Generalized Typestate Checking for Data Structure Consistency

Patrick Lam, Viktor Kuncak, and Martin Rinard

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Abstract. We present an analysis to verify abstract set specifications for programs that use object field values to determine the membership of objects in abstract sets. In our approach, each module may encapsulate several data structures and use membership in abstract sets to characterize how objects participate in its data structures. Each module's specification uses set algebra formulas to characterize the effects of its operations on the abstract sets. The program may define abstract set membership in a variety of ways; arbitrary analyses (potentially with multiple analyses applied to different modules in the same program) may verify the corresponding set specifications. The analysis we present in this paper verifies set specifications by constructing and verifying set algebra formulas whose validity implies the validity of the set specifications.
We have implemented our analysis and annotated several programs (75-2500 lines of code) with set specifications. We found that our original analysis algorithm did not scale; this paper describes several optimizations that improve the scalability of our analysis. It also presents experimental data comparing the original and optimized versions of our analysis.

1 Introduction

Typestate systems [7, 10, 12, 13, 21, 30] allow the type of an object to change during its lifetime in the computation. Unlike standard type systems, typestate systems can enforce safety properties that depend on changing object states.

This paper develops a new, generalized formulation of typestate systems. Instead of associating a single typestate with each object, our system models each typestate as an abstract set of objects. If an object is in a given typestate, it is a member of the set that corresponds to that typestate. This formulation immediately leads to several generalizations of the standard typestate approach. In our formulation, an object can be a member of multiple sets simultaneously, which promotes modularity and typestate polymorphism. It is also possible to specify subset and disjointness properties over the typestate sets, which enables our approach to support hierarchical typestate classifications. Finally, a typestate in our formulation can be formally related to a potentially complex property of an object, with the relationship between the typestate and the property verified using powerful independently developed analyses such as shape analyses or theorem provers.

We have implemented the idea of generalized typestate in the Hob program specification and verification framework [23, 24]. This framework supports the division of the program into instantiable, separately analyzable modules. Modules encapsulate private state and export abstract sets of objects that support abstract reasoning about the encapsulated state. Abstraction functions (in the form of arbitrary unary predicates over the encapsulated state) define the objects that participate in each abstract set. Modules also export procedures that may access the encapsulated state (and therefore change the contents of the exported abstract sets). Each module uses set algebra expressions (involving operators such as set union or difference) to specify the preconditions and postconditions of exported procedures. As a result, the analysis of client modules that coordinate the actions of other modules can reason solely in terms of the exported abstract sets and avoid the complexity of reasoning about any encapsulated state.

When the encapsulated state implements a data structure (such as list, hash table, or tree), the resulting abstract sets characterize how objects participate in that data structure. The developer can then use the abstract sets to specify consistency properties that involve multiple data structures from different modules. Such a property might state, for example, that two data structures involve disjoint objects or that the objects in one data structure are a subset of the objects in another. In this way, our approach can capture global sharing patterns and characterize both local and global data structure consistency.

The verification of a program consists of the application of (potentially different) analysis plugins to verify 1) the set interfaces of all of the modules in the program and 2) the validity of the global data structure consistency properties. The set specifications separate the analysis of a complex program into independent verification tasks, with each task verified by an appropriate analysis plugin [23]. Our approach therefore makes it possible, for the first time, to apply multiple specialized, extremely precise, and unscalable analyses such as shape analysis [27,28] or even manually aided theorem proving [31] to effectively verify sophisticated typestate and data structure consistency properties in sizable programs [23,31].

Specification Language Our specification language is the full first-order theory of the boolean algebra of sets. In addition to basic typestate properties expressible using quantifier-free boolean algebra expressions, our language can state constant bounds on the cardinalities of sets of objects, such as "a local variable is not null" or "the content of the queue is nonempty", or even "the data structure contains at least one and at most ten objects". Because a cardinality constraint counts all objects that satisfy a given property, our specification language goes beyond standard typestate approaches that use per-object finite state machines. Our specification language also supports quantification over sets. Universal set quantifiers are useful for stating parametric properties; existential set quantifiers are useful for information hiding. Note that quantification over sets is not directly expressible even in such sophisticated languages as first-order logic with transitive closure. [1] Despite this expressive power, our set specification language is decidable and extends naturally to Boolean Algebra with Presburger Arithmetic [22].

