On Our Experience with Modular Pluggable Analyses

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

Abstract

We present 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 plugin for modules that encapsulate linked data structures such as lists and trees, and an array plugin in which abstract set membership is determined by membership in an array. Our experimental results indicate that our approach makes it possible to effectively combine multiple analyses to verify properties that involve objects shared by multiple modules, with each analysis analyzing only those modules for which it is appropriate.

1 Introduction

Data structure consistency is important for successful program execution — if an error corrupts a program's data structures, the program can quickly exhibit unacceptable behavior or even crash. Motivated in part by the importance of this problem, researchers have developed algorithms for verifying that programs preserve important consistency properties [3, 7, 13, 29–31].

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. 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.

1.1 Target Application Class

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.

We have implemented our analysis framework and populated this framework with three analysis plugins: 1) the flags plugin, which is designed to analyze modules that use a flag field to indicate the typestate of the ob-

jects that they manipulate; 2) the PALE plugin, which implements a shape analysis for linked data structures (we integrated Anders Møller's implementation [30] of the Pointer Analysis Logic Engine analysis tool into our system); and 3) the array verification plugin, which generates verification conditions for consistency properties of array-based data structures. Verification conditions from the array verification plugin are designed to be discharged manually using the Isabelle interactive theorem prover. We have used our analysis framework to analyze several programs; our experience shows that it can effectively 1) verify the consistency of data structures encapsulated within a single module and 2) combine analysis results from different analysis plugins to verify properties involving objects shared by multiple modules analyzed by different analyses.

1.2   Contributions

The contributions of this paper are the following:

- **Pluggable Analysis Framework:** We show how to apply multiple analyses to multiple data structures encapsulated within multiple modules, with the analysis results appropriately combined to verify properties that span multiple modules. The approach supports sharing patterns in which objects move between different data structures and patterns in which objects participate in multiple data structures simultaneously.

  We introduce *abstract sets* as the key abstraction that each analysis uses to characterize how objects participate in encapsulated data structures. The connection between sets and concrete data structure consistency properties enables modules to express the data structure participation requirements that externally accessible objects must satisfy without exposing the data structure representation to their clients. The set abstraction also enables different analyses to interoperate to verify properties that span multiple data structures and modules.

  We show how to use the common set abstraction to specify and verify *global invariants* that correlate membership of objects in different data structures analyzed by different analysis plugins (Section 3).

  We provide mechanisms — *scopes* and *defaults* — which allow developers to write strictly local specifications of procedures, without having to explicitly include global invariants. Our system then automatically conjoins these global invariants when appropriate.

- **Analysis Plugins:** We present three analysis plugins that show how our approach works in practice:

a *flag typestate analysis* for modules in which set membership is determined by the value of a flag field in each object (Section 5), the *PALE analysis* plugin for modules that manipulate linked data structures such as lists and trees (Section 6), and the *array analysis plugin* that can verify arbitrarily complex properties of array-based data structures by generating verification conditions and discharging them using an interactive theorem prover (Section 7). The typestate plugin can be thought of as a scalable plugin that propagates and verifies membership of objects in global sets; the PALE plugin is an example of a more precise shape analysis plugin; and the array analysis plugin is an extreme point that in principle has no bound on the complexity of properties that it can verify. More precise analysis plugins may require more analysis time or more interaction with the user; this cost is amortized because these analyses are typically applied to instantiatable modules that encapsulate reusable data structures.

- **Experimental Evaluation:** We present our experience using our implemented system to analyze programs that require the use of multiple analysis plugins to verify important consistency and typestate properties (Section 8).

2   Example

We next discuss an example program that shows how to use our approach to verify 1) the consistency of individual data structures encapsulated in instantiatable modules, 2) that the rest of the program uses each module correctly, and 3) important properties that involve data structures encapsulated in different modules. Our example program implements the popular minesweeper game.[1] Figures 1, 2, and 3 present a linked list module used in our example minesweeper program. This module has a specification section (Figure 1), an implementation section (Figure 2), and an abstraction section (Figure 3). The abstraction section specifies the relationship between the concrete data structure implementation and the abstract set specification, and enables the PALE plugin to check that the implementation satisfies its specification.

The abstract `Content` set in Figure 1 represents the contents of the list. (The notation `Content'` denotes the new version of `Content` after a procedure executes; the unprimed `Content` denotes the old version before it executes). The procedures in the `List` module use this

---

[1]Full source code for the minesweeper example and other case studies, the interpreter for our language, and analysis engine is available at `http://cag.csail.mit.edu/∼plam/mpa`.

set to express their preconditions, postconditions, and effects. The `requires` clause of the `add` procedure, for example, requires that the parameter `e` (which the `add` procedure will insert into the list) not already be in the `Content` set. The `ensures` clause states that the effect of the `add` procedure is to add the parameter `e` to the `Content` set. The `modifies` clause indicates that the procedure modifies the `Content` set only.

Procedure specifications can also express cardinality constraints. The requires clause of the `removeFirst` procedure, for example, uses the formula `card(Content)>=1` to require that the `Content` set be nonempty upon entry.

An analysis based on monadic second-order logic over trees (as implemented in the PALE analysis tool [30]) is able to verify that the `List` implementation correctly implements its specification. However, it needs some additional information to do so. The abstraction section in Figure 3 provides this information.

This abstraction section starts by identifying the analysis plugin used to verify this module; in this case the PALE analysis plugin. This analysis plugin implements a decision procedure for the monadic second-order logic over trees and uses this decision procedure to analyze procedures that manipulate recursive linked data structures such as lists and trees [20]. To enable the application of this analysis to the `List` module, the abstraction section identifies the correspondence between the abstract sets in the specification and the concrete data structure encapsulated inside the module. The statement `Content = {x : Entry | "root<next*>x"};` defines the `Content` set to be all objects `x` reachable by following `next` fields starting from the `root` variable[2].

The analysis uses this correspondence to translate the `requires`, `ensures`, and `modifies` clauses (expressed in terms of abstract sets) into properties of the concrete data structure (which in this case are expressed in monadic second-order logic over the objects and fields in the concrete heap). For example, the translated precondition of `add` is `!root<next*>e`, which states that `e` is not reachable by following `next` fields starting at the `root`. The analysis then uses the translated `requires` clause as a precondition and the translated `ensures` clause as a postcondition of each procedure. Note that other modules need not be aware of how membership in `Content` is determined; they simply use the `Content` set in their own specifications, as needed.

So far, we have presented a generic `List` mod-

---

[2]This implementation places the `next` field directly in the `Entry` objects. Our approach also supports the more common implementation that uses auxiliary encapsulated list objects to refer to the `Entry` objects; in that implementation the auxiliary list objects (and not the `Entry` objects) contain the `next` fields.

```
spec module List {
   format Entry;
   sets Content : Entry;

   proc add(e : Entry)
     requires not (e in Content)
     modifies Content
     ensures Content' = Content + e;

   proc removeFirst() returns f : Entry
     requires card(Content)>=1     // Content nonempty
     modifies Content
     ensures (Content' = Content - f) &
             card(f)=1 & (f in Content);

   proc isEmpty() returns b : bool
     ensures not b <=> (card(Content') >= 1);
}
```

Figure 1: Linked List Specification Section

```
impl module List {
   format Entry { next : Entry; } // see footnote 2
   reference root : Entry;

   proc add(e : Entry) {
     /* add to the beginning of the list */
      if (root==null) {
       root = e;
       e.next = null;
     } else {
         e.next = root; root = e;
     }
   }

   proc removeFirst() returns f : Entry {
     Entry e = root;
     root = root.next;
     e.next = null;
     return e;
   }

   proc isEmpty() returns b : bool {
     return root == null;
   }
}
```

Figure 2: Linked List Implementation Section

```
abst module List {
   use plugin "PALE";
   Content = {x : Entry | "root<next*>x"};

   invariant "type Entry = {
               data next : Entry;
             }";
   invariant "data root:Entry;";
}
```

Figure 3: Linked List Abstraction Section

ule. In the minesweeper example, we instantiate this `List` module as an `UnexposedList`, which stores minesweeper board cells which have not yet been cleared. The instantiation mechanism of our language substitutes the generic `Entry` format with the `Cell` format used in the rest of the minesweeper example. Conceptually, instantiation creates a fresh copy of the instantiated module, carrying out substitutions as appropriate to make the generic implementation applicable to the particular use at hand.

2.1 Verifying Cross-Module Properties

We next illustrate how our approach enables the verification of properties that span multiple modules. Our minesweeper implementation has several modules (see Figure 4): a game board module (which represents the game state), a controller module (which responds to user input), a view module (which produces the game's output), an exposed cell module (which stores the exposed cells in an array), and an unexposed cell module (which stores the unexposed cells in an instantiated linked list). 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.

