Crosscutting Techniques in Program Specification and Analysis

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

ABSTRACT

We present three aspect-oriented constructs (*formats*, *scopes*, and *defaults*) that, in combination with a specification language based on abstract sets of objects, enable the modular application of multiple arbitrarily precise (and therefore arbitrarily unscalable) analyses to scalably verify data structure consistency properties in sizable programs. Formats use a form of field introduction to group together the declarations of all of the fields that together comprise a given data structure. Scopes and defaults enable the developer to state certain data structure consistency properties once in a single specification construct that cuts across the preconditions and postconditions of the procedures in the system. Standard approaches, in contrast, scatter and duplicate such properties across the preconditions and postconditions. We have implemented a prototype implementation, specification, analysis, and verification system based on these constructs and used this system to successfully verify a range of data structure consistency properties in several programs.

Most previous research in the field of aspect-oriented programming has focused on the use of aspect-oriented concepts in design and implementation. Our experience indicates that aspect-oriented concepts can also be extremely useful for specification, analysis, and verification.

Keywords

aspect-oriented programming, program verification, static analysis tools, crosscutting concerns

1. INTRODUCTION

A key principle of good software engineering is to collect related items together into a single program unit. Each collection of related items comprises a *concern* of the program; a decomposition of the program into appropriate concerns minimizes interactions between program units, making the program easier to develop, understand, and maintain. Over the years a set of standard program units has emerged: record declarations group together related data fields, procedures group together related computations and actions, and classes group together related data fields and computations that operate on those fields.

A key insight behind aspect-oriented programming is that the exclusive use of the standard set of program units may force developers to scatter (and in some cases, duplicate) items from a single concern across many different units. In response to this situation, researchers have developed new kinds of program units (such as aspects [13] and hyperslices [35]) and new composition mechanisms (such as dynamic weaving [28] and hyperslice compositors [35]). Together, these new aspect-oriented constructs may enable developers to improve the structure of their programs by collecting previously scattered items from the same concern into more intellectually coherent program units.

This paper discusses our use of aspect-oriented concepts in the Hob program specification, analysis, and verification system [18–21,37]. The goal is to specify data structure consistency properties, then verify that the program preserves these properties. The approach is to encapsulate each data structure inside a separate module, then apply potentially different, arbitrarily precise (and therefore arbitrarily unscalable) analyses in a scalable, modular fashion to verify that each module correctly preserves the consistency properties of the data structures that it encapsulates. A common specification language based on abstract sets of objects enables the analyses to interoperate to verify consistency properties that involve multiple data structures encapsulated in multiple modules and analyzed by different analyses.

The traditional approach to modular specification and verification uses assume/guarantee reasoning: each procedure has a specification that consists of a precondition (the properties that it assumes hold upon entry) and a postcondition (the properties that it guarantees will hold upon exit if the precondition holds upon entry). We have found, however, that a phenomenon we call *specification aggregation* makes this approach fundamentally unscalable. The problem is that each procedure's specification must typically contain, as part of its precondition and postcondition, the preconditions and postconditions of any procedures that it invokes. These properties therefore become scattered across the procedure specifications as the preconditions and postconditions of callees are duplicated in the preconditions and postconditions of their (transitive) callers, with the specification and verification overhead becoming prohibitive as one moves up the procedure invocation hierarchy.

Note that aspect-oriented constructs were originally designed (in a different context) to eliminate similar kinds of scattering and duplication. Indeed, we have been able to use constructs either borrowed from or conceptually related to those in aspect-oriented programming languages to eliminate specification aggregation and make the modular verification of data structure consistency properties feasible for programs of arbitrary size. *Formats* provide a form of field introduction [6] that enables developers to group together, in a single module, all of the declarations of all of the fields (potentially from different kinds of objects) that together comprise the data structure that the module encapsulates. Standard approaches, in contrast, group together all of the declarations of all of the fields of a given record or object, with fields from the same data structure scattered across the record declarations or classes of the objects that may participate in that data structure. *Scopes* and *defaults* enable the developer to state a property once in a single location; standard approaches require the developer to explicitly duplicate the property across the preconditions and postconditions of procedures that either require the property themselves or (transitively) invoke procedures that require the property. Scopes can be seen as enabling the effective specification and verification of data structure consistency properties in certain types of program hyperslices [35].

1.1 Formats

Our approach supports the deployment of multiple analyses, with each analysis specialized to verify an arbitrarily complex but potentially quite narrow class of properties characteristic of a corresponding class of data structures. Each analysis operates on a single module to verify that 1) the module preserves the consistency of any encapsulated data structures and 2) each procedure in the module conforms to its specification.

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 analyses. To eliminate the possibility that one module may corrupt another's data structure (and to ensure that each analysis algorithm analyzes all of the relevant code), modules encapsulate fields, *not objects*. Each module uses format declarations to add the fields that implement its encapsulated data structure to the objects that participate in this data structure. The complete set of fields of a given object is equal to the union of all of the fields in all of the format declarations that involve that object. In general, these format declarations may appear distributed across multiple modules.

A key insight behind the format construct is that a field is more closely related to other fields in the same data structure than to other fields in the same object; a field declaration construct should therefore group together fields from the same data structure rather than fields from the same object, *even though fields from the same data structure may occur in different kinds of objects*. From an aspect-oriented perspective, formats isolate a concern (the set of fields that comprise a given data structure) that cuts across the objects in that data structure. Formats support many of the same kinds of program structuring strategies as other field introduction constructs from a variety of aspect-oriented languages [6, 13, 35].

1.2 Scopes

Systems often maintain important invariants that involve multiple data structures and cross encapsulation boundaries. Such invariants are typically violated (temporarily and legitimately) as modules execute a coordinated sequence of data structure updates. Because these invariants involve objects that participate in different data structures encapsulated in different modules analyzed by different analyses, the analyses must somehow interoperate if they are to successfully verify the invariant.

Our approach uses *scopes* to identify invariants that involve multiple data structures encapsulated in multiple modules. Each scope has an invariant (a property expressed in the common abstract set specification language), a set of exported modules, and a set of local modules. The exported modules can be invoked from outside the scope; the local modules can be invoked only from within the scope. Together, the exported and local modules include all of the procedures that may directly modify the data structures that affect the truth value of the invariant. When the analysis processes an exported module it ensures that, if the invariant holds upon entry to the exported modules, then it is correctly restored upon exit.

Scopes eliminate scattering and duplication within the specification because they eliminate the need for callers of procedures in the exported modules to include invariants explicitly in their preconditions and postconditions. The resulting specifications are smaller, simpler, and more modular. From an aspect-oriented perspective, scopes isolate a concern (the invariant) that cuts across the specifications of the (transitive) callers of procedures in exported modules.

1.3 Defaults

Sometimes a data structure must satisfy a given property before clients can successfully invoke some of its operations. The canonical example is initialization: some data structures must be properly initialized before clients can use them at all. With standard approaches, the precondition of each operation in this kind of data structure explicitly requires the data structure to be initialized. This requirement then scatters throughout the system as it is duplicated in the specifications of the (transitive) clients of the data structure.

