Modular Pluggable Analyses for Data Structure Consistency

Viktor Kuncak, Patrick Lam, Karen Zee, and Martin Rinard

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Abstract. We describe a technique 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 an interface that uses formulas in a set algebra to characterize the effects of operations on the encapsulated data structures. 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 have implemented our system and deployed three pluggable analyses into it: a flag analysis for modules in which abstract set membership is determined by a flag field in each object, a shape analysis plugin, PALE, for modules that encapsulate linked data structures such as lists and trees, and a theorem proving plugin for analyzing arbitrarily complicated data structures including data structures that use arrays. 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.

1 Introduction

A data structure is consistent if it satisfies the invariants necessary for the normal operation of the program. Data structure consistency is important for successful program execution—if an error corrupts 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 [3, 5, 7, 21, 26, 42, 43, 46].

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 new algorithm that can analyze some specific set of consistency properties, we instead 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. The analyses use a common abstraction based on sets of objects to communicate their analysis results. Our approach therefore 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 manually discharged using a theorem prover or proof checker). The rest of the program uses these modules but does not directly manipulate the encapsulated data structures. These modules can then be analyzed by more efficient analyses that operate primarily at the level of the common set abstraction. These analyses can be viewed as a generalization of typestate analyses [14, 16–18, 33, 48], with the typestate of an object given by the sets that contain the object. Such an analysis simultaneously 1) ensures that the rest of the program respects the preconditions of data structures operations (that is, adheres to a protocol that guarantees the correct use of a data structure), and 2) verifies 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 [37, 39] and 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 [38]; 2) the PALE plugin, which implements shape analysis for linked data structures using the PALE tool [42]; 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 [45]. We used our analysis framework to analyze several programs; our experience shows that it can effectively 1) verify the consistency of data structures encapsulated within a single module and 2) combine analysis results from different analysis plugins to verify properties involving objects shared by multiple modules analyzed by different analyses.

In the rest of the paper, we present the basic concepts of our system; see [37] for additional details. We use a running example to illustrate these concepts.

2 Minesweeper Example

Our example program 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 annotate each unexposed cell as marked or unmarked.

Our implementation uses the standard model-view-controller (MVC) design pattern [23]. 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 responds to user input; the view module produces the game's output; the exposed cell module uses an array to store the cells exposed by the player in the course of the current game; and the unexposed cell module uses an instantiated linked list to store the cells that have not been exposed yet. 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 [39].)

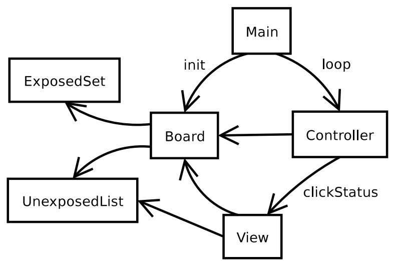

Fig. 1. Modules in Minesweeper implementation

Although of moderate size, 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:

– The set of unexposed cells in `UnexposedList` module is an acyclic doubly-linked list with all `prev` references being the inverse of `next` references.
– The iterator of the `UnexposedList` module is either null or points inside the list.
– If the system is initialized, then the array storing the exposed cells in the `ExposedSet` module has a non-negative size.
– The set of unexposed cells maintained in the `board` module using a flag is identical to the set of unexposed cells maintained in the linked `UnexposedList` data structure.
– The set of exposed cells maintained in the `board` module using a flag is identical to the set of exposed cells maintained in the `ExposedSet` array.
– Unless the game is over, the set of mined cells is disjoint from the set of exposed cells.
– The sets of exposed and unexposed cells are disjoint.
– At the end of the game, all cells are revealed; *i.e.* the set of unexposed cells is empty.

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 through an example of a doubly-linked list with an iterator, which corresponds to the `UnexposedList` module in the minesweeper example.

3 Modules in Hob

The basic unit of analysis in our system is a module. By separating a program into multiple modules, our system can verify different parts of the program independently, and 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.

