The Hob System for Verifying Software Design Properties

by

Patrick Lam

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2007

Author .....
Department of Electrical Engineering and Computer Science
February 1, 2007

Certified by .....
Martin Rinard
Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

The Hob System for Verifying Software Design Properties

by

Patrick Lam

Submitted to the Department of Electrical Engineering and Computer Science
on February 1, 2007, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This dissertation introduces novel techniques for verifying that programs conform to their designs. My Hob system, as described in this dissertation, allows developers to statically ensure that implementations preserve certain specified properties. Hob verifies heap-based properties that can express important aspects of a program's design. The key insight behind my approach is that Hob can establish detailed software design properties—properties that lie beyond the reach of extant static analysis techniques due to scalability or precision issues—by focussing the verification task. In particular, the Hob approach applies scalable static analysis techniques to the majority of the modules of a program and very precise, unscalable, static analysis or automated theorem proving techniques to certain specific modules of that program: those that require the precision that such analyses can deliver. The use of assume/guarantee reasoning allows the analysis engine to harness the strengths of both scalable and precise static analysis techniques to analyze large programs (which would otherwise require scalable, imprecise analyses) with sufficient precision to establish detailed data structure consistency properties, *e.g.* heap shape properties. A set-based specification language enables the different analysis techniques to cooperate in verifying the specified design properties. My preliminary results show that it is possible to successfully verify detailed design-level properties of benchmark applications: I have used the Hob system to verify user-relevant properties of a water molecule simulator, a web server, and a minesweeper game. These properties constrain the behaviour of the program by stating that selected sets of objects are always equal or disjoint throughout the program's execution.

Thesis Supervisor: Martin Rinard
Title: Professor

Dedicated to the memory of Raja Vallée-Rai (1975-2004).

Acknowledgments

First, I must acknowledge my advisor Martin Rinard for guiding me through this degree. His intuitions (usually spot-on) have certainly helped me design and execute the research in this thesis. Martin has always been accessible and helpful to me. I would also like to thank Butler Lampson for his ability to spot inconsistencies and non sequiturs in this document which eluded my notice.

Collaborating with Viktor Kuncak has certainly been difficult at times. I'd like to thank him, though; together, we managed to get some research results that, I believe, exceeded what we could have achieved separately.

Viktor is responsible for the idea of incorporation in the flags plugin. The Bohne plugin was primarily developed by Thomas Wies in collaboration with Viktor. The theorem proving plugin was developed and used by Karen Zee and Viktor.

An important component of a graduate degree—perhaps the most important part of the learning experience—is that of belonging to a research group. Radu Rugina, Maria-Cristina Marinescu, Darko Marinov, C. Scott Ananian, Brian Demsky, Alexandru Sălcianu, Karen Zee, Viktor Kuncak, and new students Michael Carbin and Zoran Dzunic have contributed immensely to my experience at MIT. I'd also like to explicitly thank Brian for not getting us into an avalanche.

Mary McDavitt, administrative assistant extraordinare, has provided invaluable support to our group, and has furthermore helped proofread selected passages of this thesis. I appreciate all of her help.

Jonathan Babb mentioned Viktor, Karen and me for being his last officemates in his acknowledgements. I'd like to thank him for being my first officemate here at MIT and setting the tone for the next six years of graduate school.

Seven years ago, I thanked Marie-Pascale Desjardins [57]. I am fortunate in that I can once again thank her for her love, her support, and her acceptance of my quirks, which she has, by now, had ample time to discover.

This research was partially supported by Canada's Natural Science and Engineering Research Council as well as le Fonds québécois de la recherche sur la nature et les technologies.

Contents

1 Introduction **17**
- 1.1 Scalability and Diversity . . . 18
- 1.2 Approach Based on Abstract Set Specifications . . . 20
  - 1.2.1 Two novel specification-level constructs . . . 21
- 1.3 Verifying Program Properties . . . 21
- 1.4 Rationale . . . 22
- 1.5 Results . . . 23
- 1.6 Limitations . . . 24
- 1.7 Contributions . . . 26
- 1.8 Structure . . . 27

2 Hob Implementation Language **29**
- 2.1 Example: Doubly-Linked List Implementation . . . 29
  - 2.1.1 Explicit module definitions . . . 29
  - 2.1.2 Static module instantiation . . . 29
  - 2.1.3 Type and variable declarations . . . 30
  - 2.1.4 Procedures . . . 31
  - 2.1.5 Executing Hob programs . . . 33
- 2.2 Implementation Language Grammar . . . 33
- 2.3 Operational Semantics . . . 33
- 2.4 Discussion . . . 38
  - 2.4.1 Implications of encapsulating fields . . . 38
  - 2.4.2 Implications of static instantiation . . . 39

3 Hob Specification Language **41**
- 3.1 Example: Doubly-Linked List Specification . . . 42
  - 3.1.1 Specification module definitions and instantiations . . . 42
  - 3.1.2 Specification variable definitions . . . 42
  - 3.1.3 Procedure definitions . . . 43
- 3.2 Example: Global Properties (Scopes) . . . 44
  - 3.2.1 A global invariant . . . 44
  - 3.2.2 Specifying global invariants . . . 46
  - 3.2.3 Verifying global invariants . . . 48
  - 3.2.4 Specification aggregation . . . 48
- 3.3 Example: Global Properties (Defaults) . . . 49

3.4 Specification Language Grammar . . . 49
3.4.1 Core specification language . . . 50
3.4.2 Scopes . . . 51
3.4.3 Defaults . . . 56
3.5 Discussion . . . 58
3.5.1 Scopes and specification aggregation . . . 59
3.5.2 Advantages and disadvantages of defaults . . . 60
3.5.3 Implications of using a set specification language . . . 60
3.5.4 Comparison: Static analysis and testing . . . 63

4 Hob Abstraction Languages 65
4.1 Analysis Approach . . . 66
4.1.1 Specifying Hob abstraction functions . . . 68
4.1.2 Common abstraction module grammar . . . 68
4.2 Flags Abstraction Module Language . . . 70
4.2.1 Example: Flag abstraction module . . . 72
4.2.2 Loop invariant inference . . . 73
4.2.3 Using the flag analysis plugin . . . 79
4.3 Bohne Abstraction Module Language . . . 80
4.3.1 Example: Bohne abstraction module . . . 81
4.3.2 Using the Bohne analysis plugin . . . 83
4.4 Theorem Proving Abstraction Module Language . . . 85
4.4.1 Example: Theorem proving abstraction module . . . 86
4.4.2 Using the theorem proving analysis plugin . . . 87
4.4.3 Expressive power of the theorem proving plugin . . . 87
4.5 How Abstraction Modules Enable Checking of Global Properties . . . 89

5 Ensuring Consistency Properties 93
5.1 Analysis Plugin Responsibilities . . . 93
5.2 Developing New Analysis Plugins . . . 96
5.3 Hob Analysis Driver . . . 97

6 Flags Analysis Plugin 103
6.1 Flags Analysis Example . . . 103
6.2 Flags Analysis Algorithm . . . 106
6.3 Incorporation . . . 108
6.4 Transition Relations . . . 109
6.5 Verifying Implication of Dataflow Facts . . . 112
6.6 Loop Invariant Inference . . . 112
6.7 Boolean Algebra Formula Transformations . . . 112
6.8 Evaluating Formula Optimization Impact . . . 115

7 Experience **117**
7.1 Data Structure Implementations 117
7.1.1 Tree data structure 117
7.1.2 Stack data structure 121
7.2 Water 121
7.3 HTTP Server 124
7.4 Minesweeper 127
7.5 Implications of Modular Analysis 132
7.6 Summary and Reflections 132

8 Related Work **137**
8.1 Specification Languages 137
8.1.1 Expressing design information 142
8.2 Analysis Technologies and Verification Systems 143
8.3 Combining Static Analyses 149

9 Conclusion **151**
9.1 Future Work 152
9.2 Implications 154

List of Figures

2-1 Doubly-linked list implementation, part 1 34
2-2 Doubly-linked list implementation, part 2 35
2-3 Formats example 36
2-4 Grammar for Hob implementation language 36
2-5 Operational semantics for implementation language 37

3-1 Doubly-linked list specification 45
3-2 Scope invariant example 47
3-3 Illustration of scopes example 47
3-4 Defaults example 50
3-5 Syntax of the Module Specification Language 51
3-6 Syntax of Scope Declarations 53
3-7 Scope Entry and Exit Points 53
3-8 Pointcut Language for Defaults 57

4-1 Abstraction Language Grammar 69
4-2 Example List Abstraction Module 71
4-3 Example Flag Abstraction Module 73
4-4 Procedure containing a loop 75
4-5 Procedures called within the loop 75
4-6 Grammar for Flag Abstraction Modules 79
4-7 Bohne abstraction body for doubly-linked list 83
4-8 Grammar for Bohne Abstraction Modules 85
4-9 Example Theorem Proving Abstraction Section 86
4-10 Grammar for Theorem Proving Abstraction Modules 88
4-11 Model scope from Minesweeper example 90
4-12 Module visibility by various analysis plugins 91

5-1 Overview flowchart for generic analysis plugin 94
5-2 Detailed flowchart for generic analysis plugin 95
5-3 Driver state after parsing `minesweeper` files 98
5-4 Driver state after processing `minesweeper` static module instantiations 99
5-5 Driver state after adding `minesweeper` scope dependencies 99
5-6 Driver state after adding `minesweeper` inter-module dependencies . . 100
5-7 Commands generated by Hob analysis driver 101

6-1 Minesweeper `Board` specifications, implementations, and abstractions 104
6-2 Flowchart for flags analysis plugin 107
6-3 Pseudo-code for Loop Invariant Inference Algorithm 113

7-1 Implementation of `TreeSet` insert procedure 120
7-2 Module dependency diagram for water benchmark 122
7-3 Module dependency diagram for web server 125
7-4 Module dependency diagram for Minesweeper implementation 128
7-5 Doubly-Linked List Client 130

List of Tables

6.1 Formula sizes before and after transformation 116

Chapter 1

Introduction

Design information can greatly contribute to understanding the structure of a software artifact and the underlying assumptions behind that artifact. During the initial development phase of a software system, design information enables developers to communicate productively while collaboratively implementing software systems, guides them through the implementation process, and helps them pinpoint the causes of software defects. In subsequent phases of software development and maintenance, design information can help developers to avoid introducing pernicious errors based on misunderstandings of a system's design and understand how to most effectively add new features to software. Today, however, design information is typically codified at an early stage of a program's development and rarely kept up-to-date: because no tool can currently automatically verify design properties, it is difficult to keep design information current as designs and implementations evolve. It is therefore difficult for developers to take advantage of the benefits offered by up-to-date, valid design information as systems progress through their entire lifecycles.

This dissertation presents a suite of techniques that allow developers to specify important design properties of software systems and to automatically verify that their implementations satisfy the specified design properties. These techniques focus on properties that pertain to data structures. Since a data structure's state can often be productively summarized, in abstract terms, by the set of heap objects that the data structure contains, this dissertation focusses on data structure properties that can be expressed in terms of the boolean algebra of sets. (In this dissertation, the term "abstract set" denotes a named set of heap-allocated objects, abstracting away from the question of how the set is defined.) Abstract sets therefore enable developers to describe the state of a software system in abstract terms—developers may instead reason about the contents of different data structures without requiring them to reason about the implementation details behind each data structure.

Relationships between abstract sets constitute a key type of design information, and being aware of such relationships can help developers better understand the structure of their systems. This dissertation therefore proposes a set specification language, containing the boolean algebra of sets, for expressing relationships between sets. Such a specification language enables developers to state equality, containment, or disjointness relationships between sets and objects, and to verify that these properties hold

in the implementations of their systems. Because design properties often relate the contents of different sets (and therefore data structures), the techniques presented in this dissertation therefore enable developers to state and verify design-level properties about a software system's design and verify that they hold across all executions of a software system. Such design-level properties may be most appropriately expressed as invariants, which hold throughout a program's execution, or as preconditions and postconditions, which hold at specific points in the execution.

Reasoning in terms of abstract set specifications is only meaningful if data structures properly implement their set abstractions. Implementations of data structures tend to be intricate and the necessary low-level concrete consistency properties are typically difficult to verify. While static analysis techniques which can verify certain classes of data structure implementations do exist, two challenges currently limit the applicability of existing data structure analysis techniques. The first challenge is diversity: there exists a wide range of data structures, and yet each existing analysis technique only applies to a subset of these data structures. The second challenge is scalability: because data structure consistency properties are so complex, analysis techniques must build detailed summaries of the implementation's state. Such summaries are computationally expensive to reason about, so existing analysis techniques do not scale to even moderately-sized programs.

A key insight of this dissertation is that it is possible to use a modular analysis approach to overcome both the diversity and scalability problems: by applying existing sophisticated static analyses in concert and using them to decide whether procedures satisfy their set specifications, it becomes possible to handle many different classes of data structure implementations; and by only applying expensive analyses to those portions of a program that require them, the overall analysis can terminate in a reasonable amount of time.

In summary, my research enables developers to verify both low-level concrete data structure consistency properties, which ensure that data structures satisfy necessary internal invariants, and abstract high-level properties, which relate the contents of multiple data structures. These properties therefore help developers maintain verified, up-to-date design information. I expect that such design information will improve developer productivity by giving developers reliable information about their programs, especially when multiple developers participate in the development of a software system and have roles that change over time.

1.1 Scalability and Diversity

Researchers have developed a range of static program analyses for verifying programs that manipulate data structures, and in particular for verifying that programs preserve important data structure consistency properties in all possible executions. Shape analyses, for instance, verify that programs correctly manipulate linked heap data structures. Static analysis techniques work by constructing models of the program's state and actions that overapproximate its semantics. Each analysis technique uses a set of models that is specific to the targetted set of programs and properties.