Our minesweeper implementation uses the standard model-view-controller (MVC) design pattern [15]. The `board` module (which stores an array of `Cell` objects) implements the model part of the MVC pattern. Each `Cell` object may be mined, exposed or marked. The `board` module represents this state information by contributing `isMined`, `isExposed` and `isMarked` flags to `Cell` objects. At an abstract level, the sets `MarkedCells`, `MinedCells`, `ExposedCells`, `UnexposedCells`, and `U` (for Universe) represent sets of cells with various properties; the `U` set contains all cells known to the board. The board also uses a flag `gameOver`, which it sets to `true` when the game ends.
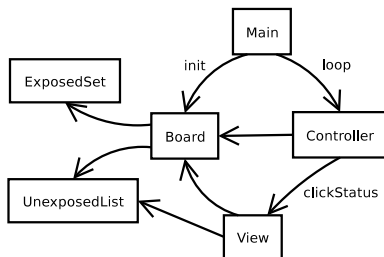

Figure 4: Modules in Minesweeper implementation

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

- Unless the game is over, the set of mined cells is disjoint from the set of exposed cells.

- The sets of exposed and unexposed cells are disjoint.

- The set of unexposed cells maintained in the `board` module is identical to the set of unexposed cells maintained in the `UnexposedList` list.

- The set of exposed cells maintained in the `board` module is identical to the set of exposed cells maintained in the `ExposedSet` array.

- At the end of the game, all cells are revealed; *i.e.* the set of unexposed cells is empty.

We next explain how our system verifies the fourth and fifth properties listed above. Note that the `board` module, which is analyzed by the flag plugin, defines its sets using flag values, so that `board`'s set of exposed cells consists of all objects with the field `isExposed` set to `true`, whereas the `ExposedSet` module defines its contents set by array membership.

Although our system focuses on using sets to model program state, not every module needs to define its own abstract sets. Indeed, certain modules may not define any abstract sets of their own, but instead coordinate the activity of other modules to accomplish tasks. The view and controller modules are examples of such modules. The view module has no state at all; it queries the board for the current game state and calls the system graphics libraries to display the state.

Because these modules coordinate the actions of other modules — and do not encapsulate any data structures of their own — the analysis of these modules must operate solely at the level of abstract sets. We therefore analyze these modules using a subset of the flag plugin. This subset tracks abstract set membership, solves formulas in the boolean algebra of sets, and incorporates the effects of invoked procedures as it analyzes each module. It does not, however, need to reason about the correspondence between the concrete data structure representations and the abstract sets.

The analysis of the view and controller modules illustrates a core idea behind our approach: we use faster, less-detailed analyses on high-level modules that primarily coordinate the actions of other modules, and apply more-precise analyses to verify leaf modules that encapsulate implementations of sophisticated data structures.

Note that the set abstraction supports typestate-style reasoning at a per-object level (for example, all objects in the `ExposedCells` set can be viewed as having a conceptual typestate `Exposed`). Our system also supports the notion of global typestate: for instance,

the `board` module has a global `gameOver` variable which indicates whether or not the game is over. Using this variable and the definitions of sets, we maintain the global invariant

$$\texttt{gameOver} \mid \texttt{disjoint(MinedCells, ExposedCells)}.$$

This global invariant connects a global typestate property — is the game over? — with a object-based typestate state property evaluated on objects in the program — there are no mined cells that are also exposed. Our analysis plugins verify these global invariants by conjoining them to the preconditions and postconditions of methods. Note that global invariants must be true in the initial state of the program; if some initializer must execute to establish an invariant, then the invariant can be guarded by a global typestate property.

Another invariant concerns the correspondence between the `ExposedCells` and `UnexposedCells` sets with the `ExposedSet.Content` and `UnexposedList.Content` sets.

```
(ExposedCells = ExposedSet.Content) &
(UnexposedCells = UnexposedList.Content)
```

Recall that, in our example, the `ExposedSet`, the `UnexposedList`, and the `board` are all implemented in different modules, and are analyzed by different analysis plugins. This invariant verifies that a set defined by field values is equal to a set defined by reachability in the heap, and that a set defined by field values is equal to a set defined by membership in an array.

Our analysis ensures this property by conjoining it to `ensures` and `requires` clauses of appropriate procedures. The `board` module is responsible for maintaining this invariant. However, the flag analysis used for the board module does not, in isolation, have the ability to verify the invariant, because it cannot reason about the heap structure of the program. Because we have a common set specification language, though, the flag analysis can successfully use the `ensures` clause of its callees, along with its own analysis tracking `ExposedCells` membership, to guarantee the invariant.

3 Modular Analysis Framework

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

3.1 Implementation Language

Implementation sections for modules in our system are written in a standard memory-safe imperative language supporting arrays and the dynamic allocation of objects.[3] Analysis plugins use our system's core libraries to easily manipulate abstract syntax trees for this imperative language. Using these libraries, we have implemented an interpreter for our language; it would also be straightforward to write a compiler for our language.

We point out one special feature of our imperative language, which we call *formats*. Formats aid modular reasoning about shared objects by encapsulating fields while allowing modules to share objects. When the program creates an object with format T, the newly-created object contains the fields contributed to format T by all modules in the program [8]. A simple type checker for the implementation language statically ensures that each module accesses only fields that it has contributed to an object. Note that no analysis plugin needs the full layout of an object; it will only need the fields which the module under analysis has contributed to that object.

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

3.2 Specification Language

Figure 5 presents the syntax for the module specification language. A specification section contains a list of set definitions and procedure specifications, and lists the names of formats used in set definitions and procedure specifications. Set declarations identify the module's abstract sets, while boolean variable declarations identify the module's abstract boolean variables. Each procedure specification contains a `requires`, `modifies`, and `ensures` clause. The `modifies` clause identifies sets whose elements may change as a result of executing the procedure. The `requires` clause identifies the precondition that the procedure requires to execute correctly; the `ensures` clauses identifies the postcondition that the procedure ensures when called in program states that satisfy the `requires` condition. Both `requires` and `ensures` clauses use arbitrary first-order formulas B in the language of boolean algebras extended with

---

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

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

Figure 5: Syntax of Module Specification Language

cardinality constraints. Specification sections may also contain invariants in the same language; these invariants are automatically conjoined with `requires` and `ensures` clauses of procedures in that module. Free variables of these formulas denote abstract sets declared in specification sections. The expressive power of such formulas is the first-order theory of boolean algebras, which is decidable [21,28]. The decidability of the specification language ensures that analysis plugins can precisely propagate the specified relations between the abstract sets.

3.3 Analysis Overview

The analysis of a module M is performed by the analysis plugin specified in the abstraction section of module M. The abstraction section of module M establishes the connection between the specification and implementation sections of module M. Each analysis plugin augments the generic syntax of abstraction sections with a plugin-specific *plugin annotation language*. The plugin annotation language is used to define the mapping between the concrete and abstract representations of sets. The abstraction section of module M may additionally state representation invariants for the data structure implementing the abstract sets. The responsibility of each plugin is to guarantee that each procedure satisfies its specification; it may do so by any means practical. The specification of a procedure is derived from the abstract `requires`, `modifies`, and `ensures` clauses using the definitions of abstract sets as well as the representation invariants [25]. We also require that a procedure never violates the preconditions of its callees.

Figure 6 illustrates our analysis of the `board` module from minesweeper: to ensure that `board` meets its specification, the flag 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.

We have implemented three plugins in our analysis framework: a flags plugin, which assigns set membership based on field values (Section 5), a PALE plugin, which assigns set membership based on heap reachability (Section 6) and an array plugin, which assigns set membership based on array membership (Section 7).
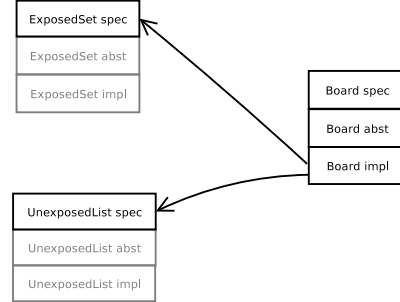

Figure 6: Checking implementation of minesweeper board

4 Scopes and Defaults

In this section, we present the notions of scopes and defaults. These notions enable developers to write more concise specifications when using our modular analysis framework.

Scopes serve two purposes: they enable the specification and verification of cross-module invariants by identifying the subset of a program in which an invariant is expected to hold, and they combat specification aggregation by hiding irrelevant sets from callers. Scopes are key to our system's verification of invariants containing sets from different modules: by designating certain modules as public access points, we ensure that scope invariants always hold outside their declaring scope. Scopes also shield callers from irrelevant detail: only sets from exported modules are visible to modules in different scopes. This serves to bound the detail required in procedure specifications: the specification of procedure p belonging to scope $\mathcal{C}$ need only contain the effects of procedures on sets in $\mathcal{C}$ and exported sets outside $\mathcal{C}$.