3.1 Implementation Section

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

```
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

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 therefore 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.

3.2 Specification Section

In conventional programming languages, an interface of a module contains only type declarations that indicate the format of procedure parameters, but do not describe the behavior of procedures; this behavior is typically left to the informal documentation of the module and is not checked by the compiler. In contrast, the specification section of a module in our language contains a procedure contract for each public procedure of the module.

Figure 3 presents the specification section for our iterable doubly-linked list module. To describe the behavior of procedures without exposing implementation details,

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

  proc isEmpty() returns e:bool
    ensures not e <=> (card(Content') >= 1);
  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 initIter()
    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

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 (notation `in` denotes subset).

Procedure contracts. A procedure contract consist of a `requires` clause that specifies a condition that must hold before calling a procedure, and an `ensures` clause that specifies the postcondition that the procedure guarantees. 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 $\{x\}$ denoting a reference to object x and $\emptyset$ denoting a null reference. In particular, the conjunct `card(n)=1` in the precondition of `add` indicates that the parameter `n` is not `null`. 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 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 frame condition $\bigwedge_S S' = S$ in the `ensures` clause, where S ranges over sets not listed in the `modifies` clause. For example, the modifies clause of the `add` procedure means that the `add` procedure does not change the `Iter` set.

Boolean algebra of sets. Procedure preconditions, postconditions, and invariants are first-order logic formulas in the language of Boolean algebras of sets (that is, monadic second-order logic of a set), which is a decidable theory for reasoning about sets of uninterpreted elements [31, 40]. Formulas in the language of Boolean algebras 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) r k` on sets with constant cardinality bound `k` and some ordering or equality relation `r` $\in \{=, <, \leq, \geq, >\}$. 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 with an iterator 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 follow the implementation and the intended use of these procedures. The developer initiates the iteration by calling `openIter`, which initializes `Iter` to contain all members of `Content`; this procedure requires that `Iter` be empty upon entry. The `openIter` precondition requires that each iteration must end before a new iteration may begin. 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 as follows: the analysis assumes the invariant upon entry to the module and proves it upon exit; because the sets `Iter` and `Content` are encapsulated within the `DLLIter` module, showing that the invariant holds in the initial program state and that the invariant always holds upon exit is sufficient to show that the invariant always holds outside the `DLLIter` module. Note that, together, the invariant `Iter in Content` and the `openIter` and `nextIter` specifications naturally express the essence of iteration over a set.

In summary, a specification section of a module contains specification variables, invariants between specification variables, and a procedure contract for each public procedure, where a contract may contain a `requires`, `modifies`, and `ensures` clause.

3.3 Abstraction Section

The abstraction section of a module establishes the connection between the implementation and the specification section of a module.

```
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

Abstraction function. Note that the implementation section manipulates concrete variables; on the other hand, procedure contracts are written in terms of abstract specification variables. To verify that a procedure implementation conforms to its contract, it is therefore necessary to give the definition for specification variables. These definitions provide an abstraction function that maps the concrete state into the abstract state. An abstraction section of a module therefore defines the meaning of each specification variable in terms of concrete variables. Figure 4 presents the abstraction section of the iterable list module. The module 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.

Analysis plugins. Given the meaning of specification variables, procedure contracts yield pre and postconditions expressed in terms of concrete states; this pre/postcondition pair is the correctness condition for the implementation of the procedure. Note, however, that verifying different data structures may require different reasoning techniques. Our system supports this view by allowing each module to be analyzed using a different analysis plugin. The `use plugin` keywords in the abstraction section of a module specify the plugin that the system should invoke to verify the module. As Figure 4 indicates, the Hob system will invoke the PALE plugin to analyze the iterable list module; this choice of plugin is appropriate because the PALE tool is specialized for analyzing linked data structures. Because the PALE plugin handles the translated procedure contracts, the syntax of set specification variables is the syntax used by the PALE tool [42] and is written in quotes.

Representation invariants. The `requires` clauses and specification module invariants indicate those preconditions of operations that are expressible solely in terms of abstract specification variables, for example the condition `Iter in Content`. However, many data structure consistency properties are not expressible in terms of desired specification variables, because the abstraction function loses information. In the iterable list example, one such property is the fact that the `prev` fields are the inverse of the corresponding `next` fields. Such a property is needed for the correct implementation of `remove` operation, but is not expressible in terms of `Content` and `Iter`; the definitions of these sets do not even depend on the `prev` field. We use representation invariants to specify such properties. The abstraction section of each module specifies the representation invariants using a language specific to the analysis plugin. Representation invariants in Figure 4 use the notation of graph types [42] to indicate that 1) a doubly linked list is a linked data structure whose backbone is rooted at the global reference variable `root` 2) that the list backbone is spanned by the `next` field, and 3) that the `prev` field is the inverse of the `next` field. Because representation invariants mention concrete variables of a module, they are only visible while analyzing the implementation of a module. To be able to assume that the invariant holds at the entry of public procedures, the invariant is proved when the control leaves the module, and is also proved to hold 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).

3.4 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 possible renaming of its formats. 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` format replaced by the `Cell` format. Modules generated using instantiation otherwise behave no differently from other modules.

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.