The Flag Analysis Plugin The present paper describes the flag analysis plugin, which uses the values of integer and boolean object fields (flags) to define the meaning of abstract sets. It verifies set specifications by first constructing set algebra formulas whose

[1] The first-order logic with transitive closure is the basis of the analysis [28]; our modular pluggable analysis framework [23] can incorporate an analyzer like TVLA [28] as one of the analysis plugins.

validity implies the validity of the set specifications, then verifying these formulas using an off-the-shelf decision procedure. The flag analysis plugin is important for two reasons. First, flag field values often reflect the high-level conceptual state of the entity that an object represents, and flag changes correspond to changes in the conceptual state of the entity. By using flags in preconditions of object operations, the developer can specify key object state properties required for the correct processing of objects and the correct operation of the program. Unlike standard typestate approaches, our flag analysis plugin can enforce not only temporal operation sequencing constraints, but also the generalizations that our expressive set specification language enables.

Second, the flag analysis plugin can propagate constraints between abstract sets defined with arbitrarily sophisticated abstraction functions in external modules. The plugin can therefore analyze modules that, as they coordinate the operation of other modules, indirectly manipulate external data structures defined in those other modules. The flag analysis can therefore perform the intermodule reasoning required to verify global data structure invariants such as the inclusion of one data structure in another and data structure disjointness. Because the flag plugin uses the boolean algebra of sets to internally represent its dataflow facts, it can propagate and verify these constraints in a precise way.

To evaluate our flag analysis, we have annotated several benchmark programs with set specifications. We have verified our benchmarks (in part) using the flag analysis algorithm described in Section 3, with MONA [19] as the decision procedure for the boolean algebra of sets. We found that our original analysis algorithm did not scale. This paper describes several optimizations that our analysis uses to improve the running time of the algorithm and presents experimental data comparing the original and optimized versions of our analysis.

2 Specification Language

Our system analyzes programs in a type-safe imperative language similar to Java or ML. A program in our language consists of one of more modules; each module has an implementation section, a specification section, and an (analysis-specific) abstraction section. We next give an overview of the specification section.

Figure 1 presents the syntax for the specification section of modules in our language. This section contains a list of set definitions and procedure specifications and lists the names of types used in these 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 `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. The `modifies` clause identifies sets whose elements may change as a result of executing the procedure. For the purposes of this paper, `modifies` clauses can be viewed as a special syntax for a frame-condition conjunct in the `ensures` clause. The variables in the `ensures` clause can refer to both the initial and final states of the procedure. Both `requires` and `ensures` clauses use arbitrary first-order boolean algebra formulas B extended with cardinality constraints. A free variable of any formula appearing in a module specification denotes an abstract set or boolean variable declared in that specification; it is an error if no such set or boolean variable has been declared. The expressive power of such formulas is the first-order theory of boolean algebras, which is decidable [20, 26]. The decidability of the specification language ensures that analysis plugins can precisely propagate the specified relations between the abstract sets.

$$
\begin{array}{rl}
M ::= & \textsf{spec module}\ m\ \{(\textsf{type}\ t)^*(\textsf{set}\ S)^*(\textsf{predvar}\ b)^*\ P^*\} \\
P ::= & \textsf{proc}\ pn(p_1 : t_1, \ldots, p_n : t_n)[\textsf{returns}\ r : t] \\
 & [\textsf{requires}\ B]\ \ [\textsf{modifies}\ S^*]\ \ \textsf{ensures}\ B \\
B ::= & SE_1 = SE_2 \mid SE_1 \subseteq SE_2 \mid \textsf{card}(SE){=}k \\
\mid & B \wedge B \mid B \vee B \mid \neg B \mid \exists S.B \mid \forall S.B \\
SE ::= & \emptyset \mid p \mid [m.]\ S \mid [m.]\ S' \\
\mid & SE_1 \cup SE_2 \mid SE_1 \cap SE_2 \mid SE_1 \setminus SE_2
\end{array}
$$

Fig. 1. Syntax of the Module Specification Language

3 The Flag Analysis

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 2 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 4 describes these formula optimizations. These transformations greatly improved the performance of our analysis and allowed our analysis to verify larger programs.

