Generalized Typestate Checking Using Set Interfaces and Pluggable Analyses

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

Abstract

We present a generalization of standard typestate systems in which the typestate of each object is determined by its membership in a collection of abstract typestate sets. This generalization supports typestates that model participation in abstract data types, composite typestates that correspond to membership in multiple sets, and hierarchical typestates. Because membership in typestate sets corresponds directly to participation in data structures, our typestate system characterizes global sharing patterns.

In our approach, each module encapsulates a data structure and uses membership in abstract sets to characterize how objects participate in its data structure. Each analysis verifies that the implementation of the module 1) preserves important internal data structure representation invariants and 2) conforms to a specification that uses formulas in a set algebra to characterize the effects of operations on the data structure. The analyses use the common set abstraction 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.

1 Introduction

Typestate systems allow the type of an object to change during its lifetime in the computation, enabling the typestate system to enforce safety properties that depend on changing object states of the objects and increasing the precision of the abstractions in the type system (in standard type systems, an object's type never changes). The standard typestate approach assigns, at each program point, a single state to each object. The states change in response to program actions [21].

This paper presents a new formulation of typestate systems. Instead of associating a single state with each object, our system instead 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 the following generalizations of the standard typestate approach:

- **Abstract Data Types:** For typestate purposes, abstract data types can be viewed as maintaining several abstract sets of objects. For example, a list abstract data type maintains the set of objects in the list, while a tree maintains the set of objects in the tree. With this perspective, the typestate of an object is a function of its participation in the abstract data type as reflected in its membership in the data type's abstract sets of objects.

- **Orthogonal Composition:** In standard typestate systems, each object has a single atomic typestate. In our formulation, however, an object can be a member of multiple sets simultaneously. This promotes composite typestate structures in which the developer endows each component with a collection of abstract sets, with each set corresponding to an aspect of the typestate relevant to the component. With this kind of structure, each object's typestate is an orthogonal composition of the typestate aspects from each of the components in which it participates. Examples include composite typestates for objects that participate in multiple data structures and objects that play multiple roles within a single component.

  The advantages of this approach include better modularity (because each component deals only with those aspects of the typestate that are relevant for its operation) and support for polymorphism (because each component can operate successfully on multiple objects that participate in different ways in other components).

- **Hierarchical Typestates:** Hierarchical classification via inheritance is a key element of the type systems in most object-oriented languages, but is completely absent in existing flat typestate systems. Our formulation cleanly supports typestate hierarchies — a collection of sets can partition a more general set, with the subset inclusion ordering capturing the hierarchy.

- **Sharing and Typestates:** Sharing via aliased object references has caused problems for standard typestate systems — it has been difficult to ensure that if the program uses one reference to access the object and change its typestate, the declared type of other references is appropriately adjusted. Restrictions adopted to ensure soundness have included the elimination of aliasing [21, 6], requiring typestate restoration after temporary changes [9], and allowing only monotonic typestate changes [10]. To the best of our knowledge, the role system [15] is the only typestate system to support both sharing and nonmonotonic typestate changes.

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

The basic utility of typestate systems is to enforce safety properties, specifically to ensure that the program never applies an operation to an object if the object is not in the correct typestate. We believe our generalization of the standard typestate approach substantially improves the sophistication of the properties that the type system can verify, its ability to enforce important safety properties, and the ability of typestate systems to support modular development practices. It also allows our system to much more effectively support many software engineering activities such as understanding the global sharing patterns in large programs (by verifying which objects participate in which data structures), verifying the absence of undesirable interactions (by enforcing the absence of sharing for selected objects), and understanding the sequences of actions that the program may generate (by enforcing finite state protocols through constraints on the movement of objects between abstract sets).

1.1 Verification via Pluggable Analyses

In our approach, there is an intimate connection between typestate checking and verifying data structure consistency properties. Because each object's typestate reflects how it participates in different data structures, the soundness of the checker depends on its ability to verify that data structures remain consistent throughout the execution of the program.

Over the years, researchers have developed many algorithms for verifying that programs preserve important consistency properties [2, 11, 17, 19]. But 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 set of consistency properties, we instead propose a technique that leverages the generalized typestate information to enable the application of multiple pluggable analyses to the same program, with each analysis applied to the data structures for which it is appropriate. The key features of this technique include:

- **Modular Analysis, Shared Objects, and Encapsulated Fields:** In our approach, the program contains a set of modules, each of which encapsulates the implementation of one of the data structures. Instead of attempting to analyze the entire program, each analysis operates on a single module. By focusing each analysis on only those parts of the program that are relevant for the properties it is designed to verify, we enable the application of sophisticated analyses to sizable programs composed of multiple modules.