Note that, in both contexts, the client code 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 a simpler but more efficient plugin to verify the consistency of data structure uses. In particular, Hob verifies that the precondition for `nextIter`, namely that 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, after successful completion, that the unexposed set is empty, by showing that the `Iter` set equals the `Content` set during every iteration, and that the `Iter` set is empty upon successful completion. The proof of this precondition succeeds because, before each loop iteration, the `Iter` and `Content` sets are equal; in each iteration, `nextIter` removes an element from the `Iter` set, and `setExposed` removes the element from the `Content` set, preserving the invariant. Since `Iter` is empty after the loop, so is `Content`.

```
proc revealAllUnexposed() {
    UnexposedList.openIter();
    bool b = UnexposedList.isLastIter();
    while "... & (b’ <=> (UnexposedList.Iter’ = {})) &
                 (UnexposedList.Iter’ = UnexposedList.Content’)" (!b) {
        Cell c = UnexposedList.nextIter();
        setExposed(c, true);
        b = UnexposedList.isLastIter();
    }
}

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;
}
```

Fig. 5. Doubly-Linked List Client

Our analysis uses an additional (developer-specified) invariant to ensure that the `openIter` precondition holds prior to calls to `openIter`:

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

The analysis checks this invariant by assuming it holds upon entry to the `Board` module and checking that it is true upon exit. The specification also declares the default `not peeking`, which implies that (unless the default is explicitly suspended) the `Iter` set is empty upon entry to each procedure. Within the `peek` procedure, our analysis explicitly tracks the emptiness of the `Iter` set; it guarantees emptiness before the second `openIter` call and upon exit.

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 which performs a simple 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; 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 maintenance of data structure consistency, and without 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 sequel 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.

4 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 section of M, as well as the specification sections of all modules whose procedures M invokes. Given these requirements, the details of the analysis are entirely plugin-specific, which gives our system great flexibility in leveraging different analysis techniques.

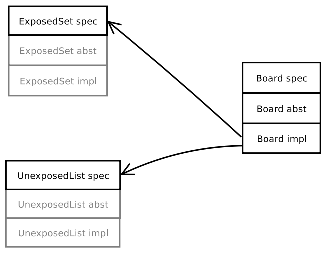

Fig. 6. Checking implementation of minesweeper board

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

In the following sections, we describe the three plugins that we have implemented in our analysis framework.

4.1 The PALE Analysis Plugin

Our PALE plugin uses a previously implemented tool, Pointer Assertion Logic Engine [42] (PALE). We have incorporated PALE into our framework with essentially no changes to the tool itself.

Using PALE plugin. As illustrated in Section 3, the developer specifies the abstraction function for a graph type data structure 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 "`data root: Node`" declaration; or the declaration of a non-backbone routing restriction, such as the "`pointer current: Node`" declaration.

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 the data fields. In addition to data fields, a member object has the 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 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, which enables objects to participate in multiple data structures.

The PALE plugin use the 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 loop invariants in implementation modules verified by the PALE plugin contain two parts. The first part contains concrete data structure properties, and is literally transferred into the PALE implementation language. The second part contains abstract set properties, and is translated in the same way that the requires and ensures clauses are translated. Our translation also elides integer variables from the input program; integer variables are not supported by the PALE input language.

A sketch of the translation is the following. 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:

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

A formula $B(S_1, \ldots, S_n)$ in the boolean algebra of sets then corresponds to the formula

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

The translator then uses $\mathtt{isS}_i$ predicates to translate the specification of a procedure p. In the example of the `remove` procedure whose implementation is in Figure 2 and whose contract is in Figure 3, we obtain PALE code of the form:

```
set Content : Node;
set Iter : Node;
/* precondition */
[isContent(Content) ∧ isIter(Iter) ∧ n ∈ Content ∧ Iter ⊆ Content]

{stmts}

/* postcondition */
[existset Content′ of Node : isContent(Content′) ∧
 existset Iter′ of Node : isIter(Iter′) ∧
 Content′ = Content \ {n} ∧ Iter′ = Iter \ {n} ∧ Iter′ ⊆ Content′]
```

We note that the actual translation is slightly more complicated, in part because our PALE plugin introduces additional instrumentation fields that conceptually store all objects that are not part of the currently analyzed data structure.

Consequences. The PALE analysis package implements a sophisticated analysis that can verify detailed properties of complex linked data structures. It is clearly infeasible (for scalability reasons) to use PALE to analyze anything other than encapsulated data structure implementations. 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. In the next section we show how to verify potentially even more detailed properties using theorem proving.