3.1 Operation of the Analysis Algorithm

The flag analysis verifies a module M by verifying each procedure of M. To verify a procedure, the analysis performs abstract interpretation [5] with analysis domain el-

$$
\begin{aligned}
M &::= \textsf{abst module}\ m\ \{D^*\ P^*\} \\
D &::= \textsf{id}{=}D_r; \\
D_r &::= D_r \cup D_r \mid D_r \cap D_r \mid \textsf{id} \mid \{x : T \mid x.f{=}c\} \\
P &::= \textsf{predvar}\ p;
\end{aligned}
$$

Fig. 2. Syntax of the Flag Abstraction Language

ements 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 may also contain unprimed and primed boolean variables b and b' representing the pre- and post-values of local and global boolean variables. The definitions in the abstraction sections of the module provide the interpretations of these variables. 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 p [6, 17, 29].

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 [5]. The analysis also allows the developer to provide loop invariants directly. [2] 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

[2] Our typestate analysis could also be adapted to use predicate abstraction [1, 2, 16] to synthesize loop invariants, by performing data flow analysis over the space of propositional combinations of relationships between the sets of interest, and making use of the fact that the boolean algebra of sets is decidable. Another alternative is the use of a normal form for boolean algebra formulas.

composing the relations defined by the 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

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

3.2 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 the 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,

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

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:

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

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

$$\mathcal{F}(\mathtt{x} = \mathtt{y}) = (\mathtt{x}' = \mathtt{y}) \wedge \mathsf{frame}_x \qquad \mathcal{F}(\mathtt{x} = \mathsf{null}) = (\mathtt{x}' = \emptyset) \wedge \mathsf{frame}_x$$
$$\mathcal{F}(\mathtt{x} = \mathsf{new}\ t) = \neg(\mathtt{x}' = \emptyset) \wedge \textstyle\bigwedge_S (\mathtt{x}' \cap S = \emptyset) \wedge \mathsf{frame}_x$$

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

$$\mathcal{F}(\mathtt{x.f} = \mathtt{c}) = R' = R \cup \mathtt{x} \wedge \textstyle\bigwedge_{S \in \mathsf{alts}(R)} S' = S \setminus \mathtt{x} \wedge\ \mathsf{frame}_{\{R\} \cup\, \mathsf{alts}(R)}$$

where $\mathsf{alts}(\mathtt{R}) = \{S \mid$ abstraction module contains $S = \{x : T \mid x.\mathtt{f} = c_1\}, c_1 \neq c.\}$

The rules for reads and writes of boolean fields are similar but, because our analysis tracks the flow of boolean values, more detailed:

$$\mathcal{F}(\mathtt{x.f} = \mathtt{b}) = \begin{pmatrix} b \wedge B^{+'} = B^+ \cup \mathtt{x} \\ \wedge \bigwedge_{S \in \mathsf{alts}(\mathtt{B}^+)} S' = S \setminus \mathtt{x} \end{pmatrix} \wedge \begin{pmatrix} \neg b \wedge B^{-'} = B^- \cup \mathtt{x} \\ \wedge \bigwedge_{S \in \mathsf{alts}(\mathtt{B}^-)} S' = S \setminus \mathtt{x} \end{pmatrix}$$
$$\wedge \mathsf{frame}_{\{B\} \cup \mathsf{alts}(B)}$$
$$\mathcal{F}(\mathtt{b} = \mathtt{y.f}) = (b' \Leftrightarrow y \in B^+) \wedge \mathsf{frame}_b.$$

where $B^+ = \{x : T \mid x.\mathtt{f} = \mathsf{true}\}$ and $B^- = \{x : T \mid x.\mathtt{f} = \mathsf{false}\}$.
Finally, we have some default rules to conservatively account for expressions not otherwise handled,

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

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

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

where both $\mathsf{ensures}_1$ and $\mathsf{requires}_1$ 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,