Analyzing different data structures therefore requires different abstract models: analyzing linked lists is very different from analyzing arrays, which is in turn quite different from analyzing data structures encoded using bit manipulations. It is hard to imagine any single analysis abstraction which can effectively handle all data structures of interest. Therefore, any technique which verifies consistency properties that cut across multiple data structures must somehow combine analysis results from different static analyses, each of which uses its private model of the program.

An additional challenge to verifying data structure consistency properties is the issue of scalability. The design of any static analysis technique involves a fundamental tradeoff between precision and scalability: to verify more detailed program properties, a static analysis must build more detailed models of the program's state and more accurate abstractions of how the program manipulates its state. It is, of course, expensive to construct and maintain detailed models and abstractions. The computational cost of state-of-the-art shape analyses, suitable for verifying key data structure consistency properties, is typically super-exponential in the size of the program fragment being verified, and the literature therefore does not contain successful reports of applications of shape analysis to more than hundreds of lines of code at a time.

Modular verification, in the form of assume/guarantee reasoning[1], is a well-known technique in the program verification community. However, modular verification has always been difficult to apply in practice. Firstly, it has been difficult to choose a notation for expressing program properties which is suitable for modular verification. This notation must be sufficiently expressive to enable developers to express interesting properties, yet it must be concise enough so that the specifications remain tractable. Secondly, even with a suitable notation, it has been difficult to find appropriate techniques for automatically verifying that implementations actually conform to their stated properties. The key insights in this dissertation are that the use of a common set specification language and the pluggable analysis approach enable the productive use of modular reasoning for the verification of data structure consistency properties. The set specification language enables the encapsulation of data structures behind sufficiently rich abstraction barriers such that once an analysis proves that an implementation conforms to its set interface, other analyses can productively use this analysis result to guarantee data structure consistency properties. Furthermore, the specification language contains notions that enable developers to control the growth of specifications throughout the program. My results show that the approach presented in this dissertation can soundly and practically apply arbitrarily precise—and hence arbitrarily unscalable—analyses to only those portions of an implementation that need that precision.

[1]Because this dissertation focusses on sequential programs, assume/guarantee reasoning is equivalent to reasoning based on preconditions and postconditions. The general formulation of assume/guarantee reasoning is more general than preconditions and postconditions in that it relates the actions of the system and its environment. In concurrent programs, the system and its environment may act concurrently.

1.2 Approach Based on Abstract Set Specifications

The Hob system presented in this dissertation analyzes programs consisting of a collection of program modules. Each module contains specifications and implementations. *Set-based specifications* are a key part of the Hob methodology; they allow developers to state properties of the heap by stating properties of sets of heap objects. Because the contents of data structures can often be characterized using sets, Hob's set-based specification language enables developers to express important global data structure consistency properties relating the contents of different data structures without needing to understand the internal design of each data structure. The developer may instead assume that each data structure encodes a set. This enables the developer to reason about the state of a program by reasoning about its sets. Furthermore, set-based specifications serve as an analyzable abstraction of the program state: the use of set-based specifications as a common specification notation allows each of the static analyses that comprise the Hob system to track the abstract state of the heap relatively efficiently.

Such an approach to the verification of global properties, of course, relies on data structures correctly implementing their set abstractions. The Hob system allows data structure implementors to specify internal data structure properties—relationships between the abstract state, expressed in terms of sets, and the concrete state, expressed in terms of properties of heap objects. *Abstraction functions* and *invariants* relate the abstract and concrete states. The Hob system enables developers to use different static analysis techniques to verify each module by supporting *analysis plugins*. Each analysis plugin processes a particular family of abstraction functions and decides whether or not implementations conform to their specifications, using the provided abstraction functions.

Because different data structures may be analyzed using different analysis techniques, and because Hob's common set specification language enables developers to uniformly express properties about different data structures, the Hob system enables developers to verify implementations using a variety of analysis techniques. For instance, developers can state and verify the property that two data structures share no elements, even if these data structures are implemented using completely different data structures. Data structure consistency properties can, in general, describe how program modules may interact.

Once Hob has verified that all modules satisfy their contracts, then the program's data structure consistency properties are guaranteed to hold. Note that Hob's analysis task is structured in terms of *assume/guarantee reasoning*: developers express program data structure consistency properties in terms of the assumptions that procedures may expect to hold upon entry, as well as the conditions that procedures guarantee upon successful completion. Assume/guarantee reasoning in Hob works at two levels: first, modules assume that their client modules properly implement their interfaces, and second, modules may rely on their preconditions holding upon entry. The Hob system discharges the relevant guarantees when it encounters them during the analysis task.

1.2.1 Two novel specification-level constructs

In the Hob system, modular verification depends on the availability of program specifications. The size and complexity of program annotations is a critical parameter determining the feasibility of assume/guarantee reasoning, in terms of both annotation and analysis effort. Having observed that certain clauses tended to cut across specification statements in different parts of the program, and that these clauses tended to accumulate towards the top of the program's call graph, I invented and implemented two specification-level constructs that proved useful in reducing the size and the complexity of annotations. These constructs made it easier both to write annotations and to reason about them; they enable a second kind of scalability in the Hob system. Hob therefore contains scalability constructs for both the analysis task and the specification task.

Scopes are a construct for grouping together modules. Scopes contain scope invariants, which are logical formulas correlating the state of the contained modules. A scope invariant might state, for instance, that a program has two sets that are always disjoint. These formulas may be temporarily violated inside the associated scope, but are verified at scope boundaries, and therefore hold universally throughout the program's execution. In particular, scope invariants must hold in the program's initial state. Scope invariants simplify both program annotation and program analysis: they simplify the annotation task by allowing the developer to omit clauses from the annotation; furthermore, they simplify the analysis task by relieving the analysis of the responsibility for proving the invariant, except at certain crucial program points. A simple worst-case estimate for a modestly-sized program with a call depth of 6 shows that the use of scopes can reduce aggregate specification size from 384 clauses to 64 clauses and maximum specification size from 64 to 1.

I also observed that some clauses hold almost everywhere in the program, but not everywhere, and are in fact false in the program's initial state; these clauses are therefore not appropriate for use as scope invariants. Because these clauses should not need to be explicitly stated throughout a module's specification, I implemented the *default* construct, which simplifies annotations by conjoining such clauses to procedure preconditions at arbitrary points in the program's specifications. I adapted the notion of a pointcut from aspect-oriented programming to enable developers to specify where these clauses should hold.

I expect that these constructs will help developers to annotate programs. This dissertation therefore contains an evaluation of how scopes and defaults help developers specify programs.

1.3 Verifying Program Properties

Hob's approach decomposes the analysis of a program into the analysis of its component modules. Some of these modules are reusable generic library modules, while others contain application-specific code. Library modules may be implemented using a range of techniques: some modules might store objects in structures like arrays and

linked lists, while others could go as far as using bit-level manipulation to efficiently store and retrieve information. The sophistication of data structure consistency properties places them beyond the reach of scalable analysis techniques, while the diversity of these properties makes it hard to imagine that any single analysis could verify the full range of data structure consistency properties.

The problems of scalability and diversity inspired Hob's *analysis plugin* approach. Instead of attempting to use a single analysis to verify all of a program's interfaces, the Hob system is made up of a number of analysis plugins, each of which is designed to verify a narrow class of targeted consistency properties. Hob's analysis plugins currently include a field-value based analysis, a shape analysis, and an analysis that uses interactive theorem proving tools. When presented with a module to analyze, the Hob analysis driver uses an analysis for that particular module, as directed by the developer. No matter which analysis plugin is used, though, library modules only need to be verified once; as long as the module has been successfully verified, developers may subsequently rely on the module's specification as a correct summary of the behaviour of the module. Note that despite the pervasive use of unscalable analyses, the overall Hob approach can scale, since it verifies the program one procedure at a time, using assume/guarantee techniques, and communicates analysis information between procedures using the common set specification language.

1.4 Rationale

A key contribution of this dissertation is its thesis that set specifications allow designers and developers to state, communicate, and enforce design-level information about programs. The Hob approach enables developers to abstract a program's state into a collection of sets of heap objects and express design information in terms of 1) set membership constraints for objects, and 2) relationships between set contents. The Hob program verification framework then uses set specifications to automatically verify design information and ensure that the program satisfies the stated design constraints. A key part of the set specification language is its support for scalability at the specification level: the notions of scopes and defaults enable developers to write more concise specifications.

Set membership constraints allow developers to specify that objects must have particular states before certain actions may occur. Such constraints therefore enable developers to encode necessary dependencies between program operations on heap objects. In particular, an object's participation in a module's sets gives insight as to how the object is participating in the computations being carried out by that module. When different modules work together, objects will often carry correlated set memberships in the various modules. Conversely, when a program consists of independent and loosely-coupled submodules, objects may carry orthogonal set memberships in different modules.

Hob's set specifications enable developers to select an appropriate collaboration model for the modules in a program and to encode that collaboration model in a verifiable form. To this end, the Hob approach also allows developers to express and

enforce required relationships between sets. Developers may express domain-specific properties by requiring that sets (or combinations of sets: unions, intersections, set differences) always be either empty or nonempty. Hob therefore enables developers to succinctly describe anticipated global program states and allowable state transitions in terms of set-based constraints.

Set specifications therefore enable the Hob framework to automatically verify design information. Note that the targetted expressibility of the set specification notation allows developers to state relevant properties of the program state, while the analyzability of the notation enables analysis plugins to verify that implementations conform to their designs.

1.5 Results

In an effort to evaluate how the Hob approach works in practice, we have built a prototype implementation of the Hob framework and used this implementation to successfully verify a number of benchmark programs. This dissertation describes my experience using the Hob system to implement and specify design information for three programs: a simulation of water molecules; an implementation of an HTTP 1.1 server; and an implementation of the popular Minesweeper game. The water simulation contains 10 modules, 2000 lines of implementation and 500 lines of specifications. The HTTP server contains 14 modules, 1200 lines of implementation, and 300 lines of specifications. The minesweeper implementation contains 6 modules, 787 lines of implementation and 328 lines of specifications. While these applications are relatively modest in size (due in part to the difficulty of translating applications into the Hob implementation language), they demonstrate that it is possible to successfully apply the Hob methodology for program verification—in my experience, it was never necessary to verify more than one procedure at a time.

The sets in the HTTP 1.1 server include sets of request headers, response headers, and sets that capture design information related to a server-side cache. The sets in the Minesweeper game include sets of hidden and exposed cells. These sets are implemented using linked heap data structures and verified using shape analysis techniques. The design of the Hob implementation language permits the shape analysis to inspect just the library modules that manipulate the linked data structures rather than the entire program (which would be infeasible using current shape analysis technology due to scalability issues).

I was surprised to discover that abstract set specifications could express outward-looking user-relevant program properties. For instance, the web server's set specifications state that response headers are emptied between requests; that is, no response would contain stale headers from the previous response. Also, in the minesweeper application, set specifications state that exposed cells are disjoint from mined cells unless the game is over. To my knowledge, Hob is the first system that enables developers to state and verify program properties that are relevant to end users.

1.6 Limitations

The research described in this dissertation and embodied in the Hob analysis tool has some limitations which arise from design decisions made early on in the project's lifetime. This section discusses limitations in the Hob implementation and specification languages and the annotation burden involved in specifying program behaviour.

I designed the Hob implementation language to be syntactically similar to Java at a statement level. I decided to use a custom procedural implementation language as a convenient way to explore the automatic verification of data structure consistency properties while avoiding inessential complexities of a full-fledged programming language. In particular, I omitted common object-oriented features such as inheritance, dynamic dispatch, and object-based encapsulation. In my experience, it was relatively straightforward (if time-consuming) to port Java code to the Hob implementation language. When comparing Java and Hob it is important to keep in mind that Hob has two constructs that approximately correspond to Java's classes: 1) formats are used to represent memory cells, and 2) modules are used to structure a program into its main constituent parts. The static module instantiation in Hob is less general than the dynamic instantiation of classes with methods in Java, but it encourages developers to express the static architecture of an application and aids verifiability. Java programs built using stylized static instantiation idioms would also be easier to analyze than arbitrary Java programs.

Hob programs are specified using set-based specifications. While I believe that set-based specifications are quite appropriate for reasoning about program behaviour, certain properties are not expressible in the Hob specification language. For instance, developers cannot state that a map data structure links particular key and value objects. The use of a more expressive specification language would permit developers to state and verify more detailed program properties. Such a specification language, however, would enable developers to write more detailed specifications which could be more unwieldy and therefore both harder to understand and more expensive to verify conformance against.

While Hob can state and verify relationships between the set of keys and the set of values in its interface specification language (for instance, no object should be both a key and a value simultaneously), Hob cannot state that a particular key is related to a particular value. That is, the Hob specification language cannot express relations between heap objects. Its modelling of maps (*e.g.* hash maps) can therefore only discuss the set of objects which act as keys and the set of objects which act as values. Nevertheless, our experience shows that many interesting data structure properties can be expressed using just the boolean algebra of sets. Such descriptions may not be full specifications of the behaviour of operations, but they do indicate important partial correctness properties, so I believe they make a useful trade-off between the expressive power and tractability of the analysis. I chose to explicitly omit integer and floating-point arithmetic from the Hob specification language.[2] While many

[2] In [55], we describe how to decide Boolean Algebra with Presburger Arithmetic; the Hob system's core specification language could be extended to support BAPA.

data structure consistency properties do depend on general integer and floating-point arithmetic, I believe that, in most cases, these properties can be handled as local consistency properties, and therefore do not need to be expressed to clients. Note that the set specification language does not support sets of pairs or sets of sets, only sets of uninterpreted elements. This is why it can be characterized using the Boolean algebra of sets and decided in elementary time [53] and in practice often belongs to the quantifier-free fragment that can be decided in non-deterministic polynomial time.