  One factor that complicates this approach is the need for objects to participate in multiple data structures and therefore the need to share objects between modules analyzed by different algorithms. To eliminate the possibility that one module may corrupt another's data structure (and to ensure that each algorithm analyzes all of the relevant code), modules encapsulate fields *(and not objects):* the underlying language prevents one module from accessing the fields of another module. Each module therefore encapsulates all of the fields required to implement its data structure; objects that participate in multiple data structures from multiple modules contain fields from each of these modules.

- **Specification via Set Abstraction:** Each module has an implementation and a specification. It is the responsibility of the analysis to verify that the implementation correctly implements the specification. Instead of exposing the implementation details of the encapsulated data structure, the specification uses a collection of abstract sets to summarize the effect of each procedure. This collection of sets characterizes how objects participate in various data structures. For example, the specification for a linked list might have an abstract set that contains all of the objects in the linked list. The specification for the insert procedure would indicate that the procedure adds the inserted object into the set; the specification for the remove procedure would indicate a corresponding removal.

- **Abstraction Functions and Internal Data Structure Consistency:** Each analysis uses an abstraction function to establish the connection between the concrete data structure implementation and abstract set membership. This abstraction function enables the analysis to translate the set membership properties of objects that cross module boundaries back into concrete data structure properties. These concrete properties are often crucial for preserving the internal consistency of the data structure.

  For example, a list insert procedure may require that an object to be inserted must not already be a member of the abstract set containing all of the objects in the list. This set membership property, in turn, enables the analysis to verify that the object is not already in the list, which may be the precondition required to preserve the internal consistency of the list data structure.

- **Invariants Involving Multiple Modules:** Systems often have consistency properties that involve multiple data structures and therefore cross encapsulation boundaries. For example, some systems may require the objects that participate in two data structures to be disjoint; others may require that every object in one data structure to also be present in another. Note that because these properties involve objects shared across multiple modules, different analyses must somehow interoperate if they are to successfully verify the property.

  In our approach, these kinds of invariants are expressed using a boolean algebra of abstract set inclusion properties and locally verified at the appropriate program points by each analysis. This approach eliminates the need to apply complex (and therefore potentially unscalable) analyses across large regions of the program. It instead promotes the appropriately focused application of arbitrarily sophisticated analyses to individual modules within large systems, with the results of these analyses combined to enable the verification of broad properties that involve multiple modules.

- **Analysis Scopes:** Data structure updates may legitimately violate invariants as long as they restore the invariants before they complete. For invariants that involve multiple modules, this restoration usually requires the coordinated invocation of procedures from multiple modules.

Our approach uses *analysis scopes* to identify the regions of the program in which each invariant may be legitimately violated. Each analysis scope contains an invariant and a collection of modules that are analyzed together to verify the invariant. Some of these modules are exported and can be invoked from outside the scope; the other modules may be invoked only from within the scope. When our analysis verifies an exported module it ensures that, if the invariants hold upon entry to the exported modules, they are restored upon exit. For properties that involve multiple modules, this approach verifies that procedure invocations are properly coordinated to preserve the invariants.

Together, these features enable the focused application of a full range of precise, sophisticated analyses to programs that contain multiple data structures encapsulated in multiple modules. They promote the development of a range of pluggable analyses that developers can deploy as necessary to verify important data structure consistency properties. Abstract sets enable different analyses to communicate and interoperate to verify properties that cross module boundaries to involve multiple data structures. Our approach supports the appropriately verified participation of objects in multiple data structures, including patterns in which objects migrate sequentially through different data structures and patterns in which objects participate in multiple data structures simultaneously.

1.2 Characterizing Global Sharing

When combined with a programming language that encapsulates all object references inside modules [16], our approach enables the typestate system to capture global sharing properties. Together, all of the sets in an object's typestate completely characterize its participation in all of the data structures in the program. An absence of sharing shows up as a typestate assertion that an object is not a member of two distinct sets in a given collection of sets.

Compare this abstract concept of sharing with the standard concrete concept of sharing based on the aliasing of object references in the heap. In the standard concept, the program accesses objects by following object references; two objects share another object if they both have a reference to the shared object. In our typestate concept, the program accesses objects by invoking operations in modules that return references to objects; two modules share an object if the object is in a typestate set from each of the modules.

We believe this approach makes it possible, for the first time, to obtain an analysis that is both precise enough to verify sharing properties in programs with quite sophisticated data structures, yet scalable enough to characterize such properties in large programs composed out of multiple modules.

1.3 Contributions

This paper makes the following contributions:

- **Generalized Typestate Framework:** It presents a new generalized typestate framework based on the concept of object membership in abstract typestate sets. This framework supports the clean generalization of the standard flat typestate approach to support hierarchical typestates and typestates composed of the orthogonal composition of multiple typestate elements.
- **Pluggable Analyses:** It presents an approach that enables the focused application of multiple precise analyses to multiple data structures encapsulated within multiple modules, with the analysis results appropriately combined to verify properties that involve 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. It also supports the use of analysis scopes to identify the regions of the program that the analysis must process to verify invariants involving multiple modules.
- **Global Sharing:** It presents an abstract model of sharing in programs composed out of multiple modules and shows how our typestate system makes it possible to completely characterize global, large-scale object sharing patterns in such programs. We claim that our approach is therefore the first solution to the global sharing problem in programs with multiple modules and sophisticated data structures.