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

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

$$\begin{array}{rlrl}
f^+(\mathsf{true}) &= \mathsf{true} & f^-(\mathsf{true}) &= \mathsf{false} \\
f^+(\mathsf{false}) &= \mathsf{false} & f^-(\mathsf{false}) &= \mathsf{true} \\
f^+(!e) &= f^-(e) & f^-(!e) &= f^+(e) \\
f^+(x\,\mathtt{!=}e) &= f^-(x\mathtt{==}e) & f^-(x\,\mathtt{!=}e) &= f^+(x\mathtt{==}e) \\
f^+(e_1\ \mathtt{\&\&}\ e_2) &= f^+(e_1) \wedge f^+(e_2) & f^-(e_1\ \mathtt{\&\&}\ e_2) &= f^-(e_1) \vee f^-(e_2)
\end{array}$$

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

$$\begin{array}{rlrl}
f^+(x.f) &= x \subseteq B & f^-(x.f) &= x \not\subseteq B \\
f^+(x.f\mathtt{==}\mathsf{false}) &= x \not\subseteq B & f^-(x.f\mathtt{==}\mathsf{false}) &= x \subseteq B
\end{array}$$

where $B = \{x : T \mid x.\mathtt{f} = \mathsf{true}\}$; analogously, let $R = \{x : T \mid x.\mathtt{f} = \mathsf{C}\}$. Then,

$$f^+(x.f\mathtt{==}\mathsf{C}) = x \subseteq R \qquad f^-(x.f\mathtt{==}\mathsf{C}) = x \not\subseteq R.$$

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

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

Finally, we have a catch-all condition,

$$f^+(*) = \mathsf{true} \quad f^-(*) = \mathsf{true}$$

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 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.

3.3 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[3]. The implication to be verified, $B \Rightarrow A$, is a formula in the boolean algebra of sets. We use the MONA decision procedure to check its validity [18].

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.

Smart Constructors. The constructors for creating boolean algebra formulas apply peephole transformations as they create the formulas. Constant folding is the simplest peephole transformation: for instance, attempting to create $B \wedge \mathsf{true}$ gives the formula B. Our constructors fold constants in implications, conjunctions, disjunctions, and negations. Similarly, attempting to quantify over unused variables causes the quantifier

[3] 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.

to be dropped: $\exists x.F$ is created as just F when x is not free in F. Most interestingly, 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 SSA form conversion in weakest precondition computation [14, 25].

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 $\exists x.x = x_1 \wedge F(x)$, 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 [4]. 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 formulas after applying incorporation and before invoking the decision procedure. Our global simplification step rebuilds formulas bottom-up and applies simplifications to each subformula.

5 Other Plugins

In addition to the flag plugin, we also implemented a shape analysis plugin that uses the PALE analysis tool to verify detailed properties of linked data structures such as lists and trees. This plugin represents an extreme case in the precision of properties that fully automated analyses can verify. Nevertheless, we were interested in verifying even more detailed and precise data structure consistency properties. Namely, we sought to verify properties of array-based data structures such as hash tables, which are outside the scope of the PALE tool. We therefore implemented a theorem proving plugin which generates verification conditions suitable for partially manual verification using the Isabelle proof checker [31]. One of the goals of this effort is build up a library of instantiable verified data structure implementation modules. Ideally, such a library would eliminate internal data structure consistency as a concern during development, leaving developers free to operate exclusively at the level of abstract sets to concentrate on broader application-specific consistency properties that cut across multiple data structures.

6 Experience

We have implemented our modular pluggable analysis system, populated it with several analyses (including the flag, shape analysis, and theorem prover plugins), and used the system to develop several benchmark programs and applications. Table 1 presents a subset of the benchmarks we ran through our system; full descriptions of our benchmarks (as well as the full source code for our modular pluggable analysis system) are available at our project homepage at `http://cag.csail.mit.edu/~plam/hob`. Minesweeper and water are complete applications; the others are either computational patterns (compiler, scheduler, ctas) or data structures (prodcons). Compiler models a constant-folding compiler pass, scheduler models an operating system scheduler, and ctas models the core of an air-traffic control system. The board, controller, and view modules are the core minesweeper modules; atom, ensemble, and h2o are the core water modules. The **bold** entries indicate system totals for minesweeper and water; note that minesweeper includes several other modules, some of which are analyzed by the shape analysis and theorem proving plugins, not the flag plugin.