Defaults allow the developer to state a property and efficiently identify, using a pointcut specifier, the program points where the property must hold. The simplest pointcut specifier identifies every precondition and postcondition in the program (the developer can explicitly suspend the default if it does not hold in a specific precondition or postcondition). More sophisticated pointcuts enable the programmer to exclude certain classes of procedures (such as procedures that initialize data structures or transitively invoke such initialization procedures).

For data structures with initialization constraints, defaults enable the developer to eliminate any mention of the initialization constraint from the vast majority of the program. With an accurate pointcut that excludes the initialization procedures, the initialization constraint appears only in the default. Even with a less accurate pointcut that applies the default to all of the procedures, only the specifications of the initialization procedures mention the constraint (because they must explicitly suspend it). In practice, this approach eliminates virtually all occurrences of the initialization constraint.

Defaults therefore eliminate the scattering and duplication that is otherwise present in the specifications of programs with data structures that require initialization (or other conditions that would otherwise propagate across large regions of the program). From an aspect-oriented perspective, defaults isolate a concern (the default property) that cuts across the specifications of the (transitive) callers of procedures whose preconditions require the default property to hold.

1.4 Contributions

A primary contribution of this research is the combination of these aspect-oriented constructs to work together to enable the scalable automatic verification of arbitrarily precise and sophisticated data structure consistency properties: a goal that has, to this point, appeared to be completely beyond the reach of automated program analysis techniques. We claim the following contributions:

- **Specification Aggregation:** The identification of specification aggregation as a key problem that prevents standard assume/guarantee reasoning approaches from scaling to sizable programs.
- **Aspect-Oriented Approach:** The recognition that aspect-oriented constructs can substantially simplify the specification, analysis, and verification of sizable programs.
- **Aspect-Oriented Constructs:** The identification of three aspect-oriented implementation and specification constructs (namely, formats, scopes, and defaults) that, working together, largely eliminate the specification aggregation problem for data structure consistency properties.
- **Multiple Analyses:** The recognition that these three aspect-oriented implementation and specification constructs, in combination with a specification framework based on abstract sets of objects, make it possible to apply multiple arbitrarily precise, arbitrarily narrow, and arbitrarily unscalable analyses in a general, scalable way to verify sophisticated data structure consistency properties in sizable programs.
- **Implementation and Specification Languages:** The realization of these concepts in concrete specification and implementation languages.
- **Analysis and Verification System:** A prototype program analysis and verification system that contains multiple analyses and can use these analyses to verify a range of data structure consistency properties.
- **Experience:** Our experience using our prototype to verify data structure consistency properties in several complete programs. We have been able to verify detailed consistency properties of individual data structures, then use these properties to verify larger properties that involve multiple data structures analyzed by different analyses.

Consider the implications. The world is full of capable program analysis researchers who are, given adequate resources, able to deliver a virtually unlimited supply of precise, specialized (but potentially unscalable) analyses that can verify almost any data structure consistency property in appropriately-sized program fragments. But for these analyses to have any practical impact, they must be embedded within a framework that 1) effectively deploys each analysis to verify appropriate properties in appropriate regions of the program and 2) supports the combination of the analysis results to verify properties that depend on regions of the program analyzed by different analyses. Our research provides just such a framework.

2. EXAMPLE

We next present an example that illustrates how formats, scopes, and defaults promote the implementation, specification, and modular verification of data structure consistency properties in the Hob program implementation, specification, and analysis system[1] [18–20, 37]. Figure 1 presents a module that encapsulates a doubly-linked list implementation; this module is part of a larger program that implements the popular Minesweeper game. The list implementation consists of a set of `next` and `prev` fields distributed across a set of `Node` objects. The list data structure also contains a header node, which simplifies the implementation of the `add` and `remove` procedures.

The `format` declaration in Figure 1 adds `next` and `prev` fields to each `Node` object. These fields implement the list and are accessible only within the `List` module. Note that `Node` objects may also participate in other data structures, in which case the modules that encapsulate those data structures would use similar `format` statements to add any additional required fields. In this way, `format` statements support field introduction declarations in which each module adds the fields it needs to the objects that participate in its data structure.

Formats are aspect-oriented in that they cut across the set of objects that participate in a given data structure to group together the declarations of the fields that implement that data structure. To the best of our knowledge, field introduction declarations were first proposed in [6] and have since become available in aspect-oriented languages [13, 35]. They support modular analysis because they eliminate the possibility that one module may corrupt another module's data structure. They also enable each analysis to easily locate all of the code that may affect any of the fields in the analyzed data structure.

Programs may instantiate a module multiple times. The code in Figure 2, for example, instantiates the `List` module twice to create two lists of `Cell` objects: a list of exposed `Cell`s, and a list of hidden `Cell`s. Minesweeper uses such lists to maintain sets of exposed and hidden cells on the Minesweeper board. The instantiation statements in Figure 2 use the construct `Node <- Cell` to replace the `Node` type with the `Cell` type to obtain lists of `Cell`s rather than lists of `Node`s. In effect, the `Node` type is simply a placeholder in the `List` module.

Figure 3 presents the `Board` implementation module. This module uses an array `cells` of `Cell` objects to represent the board. It also uses a `format` statement to add several fields to `Cell` objects. These fields join the `next` and `prev` fields from the `ExposedList` and `HiddenList` modules in `Cell` objects, but are accessible only inside the `Board` module. Note

[1] See `http://cag.csail.mit.edu/~plam/hob/` for more information about the Hob project.

```
impl module List {
  format Node {
    next : Node; prev : Node;
  }

  var initialized : bool;
  var root : Node;

  proc init() {
    root = new Node();
    initialized = true;
  }

  proc add(n : Node) {
    Node nn = root.next;
    n.next = nn;
    if (nn != null) {
      nn.prev = n;
    }
    n.prev = root;
    root.next = n;
  }

  proc remove(n : Node) {
    if (n==current) {
      current = current.next;
    }
    Node prv, nxt;
    prv = n.prev;
    nxt = n.next;
    prv.next = nxt;
    if (nxt!=null) {
      nxt.prev = prv;
    }
    n.next = null;
    n.prev = null;
  }
}
```

Figure 1: Implementation Section of `List` Module

```
impl module ExposedList = List with Node <- Cell;
impl module HiddenList = List with Node <- Cell;
```

Figure 2: `List` Instantiations

```
impl module Board {
  format Cell {
      init, isMined, isExposed, isMarked : bool;
      i, j : int;
      init : bool; }
  var cells:Cell[][];
  ...
  proc setExposed(c:Cell; v:bool) returns gameOver:bool
  {
    if (c.isExposed) ExposedList.remove(c);
    else HiddenList.remove(c);
    c.isExposed = v;
    if (v) ExposedList.add(c);
    else HiddenList.add(c);
    if (v && c.isMined) return true;
    else return false;
  }
}
```

Figure 3: Implementation Section of `Board` Module

that the `Board` data structure and list data structures have redundant information: specifically, the `isExposed` field in the `Board` data structure indicates whether its `Cell` object is exposed or hidden; the `ExposedList` and `HiddenList` data structures also maintain this information. Our analysis can verify invariants associated with such redundant state — see Section 5 and [18, 19].