2 Example

We next present a process scheduler example that illustrates how modules can use typestate sets to capture important data structure invariants and typestate changes. The scheduler maintains a list of running processes and a priority queue of suspended processes. There are three modules: 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 interface to the scheduler). Each of these modules has three submodules: an implementation module (which contains the code for the implementation of the procedures in the module), a specification module (which uses the module's sets to specify the procedure interfaces), and an abstraction module (which specifies the relationship between the concrete data structures in the implementation module and the typestate sets in the specification module). Finally, there is also the *scope* declaration for the scheduler, which identifies the modules that participate in the scope, specifies which modules are exported, and contains an invariant (expressed using the typestate sets) that captures the relationships between the typestate sets from different modules.

2.1 Running List Module

Figure 1 presents the running list implementation module. The module maintains a reference `root` to the first `Process` object in the list. The `format` statement specifies that all `Process` objects have a `next` and `prev` field that together implement a circular doubly-linked list. These fields are accessible only within the `RunningList` module. [1] Note that other implementation modules may also use additional `format` statements to add their own fields to `Process` objects. When the program runs, each `Process` object will

[1]This implementation places the `next` and `prev` fields directly in the `Process` objects [5]. Our approach also supports the more common implementation that uses auxiliary encapsulated list objects to refer to the `Process` objects; in this implementation the auxiliary list objects (and not the `Process` objects) contain the `next` and `prev` fields.

contain all of the fields declared in all of these `format` statements. The module exports two procedures: the `add` procedure, which inserts its parameter `p` into the running list, and the `remove` procedure, which removes its parameter `p` from the running list.

Figure 2 presents the specification module for the running list. The specification has a single abstract set, `InList`, which contains all of the `Process` objects in the running list. The `requires` clause of the specification of the `add` procedure requires the parameter `p` to not already be in `InList`. The `ensures` clause states that the effect of the `add` procedure is to add the parameter `p` to `InList` (the notation `InList'` denotes the new version of `InList` after the `add` procedure executes; the unprimed `InList` denotes the old version before it executes). The `modifies` clause indicates that the procedure modifies the `InList` set only.

We would like to be able to use an appropriate analysis to verify that the running list implementation satisfies its specification. An analysis based on monadic second-order logic over trees (as implemented, for example, in the PALE analysis tool [17]) is able to verify this correspondence, but it needs some additional information to do so. The abstraction module in Figure 3 contains this information.

The abstraction module starts by identifying the plugin to use to perform the verification, in this case the `GraphTypes` plugin. It then specifies the abstraction function that establishes the correspondence between the concrete data structure implementation and the abstract sets in the specification. In this case, the `InList` set is defined to be all objects reachable by following `next` fields starting from the `root`.

To verify the correspondence, the analysis plugin establishes a simulation relation between the specification and the implementation. The plugin shows the simulation relation for each public procedure of the module by first using the abstraction function to map the `requires`, `ensures`, and `modifies` clauses to the precondition and postcondition of the concrete data structure, and then verifying the implementation of the procedure with respect to this precondition and postcondition.

For example, to show the simulation relation for the `add` procedure, the plugin assumes that `p` is not in the list of nodes reachable from `root` and shows that the set of reachable objects at the end of the procedure is equal to the original set extended with `p`.

The simulation relation need not hold unless the data structure satisfies several internal consistency properties; we call such properties *representation invariants*. The abstraction module identifies these properties using an `invariant` statement to specify that `root` points to a circular doubly-linked list, as identified by the `List` graph type declaration. As appropriate for the `GraphTypes` plugin, this declaration uses (a syntactic sugar for) monadic second-order logic to specify that the `prev` field is the inverse of the `next` field. During the analysis of the implementation module, the plugin assumes that this invariant holds at the start of each procedure and proves that it holds at the end of each procedure. In effect, the representation invariants are conjoined with the precondition and postcondition of each public procedure. The circular doubly-linked list invariant is crucial for proving the simulation relation: unless `prev` is an inverse of `next`, the procedure `remove(p)` could not guarantee the removal of `p` from the set of reachable nodes of the list.