	Number of modules	Lines of spec	Lines of impl
prodcons		41	50
compiler		75	143
scheduler		34	22
ctas		49	53
board		78	168
controller		43	133
view		43	372
minesweeper	7	236	750
atom		31	64
ensemble		164	883
h2o		158	420
water	10	582	1976

Table 1. Benchmark characteristics

We next present the impact of the formula transformation optimizations, then discuss the properties that we were able to specify and verify in the minesweeper and water benchmarks.

6.1 Formula Transformation Optimizations

We analyzed our benchmarks on a 2.80GHz Pentium 4, running Linux, with 2 gigabytes of RAM. Table 2 summarizes the results of our formula transformation optimizations.

Each line summarizes a specific benchmark with a specific optimization configuration. A ✓ in the "Smart Constructors" column indicates that the smart constructors optimization is turned on; a × indicates that it is turned off. Similarly, a ✓ in the "Optimizations" column indicates that all other optimizations are turned on; a × indicates that they are turned off. The "Number of nodes" column reports the sizes (in terms of AST node counts) of the resulting boolean algebra formulas. Our results indicate that the formula transformations reduce the formula size by 2 to 60 times (often with greater reductions for larger formulas); the Optimization Ratio column presents the reduction obtained in formula size. The "MONA time" column presents the time spent in the MONA decision procedure (up to 73 seconds after optimization); the "Flag time" column presents the time spent in the flag analysis, excluding the decision procedure (up to 477 seconds after optimization). Without optimization, MONA could not successfully check the formulas for the compiler, board, view, ensemble and h2o modules because of an out of memory error.

	Optimizations	Smart Constructors	Number of nodes	Optimization ratio	MONA time	Flag time
prodcons	✓	✓, ×	12306	2.46	0.17	0.03
	×	✓,×	30338	1.00	0.27	0.04
compiler	✓	✓	15854	32.06	0.45	5.10
	✓	×	28003	18.15	0.60	6.19
	×	✓,×	508375	1.00	N/A	60.27
scheduler	✓	✓,×	442	2.44	0.05	0.04
	×	✓,×	1082	1.00	0.12	0.14
ctas	✓	✓,×	2874	3.18	0.21	0.12
	×	✓,×	9141	1.00	12.79	0.33
board	✓	✓	28658	41.43	1.92	18.89
	✓	×	106550	11.14	11.45	29.27
	×	✓	926321	1.28	N/A	134.94
	×	×	1187379	1.00	N/A	151.46
controller	✓	✓	6759	4.23	0.41	0.18
	✓	×	7101	4.02	0.41	0.18
	×	✓,×	28594	1.00	3.08	0.54
view	✓	✓	15878	59.08	1.07	12.38
	✓	×	53925	17.39	1.45	18.88
	×	✓,×	93800	1.00	N/A	263.15
atom	✓	✓	9677	3.14	0.53	0.13
	✓	×	10244	2.97	0.54	0.13
	×	✓,×	30447	1.00	40.95	0.43
ensemble	✓	✓	120279	20.60	50.90	34.15
	✓	×	148748	16.66	105.59	47.06
	×	✓,×	2478004	1.00	N/A	464.52
h2o	✓	✓	205933	4.32	73.80	477.01
	✓	×	206167	4.31	81.85	475.86
	×	✓,×	889637	1.00	N/A	1917.99

Table 2. Formula sizes before and after transformation

6.2 Minesweeper

We next illustrate how our approach enables the verification of properties that span multiple modules. Our minesweeper implementation has several modules: 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 modules and 236 non-blank lines in the specification and abstraction modules.

Minesweeper 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 using the `isMined`, `isExposed` and `isMarked` fields of `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 global boolean variable `gameOver`, which it sets to `true` when the game ends.

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

– The sets of exposed and unexposed cells are disjoint; unless the game is over, the sets of mined and exposed cells are also 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.