2.1 List Specification

In addition to an implementation section, each module also has a specification section. Figure 4 presents the specification section for the `List` module. The specification uses an abstract boolean flag `ready` (which is true when the list has been initialized and false otherwise) and an abstract set of `Node` objects `Content` (which contains the set of `Node` objects in the list) to state, for each procedure, a `requires` clause (the precondition of the procedure), a `modifies` clause (the flags and sets that the procedure may modify), and an `ensures` clause (the postcondition of the procedure). Note that the `ready` and `Content` specification variables exist only for specification, analysis, and verification purposes: they do not exist when the program runs.

```
spec module List {
  specvar ready : bool;
  specvar Content : Node set;

  proc init()
    requires not ready
    modifies ready
    ensures card(Content)=0 & ready';

  proc add(n : Node)
    requires ready & card(n)=1 & not (n in Content)
    modifies Content
    ensures (Content' = Content + n);

  proc remove(n : Node)
    requires ready & card(n)=1 & (n in Content)
    modifies Content, Iter
    ensures (Content' = Content - n) &
            (Iter' = Iter - n);
}
```

Figure 4: Specification Section of `List` Module

All `requires` and `ensures` clauses use a specification language based on the boolean algebra of sets (this language includes cardinality constraints on the number of objects in the sets). For example, the precondition of the `add` procedure requires 1) the list to be initialized, 2) the parameter `n` to point to a `Node` object and not be `null` (in other words, the set of objects to which `n` points must be of size 1), and 3) the parameter `n` to not be in the set `Content` of objects already in the list. The postcondition states that the set `Content'` in the list after the `add` procedure executes is equal to the union of the set `Content` from before `add` executes with the set containing the object referenced by the parameter `n`.

2.2 List Abstraction

Finally, each module has an abstraction section that establishes the connection between the implementation and the specification. Figure 5 presents the abstraction section for the `List` module. This abstraction section uses object fields to define abstraction functions that provide the meaning of the abstract flags and sets of objects in the specifications. For example, the `Content` statement in Figure 5 defines the `Content` set to be the set of all objects reachable by following `next` fields starting from the header node.

In general, the abstraction section uses a specification language that is specific to whatever analysis is used to analyze the module. In our example, the abstraction section uses notation based on the monadic second-order logic over trees and is designed for an analysis (the PALE analysis) that implements a decision procedure for this logic [15, 24].

```
abst module List {
    use analysis "PALE";
    Content = { n : Elem | "root<next.next*>n" };
    ready = initialized;
    invariant "type Elem = {
      data next:Elem;
      pointer prev:Elem[this^Elem.next = {prev}];
    }";
    invariant "data root : Elem;";
    invariant "(!ready => root=null) &
               (ready  => root != null)";
}
```

Figure 5: Abstraction Section of `List` Module

The procedures in the `List` module rely on the consistency of the list data structure for their correct operation. We call such internal consistency properties *representation invariants* [23]. The abstraction section in Figure 5 uses `invariant` statements and notation based on the monadic second-order logic over trees to specify that the `next` and `prev` fields are inverses. During the analysis of the implementation module, the analysis 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 procedure. The doubly-linked list invariant is crucial: if `prev` is not an inverse of `next`, `remove(n)` may not correctly remove `n` from the list.

In general, the Hob system analyzes individual modules as follows. For each module, Hob examines the implementation, specification, and abstraction sections of that module, as well as the specifications of all procedures that the module invokes. Hob first uses the abstraction function (from the abstraction section) to translate the `requires` and `ensures` clauses into the internal representation of the specialized analysis that will analyze the module (the abstraction section specifies which analysis to use). Hob then conjoins the representation invariant to the translated `requires` and `ensures` clauses. Finally, Hob invokes the specialized analysis to verify that each procedure conforms to its translated `requires` and `ensures` clauses.

The `List` module is an example of a *leaf* module that does not invoke any other modules. The analysis of this module therefore takes as input the three sections in Figures 1, 4, and 5. Hob uses the definition of the `Content` set in Figure 5 to translate the specifications in Figure 4 into the specification language of the PALE analysis [24] (see [18] for details). Finally, Hob invokes the PALE analysis to verify that each procedure conforms to its (translated) specification.

2.3 Defaults in List Specification

Recall that the specification of each procedure in Figure 4 (except the `init` procedure) requires the list to be properly initialized. This requirement causes the `ready` flag to appear in the precondition of every procedure. In the absence of some mechanism to eliminate this explicit requirement, specification aggregation would cause the `ready` flag to propagate up the call hierarchy to appear in the preconditions and postconditions of the (transitive) callers of `List` procedures.

```
spec module List {
  ...
  proc init()
    suspends I
    requires not ready
    modifies ready
    ensures card(Content)=0 & ready';
  ...
}
```

Figure 6: List Specification for `init` With Defaults

Defaults largely eliminate this form of specification aggregation. The developer can use the following default statement to specify that the `ready` flag is true by default at all preconditions and postconditions of all procedures in the program. It would also be possible to use a pointcut specification (see Section 4) to identify a more precise set of procedures (in this case, all procedures not involved in the initialization of the program).

```
default I = ready;
```

With this default, the `ready` flag need not explicitly appear in any of the `List` procedure specifications (and therefore need not appear in the specifications of transitive callers of these procedures) except for the `init` procedure — the precondition for `init` requires the `ready` flag to be false on entry and ensures that the `ready` flag is true on exit. As shown in Figure 6, the `init` specification therefore explicitly suspends the default so that it can express these properties.

Note that the `default` construct eliminates specification aggregation only if the pointcut precisely identifies the procedures where the default property holds. In our example, the preconditions of all of the (transitive) callers of the `init` procedure must use a `suspend` clause to specify that the `ready` flag is false upon entry. In practice, programs tend to contain a small initialization phase that initially suspends all of the initialization defaults, then establishes the defaults one by one as it invokes the initialization procedures. At the end of this phase, all of the defaults hold and they need not appear in the remainder of the program.

2.4 Scopes in Minesweeper

Representation invariants correspond to standard data structure invariants — each representation invariant is encapsulated inside a single module, verified by the analysis that processes that module, and stated in the internal specification language of that analysis. Some invariants, however, involve multiple data structures encapsulated inside different modules analyzed by different analyses. Verifying these kinds of invariants requires multiple analyses to interoperate within a scope that includes all of the modules.

Figure 7 contains the `Model` scope from the Minesweeper program. This scope contains an invariant that specifies a variety of properties that involve multiple data structures in different modules — for example, the first property states that the set of all exposed cells from the `Board` module (as determined by the values of the `isExposed` flags in the `Board` data structure) is the same as the set of cells stored in the `ExposedList` module. The scope contains the `Board`,

```
scope Model
{
 modules Board, ExposedList, HiddenList;
 exports Board;
 invariant (Board.ExposedCells = ExposedList.Content) &
           (Board.HiddenCells = HiddenList.Content) &
           (Board.ready => ExposedList.ready) &
           (Board.ready => HiddenList.ready) &
           (Board.peeking | (card(HiddenList.Iter) = 0));
}
```

Figure 7: The Minesweeper `Model` scope

`ExposedList`, and `HiddenList` modules. Together, these modules define all of the abstract flags and sets in the invariant. The system verifies that the invariant holds whenever these modules are not executing (and therefore potentially updating the data structures that determine the abstract flags and sets in the invariant).

