On Modular Pluggable Analyses Using Set Interfaces

Patrick Lam, Viktor Kuncak, and Martin Rinard
Technical Report 933
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
{plam,vkuncak,rinard}@csail.mit.edu

Abstract. We present a technique that enables the focused application of multiple analyses to different modules in the same program. Our research has two goals: 1) to address the scalability limitations of precise analyses by focusing the analysis on only those parts of the program that are relevant to the properties that the analysis is designed to verify, and 2) to enable the application of specialized analyses that verify properties of specific classes of data structures to programs that simultaneously manipulate several different kinds of data structures.

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.

We characterize the key soundness property that an analysis plugin must satisfy to successfully participate in our system and present several analysis plugins that satisfy this property: a flag plugin that analyzes modules in which abstract set membership is determined by a flag field in each object, and a graph types plugin that analyzes modules in which abstract set membership is determined by reachability properties of objects stored in tree-like data structures.

Keywords: Program Analysis, Program Verification, Shape Analysis, Typestate, Formal Methods, Programming Language Design

1 Introduction

Data structure consistency is important for the successful execution of programs — if an error corrupts a program's data structures, the program can quickly exhibit unacceptable behavior or even crash. Motivated in part by the importance

VK0116, Produced March 2, 2004, 10:31pm.

of this problem, researchers have developed many algorithms for verifying that programs preserve important consistency properties [5, 7, 9, 19, 24, 50, 55].

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

This paper presents a new perspective on the data structure consistency problem: instead of attempting to develop a new algorithm that can analyze some set of consistency properties, we instead propose a technique that developers can use to apply multiple pluggable analyses to the same program, with each analysis applied to the 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 be-

tween 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, then 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.

1.1 Contributions

This paper makes the following contributions:

– **Pluggable Analyzes:** It shows how to apply 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.
– **Set Abstraction:** It introduces the use of abstract sets as the key abstraction that each analysis uses to characterize how objects participate in encapsulated data structures. The connection between sets and concrete data structure consistency properties enables modules to express the data structure participation requirements that externally accessible objects must satisfy without exposing the data structure representation to their clients. The set abstraction also enables different analyses to interoperate to verify properties that span multiple data structures and modules.
– **Analysis Scopes:** It shows how to use analysis scopes to identify the regions of the program that the analysis must process to verify invariants involving multiple modules.
– **Analysis Plugins:** It presents two plugins: a flag typestate plugin designed to analyze modules in which set membership is determined by the value of a flag field in each object and a tree reachability plugin designed to analyze modules in which set membership is determined by reachability properties in tree-like data structures.

1.2 Structure

The remainder of this paper is structured as follows. Section 2 presents an example that introduces our technique for writing modular programs and checking them using analysis plugins. Section 3 presents the syntax and semantics of our implementation language. Section 4 presents our module specification language. Section 5 presents scopes, which are used to express and verify inter-module consistency properties. Section 6 describes our analysis technique in detail, defines the responsibilities of analysis plugins, and presents the flag and graph types plugins. Section 7 presents the related work, and Section 8 concludes.

2 Example

We next present a process scheduler example. The scheduler maintains a list of running processes and a priority queue of suspended processes. The `wakeUpFirst` procedure selects a process from the priority queue and moves it to the running list; the `suspend` procedure removes the process from the running list and inserts it back into the queue. There are three modules: the running list module, the priority queue module, and the scheduler module, which invokes procedures in the running list and priority queue 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 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 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.

Our technique allows the application of 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 [50] based on previous work on graph types [18, 31, 35]) 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 graph types 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`.

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

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 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`.

```
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) then {
       if (p.next == root)
           then { p.next := null; p.prev := null; root := 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;
     } } }
```

Fig. 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;  }
```

Fig. 2. Running List Specification Module

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

Fig. 3. Running List Abstraction Function Module

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 graph types 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. Section 3 discusses the common implementation language; Section 4 discusses the common specification language. But each abstraction module is written in a language appropriate for its corresponding 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 `RunningList` module, the correspondence between implementation

```
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 { ... }
}
```

Fig. 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;
}
```

Fig. 5. Priority Queue Specification Module

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

Fig. 6. Priority Queue Abstraction Module

and specification is verified using the graph types 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 [35].

2.3 Scheduler Module

Figure 7 presents the `Scheduler` implementation module. This 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 `Running` set of running processes and the `Suspended` set 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.

```
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); }
}
```

Fig. 7. Scheduler Implementation Module

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

   proc suspend(p : Process)
     requires p in Running
     modifies Running, Suspended
     calls RunningList.remove, SuspendedQueue.add
     ensures Suspended' = Suspended + p and
             Running' = Running - p;

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

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

Fig. 8. Scheduler Specification Module

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

Fig. 9. Scheduler Abstraction Module

The abstraction module in Figure 9 uses the `status` flag to define the `Running` and `Suspended` sets. The flags plugin described in Section 6.3 can use this abstraction function to verify that the scheduler implementation correctly implements its specification.

When verifying 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 in modules outside the `Scheduler` module. It turns out that the relevant modules, `RunningList` and `SuspendedQueue`, are analyzed using an entirely different plugin, namely the tree reachability 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.

2.4 Scope Invariants

The process scheduler should satisfy several properties that involve data structures in multiple 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 module — such uncoordinated calls could cause the scheduler data structures to fall out of synch with each other, violating 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. The purpose of scopes is to specify invariants that involve multiple modules, specify a policy on when the invariants should hold, and control access to module procedures from outside the scope.

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

Fig. 10. Scope Declarations

Our example scope contains contains an invariant with three clauses that together express the set equality and disjointness properties discussed above. It also identifies a list of modules within which the invariant may be violated (these include the `Scheduler`, `RunningList`, and `SuspendedQueue` modules). Finally, it exports the `Scheduler` module, indicating that the `RunningList` and `SuspendedQueue` modules can be called only from within the other modules in the scope. `Scheduler` procedures, on the other hand, can be invoked from outside the scope. The flag 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.

Note that the flag analysis uses the specifications of the `SuspendedQueue` and `RunningList` modules (which are expressed in terms of abstract sets) to verify the invariant. Note also that these specifications are verified using an independent shape analysis based on graph types that is capable of analyzing recursive linked data structures. 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.

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. The developer could enforce this constraint by defining the abstract set of objects that satisfy the property, the abstract set of objects in the data structure, and a scope invariant that requires the first set to be a superset of the second. This invariant would enable analyses to recognize that all objects fetched from the data structure satisfy the property.

Consider a program that first modifies the fields of an object in a way that may violate the property, then removes the object from the data structure. In period between the modification and the removal, the analysis tracks the violation of the invariant and hence is able to recognize that objects fetched from the data structure during this period may not satisfy the property. 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 the invariant.

3 Implementation Language

The implementation language is a simple imperative language with procedures, object references, and dynamic object allocation. All modules are implemented in this implementation language. With the exception of formats, which enable the distributed declaration of encapsulated object fields, the language is fairly standard.