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. Our analysis is capable of ensuring the validity of these modules, since it can track abstract set membership, solve formulas in the boolean algebra of sets, and incorporate the effects of invoked procedures as it analyzes each module. Note that for these modules, our analysis need not reason about any correspondence between concrete data structure representations and abstract sets.

The set abstraction supports typestate-style reasoning at the level of individual objects (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. The `board` module, for example, has a global `gameOver` variable which indicates whether or not the game is over. The system uses this variable and the definitions of relevant sets to maintain the global invariant `gameOver | 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 variable.

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

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

Our analysis verifies this property by conjoining it to the `ensures` and `requires` clauses of appropriate procedures. The `board` module is responsible for maintaining this invariant. Yet the analysis of the board module does not, in isolation, have the ability to completely verify the invariant: it cannot reason about the concrete state of `ExposedSet.Content` or `UnexposedList.Content` (which are defined in other modules). However, the `ensures` clauses of its callees, in combination with its own reasoning that tracks membership in the `ExposedCells` set, enables our analysis to verify the invariant (assuming that `ExposedSet` and `UnexposedList` work correctly).

Our system found a number of errors during the development and maintenance of our minesweeper implementation. We next present one of these errors. 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);
```

6.3 Water

Water is a port of the Perfect Club benchmark MDG [3]. 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, then uses the new simulation state to compute the potential and kinetic energy of the system.

Water 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 simulation parameters, which are stored in a text file and loaded at the start of the computation. This module defines two boolean variables, `Init` and `ParmsLoaded`. If `Init` is true, then the module has been initialized, *i.e.* the appropriate arrays have been allocated on the heap. If `ParmsLoaded` is true, then the simulation parameters have been loaded from disk and written into these arrays. Our analysis verifies that the program does not load simulation parameters until the arrays have been allocated and does not read simulation parameters until they have been loaded from the disk and written into the arrays.

The fundamental unit of the simulation is the atom, which is encapsulated within the `atom` module. Atoms cycle between the *predicted* and *corrected* states, with the `predic` and `correc` procedures performing the computations necessary to effect these state changes. A correct computation will only predict a corrected atom or correct a predicted atom. To enforce this property, we define two sets `Predic` and `Correc` and populate them with the predicted and corrected atoms, respectively. The `correc` procedure operates on a single atom; its precondition requires this atom to be a member of the `Predic` set. Its postcondition ensures that, after successful completion, the atom is no longer in the `Predic` set, but is instead in the `Correc` set. The `predic` procedure has a corresponding symmetric specification.

Atoms belong to molecules, which are handled by the `H2O` module. A molecule tracks the position and velocity of its three atoms. Like atoms, each module can be in a variety of conceptual states. These states indicate not only whether the program has predicted or corrected the position of the molecule's atoms but also whether the program has applied the intra-molecule force corrections, whether it has scaled the forces acting on the molecule, etc. We verify the invariant that when the molecule is in the predicted or corrected state, the atoms in the molecule are also in the same state. The interface of the `H2O` module ensures that the program performs the operations on each molecule in the correct order — for example, the `bndry` procedure may operate only on molecules in the `Kineti` set (which have had their kinetic energy calculated by the `kineti` 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 velocities over time. The ensemble module uses boolean predicates to track the state of the computation. When the boolean predicate `INTERF` is true, for example, then the program has completed the interforce computation for all molecules in the simulation. Our analysis verifies that the boolean predicates, representing program state, satisfy the following ordering relationship:

$$\texttt{Init} \leadsto \texttt{INITIA} \leadsto \texttt{PREDIC} \leadsto \texttt{INTRAF} \leadsto \texttt{VIR} \leadsto \texttt{INTERF} \leadsto \cdots$$

Our specification relies on an implication from boolean predicates to properties ranging over the collection of molecule objects, which can be ensured by a separate array analysis plugin [23].

These properties 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, may have 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.

7 Related Work

Typestate systems track the conceptual states that each object goes through during its lifetime in the computation [7, 9–12, 30]. They generalize standard type systems in that the typestate of an object may change during the computation. Aliasing (or more generally, any kind of sharing) is the key problem for typestate systems — if the program uses one reference to change the typestate of an object, the typestate system must ensure that either the declared typestate of the other references is updated to reflect the new typestate or that the new typestate is compatible with the old declared typestate at the other references.