For the invariant to hold, the `Board` module must properly coordinate invocations of the `ExposedList` and `HiddenList` procedures. The `Model` scope therefore exports only the `Board` module — procedures outside the scope may invoke `Board` procedures, but not `ExposedList` or `HiddenList` procedures. This encapsulation ensures that the `Board` controls the operation of the `ExposedList` and `HiddenList` modules and enables the `Board` to preserve the invariant.

Many of the procedures in the `Board` module rely on the `Model` scope invariant for their correct operation. Without this scope, all of the procedures that rely on this invariant would have to explicitly include the invariant in their `requires` clauses. Because of specification aggregation, the invariant would then propagate up the call hierarchy to appear in the preconditions and postconditions of (transitive) callers of `Board` procedures.

Scopes eliminate this form of specification aggregation — they take properties that would otherwise appear duplicated in preconditions and postconditions throughout the program and place them in a single scope that conceptually cuts across all of these preconditions and postconditions.

Our system verifies scope invariants as follows. It first checks that the invariant holds in the initial state of the program (it verifies that the invariant is true in the initial state when sets are initialized to the empty set and flags are initialized to false). It also checks that the form of the invariant is acceptable: each abstract flag and set in the invariant must be defined in modules in the scope. Because a module may modify only its own sets and flags, no module outside the scope can directly affect the invariant. Our system then conjoins the invariant to the `requires` and `ensures` clauses of all procedures in exported modules and proceeds to analyze the procedures of exported modules to verify that they conform to the augmented `requires` and `ensures` clauses. This analysis ensures that the scope invariant holds at each of the scope's entry and exit points.

```
spec module Board
{
  specvar MarkedCells, MinedCells,
          ExposedCells, HiddenCells, U : Cell set;
  specvar init, gameOver : bool;
  default I = init;

  proc setExposed(c:Cell; v:bool) returns gameOver:bool
  requires card(c)=1
  modifies ExposedCells, HiddenCells,
           ExposedList.Content, HiddenList.Content
  ensures
    v => ((ExposedCells' = ExposedCells + c) &
        (UnexposedCells' = UnexposedCells - c);
  ...
}
```

Figure 8: Specification Section of `Board` Module

```
abst module Board
{
  MarkedCells = U cap { x : Cell | x.isMarked = true };
  ExposedCells = U cap { x : Cell | x.isExposed = true };
  UnexposedCells = U cap { x : Cell | x.isExposed = false };
  MinedCells = U cap { x : Cell | x.isMined = true };
}
```

Figure 9: Abstraction Section of `Board` Module

Consider the analysis of the `setExposed` procedure in Figure 3 (Figure 8 presents the specification of this procedure). The only explicit precondition of `setExposed` is the property that the parameter `c` is not null. However, because `Board` is an exported module in the `Model` scope, our analysis system conjoins the the invariant in Figure 7 to the explicit precondition of `setExposed`. Furthermore, the `I` default from Figure 8 applies to `setExposed` because it is not explicitly suspended, so the resulting precondition is the conjunction of 1) the explicit precondition `card(c)=1`, 2) the scope invariant in Figure 7, and 3) the default formula `ready`. Given this precondition, our flag typestate analysis [19] successfully verifies that `setExposed` conforms to its specification in Figure 8. In particular, the analysis uses the set equality `ExposedCells = ExposedList.Content` and the definition of the `ExposedCells` set from Figure 9 to verify that the `if` statement condition `c.isExposed` implies the precondition of the `ExposedList.remove`. Note that Hob uses assume/guarantee reasoning to avoid descending into the `ExposedList.remove` procedure. It instead relies on the specification in Figure 4 to determine the effect of the procedure call. Note also that the analysis of `setExposed` successfully reasons about the precondition of `ExposedList.remove` without being aware of the internal representation of the `List` module. At the end of the procedure, the analysis proves the explicit postcondition of `setExposed`, the default, and the scope invariant in Figure 7. In general, the relationships between sets that the analysis preserves can correspond to relations between complex data structures, implemented using linked data structures and even arrays [17,37]; our abstraction mechanism successfully decouples such complex properties into conformance of data structures with respect to set specifications and relations between resulting abstract sets.

3. SCOPES

Developers use scopes to identify invariants and regions of the program relevant to some concern; our system automatically verifies that the invariants hold at appropriate program points. Figure 10 presents the syntax of scope declarations. A scope groups together a set of modules, some of which are exported. Only procedures in exported modules may be called from outside the scope; modules which are not exported are local to that scope. A scope may also state a *scope invariant*, which is a formula verified to be true in the initial state, assumed to hold whenever the program enters a scope, and verified whenever the program exits that scope.

Set Stationarity Check. Our system checks that the scope invariant uses only sets and flags that are defined in the scope's modules. This *set stationarity check* ensures that only the procedures in the scope can affect the values of the sets and flags of the invariant. Verifying that the invariant holds in the initial state and at scope exit points therefore ensures that it always holds at scope entry points.

Scope Call Check. One of the basic concepts behind scopes is that procedures in exported modules control the operation of local modules to ensure the preservation of the scope invariant. Our system therefore checks that no procedure outside a given scope directly invokes a local procedure of that scope. We formalize this *scope call check* as follows.

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. A procedure call from M' to M passes the scope call check if and only if M is exported in precisely the scopes $C \in \mathsf{scopes}(M) \setminus \mathsf{scopes}(M')$ of the scope difference. More precisely, we say that a module M' calls a module M if the body of some procedure of M' contains a call to a procedure in M. We require the following condition to be satisfied for every pair of modules (M', M): if module M' calls module M, then

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

Note that this definition combines the calling restrictions from all relevant scopes: if M is a local module in some scope C, only modules that are also in C can call M.

Scope Reentrancy Check. In general, a call site inside a given scope may (potentially transitively) call an exported procedure from the same scope (which will assume the scope invariant). We call such a call site a *reentrant* call site.

Our system ensures that scope invariants hold on entry to exported procedures, in part, by requiring scope invariants to hold at all reentrant call sites. It is the developer's responsibility to identify reentrant call sites (it would also be possible to automatically detect such call sites). A simple link-time check (the *call reentrancy check*) ensures that the developer has correctly identified all reentrant sites.

Entering and Exiting Scopes. A program can exit a scope in two places: at the exit point of an exported procedure, or at a call site that invokes either a procedure outside the scope or an exported procedure in the same scope. We call such a call site an *external* call site. The program can enter a scope in two places: at the entry point of an exported procedure, or at the return point of an external call site.

Figure 11 presents an example that illustrates the possible cases. The entry point of each procedure in the exported module `M` is an entry point for the scope `C`. The exit points of these procedures are scope exit points. Call sites from procedures inside `C` (in the example, from procedures in the local module `Q`) to procedures outside `C` are scope exit points. The corresponding return points after the call sites are scope entry points. Finally, call sites from procedures inside `C` (in the example, from procedures in the local module `P`) to procedures in exported modules in `C` are also scope exit points. The corresponding return points after the call sites are also scope entry points.