4.2 The Theorem Proving Plugin

The theorem proving plugin [50] generates verification conditions using weakest preconditions 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. The state of the `ExposedSet` 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.

```
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;
    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;
    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

One of the procedures in the `ExposedSet` module is the `add` procedure, which adds a `Node` to the set of `Node`s 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, 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:

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

```
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
    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

The analysis then conjoins both the precondition and postcondition with the representation invariants specified in the abstraction section. In our example we have 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 [22]. The result of the translation is given in Figure 10.

```
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

Using weakest precondition semantics, the analysis then creates a formula from the translated code; the validity of this formula implies the conformance of the procedure with respect to its specification.

To simplify the task of discharging the resulting verification condition, the formula is split into as many sequents as possible by performing a simple non-backtracking natural-deduction search through connectives $\forall$, $\Rightarrow$, $\wedge$. The analysis then attempts to verify each sequent in turn. It first searches a library of previously proven lemma for a match to the current sequent. If it does not find a match, the analysis attempts to discharge the sequent using the proof hints that may be supplied in the procedure code.

If no hint is supplied, it invokes Isabelle's built-in simplifier and classical reasoner with array axioms.

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

4.3 The Flag Plugin

Our flag analysis [36] 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, as well as the correspondence between the concrete and the abstract boolean variables.

```
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

As an illustration, Figure 11 presents the abstraction section of the `board` module, which contains definitions of sets `MarkedCells`, `ExposedCells`, `UnexposedCells`, and `MinedCells` as well as several global boolean variables. All introduced sets are intersected with a set of initialized variables, which ensures that all abstract sets are initially empty and do not change when a different module allocates an object using the `new` statement.

The flag analysis performs abstract interpretation [12] 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 [29]. In our experience, applying several formula transformations drastically reduced the size of the formulas emitted 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 [38].

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 sets `ExposedList.Content` and `UnexposedList.Content`. This allows the analysis to verify invariants such as `Board.ExposedCells = ExposedList.Content` and `Board.UnexposedCells = UnexposedList.Content` in addition to invariants such as `disjoint(MarkedCells, ExposedCells)`. The last invariant and its consequence `disjoint(ExposedCells.Content, UnexposedList.Content)` are examples of high-level data structure invariants that correlate the values of sets that correspond to multiple data structures.

Generalizations of typestate. There are several perspectives on the use of an analysis plugin such as the flag plugin. 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 3.4) 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.

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 been already 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 standard typestate systems, each object has a single atomic typestate. In our formulation, however, 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 this approach 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 completely absent in existing flat typestate systems. Our formulation cleanly supports typestate hierarchies—a collection of sets can partition a more general set, with the subset inclusion ordering capturing the hierarchy.

– **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. If an object participates in multiple data structures, its typestate characterizes this sharing by indicating its membership in multiple typestate sets, one for each data structure. This formulation supports nonmonotonic changes—the set of objects that contain an element may change arbitrarily throughout the computation.

5 Related Work

Modularity mechanism. The ability to encapsulate individual object fields in separate modules is presented in [11], and is used in the Cecil programming language [9] as well as in intermediate languages of static checking tools [21, 34].

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 [33, 42, 46]. These analyses focus on extracting or verifying detailed consistency properties of individual data structures. While these analyses are very precise, the detail of the properties that they must track have limited their scalability. One of our primary research goals is to enable the application of these sophisticated analyses in a modular fashion, with each analysis operating on only that part of the program relevant for the properties that it is designed to verify.

Typestate systems. Typestate systems track the conceptual states that each object goes through during its lifetime in the computation [15, 48]. They 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 [36, 38]. 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. Our generalizations of typestate include multiple orthogonal typestates (corresponding to multiple sets), and, most importantly, the ability to verify the actual property associated with the typestate abstraction, as opposed to taking for granted the correctness of interface specifications.

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 [40, 47] and is presented in [2, Chapter 4]. The complexity of this problem is alternating exponential time [32]. To our knowledge, the only tool that can decide the first-order theory of boolean algebras is MONA [30]; 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 algebras with Presburger arithmetic operations is presented in [35]; this extension allows reasoning about sizes of data structures.