It is important to distinguish between Hob's set-based common specification language, which was designed to be less expressive and more tractable, and the specification languages inside the abstraction modules, which express data structure representation invariants and abstraction functions. Specifications that occur inside abstraction modules are not bound by the limitations of Hob's set-based specification language; analysis plugins may use arbitrarily powerful specification languages for expressing a module's internal properties. For example, the monadic second-order logic used in the Bohne plugin can express reachability properties that are not even expressible in first-order logic. Monadic second-order logic can therefore certainly express properties that are not expressible in terms of abstract set specifications.

Hob set specifications describe properties of abstract sets, which are encapsulated within program modules. Unfortunately, this modularization is not appropriate for all programs. For instance, sometimes a data structure's encapsulation will be violated for performance reasons. Or a program's dominant decomposition may not correspond to the module boundaries which would be required for the modular analysis of a particular data structure. The scopes construct addresses this issue to some extent, if the relevant consistency properties are set-based properties. However, scopes do not handle local data structure invariants which are collaboratively maintained in multiple places in a program's implementation.

Finally, the need for program specifications imposes an annotation burden on the development process. In our experience, specifications may grow to as much of 40% of the implementation size[3]. I feel that the overhead is not overly onerous because the specifications provide additional value to developers. Program specifications serve as verified design documentation; any property stated in a specification can automatically be checked throughout a program's lifecycle and, as long as developers continue to run the Hob verification tool and ensure that it succeeds, the design information will never become outdated.

Despite these limitations, I believe that the approach embodied in the Hob system is useful for verifying software design properties. The first two issues mentioned here, about limitations of the current implementation and specification languages, could be overcome in future work. The encapsulation problem is real, but only applies to a limited number of data structures; even programs with unencapsulatable data structures may still contain other data structures whose consistency can be verified.

[3]To put this statistic in context, I sampled a number of C++ applications, including AbiWord, Rosegarden and Inkscape, and found that their header files accounted for 19% to 28% to the application size, in terms of lines of code.

Note that the partiality of the Hob approach allows it to still be helpful even if it cannot solve the whole problem. While the annotation burden has traditionally been a problem with specification-based approaches, I feel that developers will be quite willing to write specifications if they find that these specifications are useful.

1.7 Contributions

The primary contributions of this research are 1) the identification of a specification approach based on abstract sets as a suitable notation for expressing verifiable program design information; and 2) the deployment of a range of existing and novel static analysis techniques to enable the scalable automatic verification of arbitrarily precise and sophisticated data structure consistency properties. This goal has, to this point, appeared to be completely beyond the reach of automated program analysis techniques—shape analyses, for instance, scale super-exponentially with the size of the program being analyzed, and there are no successful reports of shape analysis being used on programs in the 1000-line range. This dissertation makes the following contributions.

- **Specification Approach:** This dissertation proposes a set-based specification approach which enables developers to express data structure consistency properties and verify that implementations conform to these properties. The specification language allows developers to state program properties in terms of sets of heap objects.

- **Specification Scalability:** Specifications tend to accumulate upwards in a program and often become unmanageable (due to volume) at its top levels; we call this phenomenon specification aggregation. This dissertation introduces scopes and defaults, two novel constructs that mitigate the specification aggregation problem and help developers write more concise specifications, which are therefore less likely to be contain errors. In the absence of scopes, individual specification clauses may grow exponentially due to specification aggregation.

- **Multiple Analysis Plugins:** The approach described in this dissertation makes it possible to apply multiple arbitrarily precise, arbitrarily narrow, and arbitrarily unscalable analyses in a general, scalable way to verify sophisticated set-based data structure consistency properties in sizable programs. To my knowledge, the Hob system is the first system to combine results from different static analysis techniques to verify detailed data structure consistency properties.

- **Analysis and Verification System:** This dissertation presents our implementation of the Hob program analysis and verification system, which enables the exploration of the ideas described above. It describes the various Hob analysis plugins and explains how developers can use these analyses to verify a range of data structure consistency properties.

- **Experience:** Finally, this dissertation presents our experience using the Hob system to verify software design properties in several complete programs ranging up to 2000 lines. Hob has been able to verify detailed consistency properties of individual data structures, then use these properties to verify larger software design properties that involve multiple data structures analyzed by different analyses.

Note that the first two contributions enable two orthogonal kinds of scalability. Hob's specification-based approach enables individual analysis plugins to draw valid conclusions about a procedure without having to investigate the procedure's environment. The specification scalability constructs operate at the level of specifications. These specifications enable the analysis plugins to succeed; the specification scalability constructs make it easier for developers to provide these specifications.

1.8 Structure

The remainder of this dissertation is structured as follows. Chapters 2 through 4 explain the Hob system from a user's perspective. Chapter 2 describes the Hob implementation language. Chapter 3 describes Hob's common set specification language, shared by all analysis plugins, as well as the scopes and defaults specification constructs, which enable developers to express crosscutting parts of specifications in one place (rather than scattered across program specifications). Chapter 4 describes how developers can link implementations and specifications using Hob abstraction sections. Chapter 5 starts to peek behind the scenes and explains the basic obligation of Hob analysis plugins: essentially, they must show that an implementation satisfies its specification, where the meaning of the specification is given by the abstraction function stated in the abstraction section. This chapter also explains how the Hob system ensures that all modules in a program are analyzed and how the analysis of each module is given the necessary external specifications. Chapter 6 describes how one particular Hob analysis plugin, the flags plugin, works. Chapter 7 presents my experience using the Hob framework to verify data structure consistency properties for a number of benchmark programs, including an implementation of the popular minesweeper game, a MIDI player, and an HTTP server. Chapter 8 presents related work, and Chapter 9 concludes.

Chapter 2

Hob Implementation Language

Hob modules have three sections: an implementation section, a specification section, and an abstraction section. In this chapter we present the Hob implementation language, which is a simple module-structured Java-like imperative language with references and dynamic object allocation. The implementation language is one of the unifying components of our framework, since all analysis plugins handle programs written in the implementation language. Notable features of our language include the static instantiation of modules (which enables the specification language to work with a finite number of sets) and the ability to specify different fields of objects in different modules (formats), which ensures that modules' private data remains private even different modules share heap objects.

2.1 Example: Doubly-Linked List Implementation

Figures 2-1, 2-2, and 2-3 present a pair of module declarations and a pair of module instantiations in our implementation language. The first module, `DLL`, implements a set abstract data type using a doubly-linked list. The second module, `KeyedObject`, adds an integer `key` field to the `Node` type and implements a comparator, based on key values, for `Node` objects. The example also instantiates `CellList` as a static copy of `DLL` and `KeyedObject` as a static copy of `KeyedCell`.

2.1.1 Explicit module definitions

Developers may define Hob modules either explicitly or by static instantiation. Line 1 starts the explicit declaration of the `DLL` module with the line `impl module DLL`. Our example also contains (starting on line 85) an explicit declaration of the `KeyedObject` module. Implementation modules contain type and variable declarations as well as imperative code, organized into a set of procedures.

2.1.2 Static module instantiation

The other mechanism for creating a module is to instantiate it, at compile time, from another module. Static instantiation creates a fresh copy of a pre-existing module;

the new module shares no state with the old module. Lines 84 and 97 declare static instantiations of the `DLL` and `KeyedObject` modules. On line 84, the developer states that the program contains a `CellList` module which is an identical copy of the `DLL` module, except that instances of the `Node` type are to be replaced by instances of the `Cell` type. The `CellList` is therefore a doubly-linked list of `Cell` objects. Similarly, line 97 declares that the program contains a `KeyedCell` module.

In general, a static module instantiation, *e.g.* `impl module m = M with t <- T`, declares that module `m` instantiates module `M`, substituting instances of modules or formats `T` for modules or formats `t` from the original declaration. Hob processes static instantiations by creating a separate internal copy of the instantiated module with the declared substitutions; this treatment is essentially macro expansion.

2.1.3 Type and variable declarations

Our example declares the `Node` datatype in two parts using Hob's *format* construct. Formats allow different modules to each—independently—contribute fields to a datatype. The `DLL` module contributes (on line 2) the `next` and `prev` navigation fields to the `Node` datatype, which are used to form the tree structure. The `KeyedObject` module then contributes the `key` data field. Each of these modules acts independently of other modules in adding fields. Within the code of the `DLL` module, only the `next` and `prev` fields are in scope. The `key` field is out of scope for code belonging to the `DLL` module and may not be accessed from that module.

The format mechanism identifies a field by its name and the name of the contributing module. This enables different modules to use the same name for a field without conflicts, which is especially useful in the presence of static instantiation. Each of the different instantiations of a module will have its own copy of the fields that it is contributing.

Note that the use of formats to encapsulate fields, not objects, enables our analysis plugins to go beyond the ability of standard encapsulation systems to reason modularly about the heap: multiple modules can have pointers to the same object (unlike in most other encapsulation systems) and yet still know that the fields that they have contributed to that object are unmodified by the other modules in the program. The runtime system compiles an object's complete type description by aggregating all of the distributed type declarations; this aggregated description is irrelevant to Hob's static analysis and invisible to the developer.

Line 3 declares a `root` module variable for our doubly-linked list, which enables the procedures in the module to access the heap objects representing the list. The Hob runtime system initializes this variable to `null` upon program start, and the variable exists for the lifetime of the program. However, only procedures belonging to the `DLL` module may access this variable. The `CellList` instantiation creates a distinct `root` reference, which points to a `Cell` object after substitution. Contrast global variables with local variables, as declared for instance on line 52; local variables are allocated upon entry to their declaring procedure and exist only during that procedure's lifetime.

2.1.4 Procedures

The `DLL` module contains procedures to remove and add an element from the doubly-linked list, a procedure to test an object's membership, a procedure that returns the first element of the list and one that removes the first element of the list, a procedure that tests list emptiness, and finally a procedure to clear all elements from the list. This subsection briefly describe each procedure in the `DLL` module.

Embedding information for analysis plugins

The Hob system analyzes each procedure in the program using an analysis plugin. Two of the analysis plugins in our system are the flags plugin, which we designed as a lightweight analysis for client code—code that accomplishes tasks by invoking procedures in other modules—and the Bohne plugin, which uses field constraint analysis to verify code that manipulates linked heap data structures.

Static analysis techniques can often benefit from additional developer-provided annotations. The Hob implementation language contains three ways for developers to embed information for an analysis plugin directly in the program code. We support the use of loop invariants, `assert` statements and `assume` statements. These mechanisms have no run-time effect. Instead, when it encounters an invariant, `assert` or `assume` statement, the Hob analysis engine transmits the annotation to an analysis plugin.

Because loops potentially execute an unbounded number of times and static analyses are expected to terminate in finite time, the static analysis of loops is always challenging. Hob allows developers to provide loop invariants, which help plugins efficiently reason about the behaviour of loops. Of course, many analyses are able to automatically synthesize loop invariants from the procedure specifications and implementations. Note that when loop invariant inference techniques do fail, it is generally an open problem to effectively communicate to the developer the reasons which caused the inference to fail.

An `assert` statement contains a fact which the analysis engine must statically verify. This differs from the usual meaning of `assert`, which asks the runtime environment to dynamically check the validity of the assertion. We found that assertions were a useful form of communication between the developer and the analysis. In particular, assertions allowed the developer to query the analysis plugin and discover its abilities and limitations.

An `assume` statement is another mechanism for developers to pass information to analysis plugins. Unlike `assert` statements, which ask a plugin to verify that a statement is true, `assume` statements tell an analysis plugin that a given fact holds (without verification). Often, developers understand more about how a program works than a particular analysis plugin can deduce; for instance, the developer may have some specific domain knowledge about the problem domain. The ability to transmit this domain knowledge to an analysis plugin can then be leveraged by the analysis for it to guarantee a desired data structure consistency property. Each analysis plugin may accept a different syntax for assumes, asserts, and loop invariants. Our example

presents both the flags (line 15) and the Bohne syntax (line 71) for these constructs.

`remove` procedure

The `remove` procedure (lines 5-10) uses the `prev` and `next` pointers to remove the given object from the linked list. First, `remove` handles the special case of removing the root of the list by setting `root` to `root.next` if the element to be removed is at the root of the list. Next, if the given object e has a non-null `prev` field, the `remove` procedure sets e's predecessor's `next` field to e's successor, and similarly with e's successor. Finally, the `remove` procedure ensures that the following invariant on `Node` objects continues to hold: an object is in the list if and only if its `next` and `prev` fields are both non-`null`. Chapter 5 describes this invariant, and other list invariants, in greater depth.

`removeFirst` procedure

The `removeFirst` procedure removes the first element (which is pointed to by `root`) from the linked list and returns it to the caller. The simplest way to remove a given element from a list is to use the `remove` procedure, as we do on line 16. Note the use of the `assume` statement on line 15.

`addLast` procedure

The `addLast` procedure navigates to the end of the linked list and adds the given object p to the end of the list. The `addLast` procedure first handles the special case of an empty list on lines 21–24. Next, `addLast` declares a local variable `r` which it uses to navigate to the end of the list. The analysis of this module uses the Bohne plugin in a mode that requires the developer to provide loop invariants, so lines 29–37 contain a loop invariant which is transmitted verbatim to the analysis plugin. Note that the `while` loop in the `clear` procedure did not require a loop invariant; the flags plugin used for that procedure can automatically infer loop invariants. Finally, once the variable r points to the end of the linked list, `addLast` sets the `next` field of `r` to the given object `p`, the `prev` field of `p` to `r` and the `next` field of `p` to `null`, preserving the list invariant on `null`-ness of fields which we've previously mentioned in the description of the `remove` procedure.

`clear` procedure

The `clear` procedure iterates through the elements of the list and removes them one by one. The `removeFirst` and `clear` procedures both make calls to other procedures in this module. Note that the target of these calls is known at compile-time, as the Hob implementation language does not include inheritance or dynamic dispatch.

Other procedures

The `contains` procedure iterates through the list looking for the given element. Note that `contains` uses `assert` statements. The `getFirst` procedure simply returns the root of the linked list, which is the first element of the list. Similarly, the `isEmpty` procedure tests `root` against `null`; equality indicates that the list is currently empty.