Public and Private Scope Invariants. Our system supports two kinds of scope invariants. Public scope invariants are visible throughout the program; in particular, the verification system may simply (potentially under developer guidance) assume the public scope invariant at any point in the program outside the scope. To ensure that this verification strategy is sound, the system requires the public scope invariant to hold whenever the program may exit the scope (either at the exit point of an exported procedure or at an external call site).

$$\begin{array}{lll} S & ::= & \mathsf{scope}\ C\ \{ \\ & & \quad \mathsf{modules}\ M^*\ ;\ \mathsf{exports}\ M^*\ ; \\ & & \quad [[\mathsf{public}]\ \mathsf{invariant}\ B;]\ ^*\ \ \} \end{array}$$

Figure 10: Syntax of Scope Declarations

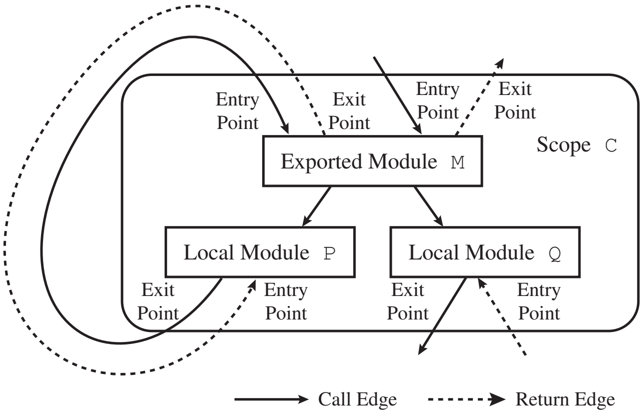

Figure 11: Scope Entry and Exit Points

In contrast, private scope invariants are not visible outside the scope. It would be possible for the verification system to require private scope invariants to hold at the same program points as public scope invariants. But because private scope invariants are not visible outside the scope, the verification system applies a less restrictive policy. Specifically, it only requires private scope invariants to hold at exit points of exported procedures and at reentrant call sites. Note that this policy allows the scope invariant to be (temporarily) violated across non-reentrant calls outside the scope. The fact that private scope invariants are not visible outside their scope ensures that this policy is sound.

Finally, the verification system assumes that the sets and flags of a given scope invariant (and more generally, all sets and flags defined in the modules in the scope) do not change across non-reentrant calls. The set stationarity check described above ensures that the computation rooted at such call sites does not affect these sets and flags.

An Alternate Treatment of Scope Invariants. It is possible to generalize the preceding treatment of scope invariants. Specifically, the system could require the developer (or an analysis) to identify, at each external call site, all of the scope invariants that any potentially (transitively) invoked procedure may assume. The verification system would then require these scope invariants to hold at the call site. A simple link-time check (similar to the link time check for reentrant call sites) would verify the correctness of the scope invariant usage information. This more general treatment eliminates the distinction between public and private scope invariants, gives the developer more control over when scope invariants are required to hold, and supports a wider range of scope invariant placement policies. The potential drawback is that it might require the developer to interact more closely with the verification system.

Scopes and Set Visibility. The sets and flags of local modules are not visible outside the enclosing scopes. In particular, the preconditions and postconditions of procedures in exported modules, the modifies clauses of such procedures, and public scope invariants must not contain sets or flags from local modules.

This design decision means that modifies clauses have a slightly different meaning in the presence of scopes with local modules. Sets and flags from local modules will be absent from the modifies clauses of all exported procedures, even if the procedures may modify some of the sets or flags. To ensure that this absence does not cause soundness violations, the analysis must assume that the procedure invoked at any reentrant call site may modify all sets and flags from the local modules of the scopes to which the module containing the call site belongs.

Multiple Scopes. In our system, a module can participate in multiple scopes simultaneously; this multiple participation enables modules to be grouped into scopes along orthogonal axes. The ability to define multiple non-disjoint scopes within a single program gives our system great expressive power.

First of all, given any region of code expressed as a set of modules, and any invariant I, a developer can introduce a scope exporting these modules, thereby precisely indicating where the invariant I should hold without imposing any unwanted additional constraints on the program structure.

Next, consider the set of all modules $M_1, \ldots, M_k$ in a program, and suppose that we wish to ensure an arbitrary set of restrictions on whether module M_i can call module M_j, given by a boolean matrix a_{ij} (with the natural property that a_{ii} is true). Then we can always define at most k scopes that precisely encode the call matrix a_{ij}. Indeed, it suffices to introduce one scope $\mathcal{C}_i$ for module M_i, make M_i be the sole local module of $\mathcal{C}_i$, and make the set of modules $\{M_j \mid a_{ji} = \mathsf{true}\}$, that are allowed to call M_i, be the set of exported modules of the scope $\mathcal{C}_i$. The set of scopes $\mathcal{C}_1, \ldots, \mathcal{C}_k$ then ensures the desired call matrix a_{ij}. In practice, programs exhibit non-trivial (even if not hierarchical) structure, which implies that many fewer than k scopes suffice to define the desired calling restrictions.

Finally, note that scopes can encode the situation where a module M exposes different subsets of its functionality to different modules, providing more or less restrictive interfaces to different clients [11]. To model this situation, write M by exposing a wide (flexible) interface, and define the proxy modules $M_1, \ldots, M_p$, each of which calls M but propagates only a subset of the functionality of M. Then create a scope with M as a local module and $M_1, \ldots, M_p$ as exported modules.

4. DEFAULTS

The default construct enables developers to state that a specific property holds at a set of procedure preconditions and postconditions (identified by a pointcut) unless explicitly suspended. The syntax of a default declaration is

$$\texttt{default}\ N(A_1, ..., A_k) : C = P$$

where N is the name of the default, the A_i are a set of optional parameter names, C is an optional pointcut specification, and P is a property expressed in the shared set specification language. As discussed in Section 2, defaults are typically used to capture initialization constraints.

$$\begin{array}{rcl}
P & ::= & P_1 - P_2 \mid P_1 \& P_2 \mid P_1 | P_2 \mid \mathsf{not}\ P \\
 & \mid & \mathsf{pre}\ S \mid \mathsf{post}\ S \mid \mathsf{prepost}\ S \\
S & ::= & S_1 - S_2 \mid S_1 \& S_2 \mid S_1 | S_2 \mid \mathsf{not}\ S \\
 & \mid & \mathsf{proc}\ pn(tn_1, \ldots, tn_n)\ \mathsf{returns}\ tn_r \\
 & \mid & \mathsf{exported}\ (\mathsf{module}\ ms) \mid \mathsf{exported}\ (\mathsf{scope}\ ss) \\
 & \mid & \mathsf{local}\ (\mathsf{module}\ ms) \mid \mathsf{local}\ (\mathsf{scope}\ ss) \\
 & \mid & \mathsf{all}\ (\mathsf{module}\ ms) \mid \mathsf{all}\ (\mathsf{scope}\ ss) \\
 & \mid & \mathsf{all} \\
pn,\ tn,\ ms,\ ss & ::= & \text{identifier} \mid \text{identifier*}
\end{array}$$

Figure 12: Pointcut Language for Defaults

Our system implements defaults by conjoining P to procedure preconditions and postconditions that 1) match the pointcut specification C and parameter names A_i (discussed below) and 2) do not explicitly suspend the default N with a specification clause "$\texttt{suspend}\,N$".