Defaults allow procedure specifications to simplify specifications in a different way. Using defaults, the developer can factor out common conjuncts that repeatedly appear in a module's procedure specifications. These conjuncts need only be written once per module, and are automatically conjoined to procedure specifications for that module, unless they are specifically *suspended* at a procedure.

$$S \quad ::= \quad \text{scope } s$$
$$\{\text{modules } M^*; \text{ exports } M^*; \text{ [invariant } B;]\ \}$$

Figure 7: Grammar for Scope Declarations

4.1 Scopes for Specifying Invariants that Cross Module Boundaries

Consider module `Main` which calls module `Worker`. The `Worker` module uses two helper modules, `Inbox` and `Outbox`, which define sets `Input` and `Output` respectively. The `Worker` module itself defines the `Jobs` set, which satisfies the cross-cutting invariant

$$I : \text{Jobs} = \text{Inbox.Input} \cup \text{Outbox.Output},$$

which must hold on entry to `Worker` and is always ensured upon exit from `Worker`. Ordinarily, specifications for procedures of `Main` must therefore include the invariant I in their own preconditions and postconditions to be able to call `Worker`; worse yet, any transitive caller of `Main` also needs to include I. We call this problem *specification aggregation*, and we describe scopes, our solution to the specification aggregation problem.

Syntax of Scopes. Figure 7 presents the syntax of scope declarations. A scope declaration contains a set of modules; a subset of these modules are declared as exported modules. Scope declarations may also contain a scope invariant.

We describe the components of a scope declaration using a typical scope C. *Exported modules* are accessible from outside C: that is, only procedures in exported modules may be called from outside C, and only sets declared in exported modules may appear in specifications outside C. *Private modules* belong to C but are not exported. Sets belonging to private modules are *private sets*. An *invariant B* is a boolean algebra formula which is guaranteed to be true in the initial state of the program, assumed to hold at all incoming boundary points, and verified at all outgoing boundary points.

Multiple Orthogonal Scopes. Note that a module can participate in multiple scopes at the same time; this multiple participation enables modules to be grouped into scopes along orthogonal axes. For most purposes, we can reason about scopes individually, since they are independent of each other. We discuss multiple scopes in the context of calling restrictions. Each module combines the calling restrictions from all of its scopes: if M is a private module in some scope C, only modules that are also in C can invoke M, and if M is exported in C[4], only modules that are not in C can call M.

---

[4]We need only disallow calls to exported modules of C if C has an invariant.

Scope Calling Condition. Our analysis verifies that the program satisfies the following scope calling condition. This condition ensures that the program's scope invariants hold at scope boundary points, defined below.

Let $\text{scopes}(M)$ denote the set of scopes C such that C declares M in its `modules` clause, and let $\text{exportingScopes}(M)$ denote the set of scopes C such that C declares M in its `exports` clause. Let the "private yard" of module M be $\text{yard}(M) = \text{scopes}(M) \setminus \text{exportingScopes}(M)$. A procedure call from M' to M is allowed if and only if M is exported in precisely the scopes $C \in \text{scopes}(M) \setminus \text{scopes}(M')$ of the scope difference. More precisely, we say that module M' calls module M if the body of some procedure in the implementation of module M' contains a call to a procedure declared in module M. We then require the following inter-scope call condition to be satisfied for every pair of modules (M', M): if module M' calls module M, then

$$\text{scopes}(M) \setminus \text{scopes}(M') \subseteq \text{exportingScopes}(M)$$
$$\wedge \quad \text{scopes}(M) \cap \text{scopes}(M') \subseteq \text{yard}(M).$$

The first conjunct of the calling condition ensures that the incoming boundary points are the only points at which execution can enter a scope, and the second conjunct ensures that between any two instances of incoming boundary points in an execution trace, at least one outgoing boundary point occurs.

Semantics of Scopes and Invariants When our analysis successfully verifies a program, it is certifying that each scope invariant holds in the defining scope's exterior; boundary points separate the interior of a scope from its exterior. Inside a scope, the invariant may be temporarily violated; our analysis then checks that the invariant is restored before the program exits the scope. The semantics for invariants is therefore that the scope invariant may be assumed to hold upon entry to its scope, and the invariant must be verified that the scope invariant holds upon exit.

The set of *incoming boundary points* is defined as the set of the entry points of procedures for exported modules of C and return points for potentially-reentrant call sites inside C. (A return point for a call site is the immediate control-flow successor of the return statement in the call site's target; potentially-reentrant call sites are those that directly invoke a method outside C which, on some execution trace, transitively call back into C.) The *outgoing boundary points* are defined as the exit points of exported procedures, plus all potentially-reentrant sites calling outside C belonging to procedures inside C. Our analysis also checks that from inside C, incoming boundary points will not be

called; in the interior, only procedures belonging to private modules of C, or outside C entirely, may be called.

Our system enables the verification of invariants by placing two constraints on how invariants are defined. First, all invariants must be true in the initial state of the program[5]. Second, an invariant in scope C may only refer to sets and booleans in scope C.

Soundness of Assuming Invariants. We justify our handling of invariants by arguing that whenever we assume an invariant, it must already be true in the underlying dynamic program state. Since the only possible unsoundness in our handling of invariants comes from assuming the invariant, we can show soundness simply by proving that an invariant is always true when our treatment assumes that invariant. In general, invariants will not be directly provable at calling sites by our analysis, because the sets mentioned in the invariant may be private sets invisible to the caller, implying that no information is available about these sets outside the scope, and in particular at the calling site.

Our condition on calling incoming boundary points from within a scope gives us the following property:

Proposition 1 *(Boundary Point Nesting) For all scopes C, all execution traces $s_0, \ldots, s_n$ and all pairs $s_{p_i}, s_{p_{i+1}}$ of incoming boundary points, there exists an outgoing boundary point s_{q_i} such that $p_i < q_i < p_{i+1}$. Similarly, between any two outgoing boundary points is an incoming boundary point.*

Two incoming boundary points will never be adjacent, because our analysis verifies that only procedures belonging to private modules of C or the exterior will be invoked inside C; in the first case, there is no incoming boundary point, and in the second case, we have constructed the outgoing boundary points such that there will always be an outgoing boundary point when the call is potentially-reentrant. Two outgoing boundary points will never be adjacent: after an exit from an exported procedure, control must flow to a procedure outside C (since only the exterior can call an exported procedure), and after a potentially-reentrant call, control also flows to a procedure outside C.

The soundness of our treatment of invariants depends on the following soundness condition on analysis plugins.

Condition 1 *(Set stationarity condition) A set may only be modified by its defining module.*

---

[5]Initialization procedures can be modelled with a `init` boolean variable: `init` $\Rightarrow$ `I` indicates that I holds after initialization. The developer would then use defaults to ensure that `init` is almost always true.

The flag, PALE and array analysis plugins presented in this paper satisfy the set stationarity condition. The following proposition is an immediate consequence of the set stationary condition:

Proposition 2 *A set may only be modified when it is in scope.*

We will prove that the invariant true at incoming boundary points by induction on instances of these points in program execution traces. This proof is in two parts: 1) for entry points of exported modules; and 2) for reentrant-call return sites. We first consider case 1), the incoming boundary points that occur as entry points of exported modules. Consider an arbitrary program execution trace $s_0, s_1, \ldots, s_n$. At the first incoming boundary point s_{i_0}, the invariant is true because it is true at s_0 and has not been changed since then (by the set stationarity condition). At subsequent incoming boundary points s_{i_k}, the invariant will also be true, since an outgoing boundary point will have executed between $s_{i_{k-1}}$ and s_{i_k} (recall we disallow calls to incoming boundary points from inside the scope, so that the program has to pass through an outgoing point), because the invariant was proven at $s_{i_{k-1}}$, and because the truth of the invariant has not changed in the exterior code between the most recent outgoing boundary point and s_{i_k}. Case 2), concerning reentrant-call incoming boundary points, is similar. Consider the sequence $t_0, \ldots, t_n$ of points between the call t_0 and its return t_n. The analysis explicitly proves the invariant at the outgoing boundary point t_0. If a trace has no incoming boundary point t_i between t_0 and t_n, then the invariant still holds at t_n, because of the set stationarity condition. For every incoming boundary point t_i, then there must also exist an outgoing boundary point t_j (by Proposition 1) at which point the invariant is explicitly shown. Between the last outgoing boundary point and t_n (a subsequence which occurs outside the scope), the invariant is preserved, implying that it holds at t_n.

Consequences of Scopes   By using scopes, developers may omit details about transitive callees which are not relevant to understanding the effects of the caller. Furthermore, scope invariants allow the developer to assume that certain invariants always hold upon entry to the scope, which enhances the expressive power of our system.