2.1.5 Executing Hob programs

Once a developer has produced a Hob program, he or she may want to execute this program. We have implemented two ways for developers to test and execute Hob programs: an interpreter and a source-to-source translator into Java. Both of these tools use the Hob infrastructure to create an abstract syntax tree from the source code. The interpreter directly executes the abstract syntax tree, whereas the compiler performs a simple translation of the abstract syntax tree into Java source code. The primary tasks of the Hob-to-Java compiler are to collect the distributed type declarations into traditional Java-style class declarations and to provide Java stubs for Hob library calls.

2.2 Implementation Language Grammar

Figure 2-4 presents the grammar for our core implementation language. An implementation module contains format declarations, module variables, and procedures. A format (format) describes a module's contribution to a concrete type. A module variable (var) contains a pointer to a heap object; module variables serve as persistent roots of data structures. A procedure (proc) contains a sequence of standard imperative statements.

The Hob implementation language's grammar has a built-in extension point: the A production allows developers to specify assertions, which are to be statically checked by analysis plugins, assumes, which are to be assumed by analysis plugins, and loop invariants. The executable code generator always ignores assertions, but each analysis plugin must check that all assertions can be guaranteed to hold at compile-time.

2.3 Operational Semantics

Figure 2-5 presents operational semantics for a simplified version of the Hob implementation language. For the purposes of the operational semantics, we assume that structured code has been converted to a control-flow graph by compilation and that expressions have been normalized into three-address code. These semantics enable us to precisely describe the task of an analysis plugin. The state of the heap is a pair $\langle s, H \rangle$, where s is a call stack of pairs $[p, r]$ and H is the garbage-collected heap. The call stack s consists of program counters p and activation records r. Note that the program counter contains static information about the program point: $\mathsf{pc}(p)$ points

```
1  impl module DLL {
2    format Node { next : Node; prev : Node; }
3    var root : Node;
4
5    proc remove(e : Node) {
6      if (e==root) root = root.next;
7      if (e.prev!=null) e.prev.next = e.next;
8      if (e.next!=null) e.next.prev = e.prev;
9      e.next = null; e.prev = null;
10   }
11
12   proc removeFirst() returns n : Node {
13     Node nn = root;
14     // assume statement is given directly to static analysis
15     assume "(nn' in Content) & card(nn') = 1";
16     DLL.remove (nn);
17     return nn;
18   }
19
20   proc addLast(p : Node) {
21     if (root==null) {
22       root = p; p.next = null; p.prev = null;
23       return;
24     }
25
26     Node r = root;
27     // first three lines are relevant to loop;
28     // remaining lines are general list invariants that we preserve
29     while "p ~= null & r ~= null & p = 'p & ~(p : 'Content) &
30            next p = null &
31            (rtrancl (lambda v1 v2. next v1 = v2) root r) &
32            (ALL v. ~(next v = p) & ~(next v = root)) &
33            (ALL v. (v : 'Content) <=>
34                rtrancl (lambda v1 v2. next v1 = v2) root v) &
35            (ALL x. x ~= null &
36                ~(rtrancl (lambda v1 v2. next v1 = v2) root x) -->
37                ~(EX e. e ~= null & next e = x) & (next x = null))"
38           (r.next != null) {
39       r = r.next;
40     }
41     r.next = p; p.prev = r; p.next = null;
42   }
```

Figure 2-1: Doubly-linked list implementation, part 1

```
43   proc clear() {
44     bool e = DLL.isEmpty();
45     while (!e) {
46       Node q = DLL.removeFirst();
47       e = DLL.isEmpty();
48     }
49   }
50
51   proc contains(e : Node) returns b : bool {
52     Node n = root;
53     while "e ~= null &
54            (rtrancl (% x y. next x = y) root n) &
55            (ALL x. (x : 'Content) <=>
56               rtrancl (% v1 v2. next v1 = v2) root x) &
57            (ALL x. next x = root --> root = null) &
58            ~((rtrancl (% x y. next x = y) root e) &
59            (rtrancl (% x y. next x = y) (next e) n)) &
60            (ALL x. x ~= null &
61            ~(rtrancl (lambda v1 v2. next v1 = v2) root x) -->
62            ~(EX e. e ~= null & next e = x) & (next x = null))"
63           (n != null) {
64       if (n == e) {
65         assert "rtrancl (% v1 v2. next v1 = v2) root e";
66         return true;
67       } else {
68         n = n.next;
69       }
70     }
71     assert "~(rtrancl (% v1 v2. next v1 = v2) root e)";
72     return false;
73   }
74
75   proc getFirst() returns e : Node {
76     return root;
77   }
78
79   proc isEmpty() returns rv:bool {
80     return root == null;
81   }
82 }
83
84 impl module CellList = DLL with Node <- Cell;
```

Figure 2-2: Doubly-linked list implementation, part 2

```
impl module KeyedObject {
  format Node { key:int; }

  proc equals(a : Node; b : Node) returns rv:bool {
    return a.key == b.key;
  }

  proc lessthan(a : Node; b : Node) returns rv:bool {
    return a.key < b.key;
  }
}

impl module KeyedCell = KeyedObject with Node <- Cell;
```

Figure 2-3: Formats example

$$
\begin{array}{rcl}
Prog & ::= & M^* \\
M & ::= & \text{impl module } m\ \{F^*V^*P^*\} \mid \text{impl module } m = M \text{ with } T \text{ <- } T\ [, T \text{ <- } T]^* \\
F & ::= & \text{format } tid\ \{Fd^*\} \\
Fd & ::= & f : T; \\
V & ::= & \text{var } v : T; \\
P & ::= & [\text{private}]\ pn(fn : T[; fn : T]^*)[\text{returns } r : T]\ \{\ \text{Ld}^*\ \text{St}^*\ \} \\
Ld & ::= & T\ l; \\
St & ::= & \{St\} \mid E_l{=}E; \mid [m.]\ pn(E) \mid \text{return } [E]\ \mid \\
 & & \text{if } (B) \text{ then } St_1 \text{ else } St_2 \mid \text{while } [A]\ (B)\ St \mid \\
 & & \text{assert } A \mid \text{assume } A \\
E_l & ::= & l \mid l.f \mid v \\
E & ::= & E_l \mid \text{new } t \mid \text{null} \mid [m.]pn(E[, E]^*) \\
T & ::= & \text{int} \mid \text{bool} \mid \text{float} \mid \text{string} \mid \text{char} \mid \text{byte} \mid tid \\
A & - & \text{analysis plugin-specific syntax for asserts, assumes and loop invariants}
\end{array}
$$

Figure 2-4: Grammar for Hob implementation language

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

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

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

Figure 2-5: Operational semantics for implementation language

to the control-flow graph node to be executed, while $\mathsf{mod}(p)$ indicates the module to which $\mathsf{pc}(p)$ belongs.

The heap contains three types of tuples. These tuples track module variable contents, field contents, and local variable contents. We write that H contains triples $\langle m, v, o\rangle$ to indicate that module variable v in module m points to 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.

In the Hob implementation language, module variables are always initialized to a default value appropriate to their type. Numeric variables are initialized to 0, `bool` variables to `false`, `string` variables to the empty string, and reference variables to `null`. Chapter 4 will describe how Hob's static analyses enforce the Hob stationarity condition by using these known initial values for concrete variables to prevent a program from carrying out unintended modifications to its abstract state. This set stationarity condition is central to Hob's ability to carry out modular verification.

2.4 Discussion

We chose to design our own implementation language to enable the best possible fit between our specifications and implementations. Our use of a custom implementation language enabled us to experiment with language design issues.

Our custom implementation language allowed us to experiment with the format construct for distributed type declarations. We found that formats aided the verification of our benchmark programs by enabling the modular analysis of programs even when these programs share objects between different modules.

The Hob implementation language requires modules to be statically instantiated. Because of static instantiation, Hob programs contain a finite number of modules. The design of Hob's specification language then ensures that each module contains a finite number of specification-level sets. This implementation language feature simplified the specification language, since it implies that the specification language only needs to work with a finite number of sets.

The tradeoff involved in using our own implementation language was that we had to port benchmark programs to our Hob language. We felt that this price was not overly onerous, especially given that we also had to provide program specifications. One design decision that was quite useful in the porting process was the choice of a subset of the Java statement syntax for Hob implementation-language statements. This decision also simplified the compilation of Hob benchmarks to Java source code for execution. While our implementation language is a Java subset at the statement level, the Hob approach provides developers with a different high-level structuring mechanism than Java does. In particular, the Hob implementation language expects programs to be structured as a collection of modules (and, for specification purposes, scopes, as described in Chapter 3.4.2). Although techniques for writing specifications for Java programs do exist [18], we felt that it was appropriate for the Hob system to use a simpler and more direct specification approach which avoids the issues involved in reasoning about specifications in the presence of exceptions and inheritance.

2.4.1 Implications of encapsulating fields

Encapsulation is critical to any modular verification effort, since it converts sound reasoning about a part of the program into sound reasoning about the whole program by showing that the rest of the program does not affect the property of interest. Unlike many standard encapsulation mechanisms [15, 11], our format mechanism works by encapsulating fields, not objects. Because only the declaring module may access the fields that it has contributed to an object, formats enable analysis plugins to reason about the contents of a field by analyzing only the module that defines the field. In particular, analyses need not analyze any other modules that may access the same objects, even though the modules may mutate shared objects. Our type system guarantees that accesses to shared objects operate on disjoint parts of these objects, so that there is no interference between modules. Formats therefore enable modules to share objects and yet do not prevent the modular analysis of the modules that do share objects. Distributed type declarations were first introduced in [14],

and AspectJ's intertype declarations allow developers to write distributed type declarations today. Distributed type declarations are clearly useful in the context of aspect-oriented programming, since they enable developers to associate data with the program code, which can be scattered around arbitrarily for aspect-oriented programs. To our knowledge, formats are a novel application of the idea of distributed type declarations to the modular verification problem.

2.4.2 Implications of static instantiation

Typically, one of the most difficult issues involved in reasoning about programs is in reasoning about how they access, and modify, a statically unbounded heap; some finitization of the program state is required. We felt that the Hob system had to support reasoning about unbounded heaps, since data structure implementations are typically engineered to work with unbounded numbers of objects. Hob therefore allows developers to use an unbounded number of data objects in programs. The static instantiation mechanism, however, encourages developers to structure their programs so that a finite number of named sets suffices to reason about the program, thereby simplifying the task of stating and verifying data structure consistency properties. In particular, the static instantiation mechanism enables developers to define data structures once and to use these data structures as needed, without forcing implementations that require the specification language to handle an unbounded number of sets.

In the Hob system, program modules must be either explicitly declared or statically instantiated. Each static instantiation creates exactly one additional program module. The total number of program modules in a Hob program is therefore finite and known at compile-time. Furthermore, each module's specification may only declare a finite number of sets. Hob programs therefore have a finite number of specification-level sets, and each set in the program has a statically determined name, which is assigned by the developer. The set specification language enables analysis plugins to verify that developer-provided constraints on named sets continue to hold throughout the program's execution and that procedures carry out changes to set memberships as stated in their specifications.

While the Hob system bounds the number of sets, it does not bound the number of objects in each set. Modules may *create* arbitrary data structures on the heap. But they may only *specify* design-level properties for a finite number of sets, where these sets are somehow related to the data structures on the heap.

As an example, consider a program which processes a sequence of requests and associates a response—in the form of a set of objects—to each request. This program could be implemented in the Hob implementation language, but the specification would not be able to directly represent the set of response sets. One workaround is to focus attention on only one response set at a time.

Specification-level sets may have a statically unbounded number of members, and the Hob framework gives analysis plugins complete latitude in assigning objects to sets. Our Hob modular verification approach succeeds in part because analysis plugins never need to know about how other analysis plugins assign membership for

their sets. In the Hob system, each plugin is only responsible for reading set specifications for external modules, and does not need to inspect the external modules' implementations.

Chapter 3

Hob Specification Language

In this chapter we explain how developers can specify data structure consistency properties for the Hob system to verify. Hob supports several different types of specifications, described below.

Procedural (local) specifications. At the most basic level, developers may provide interfaces for procedures in terms of preconditions and postconditions. The Hob system allows developers to provide this information in an abstract set specification language. Developer-provided abstraction sections connect these abstract set specifications with the concrete implementations we described in Chapter 2.

Specifications of global properties. Hob is also able to verify global data structure consistency properties. Global data structure consistency properties relate states of different program modules. For instance, modules `A` and `B` may maintain sets that are always disjoint (except possibly while `A` and `B` are executing). Global properties therefore enable developers to state and verify relationships between parts of a program analyzed using very different techniques. It is theoretically possible to manually embed these properties into procedure specifications. However, such a manual embedding would impose a heavy burden on the developer and greatly affect the maintainability of program specifications. The Hob system therefore also supports two higher-level mechanisms that help developers state and verify these global consistency properties: scopes and defaults. These mechanisms do not impose any additional requirements on the specific analyses used by the Hob system; instead, Hob desugars these mechanisms into local specifications.

Other types of specifications. The Hob system relates the abstract set specifications with concrete program states using abstraction functions and representation invariants. For instance, a linked list module may export a set `Content` representing the objects in the linked list, that is, the objects reachable from the root of the linked list through `next` fields. These abstraction functions and representation invariants are an additional form of specifications which are visi-

ble exclusively within their defining modules. Chapter 5 describes these internal invariants in more detail.

This chapter will discuss the specification language for local and global properties. Both of these properties use formulas in the boolean algebra of sets to describe desired properties of the abstract program state.

3.1 Example: Doubly-Linked List Specification

Figure 3-1 contains a complete example of a specification for the doubly-linked list `DLL`, which we presented earlier in Figures 2-1, 2-2, and 2-3. Figure 3-1 also presents a static instantiation of the `DLL` list specification module into a specification for the `CellList` module.