Defaults are useful for several reasons: they reduce the size of the specifications, eliminate the specification aggregation that would otherwise occur when default conditions would propagate up the procedure call hierarchy from procedures that require the default, and eliminate specification errors that would otherwise occur when developers inadvertently omit default properties. Developers often appear to unconsciously assume that the default holds (this is understandable as many defaults do, in fact, hold almost everywhere in a correct program) and therefore tend to write specifications that omit required default properties. Defaults can transform these incomplete, unsound, but intuitively correct specifications into complete, sound specifications.

Pointcut Specification Language. Figure 12 presents the syntax for the default pointcut specification language. The developer can use this language to identify a set of procedures S to which the default applies, then specify that the default applies to the preconditions ($\mathsf{pre}\ S$), postconditions ($\mathsf{post}\ S$), or both preconditions and postconditions ($\mathsf{prepost}\ S$) of all procedures in S. The developer may select the procedures by name, by membership in modules, or by membership in scopes. A missing pointcut indicates that the default should apply to all preconditions and all postconditions of all procedures.

Defaults and Modules. Defaults are often coupled to a specific module — for example, a data structure initialization default is typically coupled to the module that encapsulates the data structure. In such cases the developer should define the default within the corresponding module so that the instantiation of the module correctly includes the instantiation of the default (and the constraint that it enforces). Developers may also declare defaults on their own outside of any module — such declarations are typically appropriate when the default property involves multiple modules.

Default Parameter Names. If the default includes parameter names, these parameter names further constrain the set of procedures to which the default applies — if the default has a list of parameter names $A_1, \ldots, A_k$ then it applies only to procedures that have at least k parameters with formal parameter names $A_1, \ldots, A_k$. The parameter names may appear in any order in the procedure's parameter list. For example, in the Water benchmark (Section 5.2), the default

```
default padRead(p) : pre(all(module Reduce)) =
        card(p)=1 & (p in Reduce.Read)
```

applies only to preconditions of procedures in the `Reduce` module that have (at least) a parameter named `p`. When conjoined with the precondition of such a procedure, the default requires `p` to have cardinality 1 (*i.e.* it must not be null) and to be a member of the `Reduce.Read` set.

Defaults as Formula Transformers Conceptually, defaults are *formula transformers*: the defaults we have discussed so far transform preconditions and postconditions by conjoining the default property P to these formulas. The default concept smoothly generalizes to include arbitrary formula transformers that may transform formulas in more sophisticated ways. We have implemented such general formula transformers in the Hob system as part of its support for defaults. One issue is that multiple transformers may apply to a single precondition or postcondition. If the transformers do not commute, different application orders may produce different final formulas. One way to eliminate any such nondeterminism is to group formula transformers into classes (so that all transformers in the same class commute), then prioritize the classes to fix an application order for transformers that may not commute.

5. EXPERIENCE

We have implemented formats, scopes and defaults in the Hob progam implementation, specification, and analysis system. In this section, we discuss how we used these constructs in the specification and verification of several benchmark programs.

5.1 Minesweeper

We have implemented the popular Minesweeper game in our implementation language and used Hob to verify that our implementation conforms to its specification. Our Minesweeper implementation uses the standard model/view/controller (MVC) design pattern [11]. The implementation contains several modules: a game board module (which represents the game state), a controller module (which responds to user input), a view module (which produces the game's output), an exposed cell module (which stores the exposed cells in an array), and a hidden-cell module (which stores the hidden cells in a linked list). The example's `Model` scope encapsulates the complete concrete and abstract states of the game board; the scope contains the game board and the exposed and hidden cell modules.

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

- The linked list satisfies its representation invariants and is used correctly.
- The array set satisfies its representation invariants and is used correctly [37].
- Unless the game is over, the set of mined cells is disjoint from the set of exposed cells.
- The sets of exposed and hidden cells are disjoint.
- At the end of the game, all cells are revealed; *i.e.* the set of hidden cells is empty.
- The set of hidden cells maintained in the `Board` module equals the set of hidden cells maintained in the `HiddenList` list.
- The set of exposed cells maintained in the `Board` module equals the set of exposed cells maintained in the `ExposedSet` array.

The first five properties are intra-module properties enforced on the abstract state of the `Board` module, while the last two properties are inter-module properties, maintained using scope invariants.

Although our system focuses on using sets to model program state, not every module needs to define its own abstract sets — some modules simply coordinate the activity of other modules. For example, the view module does not encapsulate any abstract sets. It instead queries the board for the current game state and calls the system graphics libraries to display the state. Because such interface modules coordinate the actions of multiple modules, they often become exported modules of a scope that specifies an invariant involving the data structures encapsulated in the invoked modules.

Linking global typestate and per-object states. Note that the set abstraction supports typestate-style reasoning at a per-object level (for example, all objects in the abstract `ExposedCells` set can be viewed as having a conceptual typestate "exposed"). Our system also supports the notion of global typestate: for instance, the `Board` module has a global `gameOver` variable which indicates whether or not the game is over. Using this variable and the definitions of sets, we maintain the invariant

```
Board.gameOver |
disjoint(Board.MinedCells,Board.ExposedCells)
```

This invariant connects a global typestate property — is the game over? — with a object-based typestate state property evaluated on objects in the program — there are no mined cells that are also exposed. Conceptually, one could verify these global invariants by conjoining them to the preconditions and postconditions of methods. We use a default to automatically conjoin this particular invariant to the specifications of the procedures in the `Controller` module. For this benchmark, such a default is sufficient to guarantee our property because the `Controller` is the only module that mutates `Board` state. Note that this invariant must be maintained outside of the `Board` module (and hence outside the `Model` scope) because it is only true as a result of the way the `Board` is actually used; a different client of the `Board` module might falsify the invariant.

A scope invariant captures the correspondence between the `Board` module's `ExposedCells` and `HiddenCells` sets and the `Content` module's `ExposedSet` and `HiddenList` sets:

```
(Board.ExposedCells = ExposedSet.Content) &
(Board.HiddenCells = HiddenList.Content)
```

We verify this invariant by including it as a scope invariant for the `Model` scope, so that the system verifies that it holds at all exit points from the `Board` module. Since the scope prevents other modules from calling `ExposedSet` or `HiddenList` procedures directly, the invariant therefore holds on entry to the `Board` module. Note that scope invariants must be true in the initial state of the program. If some initializer must execute to establish a property that then remains true for the remainder of the execution, a global typestate initialization flag can turn the property into an invariant. The developer may then use a default to state that the default value of that global typestate flag is true.

In the absence of defaults, the developer would have to accumulate annotations and manually include the property at every procedure in the module.