4.2 Defaults for Simplifying Specifications by Omission

Many modules require that some initialization code be executed before normal operation of the module can proceed. Our system can represent this with an `Init`

boolean predicate attached to the appropriate module, and requiring that `Init` hold before (almost every) procedure in the module. Such a practice clutters procedure specifications with extra conjuncts.

We have created the notion of a *default* to address this problem. Defaults are named boolean clauses which are uniformly conjoined to requires and ensures clauses of procedure specifications unless they are explicitly suspended. Procedure p may declare a suspends clause; if default I is suspended in p, then the default is not applied to the requires and ensures clauses of p.

Defaults differ from scopes in that scopes talk about global cross-cutting concerns, whereas defaults talk about local properties that need to be uniformly woven into specifications inside a module. Defaults and scopes work together allow developers to focus on specifying local properties specifically of interest to a particular specification, by freeing them from the obligation of specifying details of global interest in each procedure specification.

5 The Flag Plugin

Our flag analysis verifies that modules implement set specifications in which integer or boolean flags indicate abstract set membership. The developer specifies (using the flag abstraction language) 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. Figure 8 presents the syntax for our flag abstraction modules. This abstraction language defines abstract sets in two ways: (1) directly, by stating a base set; or (2) indirectly, as a set-algebraic combination of sets. *Base sets* have the form $B = \{x : T \mid x.f{=}c\}$ and include precisely the objects of type T whose field `f` has value `c`, where `c` is an integer or boolean constant; the analysis converts mutations of the field `f` into set-algebraic modifications of the set B. *Derived sets* are defined as set algebra combinations of other sets; the flag analysis handles derived sets by conjoining the definitions of derived sets (in terms of base sets) to each verification condition and tracking the contents of the base sets. Derived sets may use named base sets in their definitions, but they may also use *anonymous* sets given by set comprehensions; the flag analysis assigns internal names to anonymous sets and tracks their values to compute the values of derived sets.

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. Section 5.4 describes these formula optimizations. These transformations greatly improved the per-

$$
\begin{array}{lll}
M & ::= & \text{abst module } m \; \{U \; I^* \; P^*\} \\
D & ::= & \text{id}{=}D_r; \\
D_r & ::= & D_r \cup D_r \mid D_r \cap D_r \mid \text{id} \mid \{x : T \mid x.f{=}c\} \\
A & ::= & \neg A \mid A \wedge A \mid A \vee A \mid B \\
P & ::= & \text{predvar } p;
\end{array}
$$

Figure 8: Syntax of Flag Abstraction Language

formance of our analysis and allowed our analysis to verify larger programs.

5.1 Operation of the Analysis Algorithm

The flag analysis verifies a module M by sequentially checking each procedure of module M. To verify a procedure, the analysis performs abstract interpretation [10] with analysis domain elements represented by formulas. Our analysis associates quantified boolean formulas B to each program point. A formula F has two collections of set variables: unprimed set variables S denoting initial values of sets at the entry point of the procedure, and primed set variables S' denoting the values of these sets at the current program point; F also contains unprimed and primed boolean variables b and b' representing the pre- and post-values of local and global boolean variables. The interpretations of these variables are given by the definitions in the abstraction section of the module. The use of primed and unprimed variables allows our analysis to represent, for each program point p, a binary relation on states that overapproximates the reachability relation between procedure entry and program point p [11, 17, 32].

In addition to the abstract sets from the specification, the analysis also generates a set for each (object-typed) local variable. This set contains the object to which the local variable refers and has a cardinality constraint that restricts the set to have cardinality at most one (the empty set represents a null reference). The formulas that the analysis manipulates therefore support the disambiguation of local variable and object field accesses at the granularity of the sets in the analysis; other analyses often rely on a separate pointer analysis to provide this information.

The initial dataflow fact at the start of a procedure is the precondition for that procedure, transformed into a relation by conjoining $S' = S$ for all relevant sets. At merge points, the analysis uses disjunction to combine boolean formulas. Our current analysis iterates `while` loops at most some constant number of times, then coarsens the formula to `true` to ensure termination, thus applying a simple form of widening [10]. The analysis also allows the developer to provide loop in-

variants directly. [6] After running the dataflow analysis, our analysis checks that the procedure conforms to its specification by checking that the derived postcondition (which includes the `ensures` clause and any required representation or global invariants) holds at all exit points of the procedure. In particular, the flag analysis checks that for each exit point e, the computed formula B_e implies the procedure's postcondition.

Incorporation. The transfer functions in the dataflow analysis update boolean formulas to reflect the effect of each statement. Recall that the dataflow facts for the flag analysis are boolean formulas B denoting a relation between the state at procedure entry and the state at the current program point. Let B_s be the boolean formula describing the effect of statement s. The incorporation operation $B \circ B_s$ is the result of symbolically computing the relation composition of relations given by formulas B and B_s. Conceptually, incorporation updates B with the effect of B_s. We compute $B \circ B_s$ by applying equivalence-preserving simplifications to the formula

$$\exists \hat{S}_1, \ldots, \hat{S}_n.\ B[S'_i \mapsto \hat{S}_i] \wedge B_s[S_i \mapsto \hat{S}_i]$$

5.2 Transfer Functions

Our flag analysis handles each statement in the implementation language by providing appropriate transfer functions for these statements. The generic transfer function is a relation of the following form:

$$[\![\mathrm{st}]\!](B) = B \circ \mathcal{F}(\mathrm{st})$$

where $\mathcal{F}(\mathrm{st})$ is the formula symbolically representing the transition relation for statement st expressed in terms of abstract sets. The transition relations for the statements in our implementation language are as follows.

Assignment statements. We first define a generic frame condition generator, used in our transfer functions,

$$\mathsf{fram}_x = \bigwedge_{S \neq x,\ S \text{ not derived}} S' = S \wedge \bigwedge_{p \neq x} (p' \Leftrightarrow p),$$

where S ranges over sets and p over boolean predicates. Note that derived sets are not preserved by frame conditions; instead, the analysis preserves the anonymous sets contained in the derived set definitions and conjoins these definitions to formulas before applying the decision procedure.

Our flag analysis also tracks values of boolean variables:

$$\begin{aligned}
\mathcal{F}(\mathsf{b} = \mathsf{true}) &= \mathsf{b}' \wedge \mathsf{fram}_b \\
\mathcal{F}(\mathsf{b} = \mathsf{false}) &= (\neg \mathsf{b}') \wedge \mathsf{fram}_b \\
\mathcal{F}(\mathsf{b} = \mathsf{y}) &= (\mathsf{b}' \Leftrightarrow \mathsf{y}) \wedge \mathsf{fram}_b \\
\mathcal{F}(\mathsf{b} = \langle\mathrm{if\ cond}\rangle) &= (\mathsf{b}' \Leftrightarrow f^+(\langle\mathrm{if\ cond}\rangle)) \wedge \mathsf{fram}_b \\
\mathcal{F}(\mathsf{b} = !e) &= \mathcal{F}(\mathsf{b} = e) \circ ((\mathsf{b}' \Leftrightarrow \neg \mathsf{b}) \wedge \mathsf{fram}_b)
\end{aligned}$$

where $f^+(e)$ is the result of evaluating the condition e, as defined below in our analysis of `if` statements. We also track the local variable object references:

$$\begin{aligned}
\mathcal{F}(\mathsf{x} = \mathsf{y}) &= (\mathsf{x}' = \mathsf{y}) \wedge \mathsf{fram}_x \\
\mathcal{F}(\mathsf{x} = \mathsf{null}) &= (\mathsf{x}' = \emptyset) \wedge \mathsf{fram}_x \\
\mathcal{F}(\mathsf{x} = \mathsf{new}\ t) &= \neg(\mathsf{x}' = \emptyset) \wedge \bigwedge_S (\mathsf{x}' \cap S = \emptyset) \wedge \mathsf{fram}_x
\end{aligned}$$

We next present the transfer function for changing set membership. If $R = \{x : T \mid x.\mathsf{f} = c\}$ is a set definition in the abstraction section, we have:

$$\begin{aligned}
\mathcal{F}(\mathsf{x.f} = \mathsf{c}) \ := \ & R' = R \cup \mathsf{x} \wedge \bigwedge_{S \in \mathsf{alts}(R)} S' = S \setminus \mathsf{x} \\
& \wedge\ \mathsf{fram}_{\{R\} \cup\ \mathsf{alts}(R)}
\end{aligned}$$

where $\mathsf{alts}(R) := R'$ such that the abstraction module contains $R' = \{x : T \mid x.\mathsf{f} = c_1\}, c_1 \neq c$.