All implementation and specification modules are written in a common language. But each abstraction module is written in a language appropriate for its corresponding

```
impl module RunningList {
   reference root : Process;
   format Process { next, prev : Process }

   proc add(p : Process) {
     if (root=null) then {
       root := p; p.next := p;  p.prev := p;
     } else { p.next := root.next; root.next := p;
              p.prev := root; p.next.prev := p;
     }
   }

   proc remove(p : Process) {
      if (p=root)
          if (p.next=root) {
            root := null;
            p.next := null; p.prev := null;
            return;
          } else root := p.next;
      Process pp, pn; pp := p.prev; pn := p.next;
      pp.next := pn; pn.prev := pp;
      p.next := null; p.prev := null;
   }
}
```

Figure 1: Running List Implementation Module

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

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

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

Figure 2: Running List Specification Module

```
abst module RunningList {
   use plugin GraphTypes;
   InList = {p : Process | p elem root.next*};
   GraphType List = {  next : List | List[$];
                       prev : List[this.~next] }
   invariant root : List | null;
}
```

Figure 3: Running List Abstraction Function Module

```
impl module SuspendedQueue {
   reference root:Process;
   format Process {
     priority : int;
     left, right : Process
   }
   proc isEmpty() returns b : boolean { ... }
   proc add(p : Process) { ... }
   proc removeFirst() returns p : Process { ... }
}
```

Figure 4: Skeleton of the Priority Queue Implementation Module

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

   proc isEmpty() returns b : boolean
     ensures b <=> InQueue = {};

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

   proc removeFirst() returns p : Process
     requires InQueue != {}
     modifies InQueue
     ensures InQueue' = InQueue - p;
}
```

Figure 5: Priority Queue Specification Module