The list iterator in our Minesweeper implementation exports two sets, `Iter`, containing the objects yet to be returned by the iterator, and `Content`, containing all objects in the list. It maintains the public module invariant[2] `Iter sub Content`. This invariant captures the property that the iterator only returns objects contained in the list. This invariant is always true outside the list iterator module.

5.2 Water

Water is a port of the Perfect Club benchmark MDG [4] to the Hob implementation language. It contains ten modules and two scopes, with approximately 2000 lines of implementation and 500 lines of specification.

The `ParametrizedEnsemble` scope exports the `Ensemble` module and encapsulates the `Simparm` module. The `Simparm` module manages the global simulation parameters, which are stored in a text file and loaded upon demand. The `ParmsLoaded` flag is true when the parameters have been loaded and stored in the appropriate data structures; the analysis verifies that the computation does not access the simulation parameters until they have been loaded. A scope invariant uses the `Init` flag in the `Ensemble` module to shield the `Simparm` flag: `Ensemble.Init => Simparm.ParmsLoaded`. Thus, in the rest of the program, the `Simparm` module is invisible, and external callers need only establish that the precondition `Ensemble.Init` holds.

The other scope in the program is the `Computation` scope, which exports the `Ensemble` module and encapsulates the `H2O`, `Skratch_pad`, and `Atom` modules. This scope makes it possible to remove all of the sets except the `Ensemble` sets from the following `modifies` clause of the `main` program:

```
modifies Ensemble.SCALEFORCES, Ensemble.INTERF,
  Ensemble.VIR, Ensemble.INTRAF, Ensemble.INITIA,
  Ensemble.Init, Ensemble.CORREC, Ensemble.BNDRY,
  Ensemble.KINETI, Ensemble.PREDIC,
  Skratch_pad.Updated, Skratch_pad.Read, Skratch_pad.Init,
  H2O.Vel, H2O.Pos, H2O.Scaled, H2O.Init, H2O.Intraf,
  H2O.Predic, H2O.Correc, H2O.Bndry, H2O.Kineti,
  Atom.Init, Atom.Correc, Atom.Predic,
  Simparm.ParmsLoaded, Simparm.Init,
```

The `Ensemble` module manages the sequence of computational steps that comprise the water simulation. It uses boolean flags to track the state of the computation; when the boolean flag `INTERF` is true, for example, then the interforce step has been carried out for all of the molecules in the simulation. The specification uses these flags to ensure that the program correctly sequences the steps in the following order:

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

The ability to automatically enforce this step sequencing information may become especially valuable in the maintenance phase of the program's lifetime, when the original designer, if available, may have long since forgotten these sequencing constraints. When appropriate, the specification encapsulates phase ordering constraints behind the interfaces of exported modules. The specification also uses cardinality constraints to prevent null pointers from being passed to procedures that require the pointers to refer to an actual object.

	Full	With Scopes and Defaults	Ratio
acc_double	257	257	N/A
atom	961	868	0.90
consts	3162	3162	N/A
ensemble	8893	6489	0.73
h2o	5472	4557	0.83
main	606	606	N/A
simparm	2308	1722	0.75
skratch_pad	1757	1368	0.78
util	82	82	N/A
water total	24820	20433	0.82
arrayset	571	570	1.00
board	4860	5191	1.07
controller	2761	2478	0.90
list	1157	1126	0.97
main	572	572	N/A
view	2353	2209	0.94
minesweeper total	12274	12146	0.99

Figure 13: Specification sizes, in bytes, for benchmarks with and without scopes and defaults

5.3 Scope and Default Evaluation

We have created and compared versions of our benchmark programs with and without scopes and defaults. Figure 13 presents the sizes of our specification modules with and without scopes and defaults. We found that the use of defaults and scopes in some cases reduced the specification size up to 27%. Our benchmarks had shallow inter-module calling depths, limiting the applicability of scopes. We expect that a program which uses multiple orthogonal and nested scopes would see an even greater reduction in specification size. In any case, we found that defaults and scopes qualitatively made our specifications much more readable by omitting conjuncts that appeared in a number of specifications. In fact, a number of procedures no longer needed any `requires` clauses at all after the use of defaults. Finally, specifications that use scopes and defaults are more likely to be correct: developers are less likely to inadvertently omit required clauses once they are placed in scopes and defaults.

6. RELATED WORK

Our work explores the use of mechanisms to address crosscutting concerns arising in static analysis of data structure consistency. We survey related work in the areas of semantics and analysis of aspect-oriented programs, crosscutting module systems, aspect-oriented specification techniques, default logic, and program checking tools in general.

Semantics and analysis of aspect-oriented programs. Ideas for addressing crosscutting concerns in design and implementation appear in [3, 14, 22, 25, 27, 35]. Classification systems for aspects are presented in [8, 16], and a semantics of aspects with dynamic join points is presented in [36]. Our module mechanism is more modest than open modules [1]; our primary goal is to support the analysis of data structure consistency in the presence of shared objects, so we preserve the control-flow structure of traditional module systems. In [30,33,34] the authors present techniques for analyzing and classifying aspects. Our contribution is somewhat dual to the analysis of aspect interactions, because we explore the use of crosscutting techniques in specifications themselves. Despite these differences, the present pa-

[2] Module invariants are special cases of scope invariants; we conceptually treat a module M with an invariant I as a scope containing the single exported module M and invariant I.

per further contributes to the view that static program analysis and aspect-oriented development techniques can benefit from each other.

Crosscutting module systems. Aspect-oriented techniques aim to modularize crosscutting concerns in software systems. A technique that is particularly relevant to our work is virtual source files. In the Decal system [12], developers may browse and effectively edit two distinct views of a software system: the usual class-based view, and a module-based view which cuts across classes to separate out parts of a program relevant to a given concern. Hob's use of formats to separate concerns is akin to Decal's module-based view; formats allow our system to verify properties of software systems which have this module-based form of crosscutting. Formats can also be viewed as a case of intertype declarations in AspectJ [13]. Scopes are related to virtual source files and hyperslices; augmenting Stellation's dynamic aggregate generation [7] or hyperslices [26] with invariants could achieve a similar effect as scope invariants. In particular, once a developer has identified a scope using a program query or a specification hyperslice, then the developer could state cross-module invariants corresponding to this scope. Hob's scopes allow developers to both state and verify such properties.

Our notion of module instantiation is related to Hyper/J's `equate into` construct: our example could be implemented in Hyper/J by creating a `ExposedList` hyperslice and an `HiddenList` hyperslice, each equating the global `List` into that hyperslice and redirecting the `List`'s reference to `Node` into a reference to `Cell`. If two different modules both contribute a field `init` to objects of class `Cell`, these two `init` fields remain distinct; our system's treatment of such fields corresponds to the Hyper/J `nonCorrespondingMerge` qualifier. Our system differs from Hyper/J with respect to access control: Hyper/J does not allow developers to restrict access to fields across slices.