3.1 Implementation Language Syntax

Figure 11 presents the syntax for the implementation language. Each implementation module may contain format declarations, module variables, and procedures. Each format declaration describes the fields that the module contributes to objects of the specified type [12, 42]. Each module variable contains a reference to an object; references serve as persistent roots of data structures. Each procedure contains a sequence of imperative statements that manipulate references and objects. Note that the language has a built-in extension point: the A production allows assertions, which analysis plugins may use (after they verify that the assertion holds) to help establish the connection with the corresponding specification module.[2]

The type checker ensures that any well-typed program accesses only fields that exist and are visible to the executing module; it also ensures the correctness of `calls` clauses from specifications. It checks three properties: that the types of the actual and formal parameters match at call sites; that each field access refers to a field declared in a format declaration from the enclosing module; and that the `calls` clause in the specification of a procedure accurately describes its implementation.

$$
\begin{array}{rl}
M ::= & \mathsf{impl\ module}\ m\ \{F^* R^* P^*\} \\
F ::= & \mathsf{format}\ t\ \{Fd^*\} \\
Fd ::= & f^* : t; \\
R ::= & \mathsf{reference}\ v : t; \\
P ::= & \mathsf{proc}\ pn(p_1 : t_1, \ldots, p_n : t_n)[\mathsf{returns}\ r : t]\ \{\ St^*\ \} \\
St ::= & \{St\} \mid Ld^* \mid E_l{:=}E; \mid [m.]\ pn(E) \mid \mathsf{return}\ r \mid \\
 & \mathsf{if}\ (B)\ \mathsf{then}\ St_1\ \mathsf{else}\ St_2 \mid \text{while } B \text{ do } St \mid \mathsf{assert}\ A \\
Ld ::= & t\ l^*; \\
E_l ::= & l \mid l.f \mid v \mid p \mid r \\
E ::= & E_l \mid \mathsf{new}\ t \mid \mathsf{null} \mid [m.]pn(E) \\
B ::= & E{=}E \mid E\, \texttt{!=} E \\
A\ - & \text{analysis plugin-specific syntax for assertions}
\end{array}
$$

Fig. 11. Implementation Language Grammar

Implementation Language Operational Semantics The operational semantics of the implementation language (see the Appendix) tracks a program state consisting of the program counter, stack, and a garbage-collected heap. The heap contains values for references, fields and local variables. The semantics assumes that structured code has been converted to a control-flow graph and that nested expressions have been normalized into three-address code.

4 Specification Language

Figure 12 presents the syntax for the module specification language. Abstract set declarations identify the module's abstract sets. Procedure specifications use

[2] The implementation language compiler ignores assertions — they are used only during the analysis and, unlike assertions in many other languages, do not dynamically check conditions that the developer expects to be true.

these sets to identify the effects of each procedure in the module. The `requires` and `ensures` clauses in the procedure specifications use arbitrary boolean clauses B over abstract sets to specify these effects. The `calls` clause specifies which procedures may be called by a procedure; this information is checked by the implementation language typechecker, and allows the construction of call graphs using only module specifications. The `modifies` clause bounds the collection of sets directly modified by a procedure; note that `modifies` clauses need not declare sets modified by the transitive callees of a procedure.

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

Fig. 12. Grammar for Module Specification Language

The expressive power of boolean clauses B is the first-order theory of boolean algebras, which is decidable [36, 45]. As described in Section 6.3, the dataflow analysis for the typestate flag plugin uses this fact to synthesize loop invariants, to implement the transfer function for each statement, and to check implication when verifying ensures clauses. We expect other plugins to use this decidability property in similar ways.

5 Specifying Inter-Module Invariants Using Scopes

In our system, *scopes* serve two purposes: they enable the specification and verification of cross-module invariants by identifying the subset of a program in which an invariant is expected to hold, and they combat specification aggregation by hiding irrelevant sets from callers. Scopes are key to the verification of invariants containing sets from different modules: by designating certain modules as public access points, we ensure that scope invariants always hold outside their declaring scope. (For verification of invariants associated with objects see [6].) Scopes also shield callers from irrelevant detail: only sets from exported modules are visible to modules in different scopes. This serves to bound the detail required in procedure specifications: the specification of procedure p belonging to scope $\mathcal{C}$ need only contain the effects of procedures on sets in $\mathcal{C}$ and exported sets outside $\mathcal{C}$.

5.1 Definition of Scopes

Figure 14 presents the syntax of scope declarations. A scope declaration contains a set of module names, some of which are exported, and a set of scope invariants. Note that a module may be a member of multiple scopes; this allows modules to be grouped along orthogonal axes of their functionality.

How Scopes Restrict Method Calls Recall that each scope declares some of its modules to be exported. When a module M is exported in a scope C, then procedures of M may be invoked from modules outside of C. Other modules M' of C are private members of that scope, and their procedures cannot be called from outside scope C. We now formally explain how scopes restrict method calls; the restrictions induced by scopes allow our technique to guarantee the soundness of invariants and of set hiding.

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

Inter-Scope Call Condition: If module M' calls module M, then

$$\mathsf{scopes}(M) \setminus \mathsf{scopes}(M') \subseteq \mathsf{exportingScopes}(M)$$

or, equivalently, $\mathsf{yard}(M) \subseteq \mathsf{scopes}(M')$, as stated above.

Hence, to introduce a new module M' such that M' is allowed to call M, we must add M' to all scopes in $\mathsf{yard}(M)$. Note also that introducing new scopes into program only makes the condition $\mathsf{yard}(M) \subseteq \mathsf{scopes}(M')$ stronger, never weaker.

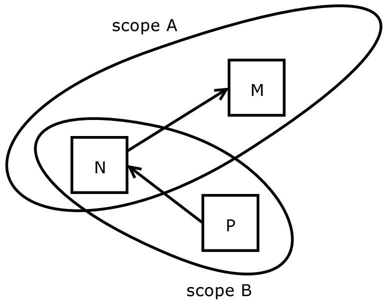
Figure 13 illustrates two scopes and two permissible inter-scope calls. In this example, we declare scopes `A` and `B`; scope `A` contains modules `M` and `N`, while scope `B` contains `N` and `P`. Module `N` is exported from `A`, while `P` is exported from `B`. The illustrated call chain from `P` through `N` to `M` is legal: `P` may call `N` because the scope difference between `P` and `N` contains the scope `A`, and `N` is exported from `A`. On the other hand, `N` may call `M` because their scope difference is $\emptyset$. The table in Figure 13 exhaustively lists all possible inter-scope calls for our example.

5.2 Uses of Scopes

Scopes allow our technique both to check invariants between sets and to combat specification aggregation.

```
scope A
  { modules M, N; exports N; }
scope B
  { modules N, P; exports P; }

spec module P
  { proc f() calls N.g; }
spec module N
  { proc g() calls M.q; }
spec module M
  { proc q(); }
```



outside → P ✓		M → P ✓	N → P ✓
outside → N ×	P → N ✓	M → N ×	
outside → M ×	P → M ×		N → M ✓

Fig. 13. Example of permissible inter-scope method calls

Scope Invariants Scope invariants are boolean clauses expressing relationships between sets in different modules. Scope invariants must be true in the initial state of the program, before any code is executed. If scope C declares scope invariant I, the basic idea is that I is assumed to hold whenever control is transferred into C, and therefore must be proved when exiting C; Section 6.1 provides full details of our policy on invariants.

$$S ::= \textsf{scope}\ s\ \{\textsf{modules}\ M^*;\ \textsf{exports}\ M^*;\ \textsf{invariant}\ B;\ \}$$

Fig. 14. Grammar for Scope Declarations

On Specification Aggregation Boolean clauses in a specification of module M refer to sets from modules in $\textsf{scopes}(M)$ plus sets from exported modules. Scopes enhance modularity by allowing developers to separate different parts of programs and ensuring that these parts only communicate by invoking exported modules, which (visibly) only manipulates sets belonging to these exported modules. In particular, one consequence of using scopes is that module M need not report the effects of a transitive callee M', if M' is unexported and $\textsf{scopes}(M) \cap \textsf{scopes}(M') = \emptyset$. 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.

6 Modular Analysis using 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 analyze a module M, the system uses the following information:

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

Plugins and Abstraction Modules The analysis of M is performed primarily by the analysis plugin specified in the abstraction module for M. Figure 15 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 of Section 4 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.

Specification Module Robustness Conditions We require two conditions on abstraction function modules. These conditions ensure that the `new` statement never changes the values of sets in specification modules. More specifically, consider module M_1 containing the statement `s: x = new t`, and set S declared in some specification module $M_2 \neq M_1$. Then, for the representation invariant I_R in M_2, the execution of statement `s` does not change the validity of I_R. Furthermore, let S' represent the contents of set S after the execution of statement `s`. Then $S' = S$.