In general, modules contain declarations for 1) program code and 2) program data. Chapter 2 described how implementation modules contain procedure implementations (written in the Hob imperative language) and concrete global variables. Specification sections contain analogous declarations for procedure specifications and abstract specification variables.

3.1.1 Specification module definitions and instantiations

Specification sections of Hob modules, like implementation sections, can contain either explicit module definitions or static module instantiations. Line 1 declares that the specification of module `DLL` follows, and line 37 declares that module `CellList` is a static instantiation of the `DLL` module which contains `Cell` objects rather than `Node` objects.

3.1.2 Specification variable definitions

Hob specification sections describe the abstract state of the module using specification variables. These specification variables can be either sets or boolean variables. Hob's set-typed specification variables do not exist at runtime. Instead, they are used exclusively in module specifications to abstractly describe the contents of data structures (as sets of objects) and hide implementation-level issues of data representation. Line 3 declares the `Content` specification variable, which contains a set of `Node` objects. (Line 2 informs the specification parser that the `Node` type will be used in this module's specifications; implementations of `DLL` will use the format construct to define the `Node` type.) Other modules may reference this `Content` set as `DLL.Content`, and the static instantiation on line 37 creates set named `CellList.Content`. For most modules, abstract boolean variables are linked to concrete boolean variables via the identity map.

The specification section does not include any information on the concrete meaning of the abstract sets that it uses. It is the sole responsibility of the abstraction section to provide a definition for a module's abstract sets (by relating concrete program

states to abstract sets). Our `Content` set, for instance, is defined in its abstraction section to be the set of objects reachable from the `root` module variable through `next` fields. Since this definition is completely irrelevant to any clients of the `DLL` module, the Hob system hides a module's set definitions outside that module.

3.1.3 Procedure definitions

The `DLL` specification module primarily contains procedure specifications for the `remove`, `removeFirst`, `addLast`, `clear`, `contains`, `getFirst`, and `isEmpty` procedures. Analysis plugins are responsible for verifying that each procedure's implementation conform to its specification. Procedure specifications contain a `requires` clause constraining the states in which it is legal to call a procedure, a `modifies` clause giving the sets which are potentially modified in the procedure and its transitive callees[1], and an `ensures` clause guaranteeing certain constraints on the program state upon return from the procedure. We describe each procedure specification in turn.

`remove` procedure

The `remove` procedure removes a given object from the set maintained by this module. It requires that the object already belong to the set and guarantees that the object is no longer in the set upon return. More precisely, any successful call to `remove` will require that, prior to the call, `e` must be non-null and must belong to the `Content` set. The procedure specification also states that `remove` modifies the `Content` set and that the set `Content'`, which denotes `Content` upon return from `remove`, contains the objects in `Content` minus the `e` parameter.

Our specification language treats procedure parameters as sets. If a parameter contains `null`, then we represent it with the empty set, and if it points to a heap object, then we represent it by a set with cardinality 1. Therefore, the constraint `card(e)=1` in the `requires` clause ensures that `e` is non-null.

`removeFirst` procedure

From the set specification point of view, the `removeFirst` procedure picks an arbitrary element from the nonempty `Content` set, removes it, and returns it to the caller. More precisely, the precondition `card(Content) >= 1` states that `Content` must be nonempty; the postcondition `card(n')=1` ensures that the return value is not null; and `Content' = Content - n'` states that the `Content` set upon return is the same as the `Content` set upon entry minus the removed object `n`, and that the object `n` belonged to `Content` prior to the call to `removeFirst`. Note that the set specification for the `removeFirst` procedure does not specify that the return value `n` was the first element of the list. Our set specification abstracts away from such details.

[1] While the Hob analysis tool requires procedures to declare sets modified in transitive callees in `modifies` clauses, a simple preprocessor can collect sets modified in transitive callees and add them to `modifies` clauses.

addLast procedure

This procedure adds the parameter `p` to the `Content` set. The specification states that prior to a legal call to `add`, the parameter `n` must be non-null (`card(n)=1`), and that `n` must not belong to `Content`. The specification also declares that this procedure modifies only the `Content` set. Finally, the specification declares that, upon return from `add`, the set `Content'`, which denotes the state of `Content` after returning from `add`, contains the objects initially in `Content` plus the given object `n`. Once again, note that the order of elements of the linked list is abstracted at the level of the set specification.

clear procedure

The `clear` procedure modifies the `Content` set by removing all elements from this set. In particular, the postcondition states that `Content` is empty upon return: `card(Content') = 0`.

Other procedures

The `contains` procedure presents the use of boolean return values in specifications. Given a non-null `e` parameter, `contains` returns true if and only if `e` is in the `Content` set. Note that this procedure does not modify any abstract state. The `getFirst` procedure returns an element belonging to the `Content` set. The `isEmpty` procedure is useful for guarding calls to procedures that require `Content` to be nonempty.

3.2 Example: Global Properties (Scopes)

Section 3.1 explained how the Hob system allows developers to state specifications for program modules. These specifications enable developers to state requires, modifies and ensures clauses for a single procedure at a time.

3.2.1 A global invariant

Some program properties involve sets belonging to multiple modules. Consider three modules, `Worker`, `Inbox` and `Outbox`. The `Worker` module maintains a set of jobs `Worker.Jobs`, while the `Inbox` module maintains a set of input jobs `Inbox.Input` and the `Outbox` module maintains a set of output jobs `Outbox.Output`. These modules need to work together to preserve the following invariant `I`:

```
I: Worker.Jobs = Inbox.Input + Outbox.Output
```

The `Worker` module guarantees that the invariant is preserved by properly coordinating updates to the `Jobs` set with calls to the `Inbox` and `Outbox` modules. The first responsibility of the analysis, in terms of verifying the invariant, is therefore to verify that the procedures in the `Worker` module preserve the invariant (and that the invariant holds in the program's initial state). Note that `Worker` may temporarily

```
1  spec module DLL {
2    format Node;
3    specvar Content : Node set;
4
5    proc remove(e : Node)
6      requires card(e)=1 & (e in Content)
7      modifies Content
8      ensures (Content' = Content - e);
9
10   proc removeFirst() returns n:Node
11     requires card(Content)>=1
12     modifies Content
13     ensures card(n')=1 & (Content' = Content - n') & (n' in Content);
14
15   proc addLast(p : Node)
16     requires card(p)=1 & not (p in Content)
17     modifies Content
18     ensures Content' = Content + p;
19
20   proc clear()
21     modifies Content
22     ensures card(Content') = 0;
23
24   proc contains(e : Node) returns b:bool
25     requires card(e) = 1
26     ensures (b' <=> (e' in Content));
27
28   proc getFirst() returns e:Node
29     requires card(Content)>=1
30     ensures card(e')=1 & (e' in Content);
31
32   proc isEmpty() returns rv:bool
33     ensures rv' <=> (card(Content) = 0);
34 }
35
36 spec module CellList = DLL with Node <- Cell;
```

Figure 3-1: Doubly-linked list specification

violate the invariant; our analysis simply needs to verify that the invariant is restored upon exit from `Worker`.

Because the design of the `Worker`, `Inbox` and `Outbox` modules relies on the `Worker` module to properly coordinate accesses to the `Inbox` and `Outbox` modules, any direct calls to `Inbox` and `Outbox` may cause the invariant to be permanently violated. Because only the modules `Worker`, `Inbox` and `Outbox` may directly modify the sets involved in I, external modules can violate the invariant only by calling `Worker`, `Inbox` or `Outbox`. The second responsibility of the analysis is therefore to prohibit direct calls to `Inbox` and `Outbox`; all calls to `Inbox` and `Outbox` must go through `Worker`.

In summary, to prove that invariant I holds, the Hob system needs to verify that the invariant holds initially, that the `Worker` module preserves the invariant, and that calls to `Inbox` and `Outbox` all originate from the `Worker` module. Together, these conditions enable an induction on program traces which permits the analysis to safely conclude that I holds upon each entry to procedures in the `Worker` module.

3.2.2 Specifying global invariants

More generally, the Hob system needs the following (developer-supplied) information to attempt to verify any invariant I: a set of modules where I may temporarily be violated; the set of exported modules which are responsible for ensuring that the invariant holds upon exit; and (of course) the invariant I itself. The developer expresses this information by specifying a *scope*.

Figure 3-2 presents the definition of our example scope. Line 1 states that the scope is named `W`. Line 2 of the scope definition states that `Worker`, `Inbox` and `Outbox` are the `modules` of scope `W`; the invariant may be temporarily violated inside these modules. Line 3 declares that the scope `W` `exports` the `Worker` module. This declaration instructs the Hob system to assume that the scope invariant holds upon entry to `Worker` and to show that the invariant is always ensured upon exit from `Worker`. Only modules that belong to the scope may invoke procedures in the non-exported `Inbox` and `Outbox` modules of the scope. Line 4 states the invariant itself using the `invariant` keyword.

Figure 3-3 illustrates the scope `W` and a module which calls `W`. In our example, the invariant I may be temporarily violated in the `Inbox`, `Outbox` and `Worker` modules. In other words, the scope `W` encapsulates these modules; we say that these modules *belong* to the scope. The `Worker` module ensures that the invariant holds upon exit from its procedures, so the scope `W` exports `Worker`. The scope invariant states that the set of jobs `Worker.Jobs` is equal to the union of the sets `Inbox.Input` and `Outbox.Output`. Note that Figure 3-3 also presents an extra module, `Server`, which invokes a procedure inside scope `W` from outside the scope. The `Server` module may only call the exported `Worker` module and not the `Inbox` or `Outbox` modules.

```
scope W {
  modules Worker, Inbox, Outbox;
  exports Worker;
  invariant Worker.Jobs = Inbox.Input + Outbox.Output;
}
```

Figure 3-2: Scope invariant example

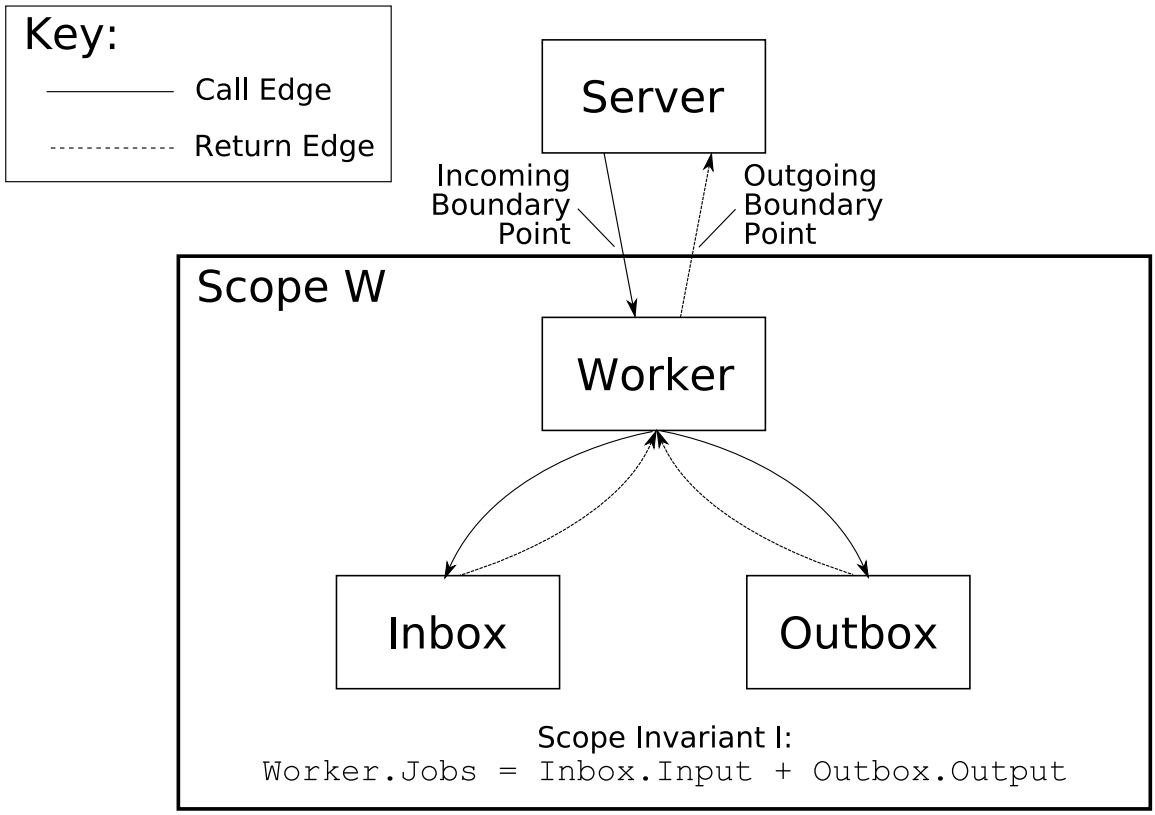

Figure 3-3: Illustration of scopes example

3.2.3 Verifying global invariants

We next discuss how the Hob system establishes whether or not a scope invariant holds. The Hob system checks that I holds in the initial state of the program. When verifying the scope's exported procedures, the Hob system appends the scope invariant to the preconditions and postconditions of those procedures. By conjoining the invariant to postconditions of exported procedures, Hob ensures that exported modules meet their responsibility of ensuring that scope invariants hold when exiting a scope. When a caller outside the scope invokes an exported procedure, the caller is responsible for ensuring that the exported procedure's precondition holds. However, the Hob system does not require that external callers show that the invariant holds. Because Hob checks scope invariants upon exit from a scope and because Hob checks that I holds in the program's initial state, programs may safely assume that the invariant holds whenever entering the scope.

Note that scope invariants are not actually appended to preconditions and postconditions; in particular, the Hob system can hide scope invariants from calling procedures. Neither the analysis system nor the developer need to explicitly write out a scope invariant outside the scope.