We also have a rule handling field reads and writes of boolean values b; it is similar to the rule above for reads and writes of integers. However, since our analysis tracks the flow of boolean values, the rules are more detailed. When $B^+ = \{x : T \mid x.\mathsf{f} = \mathsf{true}\}$ and $B^- = \{x : T \mid x.\mathsf{f} = \mathsf{false}\}$, the rule is:

$$\begin{aligned}
\mathcal{F}(\mathsf{x.f} = \mathsf{b}) &= \left(\begin{array}{c} b \wedge B^{+\prime} = B^+ \cup \mathsf{x} \\ \wedge \bigwedge_{S \in \mathsf{alts}(B^+)} S' = S \setminus \mathsf{x} \end{array} \right) \\
& \quad \wedge \left(\begin{array}{c} \neg b \wedge B^{-\prime} = B^- \cup \mathsf{x} \\ \wedge \bigwedge_{S \in \mathsf{alts}(B^-)} S' = S \setminus \mathsf{x} \end{array} \right) \\
& \quad \wedge \mathsf{fram}_{\{B\} \cup \mathsf{alts}(B)} \\
\mathcal{F}(\mathsf{b} = \mathsf{y.f}) &= (\mathsf{b}' \Leftrightarrow \mathsf{y} \in B^+) \wedge \mathsf{fram}_b.
\end{aligned}$$

Finally, we have some default rules to conservatively account for expressions not otherwise handled,

$$\mathcal{F}(x.f = *) = \mathsf{fram}_x \qquad \mathcal{F}(x = *) = \mathsf{fram}_x.$$

Procedure calls. For a procedure call `x=proc(y)`, our transfer function checks that the callee's requires condition holds, and incorporates `proc`'s ensures condition as follows:

$$\mathcal{F}(\mathsf{x} = \mathsf{proc(y)}) = \mathbf{ensures}_1(\mathsf{proc}) \wedge \bigwedge_S S' = S$$

where both $\mathbf{ensures}_1$ and $\mathbf{requires}_1$ substitute caller actuals for formals of `proc` (including the return value), and where S ranges over all local variables.

Conditionals. The analysis produces a different formula for each branch of an `if` statement `if (e)`. We define functions $f^+(e), f^-(e)$ to summarize the additional information available on each branch of the conditional; the transfer functions for the true and false branches of the conditional are thus, respectively,

$$\begin{aligned}
[\![\mathtt{if\ (e)}]\!]^+(B) &= f^+(e) \wedge B \\
[\![\mathtt{if\ (e)}]\!]^-(B) &= f^-(e) \wedge B.
\end{aligned}$$

For constants and logical operations, we define the obvious f^+, f^-:

$$
\begin{array}{llll}
f^+(\text{true}) & = & \text{true} \\
f^+(\text{false}) & = & \text{false} \\
f^+(!e) & = & f^-(e) \\
f^+(x\texttt{!=}e) & = & f^-(x\texttt{==}e)
\end{array}
\qquad
\begin{array}{llll}
f^-(\text{true}) & = & \text{false} \\
f^-(\text{false}) & = & \text{true} \\
f^-(!e) & = & f^+(e) \\
f^-(x\texttt{!=}e) & = & f^+(x\texttt{==}e)
\end{array}
$$

$$
\begin{array}{rcl}
f^+(e_1 \texttt{ \&\& } e_2) & = & f^+(e_1) \wedge f^+(e_2) \\
f^-(e_1 \texttt{ \&\& } e_2) & = & f^-(e_1) \vee f^-(e_2)
\end{array}
$$

We define f^+, f^- for boolean fields as follows:

$$
\begin{array}{rclrcl}
f^+(x.\texttt{f}) & = & x \subseteq B & f^-(x.\texttt{f}) & = & x \nsubseteq B \\
f^+(x.\texttt{f==false}) & = & x \nsubseteq B & f^-(x.\texttt{f==false}) & = & x \subseteq B
\end{array}
$$

where $B = \{x : T \mid x.\texttt{f} = \text{true}\}$; analogously, let $R = \{x : T \mid x.\texttt{f} = \texttt{c}\}$. Then,

$$
f^+(x.\texttt{f==c}) \;=\; x \subseteq R \qquad f^-(x.\texttt{f==c}) \;=\; x \nsubseteq R.
$$

We also predicate the analysis on whether a reference is `null` or not:

$$
f^+(x\texttt{==null}) \;=\; x = \emptyset \qquad f^-(x\texttt{==null}) \;=\; x \neq \emptyset.
$$

Finally, we have a catch-all condition,

$$
f^+(*) \;=\; \text{true} \qquad f^-(*) \;=\; \text{true}
$$

which conservatively captures the effect of unknown conditions.

Loops. Our analysis analyzes `while` statements by synthesizing loop invariants or by verifying developer-provided loop invariants. To synthesize a loop invariant, it iterates the analysis of the loop body until it reaches a fixed point, or until N iterations have occurred (in which case it synthesizes `true`). The conditional at the top of the loop is analyzed the same way `if` statements are analyzed. We can also verify explicit loop invariants; these simplify the analysis of `while` loops and allow the analysis to avoid the fixed point computation involved in deriving a loop invariant. Developer-supplied explicit loop invariants are automatically conjoined with the frame conditions generated by the containing procedure's modifies clause to ease the burden on the developer.

Assertions and Assume Statements. We analyze statement s of the form `assert A` by showing that the formula for the program point s implies `A`. Assertions allow developers to check that a given set-based property holds at an intermediate point of a procedure. Using `assume` statements, we allow the developer to specify properties that are known to be true, but which have not been shown to hold by this analysis. Our analysis prints out a warning message when it processes `assume` statements, and conjoins the assumption to the current dataflow fact. Assume statements have proven to be

valuable in understanding the analysis outcomes during the debugging of procedure specifications and implementations. Assume statements may also be used to communicate properties of the implementation that go beyond the abstract representation used by the analysis.

Return Statements. Our analysis processes the statement `return x` as an assignment `rv = x`, where `rv` is the name given to the return value in the procedure declaration. For all return statements (whether or not a value is returned), our analysis checks that the current formula implies the procedure's postcondition and stops propagating that formula through the procedure.

5.3 Verifying Implication of Dataflow Facts

A compositional program analysis needs to verify implication of constraints as part of its operation. Our flag analysis verifies implication when it encounters an assertion, procedure call, or procedure postcondition. In these situations, the analysis generates a formula of the form $B \Rightarrow A$ where B is the current dataflow fact and A is the claim to be verified[7]. The implication to be verified, $B \Rightarrow A$, is a formula in the boolean algebra of sets, and we check its validity using the MONA decision procedure for the monadic second-order logic of strings, which subsumes boolean algebras [18].

5.4 Boolean Algebra Formula Transformations

In our experience, applying several formula transformations drastically reduced the size of the formulas emitted by the flag analysis, as well as the time needed to determine their validity using an external decision procedure; in fact, some benchmarks could only be verified with the formula transformations enabled. This subsection describes the transformations we found to be useful and includes a performance evaluation of these transformations, comparing formula sizes and analysis running times.

Smart Constructors. The constructors for creating boolean algebra formulas apply peephole transformations as formulas are being created. The simplest peephole transformation is constant folding: for instance, attempting to create $B \wedge \text{true}$ gives the formula B. Our constructors fold constants in implications, conjunctions, disjunctions, and negations. Similarly, attempting to quantify over unused variables causes the quantifier to be dropped: $\exists x.F$ is created as just F for x not free in F. Most interest-

---

[7]Note that B may be unsatisfiable; this often indicates a problem with the program's specification. The flag analysis can, optionally, check whether B is unsatisfiable and emit a warning if it is. This check enabled us to improve the quality of our specifications by identifying specifications that were simply incorrect.

ingly, we factor common conjuncts out of disjunctions: $(A \wedge B) \vee (A \wedge C)$ is represented as $A \wedge (B \vee C)$. Conjunct factoring greatly reduces the size of formulas tracked after control-flow merges, since most conjuncts are shared on both control-flow branches. The effects of this transformations appear similar to the effects of the SSA form conversion in weakest precondition computation [14,27].

Basic Quantifier Elimination. We symbolically compute the composition of statement relations during the incorporation step by existentially quantifying over all state variables. However, most relations corresponding to statements modify only a small part of the state and contain the frame condition that indicates that the rest of the state is preserved. The result of incorporation can therefore often be written in the form $\exists x_1 . x = x_1 \wedge F(x)$, which is equivalent to $F(x_1)$. In this way we reduce both the number of conjuncts and the number of quantifiers. Moreover, this transformation can reduce some conjuncts to the form $t = t$ for some Boolean algebra term t, which is a true conjunct that is eliminated by further simplifications.

It is instructive to compare our technique to weakest precondition computation [14] and forward symbolic execution [9]. These techniques are optimized for the common case of assignment statements and perform relation composition and quantifier elimination in one step. Our technique achieves the same result, but is methodologically simpler and applies more generally. In particular, our technique can take advantage of equalities in transfer functions that are not a result of analyzing assignment statements, but are given by explicit formulas in `ensures` clauses of procedure specifications. Such transfer functions may specify more general equalities such as $A = A' \cup x \wedge B' = B \cup x$ which do not reduce to simple backward or forward substitution.