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

Fig. 15. Syntax of Abstraction Modules

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

1. **Checking Inter-Scope Call Permissions:** For each procedure invocation $M_1.p$ in the implementation of M, ensure that M_1 is exported in every scope in $\mathsf{scopes}(M_1) \setminus \mathsf{scopes}(M)$.
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 the call graph constructed from the program's set specifications, mark module-reentrant and scope-reentrant call sites. 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.

We next describe each of these steps in greater detail.

6.1 Invariant Distribution and Mapping

The first step in analysing a module M consists of preprocessing the invariants relevant to M and mapping the `requires`/`ensures` clauses into a form suitable for analysis by the appropriate analysis plugin specified in the abstraction module for M.

Invariants express key intramodule and intermodule properties of programs. We adopt the policy whereby an invariant may be temporarily violated in some program region as long as it is reestablished upon exit from that region. *Invariant distribution* implements this policy by adding appropriate checks for invariants upon exit from the region, and by allowing the the analysis to rely on the invariants when they are guaranteed to hold. To specify precisely when an invariant holds, we introduce the notion of being in control: a module M is *in control* if the activation record of some procedure p declared in M is on the top of the call stack; a scope C is in control if some module declared in C is in control.

A scope invariant for scope C holds whenever C is not in control. Similarly, a representation invariant for module M holds whenever M is not in control.

Scope Invariant Distribution When analyzing the body of procedure p in an exported module of C, the analysis distributes scope invariant I_S to p by conjoining I_S to the requires and ensures clauses of p:

$$\mathsf{ensures}_{SI}(p) = \mathsf{ensures}(p) \wedge I_S$$
$$\mathsf{requires}_{SI}(p) = \mathsf{requires}(p) \wedge I_S$$

Furthermore, invariant distribution inserts a statement $\mathsf{assert}\ I_S$ before each procedure call inside C which invokes a method outside C, forcing the analysis plugin to check I_S before the control leaves C. These assertions ensure that I_S holds, even in the presence of reentrant calls back into C.

Scope invariants must be explicitly checked prior to scope-reentrant calls. Scope invariant distribution therefore adds $\mathsf{assert}\ I_S$ before any scope-reentrant call. After returning from a call that reenters a scope C, analysis plugins are required to "flush stale state" by erasing all information about the sets declared in modules not exported in C.

The conditions inserted by invariant distribution allow the analysis to preserve the policy that either I_S is true, or C is in control. A consequence of our policy is that I_S holds at any program point outside C. In particular, I_S holds at any call site T to procedure p originating outside C, and thus need not be checked at T. This policy generalizes standard techniques for information hiding and invariant enforcement.

Module Projection Module projection is used to map `ensures` clauses at callsites which invoke procedures in different scopes and use sets which are not in scope at the caller. Consider the case where module M executes an interscope call to M'; let $\mathsf{yard}(M') \subseteq \mathsf{scopes}(M)$ and $\mathsf{scopes}(M') \not\subseteq \mathsf{scopes}(M)$. Then M may call M', but private sets—sets belonging to modules in $\mathsf{scopes}(M') \setminus \mathsf{scopes}(M)$—are not of interest to the client module M. We show how to project the definition of module M' onto sets declared in modules $\{M_1 \mid \mathsf{scopes}(M) \cap \mathsf{scopes}(M') \cap \mathsf{scopes}(M_1) \neq \emptyset\}$ that are relevant to both M and M'.

The projection technique applies to any program whose state can be described using a finite set of variables. Let $x_1, \ldots, x_n$ (or $\overrightarrow{x}$ for short) denote variables for which information should be preserved, and let $y_1, \ldots, y_m$ (or $\overrightarrow{y}$ for short) denote the variables for which information should be ignored. We then apply the transformation:

$$\begin{array}{ll} \mathsf{requires}\ r(\overrightarrow{x}, \overrightarrow{y}) & \\ \mathsf{ensures}\ e(\overrightarrow{x}, \overrightarrow{y}, \overrightarrow{x}', \overrightarrow{y}') & \end{array} \Longrightarrow \begin{array}{l} \mathsf{requires}\ \forall \overrightarrow{y}.r(\overrightarrow{x}, \overrightarrow{y}) \\ \mathsf{ensures}\ \exists \overrightarrow{y}.e(\overrightarrow{x}, \overrightarrow{y}, \overrightarrow{x}', \overrightarrow{y}') \end{array}$$

The transformation for the `requires`/`ensures` clause is justified by the relational semantics of `requires`/`ensures` clauses [28] of the form

$$\begin{array}{l} \mathsf{requires}\ r(\overrightarrow{x}, \overrightarrow{y})\ \mathsf{ensures}\ e(\overrightarrow{x}, \overrightarrow{y}, \overrightarrow{x}', \overrightarrow{y}') \equiv \\ \quad (r(\overrightarrow{x}, \overrightarrow{y}) \land \mathsf{ok}) \Rightarrow (e(\overrightarrow{x}, \overrightarrow{y}, \overrightarrow{x}', \overrightarrow{y}') \land \mathsf{ok}') \end{array}$$

with projection corresponding to existential quantification over $\overrightarrow{y}, \overrightarrow{y}'$.

Requires/Ensures Clause Mapping Let $\mathsf{sets}(M) = \{S_1, \ldots, S_n\}$ be the sets declared in the specification of module M, and let $S_i = \{x : t_i \mid F_i(x)\}$ be the definition of set S_i for $1 \leq i \leq n$. The `requires`/`ensures` clause mapping translates `requires` and `ensures` clauses, as well as assertions, into a form that exposes the definitions of sets $S_1, \ldots, S_n$ using formulas $F_1, \ldots, F_n$. (In addititio to $S_1, \ldots, S_n$, these specifications use abstract sets declared in other specification modules; these abstract sets are not affected by the mapping.)

First, consider the mapping of the assertion $\mathsf{assert}\ B(S_1, \ldots, S_n)$ where $B(S_1, \ldots, S_n)$ is a formula in the language of boolean algebras described in Figure 12. The mapping of this assertion is the formula:

$$\begin{array}{l} \mathsf{assert}\ (\ \mathsf{let}\ \bar{S}_1 = \{x : t_1 \mid F_1(x)\}\ \mathsf{in} \ldots \mathsf{let}\ \bar{S}_n = \{x : t_n \mid F_n(x)\}\ \mathsf{in} \\ \qquad\qquad B(\bar{S}_1, \ldots, \bar{S}_n)\ ) \end{array}$$

We next describe the mapping of `requires`/`ensures` clauses. (Recall that all `modifies` clauses have been translated into augmented `ensures` clauses at this point.) The mapping of the `requires`/`ensures` clause

$$\begin{array}{l} \mathsf{requires}\ B(S_1, \ldots, S_n) \\ \mathsf{ensures}\ B'(S_1, \ldots, S_n, S_1', \ldots, S_n') \end{array}$$

is a `requires`/`ensures` clause with let-bindings:

$$\begin{array}{l} \mathsf{let}\ \bar{S}_1 = \{x : t_1 \mid F_1(x)\}\ \mathsf{in} \ldots \mathsf{let}\ \bar{S}_n = \{x : t_n \mid F_n(x)\}\ \mathsf{in} \\ \quad \mathsf{requires}\ B(\bar{S}_1, \ldots, \bar{S}_n) \\ \quad \mathsf{ensures}\ (\ \mathsf{let}\ \bar{S}_1' = \{x : t_1 \mid F_1'(x)\}\ \mathsf{in} \ldots \mathsf{let}\ \bar{S}_n' = \{x : t_n \mid F_n'(x)\}\ \mathsf{in} \\ \qquad\qquad B'(\bar{S}_1, \ldots, \bar{S}_n, \bar{S}_1', \ldots, \bar{S}_n')\ ) \end{array}$$

Here $F_i'(x)$ denotes the formula $F_i(x)$ where the occurence of each field f is replaced with the expression f' referring to the value of the field in the state after execution of the procedure. Note that the sets $\bar{S}_i$ are evaluated in the initial (precondition) state but are also used in the postcondition state. Moreover, the only way in which the postcondition formula refers to the initial state is through the values of sets evaluated in the initial state; there is no direct comparison of values of fields in pre and post state, which makes the verification of the `ensures` clauses easier.