3.2.4 Specification aggregation

Consider a program which maintains an invariant I. Without the scope invariant mechanism, the developer would have to explicitly include the invariant I throughout the preconditions and postconditions of the entire program (except for when it is temporarily violated). We call this the *specification aggregation* problem. Specification aggregation causes a program's top-level modules to accumulate invariants from all of its worker modules. We expect that the number of invariants would grow roughly linearly with the size of the program, so the total annotation burden would grow quadratically across the program.

In this case, the scopes mechanism solves the specification aggregation problem by automatically conjoining the invariant I to the appropriate set of exported modules. Because the Hob system automatically conjoins I at the appropriate points in the program, the developer only needs to state I once, in the scope declaration, rather than throughout the text of the program. This replaces the quadratic number of annotations woven throughout the program text (namely, the number of invariants times the number of procedures) by a linear number of annotations (since each invariant is only stated once, the overall annotation burden due to invariants is linear).

The defaults mechanism enables developers to give names to properties; defaults could be used to simulate part of the functionality of the scopes mechanism and would somewhat mitigate the specification aggregation problem. However, scopes, when applicable, have two advantages over defaults: 1) scope invariants do not need to be explicitly added to preconditions and postconditions outside the scope (which reduces the burden on analysis plugins); and 2) scopes act as a program structuring mechanism, in that they enable developers to forbid calls to the interior of a scope.

The Hob system properly handles reentrant calls in the presence of scopes. A

reentrant call occurs when a module inside a scope calls outside the scope, and the callee subsequently calls back into the scope. Hob requires that scope invariants hold at reentrant call sites, and assumes that they hold upon return.

3.3 Example: Global Properties (Defaults)

Scope invariants are program properties that hold in most program states. In our web server benchmark, the `Config` module manipulates configuration data for the webserver and maintains a boolean specification variable `ready`, which is true as soon as the module has been initialized and throughout most of the program's execution. Unfortunately, since the variable `ready` is false until the `Config` module has been initialized, `ready` does not hold in the program's initial state. Scope invariants, however, must be true initially, so `ready` is not a suitable scope invariant. We therefore invented the *default* mechanism for properties that hold in many different program states and yet are not suitable for use as scope invariants. Defaults and scopes work well together to enable developers to concisely and accurately specify program properties.

Developers specify defaults by giving three pieces of information to the Hob system: a set of procedures to which the default is applicable; a name for the default; and the clause to be applied. Hob uses the notion of a *pointcut* to specify the region where the default is to apply; a pointcut simply names (syntactically) the procedures or modules to which the default should apply. Procedures may use a default's name for fine-tuning of its applicability: if a procedure does not need a particular default, then the procedure can suspend the default. If a default is applicable to a procedure, then Hob conjoins the default's clause to the procedure's precondition or postcondition, as specified in the pointcut.

Figure 3-4 presents a pair of defaults drawn from our web server example. The `StringTokenizer` module (not shown) maintains a specification variable `S`. The default `I` (for 'initialized') states that the `ready` boolean variable is `true` at all preconditions except for those in `init` procedures (pointcut clause `not proc *.init()`) and except before the procedure `add` in module `HostList` (pointcut clause `not proc HostList.add()`. The default `S` states that the `StringTokenizer.S` set is empty at all preconditions of procedures in the `Config` module. Note also that the `init` procedure explicitly suspends the `I` default. In this particular case, the suspend and the default pointcut have the same effect, and the developer may freely choose one mechanism or the other (or both).

3.4 Specification Language Grammar

This section presents the Hob specification language grammar and the scopes and defaults extensions to the Hob specification language. The core specification language uses formulas of the boolean algebra of sets (with cardinality constraints) for requires and ensures clauses for procedures, which are organized into modules. Hob represents

```
1  spec module Config {
2      specvar ready:bool;
3
4      default I : pre(not proc *.init() && not proc HostList.add()) = ready;
5      default S = card(StringTokenizer.S) = 0;
6      proc init(argv:string[]) suspends I requires not ready
7        modifies StringTokenizer.S, Mimetypes.init, ready
8        ensures ready';
9      proc getPort() returns p:int ensures true;
10 }
```

Figure 3-4: Defaults example

the abstract state of a module's encapsulated data structures using specification-level sets.

3.4.1 Core specification language

In this section we describe Hob's set-based specification language. Because all analyses ensure that implementations conform to specifications expressed in this specification language, Hob's specification language enables different analyses to communicate in terms of a common set of program properties. Our core specification language allows developers to express specifications at the level of procedures.

Figure 3-5 presents the complete syntax for the core module specification language. A specification module consists of type and set declarations, procedures, and module specification-level invariants. Type declarations (`format t`) declare the types which will be used in set declarations and procedure parameters. The set declarations (`specvar S`) name the sets over which the boolean clauses in the specifications will range. Boolean variable declarations (`specvar` n_b) similarly name the boolean variables which will be used in specifications. A specification for procedure pn begins with an optional suspends clause (for defaults, discussed later), a requires clause expressed in boolean algebra with cardinality constraints, continues with a modifies clauses, and concludes with an ensures clause. Module invariants in specification sections are a special case of scope invariants (as described in Section 3.4.2) which apply to the declaring module. Module invariants differ from abstraction section invariants (Chapter 5) in that they are expressed in the common set specification language, rather than in an analysis plugin-specific notation.

The expressive power of boolean clauses B is the first-order theory of boolean algebras, where variables range over sets declared in some module in the program. The first-order theory of boolean algebras is decidable [88, 53], and we use this fact to compute whether implication holds between boolean clauses as well as to perform dataflow analysis in the flags analysis, as described in Chapter 6.

Boolean clauses operate on set expressions SE. A set expression may name sets S and procedure parameters p; in Hob, primed sets S' denote the contents of a set upon

$$
\begin{array}{rcl}
M & ::= & \mathsf{spec\ module}\ m\ \{(\mathsf{format}\ t)^*\ (\mathsf{specvar}\ (n_b : \mathsf{bool} \mid S : t\ \mathsf{set}))^*\ P^*\ I^*\} \\
P & ::= & \mathsf{proc}\ pn(p_1 : t_1, \ldots, p_n : t_n)\ [\mathsf{returns}\ r : t] \\
 & & [\mathsf{suspends}\ d^+]\ \ [\mathsf{requires}\ B]\ \ [\mathsf{modifies}\ S^+]\ \ \mathsf{ensures}\ B \\
I & ::= & \mathsf{invariant}\ B \\
B & ::= & n_b \mid SE_1 = SE_2 \mid SE_1 \subseteq SE_2 \mid \mathsf{card}(SE){=}k \mid \mathsf{disjoint}(SE_1, SE_2) \\
 & \mid & B \wedge B \mid B \vee B \mid \neg B \mid \exists S.B \mid \forall S.B \\
SE & ::= & \emptyset \mid p \mid [m.]\ S \mid [m.]\ S' \mid SE_1 \cup SE_2 \mid SE_1 \cap SE_2 \mid SE_1 \setminus SE_2
\end{array}
$$

Figure 3-5: Syntax of the Module Specification Language

return from a procedure in the context of requires clauses. Developers may combine set expressions using the set union, intersection and difference operators.

Requires/Ensures Clauses. Our specification language allows procedure effects to be specified using requires and ensures clauses in boolean algebra clauses B. When a procedure specification includes a modifies clause m, the Hob analysis framework adds some extra terms to the ensures clause e to give an effective ensures clause e_{eff}, which is used as the summary of that procedure's effects. In particular, in the presence of a modifies clause m, we use this augmented ensures clause to analyze the procedure:

$$e_{\mathit{eff}} := \ e \wedge \bigwedge_{S \notin m} S{=}\hat{S} \wedge \bigwedge_{n \notin m} n \Leftrightarrow \hat{n}$$

3.4.2 Scopes

Figure 3-6 presents the syntax of scope declarations. A scope declaration contains three parts: it declares a set of modules belonging to the scope; a subset of these modules—exported modules—which are visible outside the scope; and (optionally) a scope invariant, which is a formula preserved by the scope. Outside a scope, the scope's non-exported modules are invisible: modules which do not belong to a scope may not invoke procedures in, or refer to sets of, that scope's non-exported modules. Only the exported modules and their sets may be used outside the scope. The scope invariant is a formula which Hob verifies for the program's initial state and upon exit from the scope (assuming that the invariant always holds upon entry to the scope).

Handling Reentrant Calls. 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. When control reaches a reentrant call site, the scope invariant may be temporarily violated at that point. However, since the call site is a reentrant site, the flow of control may then reach a scope entry point again. At a scope's entry points, the analysis assumes that the scope's invariants hold.

Our system therefore requires scope invariants to hold at all reentrant call sites. Combined with the verification of scope invariants upon exit from a scope, this en-

sures that scope invariants always hold upon entry to a scope. 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 performed in the overall program verification described in Chapter 5.3, the *call reentrancy check*, ensures that the developer has correctly identified all reentrant sites.

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[2]. 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).

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. Private scope invariants are useful because they help the Hob system reduce the size of the overall analysis task. They are especially useful when the scope invariant mentions private sets: invariants on private sets should always be hidden.

Finally, the verification system assumes that the sets and boolean variables of a given scope invariant (and more generally, all sets and boolean variables defined in the modules in the scope) do not change across non-reentrant calls. Hob's set stationarity check ensures that only the procedures in the scope can affect the values of the sets and boolean variables of the invariant.

> **Set Stationarity Check:** A scope invariant may use only sets and boolean variables that are defined in the scope's modules.

Because of the set stationarity check, it is sufficient to verify that the invariant holds in the initial state and at scope exit points to ensure that the invariant always holds at scope entry points.

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. Such a call site is an

[2]The Hob system currently conjoins public scope invariants to all preconditions outside the scope. This is not necessary in general. For instance, a scope invariant mentioning modules `A` and `B` should not be conjoined to a precondition on sets in modules `C` and `D`.

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

Figure 3-6: Syntax of Scope Declarations

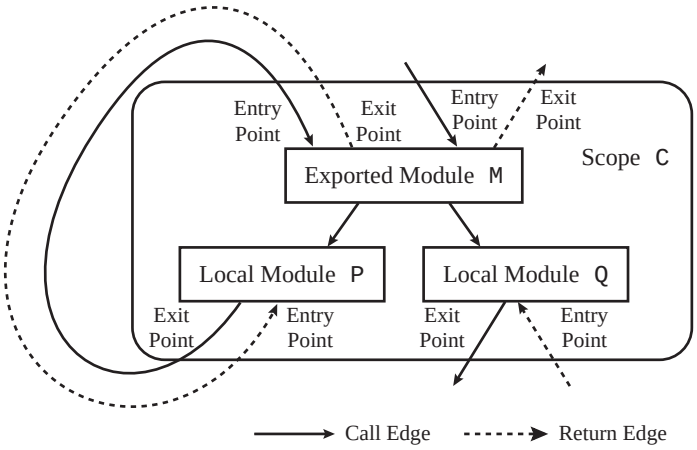

Figure 3-7: Scope Entry and Exit Points

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 3-7 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 non-exported 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 non-exported 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.

Controlling Access to Non-exported Modules. The scopes mechanism enables Hob to use properties of a program's structure to eliminate the need to check the associated scope invariants outside a scope. In particular, the Hob system only needs to ensure that scope invariants hold at certain key points, namely scope exit points. One key reason that this works is that a scope's exported procedures control the operation of non-exported modules: no non-exported module may be called from outside the scope. The Hob system ensures that non-exported modules remain private to the scope by using a *scope call check*, as described below.

Scope Call Check Consider a procedure call from module M to module M'. Then for each scope C that the target module M' belongs to, either: 1) M must also belong to scope C, or 2) M' must be exported in scope C.

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

Scopes and Set Visibility. The sets and boolean variables of non-exported 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 boolean variables from non-exported modules.

This design decision means that modifies clauses have a slightly different meaning in the presence of scopes with non-exported modules. Sets and boolean variables from non-exported modules will be absent from the modifies clauses of all exported procedures, even if the procedures may modify some of the sets or boolean variables. 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 boolean variables from the non-exported modules of the scopes to which the module containing the call site belongs.

General Modification Semantics A set S of module M' is *out of scope* for module M if there exists a scope C which does not export M', and M does not belong to C.

Consider a call from module M to procedure p of module M', and let set T of module M be out of scope for p.

1. If the call to p is labelled as a reentrant call (that is, if p includes a call back to the caller module M), then the caller must deduce, upon return from p, that T may be arbitrarily modified.
2. Otherwise, the non-reentrant call to p preserves the contents of set T.

It is sound to preserve out-of-scope sets T across non-reentrant calls: because T is defined in the calling module M, it may only be modified in M. Furthermore, since the call is non-reentrant, then T must be unmodified upon return from the call. Because the caller module's set T is out of scope for callee procedure p, the Hob system ensures that p does not explicitly mention the caller module's set T in its specification.

Verifying Scope Invariants. Having described what scopes do and how they structure the program, we next describe how Hob verifies that scope invariants hold. We have designed the Hob system so that Hob analysis plugins do not need to understand scopes or other global properties. This simplifies the design and implementation

of analysis plugins, which are solely responsible for verifying local data structure consistency properties. Briefly, Hob translates global scope invariants into requires and ensures clauses suitable for verification by analysis plugins.

- **Reentrant Call Sites.** Since potentially-reentrant sites are scope exit and entry points, the Hob framework conceptually adds an assert statement containing the invariants of all potentially-reentered scopes before that call site and an assume statement with the same invariants after the call site.

- **Private Scope Invariants.** Private invariants do not appear in formulas outside the scope. Private scope invariants are therefore conjoined to requires and ensures clauses for public procedures of exported modules when analyzing the bodies of these procedures. However, private invariants need not be conjoined to these procedures when checking validity of calls to those procedure. This is equivalent to adding an assume statement at the head of each exported procedure containing the scope invariant and an analogous assert statement at the tail of each exported procedure.