Quantifier Nesting. We have experimentally observed that the MONA decision procedure works substantially faster when each quantifier is applied to the smallest scope possible. We have therefore implemented a quantifier nesting step that reduces the scope of each quantifier to the smallest possible subformula that contains all free variables in the scope of the quantifier. For example, our transformation replaces the formula $\forall x. \forall y. (f(x) \Rightarrow g(y))$ with $(\exists x. f(x)) \Rightarrow (\forall y. g(y))$.

To take maximal advantage of our transformations, we simplify the formula after applying incorporation and before invoking the decision procedure. Our global simplification step rebuilds the formula bottom-up and applies the simplifications to each subformula.

5.5 Evaluating the Impact of Formula Transformation

Table 1 shows the result of our formula transformations. The `compiler` benchmark models a constant-folding compiler pass. The `scheduler` benchmark models an operating system scheduler. The `ctas` benchmark is the core of an air-traffic control system. The `board`, `controller` and `view` modules are the core modules of the `minesweeper` example.

We ran our benchmarks on a Pentium 4 at 2.80GHz, running Linux, with 2 gigabytes of RAM. We have reported the sizes (in terms of AST node counts) of the boolean algebra formulas created with all transformations enabled; with all transformations except for smart constructors; and with no transformations enabled. (The results with smart constructors and no other transformations were usually identical to the results with no transformations.) For each run, we have also presented the time spent in the decision procedure (under 4 seconds, optimized) and in the analysis, excluding the decision procedure (under 25 seconds, optimized). Our formula transformations reduce formula size by 2 to 70 times (with greater reductions for larger formulas); indeed, without transformation, the formulas generated by `compiler`, `board` and `view` could not successfully be checked by MONA because of an out of memory error.

	Opt	Smart Constrs	# nodes	MONA time	Flag time	Opt. ratio
compiler	✓	✓	15860	0.45	7.84	38.36
	✓	×	28009	0.60	9.68	21.72
	×	✓,×	608375	N/A	82.04	1.00
scheduler	✓	✓,×	468	0.05	0.04	2.32
	×	✓,×	1086	0.08	0.04	1.00
ctas	✓	✓,×	3410	0.23	0.11	2.85
	×	✓,×	9726	13.33	0.29	1.00
board	✓	✓	15261	1.39	9.29	39.77
	✓	×	68177	29.89	16.19	8.90
	×	✓	375919	N/A	91.46	1.61
	×	×	606967	N/A	111.04	1.00
controller	✓	✓	6840	0.47	0.28	3.24
	✓	×	7206	0.52	0.32	3.07
	×	✓,×	22145	2.93	0.74	1.00
view	✓	✓	25646	3.06	24.35	69.92
	✓	×	101872	4.45	44.56	17.60
	×	✓,×	1793295	N/A	369.90	1.00

Table 1: Formula sizes before and after transformation

6 The PALE Analysis Plugin

Unlike the flag analysis, which we designed to operate within our analysis framework, the PALE analysis is a previously implemented analysis package that we integrated into our framework. During the course of this adaptation, we did not modify the PALE analysis pack-

age itself — we instead implemented translators that enabled it to work within our analysis framework.

6.1 The PALE Analysis System

The PALE analysis system takes as input a program written in its own imperative language [30]. This program includes preconditions, postconditions, loop invariants, and graph type declarations [20]. 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 9),[8] whose values form the spanning tree *backbone* of the data structure. In addition to data fields, a graph type may contain *routing fields* [20] (such as the `prev` field in Figure 9). These routing fields are functionally determined by the backbone; the `prev` field in Figure 9, 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.

```
abst module DLLSet {
  use plugin "PALE";
  Content = {x : Entry | "root<next*>x"};

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

Figure 9: Doubly-Linked List Abstraction Section

The precondition, postcondition, and loop invariants are arbitrary formulas in monadic second-order logic. Such formulas enable the use of transitive closure over object reference fields to identify the set of all objects that participate in that data structure. Building on this base, it is also possible to specify arbitrary boolean formulas containing set inclusion and equality constraints involving these sets. For example, it is possible to specify that the sets of objects in two lists (identified using transitive closure over the `next` field) are disjoint, equal, or that one is a subset of the other. It is also possible to state set membership constraints involving the objects that variables point to.

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 monadic second-order logic. The PALE system

---

[8]Note that, in the PALE system terminology, "data" fields hold reference values.

$$
\begin{array}{rcl}
M & ::= & \text{abst module } m \ \{U \ D^* \ I^* \ \} \\
U & ::= & \text{use plugin "PALE"}; \\
D & ::= & S=\{x : T \mid F(x)\}; \\
I & ::= & \text{invariant } F; \\
F & ::= & \text{PALE specification}
\end{array}
$$

Figure 10: Syntax of PALE Plugin Abstraction Sections

uses the MONA decision procedure [18, 19] to determine the validity of these verification conditions. If all of these conditions are valid, the program satisfies its PALE specification.

6.2 Using the PALE Plugin

We next describe the information that the developer provides to enable the PALE plugin to verify that a module implementation conforms to its specification. Most of the information specific to the PALE plugin is contained in the abstraction section, whose syntax is in Figure 10.

6.2.1 Specifying Set Definitions

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. Figure 9 shows a definition of the set `Content` as the set of all `Entry` objects reachable from the `root` along the `next` field. A binary relation given by a regular expression such as `<next*>` is a shorthand for the corresponding formula with two variables definable in monadic second-order logic.

6.2.2 Specifying Representation Invariants

The developer specifies the representation invariants for the PALE plugin using `invariant` declarations in the abstraction section, as illustrated in Figure 9. The syntax of these invariants is specific to the PALE plugin. An invariant for the PALE plugin is either a graph type definition, such as the definition of the `Entry` graph type in Figure 9, or a declaration of a data structure root, such as the `data root:Entry` declaration in Figure 9.

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 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 enforces the 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.

6.3 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.

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 that takes a set as an argument:

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

A statement $B(S_1, \ldots, S_n)$ in 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 uses $\mathtt{isS}_i$ predicates to translate the specification of a procedure p as follows. Consider a specification of the form

> **requires** $B_0(S_1, \ldots, S_n)$
> **modifes** $S_{j_1}, \ldots, S_{j_m}$
> **ensures** $B_1(S_1, \ldots, S_n, S'_1, \ldots, S'_1)$

The first translation step eliminates the modifies clause, yielding

> **requires** $B_0(S_1, \ldots, S_n)$
> **ensures** $B_2(S_1, \ldots, S_n, S'_1, \ldots, S'_1)$

where

$$B_2 = B_1 \wedge \bigwedge_{i \notin \{j_1, \ldots, j_m\}} S'_i = S_i$$

The next translation step introduces logical variables $S_1, \ldots, S_n$ that correlate preconditions and postconditions. The resulting precondition/postcondition pair is:

```
set  S₁ : T₁;
...
set  Sₙ : Tₙ;
/* precondition */
[⋀ⁿᵢ₌₁ isSᵢ(Sᵢ) ∧  B₀(S₁, ..., Sₙ)]