```
abst module SuspendedQueue {
   use plugin GraphTypes;
   InQueue = {p : Process | p elem root.<left+right>*};
   GraphType Tree = {left, right : Tree | null}
   invariant root : Tree | null;
}
```

Figure 6: Priority Queue Abstraction Module

plugin. We expect all abstraction modules to specify, at a minimum, the abstraction function that establishes the connection between the implementation and the specification. We also expect that abstraction modules will often identify the representation invariants they need to establish the correspondence. The syntax of these invariants will depend on the requirements of the specific analysis plugin. In general, abstraction modules may contain any additional information useful for the analysis (such as properties of objects that are useful to track during fixpoint computation).

2.2 Priority Queue Module

The priority queue module implements a priority queue of suspended processes using a binary search tree. Figure 4 presents the skeleton of the `SuspendedQueue` implementation module. The module introduces three new fields into the `Process` format: the `priority` field is the sorting criterion for `Process` objects in the tree, whereas `left` and `right` fields implement the tree structure.

The priority queue contains three procedures. The `isEmpty` procedure checks whether the root is `null`. The `add` procedure inserts the node into the binary search tree. The `removeFirst` removes the root of the binary search tree. We omit the implementation details of this implementation.

Figure 5 presents the specification of the `SuspendedQueue` module. The specification summarizes priority queue procedures in terms of the set `InQueue`, which is the set of all `Process` objects stored in the queue: `isEmpty` tests set emptiness, `add(p)` inserts object `p` into the set, whereas `removeFirst` removes an object from the set and returns it as the result.

Figure 6 presents the abstraction module that establishes the connection between the implementation and the specification of the priority queue by defining the set `InQueue` as the set of all objects reachable along `left` and `right` fields. As in the case of the `RunningList` module, the correspondence between implementation and specification is verified using the `GraphTypes` plugin. The representation invariant specifies that the data structure referenced by `root` satisfies the property `Tree`, which is a simple graph type with only backbone edges.

2.3 Scheduler Module

Figure 7 present the `Scheduler` implementation module. The module uses a `format` declaration to add a `status` field to `Process` objects; this field is 0 if the process is suspended and 1 if the processes is running. This field encodes the conceptual state of each process (either running or suspended) and enables the module to quickly determine the status of a process. The `Scheduler` module uses the `RunningList` and `SuspendedQueue` modules to actually store the running and suspended processes.

The specification module in Figure 8 has two abstract sets: the set `Running` of running processes and the set `Suspended` of suspended processes. These sets correspond to the conceptual states that `Process` objects can be in. The specifications of the procedures (`suspend`, `hasSuspended`, and `wakeUpFirst`) therefore reflect the movement of objects between the various states. The `calls` clause in this module specifies that procedures in the `Scheduler` module may invoke procedures in the `RunningList` and `SuspendedQueue` modules. Calls clauses are used to identify potentially reentrant call sites; the analysis treats such sites differently from non-reentrant sites. Specifically, all invariants that involve the data structures or typestate sets of the analyzed module must hold at reentrant sites. These invariants can be violated at non-reentrant sites.

The abstraction module in Figure 9 uses the `status` flag to define the `Running` and `Suspended` sets. The flags plugin described in Section 3.1 can use this abstraction function to verify that the scheduler implementation correctly implements its specification.

When showing the conformance of the `suspend` procedure, the flag plugin must take into account the effects of the `RunningList.remove` and `SuspendedQueue.add` procedures, which are located outside the `Scheduler` module and analyzed using an entirely different plugin. Nevertheless, the flag plugin can take into account the effect of these procedures using their specifications, because these specifications are expressed in the common specification language based on sets.

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

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

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

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

Figure 7: Scheduler Implementation Module

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

   proc suspend(p : Process)
     requires p in Running
     modifies Running, Suspended
     ensures Suspended' = Suspended + p and
             Running' = Running - p;

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

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

Figure 8: Scheduler Specification Module

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

Figure 9: Scheduler Abstraction Module

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

Figure 10: Scope Declarations

2.4 Scope Invariants

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

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

We address these issues with *analysis scopes*; Figure 10 presents the analysis scope for our example. In general, scopes are a collection of modules and invariants; each scope may have private modules and exported modules. Scopes specify invariants that involve multiple modules, specify a policy on when the invariants should hold, and control access to modules from outside the scope.

This scope contains an invariant with three clauses that together express the set equality and disjointness properties discussed above. It also identifies a list of modules in the scope; the invariant may be temporarily violated within these modules (in our example these are the Scheduler, RunningList, and SuspendedQueue modules). It exports the Scheduler module, indicating that the RunningList and SuspendedQueue modules cannot be invoked from outside the scope. The exported Scheduler procedures, on the other hand, can be invoked from outside the scope.

The analysis of the Scheduler module assumes the invariant holds at the start of each procedure and must show that it holds at the exit of the procedure. In our example, the flag analysis uses the specifications of the running list and priority queue modules (which are expressed in terms of abstract sets) in the verification of the invariant. By encapsulating the complexity of the internal data structure properties inside the relevant modules, our technique enables the use of expensive analyses in those modules that require them, while allowing the use of simpler and faster analyses in the remainder of the program.