- **Public Scope Invariants.** Public invariants are known to hold throughout the program's execution, and can conceptually be conjoined to all preconditions and postconditions in the program outside the scope, as well as preconditions and postconditions of exported procedures. One possible optimization would conjoin public invariants to only those outside procedures that refer to sets and boolean variables used in the scope invariant.

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.

Expressive power of scopes.

Hob's scopes mechanism enables developers to specify invariants which hold across a set of modules. Scopes are more powerful than defaults: while defaults could conjoin invariants to appropriate program points, defaults do not enable the developer to forbid calls to internal modules. The scopes protection mechanism therefore increases the expressive power of the Hob language by enabling developers to ensure that, in the maintenance phase of program development, program modifications do

not inadvertently introduce calls to scope-internal modules which result in invariant violations.

Non-hierarchical program decompositions.

Our scopes mechanism furthermore enables a module to participate in multiple scopes simultaneously. This multiple participation enables modules to be grouped into scopes along orthogonal axes. By using scope invariants, developers can express properties that are common to multiple procedures belonging to multiple modules, providing a decomposition of the program layered on top of the module-based decomposition. The scope-based decomposition permits developers to encapsulate invariants that cut across modules. Scopes also enable developers to separate the underlying analysis task (as carried out, for each module, by Hob's various analysis plugins) from the set of program units that maintain a certain global invariant: many different analysis plugins can cooperate to establish a global invariant, as expressed in terms of a scope.

Invariants and Regions Where They Hold. Given any region of code expressed as a set of modules, and any invariant I, a developer can introduce a scope exporting these modules. This scope will serve to precisely indicate where the invariant I should hold, without imposing any unwanted additional constraints on the program structure.

Enforcing Arbitrary Calling Restrictions. 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.

Exposing Various Interfaces to a Module. 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 [39]. 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.

3.4.3 Defaults

The default construct enables developers to state that a specific property holds at a set of procedure preconditions and postconditions unless explicitly suspended. Devel-

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

Figure 3-8: Pointcut Language for Defaults

opers may specify the applicability of a default syntactically, by naming the modules and procedures to which the default should apply. Default declarations have the form,

$$\texttt{default } N(A_1, ..., A_k) : C = P \tag{3.1}$$

where N is the name of the default, the A_i are a set of optional parameter names, C is an optional pointcut specification (specifying where the property should be added), and P is a property expressed in the Hob set specification language. One common use of defaults is to capture initialization constraints, which always hold once a program has completed its initialization phase.

Our current 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 "**suspend** N".

Pointcut Specification Language. The two pieces of information defining a default are: (1) what is the property; and (2) where should it hold? Since Hob has a common set specification language to specify program properties, it makes a lot of sense for developers to use this set specification language to specify properties in defaults as well. Figure 3-8 presents the syntax for Hob's pointcut language, which enables developers to specify where a property should hold. The developer can use the pointcut language to identify a set of procedures S to which the default applies, then specify that the default applies to the preconditions (**pre** S), postconditions (**post** S), or both preconditions and postconditions (**prepost** S) of all procedures in S. The developer may select procedures by name, by membership in modules, or by membership in scopes. An omitted pointcut for a default specified inside a module indicates that the default should apply to all preconditions and all postconditions of all procedures of that module; for a default specified outside any module, an omitted pointcut means that the default should apply to all preconditions and postconditions in the program.

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 7.2), 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 constrains `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 can generalize to include arbitrary formula transformers that may transform formulas in more sophisticated ways. We have implemented one instantiation of such general formula transformers in the Hob system. However, 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 (and our current implementation does not guarantee a deterministic result in such a case). 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.

3.5 Discussion

In this section, we discuss various consequences of our particular choice of specification language and its features. We first discuss the *specification aggregation* problem that motivated our scopes mechanism. Next, we discuss the expressive power of the scopes mechanism, as well as the advantages and disadvantages of the defaults mechanism. We then move on to the general problem of choosing a specification language and justify our choice of a set specification language. Finally, we compare our experience with the Hob static analysis approach with the testing approach for the purpose of validating program properties.

3.5.1 Scopes and specification aggregation

Assume/guarantee reasoning, as used in the Hob system, comes at a cost: it requires specifications at boundaries of code fragments such as procedures. Consider a procedure p. Any caller of p must be able to guarantee that p's preconditions $r_1 \wedge r_2 \wedge \cdots \wedge r_n$ hold prior to its invocation. These preconditions can hold either because they are true in the program's initial state, or because they are guaranteed by the postcondition of a procedure which has been executed in the past; the preconditions, of course, must have not been subsequently violated. In principle, the developer must therefore thread conditions r_1 to r_n through all procedure preconditions and postconditions, up and down the call chain, from where they are established to where they are used. Additionally, any transitive callee q invoked from p adds its own specification burden to the preconditions of p, such that p might in fact specify preconditions r_1 through r_n *and* r_{n+1} through r_k.

Note that these preconditions may, in particular, propagate up the call chain to a procedure's callers. Of course, some callee preconditions must be established at caller sites; however, many callee preconditions are purely internal and should not be visible to the caller. Requiring callers to explicitly guarantee internal preconditions would often result in modularity violations: callers should not need to know about irrelevant details of a callee's internal state. Forcing the developer to constrain the callee's state at all callers makes reuse more difficult, since the caller must be aware of required (yet irrelevant) preconditions.

In general, developers must therefore either deal with a set of procedure postconditions, each of which potentially increases at least as fast as the size of the program; or choose some subset of these postconditions to manually propagate throughout the program specifications. If the subset is deficient, then (due to the limitations of assume/guarantee reasoning) Hob may declare that it is unsafe to call some needed procedure, or Hob may fail to prove some desired postcondition for the program as a whole. This phenomenon—the *specification aggregation* problem—forces the developer to include undesired, but mandatory, specification clauses representing future callee invariants. Such clauses cut across system specifications, yet are irrelevant to most program points: they should only appear at those program points which specifically need such clauses.

Our *scopes* mechanism was motivated by the specification aggregation problem. Scopes mitigate the cost of assume/guarantee reasoning: when providing specifications for a code fragment, the developer should only need to specify properties of that fragment. The developer should not need to specify any globally true properties which are irrelevant to that fragment: if the fragment cannot possibly affect the validity of the property, then the property will inevitably be preserved by the fragment. Scopes allow developers to specify regions in which globally true properties—*scope invariants*—are temporarily violated. Outside a scope, its invariant will generally be true.

Scopes combat specification aggregation by hiding irrelevant sets and clauses from callers. Furthermore, they enable the specification and verification of cross-module invariants by allowing developers to identify the subset of a program in which an

invariant is expected to hold. Scopes are key to our system's verification of invariants containing sets from different modules: by designating certain modules as public access points, we ensure that scope invariants always hold outside their declaring scope by verifying the scope invariant at each of a scope's exit points. Scopes also shield callers from irrelevant detail: only sets from exported modules may occur as free variables in specifications for modules in different scopes. This constraint 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 $\mathcal{C}$. In other words, procedure specifications omit all effects on sets that are private to a scope (a set is private to a scope if it is declared in a module that is not exported from that scope). Moreover, note that this irrelevant detail causes real problems for modularity. In the absence of this mechanism, a caller outside a scope would need to indicate (at the very least) that the callee's internal sets are non-deterministically modified, which is unreasonable because the outside caller has no way of knowing about the callee's private modules.

3.5.2 Advantages and disadvantages of defaults

Defaults are useful for several reasons: they reduce the size of program 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 (in situations where scopes are not applicable), and eliminate specification errors that would otherwise occur when developers inadvertently omit default properties. Developers often appear to unconsciously assume that a default holds (which 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. A disadvantage of using defaults is that when they do not hold and, for instance, cause a formula to become unsatisfiable, developers may find it difficult to debug the specification, since the offending clause was added by the default mechanism and is not immediately visible. Better tool support would mitigate this problem.

3.5.3 Implications of using a set specification language

We next discuss the advantages and disadvantages of using a set specification language to provide module interfaces. We first discuss the power of a set specification language and compare it to other possible specification languages. We then compare set specifications with less powerful alternatives, including type and typestate systems. We next justify our choice of a set specification language. The choice of a set specification language did have some drawbacks, and we outline some of them. Finally, we describe some advantages and limitations of using more powerful specification languages.

Expressive power of set specifications

Set interfaces lie somewhere in the middle of the spectrum of possible module interface languages. Less expressive interface languages include standard type systems, which fix the type of an object at instantiation time, and typestate systems [91, 90], which augment the fixed type of an object with a varying state component depending on the operations that have been performed on the object. More expressive interface languages could allow developers to specify relations between objects, as in the Jahob project. The interface language can be as detailed as the implementation language, and indeed, JML [12] permits developers to use full Java expressions in their specifications. Finally, one could permit the use of higher-order logic (as in, for instance, [42, 78]) in procedure interfaces. Each of the interface languages in this paragraph is strictly more powerful than the ones listed before it. The tradeoff is that a more powerful interface language is also more difficult to reason about; some interface languages will be undecidable. Furthermore, well-designed interface languages should enable developers to cleanly abstract away from the underlying implementation and state just the important properties of the system.

Less expressive specification languages

Our modular analysis approach needs more information than standard type systems make available, since these type systems do not permit developers to specify any but the most basic data structure properties. In particular, standard type systems are incapable of expressing the fact that objects move in and out of a program's data structures as the program executes. Typestate systems [91, 90, 24] do allow developers to express membership of objects in data structures and permit static analyses to check usage protocols, but they do not allow developers to discuss the contents of data structures in their interfaces. In other words, using a typestate system, it is only possible to discuss a program's abstract state one object at a time. For instance, developers can only verify that a given object does not simultaneously have typestates X and Y; typestate systems are not expressive enough for developers to state that no object has typestate X and typestate Y.

We next discuss the specification aggregation problem in the context of standard type systems and typestate systems. In standard type systems, the type of an object, as well as the set of type definitions, is fixed once and for all. Because type constraints cannot be violated in the course of a computation, specifications do not need to be reiterated up and down the call chain from where a property is established to where it is used.

However, in type systems which include subtyping, an object may be cast to a supertype and later back down to its actual type (the downcast problem). Standard type systems use run-time checks to ensure safety in the presence of downcasts. These run-time checks are much more tractable in the context of type systems than in the context of our set specification language, because a standard type system (which uses run-time checks) needs a much weaker safety property than does our set specification language: the fact that type definitions do not change implies that the program only

needs to verify the identity of the type, and not its definition. Recovery from type errors, however, is still challenging, since there might be no appropriate action when a precondition is violated. Parametric polymorphism enables developers to statically avoid the downcast problem, since the language will then keep the complete type information around; however, this leads to a restricted form of specification aggregation problem, because the type information must be woven through the program's execution. Because type information is more limited than our set specifications, the magnitude of the specification aggregation problem is smaller for standard type systems.

In typestate systems, objects can change typestate during the course of a program's computation. As with standard type systems, the program can easily verify at run-time that an object has the appropriate typestate. However, the fact that the program may carry out a typestate change using an alias of an object complicates static checking, and designers of typestate systems resort to various mechanisms to ensure safety, including linear type systems for objects which may change types [24]. In any case, the magnitude of the specification aggregation problem for typestate systems is similar to that for type systems.

Justification for a set specification language

Set specifications are particularly natural for developers to use because they enable developers to state object membership properties and relationships between data structures [59]. After all, many data structures are simply implementations of sets which optimize certain set operations. We feel that set specifications can express many key data structure properties and, in particular, consistency properties which relate the contents of different data structures. Such consistency properties are often crucial design properties for a system which ought to hold throughout its lifecycle; set specifications provide a concise and easy-to-understand way for developers to express and verify these properties. Our experience using set specifications has been positive.

Limitations of a set specification language

Note that our set specification language does not support the standard set theoretic construction of the integers, because our sets only contain uninterpreted elements. At a more practical level, we cannot express relations between objects in our system. For instance, our modelling of maps (*e.g.* hashmaps) can only discuss the set of objects which act as keys and the set of objects which act as values. While Hob can state and verify relationships between the set of keys and the set of values (for instance, no object should be both a key and a value simultaneously), Hob cannot state that a particular key is related to a particular value. Hob also cannot express properties of objects belonging to sets of sets. The set specification language does have the advantage of being decidable; the MONA tool [51] can decide formulas written in our set specification language.

Another limitation of Hob's specification language stems from the fact that our specification language supports only a bounded number of sets. While this language

design dramatically simplified the specification language and the resulting specifications, such a language design makes it difficult to specify properties of dynamically instantiable data structures like Java's `LinkedList` utility class. One potential solution is to use a more powerful specification language; relations enable the verification of instantiable data structures. Note that it is possible to work around the limitations of the specification language to some extent: developers could use an unbounded number of data structures in the implementation while only specifying properties of a bounded number of these data structures. Furthermore, it would be possible to "swap in" data structures and specify properties of only these "active" data structures. By carefully constructing the implementation, it would be possible to verify invariants that actually constrain an unbounded number of data structures.

More powerful specification languages

A specification language based on relations goes beyond our set specification language. In the above example, it would permit particular keys to be related to particular values. Binary relations are sufficient to encode sets and general n-ary relations. Because a relation-based specification language is more powerful, in general an interactive theorem prover might be required to reason precisely about interfaces expressed using relations. Going even further, a specification language which enables developers to state the full range of program properties, like JML [12], makes it tempting to express detailed implementation-level properties which ought to remain hidden to a module's clients. Such interfaces also are potentially as difficult to check conformance against as the original implementation, which would obviate the advantage of using a specification language to aid modular analysis.

3.5.4 Comparison: Static analysis and testing

In our experience, testing is a valuable complement to the static analysis provided by the Hob tool; since it is easy to test a program component, we found a number of straightforward errors using testing. Testing discovers many errors in implementations, and a well-tested implementation may well behave properly on the vast majority of (common) program inputs. However, Hob's static analyses guarantee that data structure consistency properties hold on all executions of a program, which is in general impossible to achieve using testing. One common weakness of testing, for example, is in detecting the faulty treatment of errors and exceptions.