{stmts}

/* postcondition */
[ existset  S'₁ of  T₁ : isS₁(S'₁) ∧
...
  existset  S'ₙ of  Tₙ : isSₙ(S'ₙ) ∧
  B₂(S₁, ..., Sₙ, S'₁, ..., S'₁)]
```

where {stmts} is the translation of the statements that implement the body of the procedure p. At this point we have a translated procedure that we can pass to the PALE analysis system for verification. Given a module to verify, our analysis driver translates all of the procedures into this form and checks if the PALE system can verify them. If so, the implementation section correctly implements its specification.

6.4 Implications

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.

7 The Array Analysis Plugin

The array analysis plugin generates verification conditions using weakest preconditions and discharges them using the Isabelle theorem prover. We have chosen this technique as a last resort for verifying arbitrarily complicated data structure implementations. The logic for specifying abstraction functions is based on typed set

theory and proof obligations can be discharged using automated theorem proving or a proof checker for manually generated proofs, which means that there is no *a priori* bound on the complexity of the data structures (and data structure consistency properties) that can be verified. In our current implementation we have explored this technique for data structures that implement sets by storing objects in global arrays. For example, we have verified the operations on abstract set Content given by an abstraction function

$$\text{Content} = \{x | \exists j.\ 0 \le j \land j < s \land x \in d[j]\}$$

where d is a global array of objects and s is an integer variable indicating the currently used part of the array.

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

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

$$\{x \mid \exists j.\ 0 \le j \land j < s' \land x \in d'[j]\} = \{x \mid \exists j.\ 0 \le j \land j < s \land x \in d[j]\} - e$$

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

3. Statement desugaring: translate statements into loop-free guarded command language (*e.g.* [14]).

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

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

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

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

8 Experience

We implemented our system and, to obtain experience using it, coded up several benchmark programs, using our system during the development of the programs. In addition to the minesweeper example presented in Section 2, we ran our analysis on programs with computational patterns from scientific computations, operating-system schedulers, air-traffic control, and program transformation passes. These benchmarks use a variety of data structures, and we have therefore implemented and verified sets, set iterators, queues, stacks, and priority queues. Table 2 illustrates the benchmarks we ran through our system. Our implementations range from singly-linked and doubly-linked lists and tree insertion (all verified using the PALE plugin) through array data structures (verified using the array membership plugin with the Isabelle theorem prover used to discharge verification conditions).

Implementation Structure. Our implementation provides an infrastructure with several general components that perform tasks required by all analyses. The implementation language component can parse and type-check implementation sections. It produces an abstract syntax tree and methods that allow analyses to conveniently access this representation. Similarly, the specification component can parse and type check specification sections and provides access to the resulting abstract syntax tree. Large parts of abstraction sections are expressed in a language that is specific to each analysis. The abstraction section component parses those parts of the abstraction section syntax that are common to all analyses and uses uninterpreted strings to pass along the analysis-specific parts. Finally, the implementation provides a driver that processes the program and invokes the appropriate analysis for each module that it encounters. Our implementation consists of approximately 10,000 lines of O'Caml code, to which the flag plugin contributes 2000 lines, the PALE plugin another 700 lines, and the array analysis plugin 1000 lines.[9]

---

[9]Full source code for our infrastructure is available at `http://cag.csail.mit.edu/∼plam/mpa`. Our Subversion source code repository is also publicly accessible at `http://plam.csail.mit.edu/svn/repos/trunk/module-language`.

	plugin	# lines spec	# lines impl
dll-stack	flag	22	15
scheduler	flag	34	22
prodcons	flag	41	50
ctas	flag	49	53
compiler	flag	75	143
atom	flag	64	29
ensemble	flag	888	152
h2o	flag	420	159
board	flag	78	168
controller	flag	43	133
view	flag	43	372
Set (SLL)	PALE	25	77
Queue (SLL)	PALE	22	34
PQueue (SLL)	PALE	22	38
Stack (SLL)	PALE	25	28
Iterator (SLL)	PALE	38	81
Set (DLL)	PALE	30	60
Queue (DLL)	PALE	26	49
Iterator (DLL)	PALE	39	68
Set Insertion (Tree)	PALE	22	71
Set (Array)	array	26	65

System totals	# modules	# lines spec	# lines impl
compiler	3	113	211
ctas	6	134	102
water	10	1921	542
prodcons	3	54	78
scheduler	3	77	128
minesweeper	7	236	750

Table 2: Benchmark characteristics

Because we implemented the flag analysis specifically for this project, it is fairly closely integrated with the rest of our infrastructure. It processes implementation and specification section directly in the abstract syntax tree representation that our infrastructure provides. The PALE analysis plugin, on the other hand, uses an off-the-shelf analysis package that was developed before the start of this project. We therefore wrote an adapter that integrates this analysis into our system, as described in Section 6.3. The array analysis plugin reads the specification, abstraction and implementation sections and produces proof obligations, using weakest preconditions, and discharges them using the Isabelle theorem prover. The developer may specify proof hints, on a per-procedure basis, that invoke arbitrarily complicated previously-proved lemmas.

8.1 Minesweeper

We earlier described some of the invariants that we successfully verify for the minesweeper example. While we were trying to verify our invariants about the minesweeper implementation, we found a number of bugs in that implementation. We now present one of the bugs that we found. The situation is that at the end of the game, minesweeper exposes the entire game board; we use `removeFirst` to remove all elements from the unexposed list, one at a time. After we have exposed the entire board, we can guarantee that the list of unexposed cells is empty:

```
proc drawFieldEnd()
    requires ExposedList.setInit & Board.gameOver &
             (UnexposedList.Content <= Board.U)
    modifies UnexposedList.Content, Board.ExposedCells,
             Board.UnexposedCells, ExposedList.Content,
             UnexposedList.Content
    ensures card(UnexposedList.Content') = 0;
```

because the implementation of the `drawFieldEnd` procedure loops until `isEmpty` returns `true`, which also guarantees that the `UnexposedList.Content` set is empty.

The natural way to write the iteration in this procedure would be:

```
while (UnexposedList.isEmpty()) {
    Cell c = UnexposedList.removeFirst();
    drawCellEnd(c);
}
```

and indeed, this was the initial implementation of that code. However, when we attempted to analyze this code, we got the following error message:

```
Analyzing proc drawFieldEnd...
Error found analyzing procedure drawFieldEnd:
    requires clause in a call to procedure View.drawCellEnd.
```

Upon further examination, we found that we were breaking the invariant ensuring that `Board.ExposedCells` equals `UnexposedList.Content`. The correct way to preserve the invariant is by calling `Board.setExposed`, which simultaneously sets the `isExposed` flag and removes the cell from the `UnexposedList`:

```
Cell c = UnexposedList.getFirst();
Board.setExposed(c, true);
drawCellEnd(c);
```

which successfully analyzes:

```
Analyzing proc drawFieldEnd... Procedure drawFieldEnd passes.
```

8.2 Stack Data Structure

Using our system, we have implemented stacks, queues, priority queues, sets, and iterators using singly-linked lists, doubly-linked lists and trees. We checked these implementations with the PALE plugin. It turns out that our initial implementations were not completely correct; our analysis pinpointed (and helped us correct) some errors in the implementations. We report, below, two bugs that were found by our PALE plugin.

For the stack, we maintain an abstract set `S` representing the content of the stack, and verify that

stack insertions actually insert the given object into the stack (S' = S + e), and that removal actually removes an object from the stack, if possible: `card(S) = 0 | (exists e:Entry. (S' = S - e) & card(e) =1))`.

Our PALE plugin checks that objects that belong to a set have consistent values for navigational fields (e.g. `next`, `prev`), and that objects that do not belong to the set have those fields set to `null`. Initially, our implementation for `removeFirst` was:

```
proc removeFirst() returns e:Entry {
    Entry res = root;
    if (root != null) root = root.next;
    pragma "removed res";
    return res;
}
```

where the `pragma` statement indicates to the analysis that it is verifying a set removal. However, the analysis reports an error while verifying this implementation. Careful inspection of this code, however, reveals that the removed object, `res`, will still have a reference to an object in the stack after removal; this is potentially problematic, as it may lead to non-list structures being present in the heap. Our plugin therefore requires us to add `res.next = null` to this procedure, so that all objects subsequently passed to this module will have `next` set to `null`.

8.3 List Iterators

We have implemented (using a singly-linked list) a set which supports iterators; it has a procedure which returns the next element in the set, until there are no more elements. We have modelled this set using a module which declares two sets, `Content` and `Iter`; the `Iter` set contains all elements which have not yet been returned by the iterator. Note that we can guarantee, by reasoning solely at the abstract set level, that the `nextIter` procedure returns every member of the `Content` set.

Our iterable set implementation, however, also supports removal. In the presence of iteration, removal has the following semantics: if we remove an object in `Iter`, then it will not be returned by subsequent calls to the iterator; if the object is not in `Iter`, then future iterations are unchanged. Our analysis caught the corner case where we remove the element which would next be returned by the iterator. Adding the following line allowed the analysis of this module to succeed:

```
if (current == e) current = current.next;
```

8.4 Program Transformations

This benchmark implements a constant-folding operation on the intermediate representation of a simple programming language. The input to this operation is an abstract syntax tree along with a list of all nodes in the tree. The output is a new tree after constant folding (this transformation replaces arithmetic expressions on constant values with the computed value of the arithmetic expression). To facilitate memory management, each tree comes with a list of nodes in the tree.

For efficiency, the transformation should, whenever possible, reuse (instead of copy) nodes from the input tree when it constructs the output tree. An implementation may therefore fail to remove reused nodes from the input tree's list, leading to premature deallocation and data structure corruption. In our system, the program eliminates this possibility by using global invariants to require the sets of nodes in the input tree list and output tree list to be disjoint. An additional global invariant requires the output tree list to contain all of the nodes in the transformed output tree.

8.5 Process Scheduler

Our process scheduler benchmark maintains a list of running processes and a priority queue of suspended processes. There are three modules in our process scheduler implementation: the `RunningList` module (which maintains the list of running processes), the `SuspendedQueue` module (which maintains the queue of suspended processes), and the `Scheduler` module (which implements the specification for the scheduler). The running list and suspended queue are verified using the PALE plugin, whereas the scheduler itself is verified with the flag plugin. Both the data structures and the core scheduler know whether a process is running or suspended: the core scheduler uses flags to indicate set membership, whereas the data structures use heap reachability to track membership. One of the global invariants ensures that the sets in the scheduler and in the data structures coincide. Our analysis also verifies that the set of running processes is always disjoint from the set of suspended processes:

```
invariant (Running = RunningList.InList) &
          (Suspended = SuspendedQueue.InQueue) &
          disjoint (Running, Suspended);
```

8.6 CTAS

The Center-TRACON Automation System (CTAS) is a set of air-traffic control tools developed at the NASA Ames research center [1]. The system is designed to help air traffic controllers visualize and manage the complex air traffic flows at centers surrounding large metropolitan airports. CTAS is structured with a central communications manager process that maintains socket connections to a graphics process, a weather process (to acquire information about the weather), a track acquisition process (to acquire radar data), and a trajectory

synthesizer (to compute predicted trajectories for the controlled aircraft).

The weather and track acquisition sockets are read-only: the communications manager simply acquires the data that they send. The graphics socket, on the other hand, is write-only: the job of the graphics process is to display the information to the control. The communications manager both reads and writes the trajectory socket: it writes the socket to send requests to project the trajectory for the controlled aircraft and reads the socket to obtain the synthesized trajectories.

We implemented a program with this communication pattern and used our system to check these access constraints. The final system ensures that all sockets are correctly initialized (and have not been closed) when the program attempts to read or write to or from the socket. Our sets include a set of writable sockets, a set of readable sockets, and a set of closed sockets. The weather and track acquisition sockets are elements of the set of readable sockets only, the graphics socket is an element of the set of writable sockets only, while the trajectory socket is an element of both sets. Note that enabling a socket to participate in multiple sets at the same time (in effect, composing the typestate out of multiple orthogonal sets) substantially simplifies the resulting typestate system. A typical example of an requires clause is that for the `readTrack` procedure:

```
proc readTrack() requires card(Track.Data)=1 &
                          (Track.Data in Sockets.Open) &
                          (Track.Data in Sockets.Readable)
                  ensures true;
```

The preconditions of the socket read and write procedures require the socket to be in the read or write set, respectively, ensuring that the program does not attempt to perform an inappropriate socket operation.

8.7 Water

Our Water benchmark is a port of the Perfect Club benchmark MDG [5] to our implementation language. It uses a predictor/corrector method to evaluate forces and potentials in a system of water molecules in the liquid state. The central loop of the computation performs a time step simulation. Each step predicts the state of the simulation, uses the predicted state to compute the forces acting on each molecule, uses the computed forces to correct the prediction and obtain a new simulation state, and uses the new simulation state to compute the potential and kinetic energy of the system.

The Water benchmark consists of several modules, including the `simparm`, `atom`, `H2O`, `ensemble`, and `main` modules. These modules contain 2000 lines of implementation and 500 lines of specification. Each module defines sets and boolean variables; we use these sets and variables to express safety properties about the computation.

The `simparm` module, for instance, is responsible for recording the simulation parameters, which are stored in a text file and loaded upon demand. This module therefore defines two boolean variables, `Init` and `ParmsLoaded`; `Init` implies that the module has been initialized, *i.e.* the appropriate arrays have been allocated on the heap, while the `ParmsLoaded` variable implies that the simulation parameters have been loaded from disk. Our analysis verifies that no simulation parameter may be requested until the parameters have been loaded.

The fundamental unit of the simulation is the atom, which is encapsulated by the `atom` module; atoms may be *predicted* or *corrected*, and the `predic` and `correc` procedures change atoms into predicted or corrected atoms if the appropriate preconditions are met. In particular, the simulation may only correct a predicted atom; to enforce this property in the specification, we define sets `Predic` and `Correc` and populate them with the set of predicted and corrected atoms, respectively. The precondition for `correc` requires that an atom is already in the `Predic` set, and ensures that, after successful completion, the atom is no longer in the `Predic` set, but is instead in the `Correc` set.

Atoms belong to molecules, which are handled by the `H2O` module. A molecule tracks the position and velocity of the three atoms belonging to that molecule; they can be in a variety of conceptual states, indicating not only whether their position has been predicted and corrected but also whether the intra-molecule force corrections have been applied, whether the molecule's forces have been scaled, etc. We verify the invariant that when the molecule has been corrected, the atoms in the molecule are also corrected. The interface of the `H2O` ensures that the operations on the molecule may only be invoked in a certain order; for instance, only molecules in the `Kineti` set (which have had their kinetic energy calculated) may be passed to the `bndry` procedure.

The `ensemble` module manages the collection of molecule objects. This module stages the entire simulation by iterating over all molecules and computing their positions and velocity over time. The ensemble module uses boolean predicates to track the state of the computation; when boolean predicate `INTERF` is true, for example, then the inter-molecule force computation has been carried out on all molecules in the simulation. Our analysis verifies that the boolean predicates, representing program state, satisfy the following ordering relationship:

$$\text{Init} \rightsquigarrow \text{INITIA} \rightsquigarrow \text{PREDIC} \rightsquigarrow \text{INTRAF} \rightsquigarrow \text{VIR} \rightsquigarrow \cdots$$

Our specification relies on an implication from boolean

predicates to properties ranging over the collection of molecule objects, which can be ensured by the array analysis plugin.

Finally, the `main` module is responsible for initializing the state of the `ensemble` module and printing out the final state of the system. In the water benchmark, the main loop can only be executed after an initial iteration of the computation has proceeded; our analysis verifies that the appropriate boolean predicate always holds before the main loop is initiated.

The `water` properties verified by our analysis help ensure that the computation's phases execute in the correct order; they are especially valuable in the maintenance phase of a program's life, when the original designer, if available, has long since forgotten the program's phase ordering constraints. Our analysis' set cardinality constraints also prevent empty sets (and null pointers) from being passed to procedures that expect non-empty sets or non-null pointers.

8.8 Discussion

Most analyses that check safety properties are perceived to be valuable because of the potential they hold for finding (and enabling the elimination of) errors in programs. Our system did identify a number of errors in programs. Furthermore, our use of the system had a profound impact on the development of our benchmark programs. In particular, the need to develop the specifications forced us to think more deeply about the intended structure and behavior of the program. We believe that this process eliminated much ambiguity about the program's behavior before we started developing the implementation sections, reducing the number of coding errors that found their way into these modules.

In general, we found abstract sets to be an appropriate formalism for our specifications. They allowed us to effectively capture, in a natural and easy to use way, many relevant properties of the data structures in our example programs. Of course, this abstraction does not capture all potentially relevant aspects (for example, ordering or mapping relationships between objects inserted into and retrieved from an encapsulated data structure), but it is decidable, natural, and worked well for us in our examples.

One surprise was that our system found substantially more errors in specification sections than in implementation sections. In particular, we sometimes found that our initial specification was not strong enough and we had to add more clauses before the properties were actually true. This process substantially improved our understanding of what the program was actually doing.

In some cases the system surprised us with the sophistication of the properties that it was able to check.

Because the modularity of our analysis approach eliminates any need for the analyses to scale to sizable programs, we were able to deploy very powerful analyses that could check quite strong program properties. Very few program analyses, for example, are able to verify that an implementation correctly processes every object in a given data structure. Nevertheless, the extreme precision of our analyses enabled them to check this kind of property in some of our test programs.

9 Related Work

We are aware of no previous research that allows multiple different analyses to analyze different parts of the program and share their results to detect or verify important properties that span parts of the program analyzed by different analyses. We survey related work in shape analysis, typestate systems, boolean algebra decision procedures, and program checking tools in general.

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 [23, 30, 31]. 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 [12, 34]. 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 [26]. 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.

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 [28, 33] and is presented in [2, Chapter 4]. The complexity of this problem is alternating exponential time [22]. To our knowledge, the only tool that can decide the first-order theory of boolean algebras is the MONA [19]; 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 [24].

Program Checking Tools. ESC/Java [13] 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. Other tools focus on verifying properties of concurrent programs [7, 29] or device drivers [3,16]. 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 analysis that check arbitrarily complicated properties within a single program.

10 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] Center-tracon automation system. http://www.ctas.arc.nasa.gov/ .

[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] T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS'02*, volume 2280 of *LNCS*, page 158, 2002.

[5] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, Nov. 1992.

[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, 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.

[8] 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.

[9] L. Clarke and D. Richardson. Symbolic evaluation methods for program analysis. In *Program Flow Analysis: Theory and Applications*, chapter 9. Prentice-Hall, Inc., 1981.

[10] 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.

[11] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[12] 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.

[13] 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.

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

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

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

[17] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *11th SAS*, 2004.

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

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

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

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

[22] 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.

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

[24] V. Kuncak and M. Rinard. The first-order theory of sets with cardinality constraints is decidable. Submitted to POPL'05, July 2004.

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

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

[27] K. R. M. Leino. Efficient weakest preconditions. KRML114a, 2003.

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

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

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

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

[32] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis problems. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc., 1981.

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

[34] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, January 1986.