One consequence of using scopes is that a module M need not report the effects of a transitive callee M' if it does not export M' and $\mathsf{scopes}(M) \cap \mathsf{scopes}(M') = \emptyset$, where $\mathsf{scopes}(M)$ is the set of all scopes C that declare M in its modules clause. Module M need not (indeed, it may not) declare sets of M' in its modifies and ensures clauses; on the other hand, a module M_0 sharing a scope with M' may refer to sets in M' in its modifies and ensures clauses, as required. The net effect is a simplification of the specification of M and a corresponding simplification in the propagation of specifications up the module hierarchy.

Moreover, the invariants of scope C are implicitly introduced at the boundaries of C as necessary to guarantee that modules outside C may assume that these invariants hold. In this way, scope invariants further simplify the specifications of individual procedures. Note that our specification language contains all boolean operations including implication. The developer may therefore choose to write a scope invariant I' of the form $F \Rightarrow I$, where F is an expression determining whether the property I should hold. By setting F to false or true, the developer can explicitly control the policy of whether the invariant should hold on procedure exit. The developer can similarly use scope invariants of the form $\bigwedge_{j=1}^{n} F_j \Rightarrow I_j$ to model a collection of modules that can be in one of n possible states. Note further that by writing invariants of the form $\bigwedge_{j=1}^{n} A_j \subseteq B_j$, the developer can use sets A_j to control (at the granularity of objects) which of the properties $B_1, \ldots, B_n$ hold.

2.5 Combining Sets and Invariants

This example illustrates how our set-based system can capture relationships between object states based on properties such as data structure participation and the contents of object fields. It is, of course, possible to build more sophisticated relationships. Consider, for example, a data structure that should contain only objects whose (primitive) fields satisfy a certain property P. The developer could enforce this constraint by defining the abstract set $S_P = \{o \mid P(x)\}$ of objects that satisfy property P, the abstract set S_D of objects stored in the data structure, and a scope invariant $S_D \subseteq S_P$. This invariant enables analyses to recognize that all objects fetched from the data structure satisfy property P.

Consider a program that first modifies the fields of an object in a way that may violate property P, then removes the object from the data structure. In the period between the modification and the removal, the analysis tracks the violation of the scope invariant $S_D \subseteq S_P$ and hence is able to recognize that objects fetched from the data structure during this period may not satisfy property P. The restoration of the invariant after the removal also restores the ability of the analysis to recognize that objects fetched from the data structure satisfy property P.

We have taken a layered approach to our typestate system: the implementation modules use a stateless type system used to specify the format of the fields in objects. The specification modules layer a generalized typestate system on top of this stateless system, with set membership determining the components that together make up the typestate of each object.

3 Analysis Plugins

To analyze a program, our technique uses an analysis plugin to check each module in turn; the program successfully verifies iff each of the modules successfully verifies in isolation. To simplify the application of the plugin to the module, our technique processes the scope declarations, any representation invariant declarations in the abstraction modules, and the specification modules to produce a precondition and postcondition for each exported procedure in the analyzed module. It is then the responsibility of the analysis plugin to verify that the postcondition holds for each procedure if the precondition holds when the procedure is invoked.

$$
\begin{array}{rcl}
M & ::= & \textsf{abst module } m \ \{U \ D^* \ I\ \} \\
U & ::= & \textsf{use plugin } p; \\
D & ::= & S{=}\{x : f | F_p(x)\}; \\
I & ::= & \textsf{invariant } A; \\
A & ::= & F_p \mid \neg A \mid A_1 \wedge A_2 \mid A_1 \vee A_2 \mid \textsf{let } S{=}\{x : f | F_p(x)\} \textsf{ in } B
\end{array}
$$

Figure 11: Syntax of Abstraction Modules

When a procedure in one scope invokes a procedure in another scope, it may be the case that the invariant of the first scope may be temporarily violated. The analysis would like to assume that 1) the precondition of the invoked procedure does not depend on this invariant, and 2) the invoked procedure has no effect on the current analysis facts from the invoking scope. This is true unless the call site is *reentrant*, i.e., unless the execution of the invoked procedure turns around and invokes a procedure that is in the invoking scope. To ensure the soundness of the analysis in this case, our technique requires the invariant of each scope to hold both before and after each potentially reentrant call site. Our technique identifies these program points and the properties that must hold; it is the responsibility of the analysis plugin to verify that the properties do in fact hold.

To analyze a module M, the system uses the following information:

- the implementation, specification, and abstraction module for M;
- the specifications of all modules whose procedures are called from the implementation module for M;
- whether any call sites in the implementation of M are potentially module-reentrant or scope-reentrant;
- the declarations of the scopes in $\textsf{scopes}(M)$; and
- the sets $\textsf{scopes}(M')$ for every module M' called from M.

Plugins and Abstraction Modules The analysis of M is performed by the analysis plugin specified in the abstraction module for M. Figure 11 presents the generic syntax of abstraction modules; each analysis plugin augments this syntax with its *plugin annotation language* (denoted A). In addition, abstraction modules may supply additional information in a form expected by the analysis plugin. The plugin annotation language is used both to write the abstraction function defining the representation of each set and to state the representation invariant. All plugin annotation languages extend the set specification language with specialized constructs of the *plugin property language* (denoted F_p) for describing properties of concrete data structures of the implementation of M. The key responsibility of the plugin is to verify that the implementation of a procedure conforms to a given `requires`/`ensures` clause expressed in the plugin annotation language.

Analysis Summary The overall process of analyzing module M consists of the following sequence of steps:

1. **Check Inter-Scope Calls:** For each procedure invocation $M_1.p$ in the implementation of M, ensure that M_1 is exported in every scope in $\textsf{scopes}(M_1) \setminus \textsf{scopes}(M)$.

2. **Modifies Clause Expansion:** For each procedure p of module M with modifies clause m and ensures clause e, augment e to yield

$$e' := e \wedge \bigwedge_{S \in U \setminus m^*} S'=S,$$

where m^* is the union of m and the modifies clauses m' of transitive callees of p, and U contains all sets in $\mathsf{scopes}(M)$ and all sets belonging to exported modules called in the implementation of p.

3. **Reentrant Call Detection** Using an inter-module call graph constructed from the program's `calls` clauses in the specifications of each module, mark module-reentrant and scope-reentrant call sites. By definition, a module-reentrant site directly calls procedure p belonging to some module $M' \neq M$, which in turn transitively calls $p' \in M$. Similarly, a scope-reentrant site directly calls p belonging to scope $C' \notin \mathsf{scopes}(M)$, which transitively calls p' belonging to $C \in \mathsf{scopes}(M)$.