Program verification and checking tools. Methodology for using formal specifications in software development include Gypsy [24], B method [1], VDM [27], Z [49], and RAISE [13]. Tool support for these methods includes verification condition generators and proof assistants, but not sophisticated static analyses such as shape analysis. Tools based on verification condition generation and theorem proving include [25, 28, 44], and, more recently, [19, 20, 41]. ESC/Java [21] is a program checking tool whose purpose is to identify common errors in programs using program specifications in a subset of the Java Modelling Language [6]. ESC/Java sacrifices soundness in that it does not model all details of the program heap, but can detect some common programming errors. A more recent effort is a sound static analysis tool which comes with a particular methodology for modular treatment of invariants in an object-oriented language that extends $C\#$ [4]. Other tools focus on verifying properties of concurrent programs [5,8] or device drivers [3,26]. One important difference between this research and our research is that our research is designed not to develop a single new analysis algorithm or technique, but rather to enable the application of multiple analyses that check arbitrarily complicated properties within a single program. Our system currently supports loose integration of analyses where each analysis applies to one module, which makes incorporation of external tools easy. An approach that proposes a tighter combination of a particular domain (uninterpreted function symbols) with an arbitrary base domain is presented in [10].

6 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.

References

1. 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*, pages 398–405. Springer-Verlag, 1991.
2. W. Ackermann. *Solvable Cases of the Decision Problem.* North Holland, 1954.
3. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, 2001.
4. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
5. 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*, volume 1102, pages 415–418, 1996.
6. L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. Technical Report NII-R0309, Computing Science Institute, Univ. of Nijmegen, March 2003.
7. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. In *ICSE 2003*, 2003.
8. S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *29th ACM SIGPLAN-SIGACT POPL*, pages 45–57. ACM Press, 2002.

9. C. Chambers and the Cecil Group. The Cecil language specification and rationale. Technical report, Department of Computer Science and Engineering, University of Washington, February 2004.
10. B.-Y. E. Chang and K. R. M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI'05*, January 2005.
11. D. R. Cheriton and M. E. Wolf. Extensions for multi-module records in conventional programming languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 296–306. ACM Press, 1987.
12. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. 6th POPL*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.
13. B. Dandanell. Rigorous development using RAISE. In *Proceedings of the conference on Software for citical systems*, pages 29–43. ACM Press, 1991.
14. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM PLDI*, 2001.
15. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *Proc. 15th ECOOP*, LNCS 2072, pages 130–149. Springer, 2001.
16. M. Fahndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proc. ACM PLDI*, 2002.
17. M. Fähndrich and K. R. M. Leino. Heap monotonic typestates. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2003.
18. J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*. Springer, 2003.
19. J.-C. Filliatre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, 2003.
20. J.-C. Filliatre and C. Marché. Multi-prover verification of c programs. In *ICFEM'04*, 2004.
21. C. Flanagan, K. R. M. Leino, M. Lilibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proc. ACM PLDI*, 2002.
22. C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proc. 28th ACM POPL*, 2001.
23. E. Gamma, R. Helm, R. Johnson, and J. Vlisside. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1994.
24. D. I. Good, R. L. Akers, and L. M. Smith. Report on Gypsy 2.05. Technical report, University of Texas at Austin, February 1986.
25. J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.
26. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *31st POPL*, 2004.
27. C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International (UK) Ltd., 1986.
28. J. C. King. *A Program Verifier*. PhD thesis, CMU, 1970.
29. 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.
30. N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. In *Proc. 5th International Conference on Implementation and Application of Automata*. LNCS, 2000.
31. D. Kozen. Complexity of boolean algebras. *Theoretical Computer Science*, 10:221–247, 1980.
32. D. Kozen. Logical aspects of set constraints. In *Proc. 1993 Conf. Computer Science Logic (CSL'93)*, volume 832 of *Lecture Notes in Computer Science*, pages 175–188, 1993.

33. V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proc. 29th POPL*, 2002.
34. 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.
35. V. Kuncak and M. Rinard. The first-order theory of sets with cardinality constraints is decidable. Technical Report 958, MIT CSAIL, July 2004.
36. P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking using set interfaces and pluggable analyses. *SIGPLAN Notices*, 39:46–55, March 2004.
37. P. Lam, V. Kuncak, and M. Rinard. On our experience with modular pluggable analyses. Technical Report 965, MIT CSAIL, September 2004.
38. P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking for data structure consistency. In *6th International Conference on Verification, Model Checking and Abstract Interpretation*, 2005.
39. P. Lam, V. Kuncak, K. Zee, and M. Rinard. The Hob project web page. http://catfish.csail.mit.edu/∼plam/hob/, 2004.
40. L. Loewenheim. Über mögligkeiten im relativkalkül. *Math. Annalen*, 76:228–251, 1915.
41. 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. to appear.
42. A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. ACM PLDI*, 2001.
43. 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.
44. G. Nelson. Techniques for program verification. Technical report, XEROX Palo Alto Research Center, 1981.
45. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
46. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.
47. 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.
48. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, January 1986.
49. J. Woodcock and J. Davies. *Using Z*. Prentice-Hall, Inc., 1996.
50. 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*, 2004.