Specifications for aspect-oriented aystems. In [9], the authors explore the applicability of model checking techniques to modular software systems. The methodology of [9] allows users of model checking tools to state and modularly prove properties of systems which consist of base modules, extension points for these base modules, and extensions to the base modules. Both the techniques of [9] and our approach take advantage of sophisticated software decomposition constructs to improve the effectiveness of program verification. Whereas [9] targets primarily finite-state properties of collaboration-based software designs, our system implements a technique for verification of complex data structure consistency properties. Hob's common specification language does not have temporal operators typically supported by model checkers; instead, Hob uses the source-level concept of program points and the ability to introduce boolean flags to represent both global and per-object temporal properties. Hob's set-based specification language goes beyond the finite state models used in model checking; the specification languages for the individual analyses are even more expressive.

Default logic. Reiter's default logic [29] is a nonmonotonic logic that extends first-order logic by adding a set of default rules. These default rules automatically add clauses to formulas in the absence of evidence to the contrary. Our notion of defaults differs from Reiter's default logic in that our defaults do not change the underlying logic of our system. We continue to use first-order logic for reasoning about program state and apply syntactic formula transformations to certain formulas in our specifications.

Program checking tools. ESC/Java [10] is a program checking tool whose purpose is to identify common errors in programs using program specifications in a subset of the Java Modelling Language (JML) [5]. ESC/Java sacrifices soundness in that it does not model all details of the program heap, but can detect some common programming errors. The Spec# programming system [2] adds similar features to C#, including the ability to specify method contracts, frame conditions and class contracts. These contracts may be verified at run-time or by the Boogie static verifier, which uses a theorem prover to discharge its verification conditions.

We discuss two key differences between our approach and the proposed Boogie approach. First, Boogie envisions the use of a single general-purpose theorem prover to discharge the generated verification conditions. Hob, on the other hand, is designed to support a diverse range of potentially narrow, specialized analyses (this range includes shape analyses, typestate analyses [19] and even interactive theorem provers [37] as well as less detailed analyses). This goal is reflected in Hob's format construct and in its abstract set specification language, both of which are designed to support a strong separation between different analyses (such a separation is necessary, of course, if multiple analyses are to cooperate to successfully analyze a single program). This approach minimizes the amount of expertise required to work within the Hob system and maximizes the ability of developers with specialized skills to contribute. We believe that enabling as many developers to contribute as possible will lead to a richer, more powerful analysis system.

Second, Boogie is designed to verify object invariants, with an object ownership mechanism supporting the hierarchical specification and verification of invariants that involve hierarchies of linked objects. This mechanism eliminates a form of specification aggregation for computations that traverse a hierarchy of owned objects — if the procedure call hierarchy matches the ownership hierarchy, each procedure need only state consistency requirements for the object that it directly accesses, not all of the child objecs that that object owns. This hierarchical specification approach is reminiscent of hierarchical access specifications in Jade [31] and hierarchical locking mechanisms in databases [32].

Hob, on the other hand, is designed to support computations organized around a flat set of data structures. The constructs that eliminate specification aggregation cut across the procedure call hierarchy rather than working within it. This adoption of cross-cutting organizational approaches reflects the maturation of computer science as a discipline — over time, the overwhelming dominance of hierarchical approaches will fade as the effectiveness of using other approaches in addition to hierarchies becomes obvious.

7. CONCLUSION

Our experience with modular pluggable analyses has identified crosscutting concerns as an issue in program specification and verification. We designed several constructs and added them to our implementation and specification languages to address these issues. We found these constructs to be useful for verifying data structure consistency: the use of formats allowed us to independently specify and analyze sets of shared objects implemented using different data structures, whereas scopes and defaults improved the locality and clarity of our specifications, and, at the same time, reduced the sizes of these specifications. Together, these constructs enabled us to build a prototype analysis system that deploys multiple precise, unscalable analyses to verify, in a scalable and modular fashion, precise data structure consistency properties in sizable programs.

8. REFERENCES

[1] J. Aldrich. Open modules: Reconciling extensibility and information hiding. *Software Engineering Properties of Languages for Aspect Technologies*, March 2004.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004: International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, March 2004.

[3] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4), Oct. 1992.

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

[5] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. Technical Report NII-R0309, Computing Science Institute, Univ. of Nijmegen, March 2003.

[6] D. R. Cheriton and M. E. Wolf. Extensions for multi-module records in conventional programming languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 296–306. ACM Press, 1987.

[7] M. C. Chu-Carroll. Supporting distributed collaboration through multidimensional software configuration management. In *Proceedings of the 10th ICSE Workshop on Software Configuration Management*, 2001.

[8] C. Clifton and G. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. Technical Report TR 02-04, Department of Computer Science, Iowa State University, Mar. 2002.

[9] K. Fisler and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Vienna, Austria, Sept. 2001.

[10] C. Flanagan, K. R. M. Leino, M. Lilibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proc. ACM PLDI*, 2002.

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

[12] D. Janzen and K. D. Volder. Programming with crosscutting effective views. In M. Odersky, editor, *18th ECOOP*, pages 195–218, 2004.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *ECOOP*, 2001.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, 1997.

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

[16] R. Laddad. *AspectJ in Action.* Manning Publications Company, Greenwich, CT, 2003.

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

[18] P. Lam, V. Kuncak, and M. Rinard. On our experience with modular pluggable analyses. Technical Report 965, MIT CSAIL, September 2004.

[19] P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking for data structure consistency. In *6th International Conference on Verification, Model Checking and Abstract Interpretation*, 2005.

[20] P. Lam, V. Kuncak, and M. Rinard. Hob: A tool for verifying data structure consistency. In *14th International Conference on Compiler Construction (tool demo)*, April 2005.

[21] P. Lam, V. Kuncak, K. Zee, and M. Rinard. The Hob project web page. http://catfish.csail.mit.edu/~plam/hob/, 2004.

[22] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Mar. 1999.

[23] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley, 2000.

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

[25] D. Moon. Object-oriented programming with flavors. In *OOPSLA*, 1986.

[26] H. Ossher and P. Tarr. Multi-dimensional separation of concerns using hyperspaces. Technical Report Research Report 21452, IBM, 1999.

[27] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, Oct. 2001.

[28] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, pages 141–147, Enschede, The Netherlands, 2002.

[29] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, pages 81–132, 1980.

[30] M. Rinard, A. Sălcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proc. 12th Symposium on the Foundations of Software Engineering*, 2004.

[31] M. C. Rinard. *The Design, Implementation and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language.* PhD thesis, Stanford University, 1994.

[32] A. Silberschatz and Z. Kedem. Consistency in hierarchical database systems. *Journal of the ACM*, 27(1):72–80, January 1980.

[33] G. Snelting and F. Tip. Semantics-based composition of class hierarchies. In *ECOOP*, 2002.

[34] M. Störzer and J. Krinke. Interference analysis for AspectJ. In *Workshop on Foundations of Aspect-Oriented Languages*, Mar. 2003.

[35] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. S. Jr. N degrees of separation: Multi-dimensional separation of concerns. In *ICSE*, 1999.

[36] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *Transactions on Programming Languages and Systems*, 26(5), 2004.

[37] K. Zee, P. Lam, V. Kuncak, and M. Rinard. Combining theorem proving with static analysis for data structure consistency. In *International Workshop on Software Verification and Validation (SVV 2004)*, Seattle, November 2004.