4. **Scope Invariant Distribution:** For every scope C that exports M, add the scope invariant of C to the following program points:
   (a) `requires` and `ensures` clause of each procedure declared in M;
   (b) each scope-reentrant call to a procedure declared outside C.

5. **Module Projection:** If a procedure p in M calls a procedure p' in module M' and the specification of p' uses a set S not declared in any of the scopes $\mathsf{scopes}(M)$, then project the specification of p' by quantifying over S.

6. **Requires/Ensures Clause Mapping:** Use the abstraction function for sets specified in the abstraction function module for M to transform the `requires` and `ensures` clause of all procedure specifications so that each clause refers to the concrete data structure from the implementation module of M instead of the abstract set specified in the specification module of M.

7. **Representation Invariant Distribution:** Add the representation invariant of M to the following program points:
   (a) `requires` and `ensures` clause of each procedure declared in M;
   (b) each module-reentrant call to a procedure declared outside M.

8. **Ensuring the Simulation Relation:** Using the analysis plugin specified in the abstraction module for M, verify that the implementation of the body of each procedure P declared in M conforms to the effective `requires`/`ensures` clause pair for P, as computed in the previous steps.

Ensuring Simulation Relation Using Plugins The responsibility of each analysis plugin is to establish that the specified abstraction function is a simulation relation between the implementation and specification modules. More specifically, when analyzing module M, an analysis plugin ensures the existence of a simulation relation r between the abstract state, containing only abstract sets, and the concrete state, where the sets declared in M are replaced by the concrete data structures from the implementation of M, and the remaining sets remain abstract. The relation r is the result of extending the abstraction function from the abstraction module of M onto the entire state; r acts as the identity function on all remaining sets. Suppose that we have verified such a simulation relation for every module. Because different modules implement data structures using disjoint fields, the composition of these relations is a simulation relation between the abstract state containing only sets and the concrete state where *all* sets are implemented using corresponding data structures.

Below, we state a condition which is sufficient to guarantee the existence of the simulation relation between the abstract state and the concrete data structure. Our condition is formulated purely in terms of the operational semantics for the implementation language and abstraction functions, obviating the need to specify an operational semantics whose states mix abstract sets and concrete data structures.

A *procedure execution fragment* is a sequence of steps in the operational semantics corresponding to the execution of a procedure; such a trace starts with the invocation and ends with the corresponding return from the procedure.

Definition 1 *Let $S_1, \ldots, S_n$ denote all sets in the program, and let $\alpha_1, \ldots, \alpha_n$ be the corresponding abstraction functions. Let $\mathcal{F}$ be a procedure execution fragment for procedure p. Let s be the first state of $\mathcal{F}$, and let s' be the final state of $\mathcal{F}$. Let r be the* `requires` *clause, and let e be the* `ensures` *clause, for p. Then, $\mathcal{F}$* conforms to its specification *iff*

$$[\![r]\!][S_1 \mapsto \alpha_1(s), \ldots, S_n \mapsto \alpha_n(s)] \;\Rightarrow$$
$$[\![e]\!][S_1 \mapsto \alpha_1(s), \ldots, S_n \mapsto \alpha_n(s), S_1' \mapsto \alpha_1(s'), \ldots, S_n' \mapsto \alpha_n(s')]$$

where $[\![\varphi]\!][x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ denotes the interpretation of formula φ where variable x_i is assigned the value v_i for $1 \leq i \leq n$. (Recall that the primed set variables S_i' in the `ensures` *clause denote the values of sets in the final state, whereas unprimed variables S_i denote the values of sets in the initial state.)*

If $\mathcal{F}$ is a procedure execution fragment for procedure p, then an *immediate subfragment* of $\mathcal{F}$ is a maximal procedure execution fragment that is a strict subfragment of $\mathcal{F}$.

Definition 2 *Let M be a module containing m sets $S_1, \ldots, S_m$ with abstraction functions $\alpha_1, \ldots, \alpha_m$. Let p be a procedure in module M. We say that procedure p* conforms to its specification, *if for all programs (contexts) containing procedure p and containing some additional sets $S_{m+1}, \ldots, S_n$ with some abstraction functions $\alpha_{m+1}, \ldots, \alpha_n$, for every procedure execution fragment of p, if all immediate subfragments of p conform to their specifications, then p conforms to its specification.*

If all procedures in a program conform to their specification, then by induction on the stack depth of execution fragments, we can show that procedure fragments of all procedure execution fragments of the program conform to their specifications. (The basis of the induction is the set of leaf procedure execution fragments containing no subfragments.)

We say that a plugin is *sound* iff whenever plugin succeeds in verifying a procedure p, then p conforms to its specification. If all modules successfully verify using the corresponding plugins, and all plugins satisfy the plugin soundness condition, then every procedure fragment conforms to its specification. In other words, the abstraction functions induce a simulation relation between the abstract states containing only sets and concrete states containing only concrete data structures.

3.1 Flag Typestate Plugin

The flag typestate plugin is designed to verify modules that use integer flags to indicate the typestate of objects containing the flag. Implementations of such modules use a different integer value for each typestate. There is an abstract set in the specification module for each typestate, and the abstraction module defines the abstraction function by specifying the correspondence between flag values and abstract sets. The developer may also specify representation invariants constraining the possible values of flags. See our technical report [16] for a discussion of the specification and analysis of modules that use the flag typestate plugin. This plugin subsumes the functionality of traditional typestate systems and goes further in allowing full boolean algebra of sets as the specification language [16].

3.2 Graph Types Plugin

The purpose of the graph types plugin is to verify properties of objects participating in recursive tree-like data structures called graph types [14]. A graph type is a dynamically allocated data structure with a distinguished set of *data fields* whose values form a *backbone* of the graph type. The backbone is a spanning tree of the data structure. In addition to data fields, a graph type may contain *routing fields* that do not belong to the spanning tree and are functionally determined by the backbone. See our technical report [16] for a discussion of the specification and analysis of modules that use the graph types plugin.