Representation Invariant Distribution Representation invariants are similar to scope invariants: they ensure that, when a module is not in control, its representation invariant holds. Representation invariants are expressed in the annotation language of the appropriate analysis plugin. Sets have unspecified values when the representation invariant does not hold. However, using information about the program's call graph, our technique ensures that modules never rely on values of sets when the relevant representation invariants are violated.

When analyzing the procedures of module M, our analysis conjoins representation invariants I_R to the mapped requires and ensures clauses:

$$\mathsf{ensures}_R(p) = \mathsf{map}(\mathsf{ensures}_{SI}(p)) \wedge I_R$$
$$\mathsf{requires}_R(p) = \mathsf{map}(\mathsf{requires}_{SI}(p)) \wedge I_R$$

In the presence of a module-reentrant callsite c, representation invariant distribution adds the condition $\mathsf{assert}\ I_R$ just before c. After returning from c, analysis plugins are required to flush stale state by removing any information about private data structures of M from the post-state of c.

As was the case with scope invariants, representation invariant distribution guarantees that module invariants always hold unless module M is in control. This ensures that the abstract sets in the specification modules accurately summarize the state of the heap at important points in the program.

6.2 Ensuring Simulation Relation Using Plugins

An analysis plugin verifies the correctness of the implementation of module M by verifying the correctness of each procedure in M. To verify a procedure p in module M, the plugin uses: 1) the implementation of procedure p; 2) `requires` and `ensures` clauses for p (after invariant distribution and mapping of Section 6.1); 3) `requires` and `ensures` clauses for all procedures called from p; and 4) the abstraction function for module M.

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; it 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 procedure 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*

$$\llbracket r \rrbracket [S_1 \mapsto \alpha_1(s), \ldots, S_n \mapsto \alpha_n(s)] \Rightarrow \\ \llbracket e \rrbracket [S_1 \mapsto \alpha_1(s), \ldots, S_n \mapsto \alpha_n(s), S'_1 \mapsto \alpha_1(s'), \ldots, S'_n \mapsto \alpha_n(s')]$$

where $\llbracket \varphi \rrbracket [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ denotes the interpretation of formula φ where variable x_i is assigned the value v_i for $1 \leq i \leq n$.

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 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.

6.3 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.

Flag Typestate Plugin Annotation Language Equation (1) presents the syntax for the flag typestate's plugin property language, which instantiates the generic syntax of Figure 15.

$$F_p ::= x.f{=}c \tag{1}$$

The abstraction function of the flag abstraction module defines set membership based on integer field values. Flag abstraction modules may also contain representation invariants.

To illustrate a concrete example, consider the `requires` clause for the `suspend` procedure from our process scheduler example of Section 2:

```
requires p in Running
```

The `Scheduler` module belongs to the `ProcessScheduler` scope, which declares the following invariant:

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

Invariant distribution gives the following augmented `requires` clause:

$$\begin{aligned}&|p| <= 1 \ \wedge \ p \subseteq \mathtt{Running} \ \wedge \ \mathtt{Running} \cap \mathtt{Suspended}{=}\emptyset\\ &\ \wedge \ \mathtt{Running} = \mathtt{RunningList.InList}\\ &\ \wedge \ \mathtt{Suspended} = \mathtt{SuspendedQueue.InQueue}\end{aligned}$$

Our mapping uses the declaration of the `Running` and `Suspended` sets,

```
    Running   = {x : Process | x.status=1};
    Suspended = {x : Process | x.status=0};
```

to translate the augmented `requires` clause into:

$$\begin{aligned}&\mathsf{let}\ \mathtt{Running\ =}\ \{\mathtt{x\ :\ Process\ |\ x.status=1}\}\ \mathsf{in}\\ &\mathsf{let}\ \mathtt{Suspended\ =}\ \{\mathtt{x\ :\ Process\ |\ x.status=0}\}\ \mathsf{in}\\ &\quad |p| <= 1 \ \wedge \ p \subseteq \mathtt{Running} \ \wedge \ \mathtt{Running} \cap \mathtt{Suspended}{=}\emptyset \ \wedge\\ &\quad \mathtt{Running} = \mathtt{RunningList.InList} \ \wedge\\ &\quad \mathtt{Suspended} = \mathtt{SuspendedQueue.InQueue}\end{aligned}$$

There are no representation invariants in this module, so the above clause is our effective requires clause.

Flag Typestate Plugin Analysis Algorithm This plugin performs dataflow analysis on the implementation language to approximate the contents of each set, as well as typestate information (in the form of set membership) for each local variable. It does this by using boolean clauses T as its dataflow fact. The dataflow fact T starts with a set of let clauses, followed by a boolean clause B. In this analysis, the let clauses are fixed throughout the analysis of a procedure; we therefore focus only on the boolean algebra clause B. Our dataflow facts conform to the following grammar (note that B is defined using a subset of the grammar from Figure 12):

$$
\begin{aligned}
T &::= B \mid \mathsf{let}\ S{=}\{x : f | F_p(x)\}\ \mathsf{in}\ T \\
B &::= E_1 = E_2 \mid E_1 \subseteq E_2 B_1 \wedge B_2 \mid B_1 \vee B_2 \mid \neg B \\
&\quad \mid\ \mathsf{card}(S){=}0 \mid \mathsf{card}(S){=}1 \mid \mathsf{card}(S){\geq}2 \\
E &::= \emptyset \mid S' \mid S \mid E_1 \cup E_2 \mid E_1 \cap E_2 \mid E_1 \setminus E_2
\end{aligned}
$$

The purpose of cardinality constraints in typestate plugin analysis is to distinguish local variables from sets; the local variables are represented as sets of cardinality 1. In the analysis, S' is used to refer to the value of set S after a statement, while S refers to the value of the set before the statement.

The initial dataflow fact at the start of a procedure is the `requires` condition for that procedure. At a merge point, our analysis combines boolean formulas using disjunction. Termination is guaranteed because our lattice is finite. Namely, there are exactly three kinds of permitted cardinality constraints, so the number of non-equivalent boolean algebra formulas is finite. Our analysis checks that the `ensures` clause holds at all exit points of the procedure; in particular, since implication is decidable, our analysis can check that for each exit point e, the computed typestate predicate B_e implies the `ensures` clause.

The transfer functions in the dataflow analysis update the boolean formulas to reflect the effect of each statement. Let B be a boolean clause denoting a relation between the state at procedure entry and the state at current program point and let B_s be the boolean clause describing the effect of statement s. The incorporation operation $B[B_s]$ changes the boolean clause B by composing it with the boolean clause B_s. We compute $B[B_s]$ by applying quantifier elimination to the formula $\exists \hat{S}_1, \ldots, \hat{S}_n.\ B[\hat{S}_i/S'_i] \wedge B_s[\hat{S}_i/S_i]$ (See [2], [40, Section 3.2] for quantifier elimination for the full first-order theory of boolean algebras, [4, 48] for fragments of boolean algebras with better complexity bounds and further references.) The result of quantifier elimination are positive boolean combinations of formulas $|S| \geq k$ and $|S| = k$. We approximate the resulting formula to eliminate set constraints with cardinality greater than 2 using the following rules:

Original Formula	Constraint	Coarsened Formula
$\mathsf{card}(S){\geq}k$	$k \geq 2$	$\mathsf{card}(S){\geq}2$
$\mathsf{card}(S){=}k$	$k \geq 2$	$\mathsf{card}(S){\geq}2$
$\mathsf{card}(S){\geq}1$		$\mathsf{card}(S){=}1 \vee \mathsf{card}(S){\geq}2$
$\mathsf{card}(S){\geq}0$		$\mathsf{card}(S){=}0 \vee \mathsf{card}(S){=}1 \vee \mathsf{card}(S){\geq}2$
otherwise		original formula

Figure 16 presents the formulas that the plugin uses to generate the transfer functions for the statements in the implementation language.

