qadeer, "Verifying properties of well-founded linked lists," in *POPL'06*, 2006.

[74] D. R. Cok, "Reasoning with specifications containing method calls and model fields," *Journal of Object Technology*, vol. 4, no. 8, pp. 77–103, September–October 2005.

[75] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte, "Verification of object-oriented programs with invariants," *Journal of Object Technology*, vol. 3, no. 6, pp. 27–56, 2004.
[76] K. R. M. Leino and P. Müller, "A verification methodology for model fields," in *ESOP'06*, 2006.
[77] T. Ball, S. Lahiri, and M. Musuvathi, "Zap: Automated theorem proving for software analysis," Microsoft Research, Tech. Rep. MSR-TR-2005-137, 2005.
[78] S. Chaki, S. K. Rajamani, and J. Rehof, "Types as models: model checking message-passing programs," in *29th ACM SIGPLAN-SIGACT POPL*. ACM Press, 2002, pp. 45–57.
[79] P. Meunier, R. B. Findler, and M. Felleisen, "Modular set-based analysis from contracts," in *Proc. 33rd ACM POPL*, J. G. Morrisett and S. L. P. Jones, Eds., January 2006, pp. 218–231.
[80] B.-Y. E. Chang and K. R. M. Leino, "Abstract interpretation with alien expressions and heap structures," in *VMCAI'05*, January 2005.

Viktor Kuncak is a doctoral candidate in the MIT Department of Electrical Engineering and Computer Science and a member of the MIT Computer Science and Artificial Intelligence Laboratory. His research interests include program analysis and verification. In the context of the Hob and Jahob projects, he developed techniques for improving software reliability by automatically proving properties of software that manipulates complex data structures. He has also worked on finite model generation, structural subtyping constraints, and role analysis.

Patrick Lam is a doctoral candidate in the MIT Department of Electrical Engineering and Computer Science and a member of the MIT Computer Science and Artificial Intelligence Laboratory. His research aims to develop techniques which enable the application of static analysis technology to the domain of program understanding, in particular by supporting the incorporation of verified design information into the software development process. His research interests also include software engineering, programming languages, and program verification.

Karen Zee is a doctoral candidate in the MIT Department of Electrical Engineering and Computer Science and a member of the MIT Computer Science and Artificial Intelligence Laboratory. Her research interests focus on the use of program analysis to automate the verification of programs. Her work in the Hob and Jahob systems center on the use of theorem provers in combination with static analysis to show strong properties about programs, and automatically inferring loop invariants using a combination of static and dynamic techniques.

Martin Rinard is a Professor in the MIT Department of Electrical Engineering and Computer Science and a member of the MIT Computer Science and Artificial Intelligence Laboratory. His research interests include parallel and distributed computing, programming languages, program analysis, program verification, software engineering, and techniques that enable software systems to execute successfully in spite of the presence of errors.