4 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 typestate systems and sharing analyses.

4.1 Typestate Systems

Typestate systems 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 [21, 6]. Generalizations support monotonic typestate changes (which ensure that the new typestate remains compatible with all existing aliases) [10] 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 [9]. It is also possible to support object-oriented constructs such as inheritance [7]. 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 [15].

Our approach generalizes existing typestate systems in several ways. It supports hierarchical typestate classification and composite typestates built out of typestate aspects from multiple modules. As our examples illustrate, composite typestates support modular software (because there is no need for one module to be aware of typestate aspects associated with other modules unless the modules deal with interdependent pieces of state); hierarchical typestates increase the expressive power of the typestate system. Our system also supports nonmonotonic typestate changes and captures the sharing patterns in the program.

4.2 Sharing Analyses

Analyzing and verifying sharing properties is a central problem in program analysis and verification. The program analysis community has focused on pointer analysis [22, 8, 1, 20] and shape analysis [4, 12, 19] as a technique for analysing the potential sharing patterns in linked data structures. The program verification community has explored a variety of logics and reasoning mechanisms for this same class of structures [18]. The focus of these research directions is on analyzing detailed local properties of individual data structures.

Our focus, on the other hand, is on characterizing global sharing properties that may involve multiple data structures in a large application. Clearly these global properties depend on the detailed local properties of individual data structures. For scalability reasons, however, we believe that any successful technique for analyzing global sharing properties must hide the complexity of these local data structure properties behind a suitable abstraction boundary. We have chosen an abstraction boundary based on membership in typestate sets. We find this approach suitable because it is a natural abstraction that captures those aspects of data structure membership that are relevant for understanding the global sharing patterns while successfully hiding the data structure complexity that makes analyzing local sharing properties such a challenging problem.

We do not see our approach as competing with existing approaches for analyzing linked data structures. Instead, we see our approach as building on the foundation that these existing approaches provide, with these approaches applied locally to produce set program translations of modules that encapsulate linked data structures. Our typestate checker can then analyze these set program translations to characterize global sharing patterns. One important aspect of our framework is its support for arbitrary analyses — given the potential complexity of the data structures that programmers may use, we believe that any one fixed technique will fail to successfully analyze some of the data structures that will appear in practice.

4.3 Program Checking Tools

ESC/Java [11] 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 [3]. ESC/Java sacrifices soundness in that it does not model all details of the program heap, but can detect some common programming errors. The LOOP project [13] offers stronger guarantees with less automation. Our framework of modular pluggable analyses [16] enables the use of a mix of powerful data flow analysis techniques that model the program heap in a sound way while offering a high degree of automation through the ability to synthesize loop invariants.

5 Conclusion

Typestate systems are designed to enforce safety conditions that involve objects whose state may change during the course of the computation. In particular, typestate systems ensure that operations are only invoked on objects that are in appropriate states.

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 abstract data types, composite typestates in which objects are members of multiple typestate sets, and hierarchical typestates. These generalizations improve the expressive power of the typestate system and increase the range of properties that it is possible to capture in this system.

Our generalization also enables the typestate system to capture sharing properties — if an object participates in multiple data structures, its typestate will indicate that it is a member of at least one typestate set for each data structure in which it participates. Our typestate system therefore effectively supports program understanding and software engineering tasks such as understanding the global sharing patterns in large programs, verifying the absence of undesirable interactions, and understanding the sequences of actions that the program may generate.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994.

[2] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *Essays Dedicated to Neil D. Jones*, volume 2566 of *LNCS*, 2002.

[3] 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.

[4] D. Chase, M. Wegman, and F. Zadek. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, pages 296–310, White Plains, NY, June 1990. ACM, New York.

[5] 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.

[6] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation*, Snowbird, UT, June 2001.

[7] R. DeLine and M. Fähndrich. Typestates for objects. In *18th ECOOP*, 2004.

[8] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*, pages 242–256, Orlando, FL, June 1994. ACM, New York.

[9] M. Fahndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the SIGPLAN '02 Conference on Program Language Design and Implementation*, Berlin, Germany, June 2002.

[10] M. Fahndrich and R. Leino. Heap monotonic typestates. In *Proceedings of the first international workshop on alias confinement and ownership (IWACO 03)*, Darmstadt, Germany, July 2003.

[11] 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.

[12] R. Ghiya and L. Hendren. Is it a tree, a DAG or a cyclic graph? a shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, pages 1–15, Jan. 1996.

[13] B. P. F. Jacobs and E. Poll. Java program verification at nijmegen: Developments and perspective. Technical Report NIII-R0318, Nijmegen Institute of Computing and Information Sciences, September 2003.

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

[15] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proceedings of the 29th Annual ACM Symposium on the Principles of Programming Languages*, Portland, OR, Jan. 2002.

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

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

[18] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01*, Paris, France, 2001.

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

[20] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, Jan. 1996.

[21] R. Strom and S. Yemini. Typestate: A programming language concept for enh ancing software reliability. *IEEE Transactions on Software Engineering*, 12(1), Jan. 1986.

[22] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995. ACM, New York.