Provided that $R = \{x : \mathsf{format}(x) | x.\mathtt{status} = v\}$ is in the abstraction module:

$$
\begin{aligned}
[\![\mathtt{x} := \mathsf{null}]\!](B) &= B[\mathtt{x} = \emptyset \wedge \textstyle\bigwedge_{\mathtt{S} \neq \mathtt{x}} S' = S] \\
[\![\mathtt{x.status} := \mathtt{v}]\!](B) &= B[R = \hat{R} \cup \mathtt{x} \wedge \textstyle\bigwedge_{S \in \mathsf{alts}(\mathtt{R})} S' = S \setminus \mathtt{x} \wedge \textstyle\bigwedge_{S \notin \{R, \mathsf{alts}(\mathtt{R})\}} S' = S] \\
[\![\mathtt{if}(\mathtt{x.status} = \mathtt{v})]\!](B) &= B[\mathtt{x} \subseteq R \wedge \textstyle\bigwedge_S S' = S] \text{ (true branch)} \\
[\![\mathtt{if}(\mathtt{x.status} = \mathtt{v})]\!](B) &= B[\mathtt{x} \cap R = \emptyset \wedge \textstyle\bigwedge_S S' = S] \text{ (false branch)}
\end{aligned}
$$

where

$\mathsf{alts}(\mathtt{R}) = R'$ such that abstraction module contains $R' = \{x : \mathsf{format}(x) | x.\mathtt{status} = v'\}$

Fig. 16. Rules for typestate analysis

For a procedure call `x=proc(y)`, our transfer function checks that `proc`'s requires condition holds, and incorporates `proc`'s ensures condition:

$$
[\![\mathtt{x} = \mathtt{proc}(\mathtt{y})]\!](B) = \begin{cases} B[\mathsf{ensures}'(\mathtt{proc})] & \text{if } B \Rightarrow \mathsf{requires}'(\mathtt{proc}) \\ \mathsf{false} & \text{otherwise} \end{cases}
$$

where both $\mathsf{ensures}'$ and $\mathsf{requires}'$ substitute caller actuals for formals of `proc`. Most modules are obliged to remove internal state from the analysis representation after a module-reentrant call; however, because the flag typestate plugin has no private internal state[3], module-reentrant calls need not be treated specially. For a scope-reentrant call, this analysis plugin, like all others, projects out all unexported sets belonging to the current scope from the result of the procedure call, $B[\mathsf{ensures}'(\mathtt{proc})]$.

Example Let $B \equiv y' = y \wedge x' = x \wedge S' = S \wedge S = x$; we show how to abstractly execute the statement `y = x`. We therefore wish to incorporate $B_s \equiv y' = x' \ \wedge \ x' = x \ \wedge \ S' = S$. Substituting and applying quantifiers gives

$$
\begin{aligned}
\exists \hat{S}, \hat{x}, \hat{y}.\ &(\hat{y} = y \ \wedge \ \hat{x} = x \ \wedge \ \hat{S} = S \ \wedge \ S = x) \ \wedge \\
&(y' = x' \ \wedge \ x' = \hat{x} \ \wedge \ S' = \hat{S}),
\end{aligned}
$$

which simplifies to $S' = S \wedge x' = x \wedge y' = x$.

6.4 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 [35]. 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* [35] (corresponding to *pointer fields* of [50]) that do not belong to the spanning tree and are functionally determined by the backbone.

[3] Program state for the flag typestate plugin is represented by set membership, as expressed by boolean clauses.

Graph Types Plugin Annotation Language The basis of the graph types plugin is weak monadic second-order logic interpreted over an infinite binary tree (MSOL). This logic is decidable; the MONA tool [33] implements a decision procedure for MSOL along with important optimizations that make the tool useful in practice [34].

$$
\begin{aligned}
F_p ::= &\ \exists x \text{ of } t.\ F_p \mid \exists S \text{ of } t.\ F_p \mid \neg F_p \mid F_{p1} \wedge F_{p2} \mid \\
\mid &\ x \in SE \mid SE_1 = SE_2 \mid SE_1 \subseteq SE_2 \mid \mathsf{card}(SE){=}k \mid R
\end{aligned}
$$

$$
R ::= x \mid R.f \mid R.{\sim}f \mid R_1 + R_2 \mid R*
$$

Fig. 17. Graph Types Plugin Annotation Language

Figure 17 presents the graph types plugin property language, which corresponds to the language in [50, Section 2]. Because it is based on monadic second-order logic, the property language allows quantification not only over objects (using $\exists x$ of $t.F$), but also over sets of objects in the heap (using $\exists S$ of $t.F$). Quantification over sets of objects allows this logic to define the transitive closure of binary relations on objects, which is important for specifying reachability properties of heap objects participating in recursive data structures. The language supports regular expressions for denoting sets of objects reachable from some root object along given paths (the notation $\sim$ denotes following a field backwards).

The property language in Figure 17 is used as part of the annotation language of Figure 15. Boolean algebra expressions and let-bindings of Figure 15 add no additional expressive power to the graph types plugin property language, because let $S = \{x : t \mid F(x)\}$ in $B(S)$ translates into $\exists S$ of $t.$ $(\forall x.x \in S \iff F(x)) \wedge B(S)$. (Note that $B(S)$ is a valid formula of second-order logic, because the logic properly contains boolean algebra of sets.)

In addition to using the property language for describing invariants and abstraction functions, the graph types plugin uses the same language for constraining the values of pointer fields in data structures. Because each pointer field in a data structure is uniquely determined by a known formula in the graph types property language, the addition of pointer fields preserves the decidability of the logic. In addition to abstraction functions and representation invariants, abstraction modules for the graph types plugin may contain graph type definitions, which introduce a graph type by specifying its data fields, and specifying pointer fields along with the formula defining the pointer field [50].

Graph Types Plugin Analysis Algorithm The analysis behind the graph types plugin is essentially the analysis implemented in the PALE tool [50]. Our main observation is that this analysis naturally fits with our technique for combining analyses that communicate through set interfaces.

Note that the let-bindings for the `requires`/`ensures` pair can be translated into logical variables. Specifically, the clause

$$
\begin{array}{l}
\mathsf{let}\ \bar{S}_1 = \{x : t_1 \mid F_1(x)\}\ \mathsf{in} \ldots \mathsf{let}\ \bar{S}_n = \{x : t_n \mid F_n(x)\}\ \mathsf{in} \\
\quad \mathsf{requires}\ B(\bar{S}_1, \ldots, \bar{S}_n) \\
\quad \mathsf{ensures}\ (\ \mathsf{let}\ \bar{S}'_1 = \{x : t_1 \mid F'_1(x)\}\ \mathsf{in} \ldots \mathsf{let}\ \bar{S}'_n = \{x : t_n \mid F'_n(x)\}\ \mathsf{in} \\
\qquad\qquad B'(\bar{S}_1, \ldots, \bar{S}_n, \bar{S}'_1, \ldots, \bar{S}'_n)\ )
\end{array}
$$

can be represented in the syntax of PALE tool by adding appropriate logical variables, precondition, and postcondition to the body c of a procedure:

$$
\begin{array}{l}
\mathsf{set}\ S_1 : t_1;\ \ldots\ \mathsf{set}\ S_n : t_n \\
[P_1(S_1) \land \ldots \land P_n(S_n) \land B(S_1, \ldots, S_n)] \\
\{c\} \\
[\exists S'_1, \ldots, S'_m.\ P_1(S'_1) \land \ldots \land P_n(S'_n) \land B'(S'_1, \ldots, S'_n, S_1, \ldots, S_n)]
\end{array}
$$

where P_i for $1 \leq i \leq n$ is given by the PALE predicate definition

$$\mathsf{pred}\ P_i(S) = \forall x. x \in S \iff F_i(x)$$

and indicates that S is equal to the value of the set S_i in the current state.

```
type List = {data next:List;}

pred roots(pointer x,y:List, set R:List) =
  allpos p of List: p in R <=> x<next*>p | y<next*>p;

pred abst(pointer l:List, set R:List) =
  allpos p of List: p in R <=> l<next*> p;

proc reverse(data list:List):List
  set R:List;
[abst(list,R)]
{
  data res:List;
  pointer temp:List;
  res = null;
  while [roots(list,res,R)] (list!=null) {
    temp = list.next;
    list.next = res;
    res = list;
    list = temp;
  }
  return res;
}
[abst(return,R)]
```

Fig. 18. A list reverse program with verification conditions that ensure that the set of elements in the list remains the same.