Most typestate systems avoid this problem altogether by eliminating the possibility of aliasing [30]. Generalizations support monotonic typestate changes (which ensure that the new typestate remains compatible with all existing aliases) [12] and enable the program to temporarily prevent the program from using a set of potential aliases, change the typestate of an object with aliases only in that set, then restore the typestate and reenable the use of the aliases [10]. It is also possible to support object-oriented constructs such as inheritance [8]. Finally, in the role system, the declared typestate of each object characterizes all of the references to the object, which enables the typestate system to check that the new typestate is compatible with all remaining aliases after a nonmonotonic typestate change [21].

In our approach, the typestate of each object is determined by its membership in abstract sets as determined by the values of its encapsulated fields and its participation in encapsulated data structures. Our system supports generalizations of the standard typestate approach such as orthogonal typestate composition and hierarchical typestate classification. The connection with data structure participation enables the verification of both local and global data structure consistency properties.

8 Conclusion

Typestate systems have traditionally been designed to enforce safety conditions that involve objects whose state may change during the course of the computation. In particular, the standard goal of typestate systems is to ensure that operations are invoked only on objects that are in appropriate states. Most existing typestate systems support a flat set of object states and limit typestate changes in the presence of sharing caused by aliasing. We have presented a reformulation of typestate systems in which the typestate of each object is determined by its membership in abstract typestate sets. This reformulation supports important generalizations of the typestate concept such as typestates that capture membership in data structures, composite typestates in which objects are members of multiple typestate sets, hierarchical typestates, and cardinality constraints on the number of objects that are in a given typestate. In the context of our Hob modular pluggable analysis framework, our system also enables the specification and effective verification of detailed local and global data structure consistency properties, including arbitrary internal consistency properties of linked and array-based data structures. Our system therefore effectively supports tasks such as understanding the global sharing patterns in large programs, verifying the absence of undesirable interactions, and ensuring the preservation of critical properties necessary for the correct operation of the program.

Acknowledgements. This research was supported by the DARPA Cooperative Agreement FA 8750-04-2-0254, DARPA Contract 33615-00-C-1692, the Singapore-MIT Alliance, and the NSF Grants CCR-0341620, CCR-0325283, and CCR-0086154.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, 2001.
2. 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.
3. 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.
4. L. Clarke and D. Richardson. Symbolic evaluation methods for program analysis. In *Program Flow Analysis: Theory and Applications*, chapter 9. Prentice-Hall, Inc., 1981.
5. 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.
6. 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.
7. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM PLDI*, 2001.

8. R. DeLine and M. Fähndrich. Typestates for objects. In *Proc. 18th ECOOP*, June 2004.
9. 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.
10. M. Fahndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proc. ACM PLDI*, 2002.
11. M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 302–312. ACM Press, 2003.
12. M. Fähndrich and K. R. M. Leino. Heap monotonic typestates. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2003.
13. J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*. Springer, 2003.
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. D. Kozen. Complexity of boolean algebras. *Theoretical Computer Science*, 10:221–247, 1980.
21. V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proc. 29th POPL*, 2002.
22. V. Kuncak and M. Rinard. The first-order theory of sets with cardinality constraints is decidable. Technical Report 958, MIT CSAIL, July 2004.
23. P. Lam, V. Kuncak, and M. Rinard. On our experience with modular pluggable analyses. Technical Report 965, MIT CSAIL, September 2004.
24. P. Lam, V. Kuncak, K. Zee, and M. Rinard. The Hob project web page. http://hob.csail.mit.edu, 2004.
25. K. R. M. Leino. Efficient weakest preconditions. KRML114a, 2003.
26. L. Loewenheim. Über mögligkeiten im relativkalkül. *Math. Annalen*, 76:228–251, 1915.
27. A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *Programming Language Design and Implementation*, 2001.
28. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.
29. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis problems. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc., 1981.
30. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, January 1986.
31. K. Zee, P. Lam, V. Kuncak, and M. Rinard. Combining theorem proving with static analysis for data structure consistency. In *International Workshop on Software Verification and Validation (SVV 2004)*, Seattle, November 2004.