Figures 18 and 19 show example programs in Pointer Assertion Logic Engine (PALE) [50]. The programs are modified versions of examples distributed with PALE. The predicate `abst` specifies a relation between the root of the list and a set which represents the content of the list. The precondition `[abst(list,R)]` indicates that the set R is the content of the list passed as the input. The

```
type Tree = {data left, right: Tree;}
pred inTree(pointer x:Tree) =
  root<(left+right)*>x;
pred abst(set R:Tree) =
  allpos x of Tree: x in R <=> inTree(x);

data root:Tree;

proc insert(data e:Tree):void
set R:Tree;
[abst(R) &
 e.left = null & e.right = null]
{ pointer current:Tree; bool go;
  if (root=null) {
    root = e;
  } else {
    current = root;
    go = true;
    while [(go => abst(R) & e.left=null & e.right=null &
                  inTree(current) & current != null) &
           (!go => abst(R union {e}))] (go) {
      if (?) {
        /* e < current */
        if (current.left=null) {
          current.left = e;
          go = false;
        } else {
          current = current.left;
        }
      } else {
        if (current.right=null) {
          current.right = e;
          go = false;
        } else {
          current = current.right;
        }
      }
    }
  }
  split [abst(R union {e})]
} [true]
```

Fig. 19. A tree insertion program with verification conditions that ensure that the set of elements in the tree is increased by the element inserted.

postcondition `[abst(return,R)]` indicates that the content of the list returned as a result is the same set R. Therefore, the specification of the first example indicates that the set of elements in the list is preserved. In the second example, the content of the tree is increased by the elements inserted.

6.5 Other Plugins

The set of analysis plugins suitable for our technique is not limited to the plugins we have presented so far.

The parametric shape analysis [55] can also be used as one of the possible plugins. One of the strengths of the parametric shape analysis approach is that it is applicable to general graph structures and not only tree-like structures. The underlying logic of [55] is first-order logic with transitive closure. As in the graph types plugin, abstraction functions can use regular expressions to define the contents of a set as the set of nodes stored in a data structure. [55] uses three-valued structures, but the approach can be adapted to use only two-valued logic [41, 53, 59].

Role logic [39] can also be used to express sets stored in data structures. For non-recursive data structures it suffices to use a decidable subset of role logic corresponding to two-variable logic with counting [27].

Interactive theorem proving techniques [23,30,47,49] can also be incorporated into our framework, in this case the abstraction module would also contain the proof script for proving the conditions arising from the simulation relation condition. Sounds versions of more automated techniques [16, 24] are also appropriate for our framework.

7 Related Work

We next discuss some further related work.

Formal Methods and Program Specification The Java Modelling Language [8] is a Java extension which allows developers to embed design information into Java source code, using Java-like syntax. Using JML, developers can specify method preconditions and postconditions, as well as invariants. These are expressed in quantified boolean formulas, primarily over fields. The JML is a framework which allows researchers to develop tools checking the specification tools; the `jmlc` tool, for instance, checks some of the JML assertions at runtime, while LOOP and ESC/Java can check some JML assertions statically. Note that checking any JML condition involving abstract sets appears to require the use of interactive theorem proving technology. Like JML, our technique also allows developers to tightly couple preconditions, postconditions and invariants to program source. Our focus, however, is on static checking of design information. Using our abstraction techniques, it is possible to harness, for instance, shape analysis technology to automatically verify the correctness of complicated implementations. Because we define the notions of modules and formats, our technique has stronger support for separating different parts of programs and ensuring that they do not interfere. JML also supports callable clause that corresponds to our calls clauses; JML uses the callable clause to ensure correct subclassing in the presence of dynamic dispatch [54].

Shape Analysis The goal of shape analysis is to verify that programs preserve consistency properties of (potentially recursive) linked data structures. Researchers have developed many shape analyses and the field remains one of the most active areas in program analysis today [10,11,21,25,26,38,39,41,43,50–53, 55, 58, 59]. Our goal is to enable developers of precise or specialized analyses (such as almost all shape analyses) to cooperatively apply their analysis, along with other analyses, in a modular fashion to large programs, with each analysis operating on only that part of the program relevant for the properties that it is designed to verify.

Typestate Systems Typestate systems track the conceptual states that each object goes through during its lifetime in the computation [15, 17, 20–22, 57] . In our approach, object state is determined by its membership in abstract sets. In addition to supporting traditional typestate classifications, this perspective also promotes the following generalizations of the standard typestate approach:

- **Orthogonal Composition:** In our system an object can be a member of sets from multiple modules simultaneously. Each object's conceptual state can therefore be an orthogonal composition of state aspects from each of the modules in which it participates. The advantages of this approach include better modularity (because each module deals only with those aspects of object states that are relevant for its operation) and support for polymorphism (because each module can operate successfully on multiple objects, each of which may participate in different ways in other modules).
- **Hierarchical Classifications:** Our system supports dynamic hierarchical classifications in which a collection of sets partitions a more general set, with subset inclusion capturing the hierarchy. Unlike the classification hierarchies in object-oriented languages, which are fixed over the lifetimes of objects, our classification hierarchies can change to reflect the effects of program actions that change the conceptual states of the objects in the program.
- **Data Structure Participation:** In standard typestate systems, the typestate of an object is determined by either the values of its fields or by its procedure invocation history [13, 15, 19, 57]. In our approach, set membership can also capture how the object participates in each data structure. We have shown how analyses can leverage the resulting correlation between abstract set membership and internal data structure representation properties to verify data structure representation invariants. An object's set membership properties can also characterize how it is (or is not) shared between multiple data structures, which may support more effective reasoning (both automated reasoning and reasoning by human developers) about the object's role in the computation.

Simulation Relations The foundation of our approach for specifying implementation modules using set-based specification modules is the notion of simulation relations [1, 3, 14, 32, 46]. We apply these general simulation relation techniques to the case of forward simulation relations that are partial functions. The domain of our simulation relations is the set of states whose data structures satisfy representation invariants; the result of mapping a state under the simulation relation is a state where concrete data structures are replaced by global sets. In our case, the forward simulation relation condition essentially reduces to verifying that the transition relation given by a procedure implementation is a subset of the transition relation given by the requires/ensures pair. The rules for mapping requires/ensures statements under an abstraction function are presented in e.g. [14, Chapter 7].

Decision Procedure for Boolean Algebras We use first-order logic formulas in the language of boolean algebras as the basis of our module specification language. The decidability of the satisfiability problem for the first-order theory of boolean algebras dates back to [45, 56] and is presented in [2, Chapter 4]. The complexity of this problem is alternating exponential time [37] (that is, exponential deterministic space [29, Page 36]). The complexity of the satisfiability problem for quantifier-free formulas in the language of boolean algebras is NP-complete [48]. Quantifier-free fragment of boolean algebras is too restrictive for our specifications because it cannot express constraints of the form $|x| \leq 1$ and $|x| = 1$, which arise naturally when using sets to model local variables and procedure parameters. The monadic class [4] allows quantified first-order variables corresponding to sets x with $|x| = 1$ as well and free second-order variables; the satisfiability problem for monadic class with equality is nondeterministic exponential time complete [44]. Due to absence of quantification over sets, the monadic class would not allow us to perform precise relation composition which is important for procedure calls, and precise projection of interfaces. Nevertheless, algorithms for deciding the quantifier free fragment and the monadic class can be applied in the cases when the specifications have the restricted form and may be useful for improving the performance of the general decision procedure. To our knowledge, the only implemented decision procedure that can decide the first-order theory of boolean algebras is MONA tool [34], which implements the more general decision procedure for monadic second-order logic over trees, and has non-elementary complexity in general.

8 Conclusion

The program analysis community has produced many precise analyses that are capable of extracting or verifying quite sophisticated data structure properties. Issues associated with using these analyses include scalability limitations and the diversity of important data structure properties, some of which will inevitably elude any single analysis.

This paper shows how to apply the full range of analyses to large programs composed of multiple modules. The key elements of our approach include modules that encapsulate object fields and data structure implementations, specifications based on membership in abstract sets, and invariants that use these sets to express (and enable the verification of) properties that involve multiple data structures in multiple modules analyzed by different analyses. We anticipate that our techniques will enable the productive application of precise analyses to verify important data structure consistency properties in large programs built out of multiple modules.

References

1. Martin Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.
2. W. Ackermann. *Solvable Cases of the Decision Problem*. North Holland, 1954.
3. Paul C. Attie and Nancy A. Lynch. Dynamic input/output automata: a formal model for dynamic systems. Technical report, MIT CSAIL, July 2003.
4. Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Set constraints are the monadic class. In *Logic in Computer Science*, pages 75–83, 1993.

5. Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, 2001.
6. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. In *Formal Techniques for Java-like Programs, Available as Technical Report 408, Department of Computer Science, ETH*, July 2003.
7. 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.
8. Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications. Technical Report NII-R0309, Computing Science Institute, Univ. of Nijmegen, March 2003.
9. William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice & Experience*, 30(7):775–802, 2000.
10. Cristiano Calcagno, Luca Cardelli, and Andrew D. Gordon. Deciding validity in a spatial logic for trees. In *ACM TLDI'02*, 2002.
11. David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proc. ACM PLDI*, 1990.
12. David R. Cheriton and Michael 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.
13. Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. ACM PLDI*, 2002.
14. Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-oriented proof methods and their comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.
15. Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM PLDI*, 2001.
16. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, COMPAQ Systems Research Center, 1998.
17. 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.
18. Jacob Elgaard, Anders Møller, and Michael I. Schwartzbach. Compile-time debugging of C programs working on trees. In *Proc. 9th ESOP*, 2000.
19. Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. 4th USENIX OSDI*, 2000.
20. Manuel Fahndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proc. ACM PLDI*, 2002.
21. Manuel Fähndrich and K. Rustan 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.
22. Manuel Fähndrich and K. Rustan M. Leino. Heap monotonic typestates. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2003.
23. Jean-Christophe Filliatre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, 2003.
24. Cormac Flanagan, K. Rustan M. Leino, Mark Lilibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proc. ACM PLDI*, 2002.
25. Pascal Fradet and Daniel Le Métayery. Shape types. In *Proc. 24th ACM POPL*, 1997.
26. Rakesh Ghiya and Laurie Hendren. Is it a tree, a DAG, or a cyclic graph? In *Proc. 23rd ACM POPL*, 1996.

27. Erich Grädel, Martin Otto, and Eric Rosen. Two-variable logic with counting is decidable. In *Proceedings of 12th IEEE Symposium on Logic in Computer Science LICS '97, Warschau*, 1997.
28. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, Inc., 1998.
29. Neil Immerman. *Descriptive Complexity*. Springer-Verlag, 1998.
30. Bart P. F. Jacobs and Erik Poll. Java program verification at nijmegen: Developments and perspective. Technical Report NIII-R0318, Nijmegen Institute of Computing and Information Sciences, September 2003.
31. Jacob L. Jensen, Michael E. Jørgensen, Nils Klarlund, and Michael I. Schwartzbach. Automatic verification of pointer programs using monadic second order logic. In *Proc. ACM PLDI*, Las Vegas, NV, 1997.
32. He Jifeng, C. A. R. Hoare, and Jeff W. Sanders. Data refinement refined. In *ESOP'86*, volume 213 of *LNCS*, 1986.
33. Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.
34. Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. In *Proc. 5th International Conference on Implementation and Application of Automata*. LNCS, 2000.
35. Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proc. 20th ACM POPL*, Charleston, SC, 1993.
36. Dexter Kozen. Complexity of boolean algebras. *Theoretical Computer Science*, 10:221–247, 1980.
37. Dexter Kozen. Logical aspects of set constraints. In *Proc. 1993 Conf. Computer Science Logic (CSL'93)*, volume 832 of *Lecture Notes in Computer Science*, pages 175–188, 1993.
38. Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Proc. 29th POPL*, 2002.
39. Viktor Kuncak and Martin Rinard. On role logic. Technical Report 925, MIT CSAIL, 2003.
40. Viktor Kuncak and Martin Rinard. On the theory of structural subtyping. Technical Report 879, Laboratory for Computer Science, Massachusetts Institute of Technology, 2003.
41. Viktor Kuncak and Martin Rinard. Boolean algebra of shape analysis constraints. In *5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, 2004.
42. K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proc. ACM PLDI*, 2002.
43. Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing and Analysis*, 2000.
44. Harry R. Lewis. Complexity results for classes of quantificational formulas. *J. Comput. Syst. Sci.*, 21(3):317–353, 1980.
45. L. Loewenheim. Über mögligkeiten im relativkalkül. *Math. Annalen*, 76:228–251, 1915.
46. Nancy Lynch and Frits Vaandrager. Forward and backward simulations – Part I: Untimed systems. *Information and Computation, 121(2)*, 1995.
47. C. Marché, C. Paulin-Mohring, and X. Urbain. The krakatoa tool for certification of java/javacard programs annotated in jml. *Journal of Logic and Algebraic Programming*, 2003. to appear.
48. Kim Marriott and Martin Odersky. Negative boolean constraints. Technical Report 94/203, Monash University, August 1994.
49. Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Automated Deduction — CADE-19*, LNCS. Springer-Verlag, 2003.
50. Anders Møller and Michael I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. ACM PLDI*, 2001.

51. G. Ramalingam, Alex Warshavsky, John Field, Deepak Goyal, and Mooly Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 83–94. ACM Press, 2002.
52. Thomas Reps, Mooly Sagiv, and Alexey Loginov. Finite differencing of logical formulas for static analysis. In *Proc. 12th ESOP*, 2003.
53. Thomas Reps, Mooly Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In *VMCAI'04*, 2004.
54. Clyde Ruby and Gary T. Leavens. Safely creating correct subclasses without seeing superclass code. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10), pages 208–228, October 2000.
55. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.
56. Thoralf Skolem. Untersuchungen über die Axiome des Klassenkalküls and über "Produktations- und Summationsprobleme", welche gewisse Klassen von Aussagen betreffen. Skrifter utgit av Vidnskapsselskapet i Kristiania, I. klasse, no. 3, Oslo, 1919.
57. Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, January 1986.
58. David Walker and Greg Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, 2000.
59. Greta Yorsh. Logical characterizations of heap abstractions. Master's thesis, Tel-Aviv University, March 2003.

Appendix: Operational Semantics

The state of the heap is a pair $\langle [p, r] \circ s, H \rangle$, where p contains the program counter and current module, r is an activation record, s is the call stack of pairs $[p, r]$, and H is the garbage-collected heap. Note that the program counter contains static information about the program point: $\mathsf{pc}(p)$ points to the CFG node to be executed, while $\mathsf{mod}(p)$ indicates the module to which $\mathsf{pc}(p)$ belongs. The heap contains several types of tuples; these track module variable contents, field contents, and local variable contents. A triple $\langle m, v, o \rangle$ in H indicates that module variable v in module m points to the heap object o. The tuple $\langle m, o_1, \mathtt{f}, o_2 \rangle \in H$ means that the field $o_1.\mathtt{f}$ encapsulated in module m points to object o_2. Finally, the triple $\langle r, \ell, o \rangle \in H$ means that the activation record r contains a local variable ℓ pointing to heap object o.

Statement	Transition	Constraints
`p: x = null;`	$\langle [p, r] \circ s, \mathcal{H} \uplus \{\langle r, \mathtt{x}, \_\rangle\}\rangle \longrightarrow$ $\langle [p', r] \circ s, \mathcal{H} \uplus \{\langle r, \mathtt{x}, \mathsf{null}\rangle\}\rangle$	
`p: x = y;`	$\langle [p, r] \circ s, \mathcal{H} \uplus \{\langle r, \mathtt{x}, \_\rangle, \langle r, \mathtt{y}, \mathrm{o}\rangle\}\rangle \longrightarrow$ $\langle [p', r] \circ s, \mathcal{H} \uplus \{\langle r, \mathtt{x}, \mathrm{o}\rangle, \langle r, \mathtt{y}, \mathrm{o}\rangle\}\rangle$	$\mathsf{type}(p, \mathtt{x}) = \mathsf{type}(p, \mathtt{y})$
`p: x = new t;`	$\langle [p, r] \circ s, \mathcal{H} \uplus \{\langle r, \mathtt{x}, \_\rangle\}\rangle \longrightarrow$ $\langle [p', r] \circ s, \mathcal{H} \uplus \{\langle r, \mathtt{x}, \mathrm{o}\rangle\}\rangle$	o fresh $\mathsf{type}(p, \mathtt{x}) = t$
`p: x = y.f;`	$\langle [p, r] \circ s, \mathcal{H} \uplus \{\langle r, \mathtt{x}, \_\rangle, \langle r, \mathtt{y}, \mathsf{id}\rangle, \langle \mathsf{mod}(p), \mathsf{id}, \mathtt{f}, \mathrm{o}\rangle\}\rangle \longrightarrow$ $\langle [p', r] \circ s, \mathcal{H} \uplus \{\langle r, \mathtt{x}, \mathrm{o}\rangle, \langle r, \mathtt{y}, \mathsf{id}\rangle, \langle \mathsf{mod}(p), \mathsf{id}, \mathtt{f}, \mathrm{o}\rangle\}\rangle$	$t = \mathsf{type}(p, \mathtt{y}) \wedge \mathsf{hasField}(\mathsf{mod}(p), t, \mathtt{f}) \wedge$ $\mathsf{type}(p, \mathtt{x}) = \mathsf{fieldType}(\mathsf{mod}(p), t, \mathtt{f})$
`p: x.f = y;`	$\langle [p, r] \circ s, \mathcal{H} \uplus \{\langle r, \mathtt{x}, \mathsf{id}\rangle, \langle \mathsf{mod}(p), \mathsf{id}, \mathtt{f}, \_\rangle, \langle r, \mathtt{y}, \mathrm{o}\rangle\}\rangle \longrightarrow$ $\langle [p', r] \circ s, \mathcal{H} \uplus \{\langle r, \mathtt{x}, \mathsf{id}\rangle, \langle \mathsf{mod}(p), \mathsf{id}, \mathtt{f}, \mathrm{o}\rangle, \langle r, \mathtt{y}, \mathrm{o}\rangle\}\rangle$	$t = \mathsf{type}(p, \mathtt{x}) \wedge \mathsf{hasField}(\mathsf{mod}(p), t, \mathtt{f}) \wedge$ $\mathsf{fieldType}(\mathsf{mod}(p), t, \mathtt{f}) = \mathsf{type}(p, \mathtt{y})$
`p: x = v;`	$\langle [p, r] \circ s, \mathcal{H} \uplus \{\langle r, \mathtt{x}, \_\rangle, \langle \mathsf{mod}(p), v, \mathrm{o}\rangle\}\rangle \longrightarrow$ $\langle [p', r] \circ s, \mathcal{H} \uplus \{\langle r, \mathtt{x}, \mathrm{o}\rangle, \langle \mathsf{mod}(p), v, \mathrm{o}\rangle\}\rangle$	$\mathsf{type}(p, \mathtt{x}) = \mathsf{varType}(\mathsf{mod}(p), v)$
`p: v = x;`	$\langle [p, r] \circ s, \mathcal{H} \uplus \{\langle r, \mathtt{x}, \mathrm{o}\rangle, \langle \mathsf{mod}(p), v, \_\rangle\}\rangle \longrightarrow$ $\langle [p', r] \circ s, \mathcal{H} \uplus \{\langle r, \mathtt{x}, \mathrm{o}\rangle, \langle \mathsf{mod}(p), v, \mathrm{o}\rangle\}\rangle$	$\mathsf{varType}(\mathsf{mod}(p), v) = \mathsf{type}(p, \mathtt{x})$
`p: goto p1;`	$\langle [p, r] \circ s, \mathcal{H}\rangle \longrightarrow \langle [p1, r, m], \mathcal{H}\rangle$	
`p: if (B) goto p1;`	$\langle [p, r] \circ s, \mathcal{H}\rangle \longrightarrow \langle [p1, r, m], \mathcal{H}\rangle$	$\mathsf{eval}(\mathcal{H}, B) = \mathsf{true}$
`p: if (B) goto p1;`	$\langle [p, r] \circ s, \mathcal{H}\rangle \longrightarrow \langle [p', r] \circ s, \mathcal{H}\rangle$	$\mathsf{eval}(\mathcal{H}, B) = \mathsf{false}$
`p: x = m2.proc(a);`	$\langle [p, r] \circ s, \mathcal{H} \uplus \{\langle r, \mathsf{a}, \mathsf{id}\rangle\}\rangle$ $\longrightarrow \langle [p'', r'] \circ [p', r] \circ s,$ $\mathcal{H} \uplus \{\langle r, \mathsf{a}, \mathsf{id}\rangle \uplus \mathsf{hProcSetup}(r', \mathtt{m2.proc}, \mathsf{id})\}$	p'' entry point for `m2.proc` r' fresh $\mathsf{argType}(\mathtt{m2.proc}) = \mathsf{type}(r, \mathsf{a})$
`p: return x;`	$\langle [p, r'] \circ [p', r] \circ s, \mathcal{H} \uplus \{\langle r', x, \mathsf{id}_x\rangle,$ $\langle r', \mathsf{retval}, X\rangle\}\rangle$ $\longrightarrow \langle [p', r] \circ s, \mathcal{H} \uplus \{\langle r, X, \mathsf{id}_x\rangle\} \setminus \{\langle r', \_, \_\rangle\}\rangle$	

where p' satisfies $\mathsf{mod}(()p') = \mathsf{mod}(p) \wedge \mathsf{pc}(()p') = \mathsf{succ}(\mathsf{pc}(()p))$ in the control-flow graph, and:

$$
\begin{aligned}
\mathsf{type}(p, \mathtt{x}) &= \text{declared format of local variable } \mathtt{x} \text{ in } p\text{'s context} \\
\mathsf{varType}(\mathsf{mod}(p), v) &= \text{declared format of variable } \mathtt{v} \text{ of module } \mathsf{mod}(p) \\
\mathsf{hasField}(\mathsf{mod}(p), t, \mathtt{f}) &= \mathsf{true} \text{ iff format } t \text{ in module } \mathsf{mod}(p) \text{ declares field } \mathtt{f} \\
\mathsf{fieldType}(\mathsf{mod}(p), t, \mathtt{f}) &= \text{declared format of field } \mathtt{f} \text{ in format } t \text{ of module } \mathsf{mod}(p) \\
\mathsf{hProcSetup}(r', \mathtt{m2.proc}, \mathsf{id}) &= \{\langle r', \mathsf{retval}, x\rangle, \langle r', fn, \mathsf{id}\rangle, \langle r', \ell_1, \mathsf{null}\rangle, \ldots, \langle r', \ell_n, \mathsf{null}\rangle\} \\
\mathsf{argType}(\mathtt{m2.proc}) &= \text{declared type of formal of } \mathtt{m2.proc}
\end{aligned}
$$

Fig. 20. Operational Semantics for Implementation Language