Furthermore, the abstractness of Hob's specification language encourages developers to think at a higher level of abstraction and enables them to express deeper properties of programs. Such properties can easily be obscured in a program's implementation. At the implementation level, design information is hidden behind a mass of details, which are necessary for implementing the design, but not useful for understanding the underlying design. We believe that the set specification language exposes design information more effectively than imperative implementation languages, since set specification languages abstract away from the details of *how* the program carries out its tasks and instead say *what* the program does. This is especially true as a pro-

gram moves through its development lifecycle through the maintenance phase: the design information may become outdated, and the original developers may move on to other projects. The Hob system enables developers to use data structure consistency properties as verified documentation. Our analysis tool verifies that these properties hold, not just at any one point in the program's life, but throughout changes by successive developers, who may not understand the program's original design at all. Our experience with Hob suggests that it is capable of recording design decisions taken by the original developers and ensuring that this design information remains up-to-date.

Chapter 4

Hob Abstraction Languages

Hob specifications are written using the Hob specification language, which enables developers to express program properties by describing changes to abstract sets of objects. Each abstract set in a specification denotes a set of concrete heap objects. Hob abstraction modules enable developers to state abstraction functions, which define the contents of abstract sets in terms of concrete heap objects. Abstraction functions therefore provide a connection between specifications (which use abstract sets) and implementations (which manipulate the concrete heap). This connection enables both developers and the Hob system to reason soundly about an implementation in terms of its higher-level set specifications.

Procedure implementations assume that certain properties of the concrete state hold upon entry and guarantee that (potentially different) properties hold upon exit. There are two main types of such properties: invariant properties and preconditions/postconditions. In the Hob approach, developers specify preconditions and postconditions using the previously-described set specification language. Because these properties are expressed in terms of sets, they constrain the abstract program state at procedure entry and exit points.

Many implementations maintain specific constraints on the concrete program state. The Hob system allows developers to specify these concrete representation invariants in abstraction sections. Analyses may then use these representation invariants as they verify the procedure implementations. In particular, they may assume that the invariants hold at the beginning of each procedure and must guarantee that the invariants hold at the end of each procedure. (Hob also supports set specification-level invariants in specification sections).

Because developers use a variety of techniques to implement sets, the Hob approach supports arbitrary static analysis techniques for analyzing these techniques. In the Hob system, we have implemented a number of static analysis techniques. Each analysis verifies whether or not a class of implementations conform to their set-based specifications, using a specific class of abstraction functions customized for that analysis. We have designed the Hob system so that each static analysis only needs to process abstraction functions corresponding to the class of implementations that it is analyzing. All of Hob's static analysis techniques are implemented in the context of *analysis plugins*, which establish that procedure implementations conform to their

specifications. Furthermore, we have designed the Hob system to be extensible: researchers may add their own analysis plugins to verify new classes of implementations, and the Hob system enables researchers to use any static analysis techniques that are appropriate for a desired class of implementations.

4.1 Analysis Approach

The Hob system delegates the central data structure consistency property analysis task—a proof that a procedure's implementation conforms to its specification, as interpreted with the provided abstraction function—to a set of analysis plugins. The Hob system currently contains four analysis plugins: the flags plugin, the Bohne and PALE shape analysis plugins, and a theorem proving plugin.

- The Hob flags plugin supports set definitions stated in terms of object field values.

- The PALE and Bohne shape analysis plugins use the monadic second-order logic for their definitions. Note that this logic is more powerful than the first-order set specification language. Bohne additionally supports nondeterministic field constraints, which enable it to verify a broader class of data structures than the PALE plugin.

- The Hob theorem proving plugin enables developers to generate verification conditions which must be manually discharged using the Isabelle interactive theorem prover. (Note that Hob can support arbitrarily powerful abstraction functions—even ones that are based on undecidable logics—by relying on developers to manually discharge the resulting verification conditions. While such a strategy is always possible, the Hob approach generally focusses on applying static analysis techniques to the program verification problem).

Once the analysis plugins have verified a program's implementations, the Hob system must somehow combine the analysis results from the different analysis plugins. We have designed the Hob set specification language specifically to enable different analyses to communicate, and the developer always states procedure preconditions and postconditions in the common set specification language. This chapter describes Hob abstraction modules, which allow analyses to link the set specifications and implementations. Hob abstraction modules contain abstraction functions and invariants. Note that abstraction modules contain yet another kind of specification information, besides the procedure specifications, scopes, and defaults that we have seen in Chapter 3.

Analysis plugins communicate information in terms of the set specification language. Set definitions are always private to a module. There are therefore two implications for analysis interoperability: any analysis plugin only needs to 1) process specifications written in the common set specification language and 2) parse its particular syntax for abstraction functions. The Hob approach enables the modular

design and implementation of analysis plugins because plugins are not responsible for processing the abstraction functions used by other modules in the program.

Conceptually, an analysis plugin verifies local data structure consistency properties for a module M by first translating references to sets belonging to module M in a procedure's precondition and postcondition into the internal representation used by the plugin, adding the appropriate invariants, and finally verifying that the implementation satisfies the (translated) postcondition on all executions through the procedure, assuming that the (translated) precondition holds. At procedure call statements, the analysis converts the internal analysis representation back into the common set specification language and verifies that the precondition of the callee is satisfied at that program point.

Some modules exclusively coordinate the activities of other modules through procedure calls. Such *coordination* modules may not define any concrete sets themselves. (Consequently, they may not manipulate any concrete sets either). These modules work at a fully abstract level and rely on other modules to access the program's concrete data structures; it is sufficient to investigate the set specifications of these modules' callees to understand what these coordination do.

However, many modules—particularly implementations of data structures—do not depend on other modules and contain mainly *leaf* procedures. Leaf procedures do not make further procedure calls; they perform concrete data structure manipulations themselves rather than delegating the work to callees. Analysis plugins that are targeted towards analyzing particular classes of data structures may therefore decline to handle procedure calls.

The Hob system also requires all analysis plugins to verify that named abstract sets are always empty in the initial state of the program. This constraint makes it possible for Hob to know the initial contents of all sets in the program without inspecting all of the abstraction modules.

Analysis plugin obligations

In summary, when analyzing a module M, an analysis plugin must:

- verify that the procedures of M satisfy their postconditions (and module invariants) assuming that their preconditions (and module invariants) hold upon entry;
- verify that preconditions for all procedure calls originating inside M are satisfied (if the analysis plugin handles procedure calls); and
- verify that all sets declared in M are empty in the initial program state.

Stationarity condition. We designed the Hob analysis approach to support the modular verification of data structure consistency properties. Modular verification requires that changes to a program's state be somehow localized. Hob plugins must therefore ensure that only the implementation module defining a set may directly

manipulate that set. One way to do so is by using the format construct: the implementation language definition guarantees that any fields that a module contributes to a format may only be accessed by that module. Therefore, if a module M's set definitions rely only on the fields that M contributes to formats, then M's sets may only be modified by module M. In general terms, the Hob system requires plugins to verify that the following stationarity condition holds:

- no set or invariant may be defined in such a way that it would be modifiable outside its defining module.

This condition ensures that, even upon return from a procedure call to another module, a module's named sets do not surreptitiously gain or lose members. As a consequence of this condition, modules only mutate sets that they define; all such mutations are declared in ensures and modifies clauses.

4.1.1 Specifying Hob abstraction functions

Hob abstraction functions exist in the context of abstraction modules. The primary purpose of an abstraction module is to enable developers to specify abstraction functions, which identify sets of concrete heap objects satisfying some property. The anatomy of Hob abstraction modules is therefore as follows.

- Because the set of properties (for naming sets) available to the developer depends on the analysis plugin used, the developer must identify *which analysis plugins* to apply.

- The developer provides *set definitions* in a notation suitable for that analysis plugin.

- The developer identifies the implementation-level *boolean variables* that appear in the module's specification sections.

- The developer may optionally state *invariants* on the concrete heap which the associated implementation module must preserve; analysis plugins are required to verify that these invariants hold upon exit from each procedure, and may assume that these invariants hold upon entry to each procedure.

4.1.2 Common abstraction module grammar

The Hob system uses a single grammar for all of its abstraction modules. However, because different analysis plugins define sets and invariants differently, each analysis needs to be able to support its own syntax for set definitions and invariants. Figure 4-1 presents the part of the abstraction module grammar that is common to all analysis plugins. Each analysis plugin n must define its own sub-grammar for the D_n and I_n productions.

$$
\begin{array}{rcl}
M & ::= & \textsf{abst module } m\ \{M_1\ |M_{\text{multi}}{}^*\ \} \\
M_1 & ::= & \textsf{use plugin } \text{“}n\text{”};\ B \\
M_{\text{multi}} & ::= & \textsf{use plugin } \text{“}n\text{”} \textsf{for } \{\ \textsf{procs } pn^*;\ B\ \} \\
B & ::= & D^*\ I^*\ P^* \\
D & ::= & id{=}D_r; \\
D_r & ::= & D_r \cup D_r \mid D_r \cap D_r \mid id \mid \{x : T \mid \text{“}D_n\text{”}\} \\
P & ::= & \textsf{predvar } p; \\
I & ::= & \textsf{invariant } \text{“}I_n\text{”};
\end{array}
$$

Figure 4-1: Abstraction Language Grammar

An abstraction module contains one or more abstraction module bodies. Each abstraction module body selects an analysis plugin and specifies invariants, set definitions, and boolean variable declarations. If an abstraction module only uses one analysis plugin, then the module itself contains the abstraction module body. Otherwise, the abstraction module must be divided into a number of sub-modules. Each sub-module chooses an analysis plugin and contains an abstraction module body. When multiple analysis plugins are used, each procedure in a module must be claimed—and therefore analyzed—by exactly one analysis plugin.

The Hob system supports two kinds of set definitions: *base* set definitions and *derived* set definitions. Each analysis plugin n must specify a syntax for base set definitions by defining the production D_n. Derived set definitions define a set by combining previously-defined sets (or "anonymous" set definitions, which are given on-the-fly during a derived set definition) using union and intersection.

Abstraction module bodies may also contain declarations of predvars. In the current version of Hob, these predicate variables are tied to boolean variables in the implementation on a one-to-one basis. Although it would be possible to support arbitrary definitions for these variables, we have not yet encountered a situation where we needed to do so.

An analysis plugin n may also specify a syntax for module invariants by defining the I_n production. Not all analysis plugins define a syntax for module invariants.

Using Multiple Analysis Plugins in a Module. In our experience, we have found that some implementation modules are best analyzed by multiple Hob analysis plugins. For instance, a given module may contain both leaf and coordination procedures, which require different static analysis techniques in general. The Hob system enables developers to analyze these kinds of implementation modules by including multiple abstraction bodies within a module's abstraction section. When an abstraction section includes multiple abstraction bodies, then each abstraction body must specify which procedures it applies to. Each procedure must be analyzed by exactly one Hob analysis plugin.

We have implemented a doubly-linked list which uses multiple analysis plugins. Figure 4-2 presents the relevant abstraction section. Lines 2 through 28 contain the `Bohne decaf` abstraction body for the `DLL` module, while lines 29 through 32

contain the `flags` abstraction body for that module. Our example abstraction module declares that the `clear` procedure is to be analyzed with the `flags` plugins, while all other procedures are to be analyzed with the `Bohne decaf` plugin.

4.2 Flags Abstraction Module Language

Hob's *flag analysis plugin* implements a typestate analysis. This typestate analysis is more general than the traditional typestate formulation [91, 90] because it uses its sets to represent all objects with a given typestate. The flag analysis plugin uses the values of integer and boolean object fields (flags) to define the meaning of abstract sets. It verifies set specifications by first constructing set algebra formulas whose validity implies the validity of the set specifications, then verifying these formulas using an off-the-shelf decision procedure [52].

The flag analysis plugin is important for two reasons. First, the flag analysis plugin can propagate constraints between abstract sets defined in external modules using arbitrarily sophisticated abstraction functions. The plugin can therefore analyze modules that, as they coordinate the operation of other modules, indirectly manipulate external data structures defined in those other modules. This enables the flag analysis to perform the inter-module reasoning required to verify global invariants relating different data structures, *e.g.* inclusion and disjointness of data structures. Because the flags plugin uses the boolean algebra of sets to internally represent its dataflow facts, it can propagate and verify these constraints without any loss of precision.

Second, flag field values often reflect the high-level conceptual state of the entity that an object represents, and flag changes correspond to changes in the conceptual state of the entity. One way to visualize this second use of the flag plugin is as follows: the plugin is, in general, responsible for tracking object membership in sets. While most sets are defined externally—that is, the flag plugin is only responsible for tracking changes to those sets by using preconditions and postconditions—some sets are defined using a specific simple class of abstraction functions, and these sets are handled directly by the plugin.

By using flags in preconditions of object operations, the developer can specify key object state properties required for the correct processing of objects and the correct operation of the program. Standard typestate approaches excel at enforcing temporal operation sequencing constraints. The use of a set specification language additionally enables developers to express, for instance, relationships between sets of objects with various typestates. Our flag analysis plugin therefore goes beyond temporal sequencing constraints and successfully verifies the more general properties which are expressible in our set specification language.

Our flags plugin supports loop invariants for reasoning about procedures that contain loops. It can either use developer-provided explicit loop invariants or infer loop invariants from available information.

```
abst module DLL {
  use plugin "Bohne decaf" for {
    Content = { n : Node |
        "rtrancl (lambda v1 v2. next v1 = v2) (next root) n" };
    Iter = { n : Node |
        "rtrancl (lambda v1 v2. next v1 = v2) current n" };

    invariant "ALL x y.
        prev x = y --> (x ~= null &
                        (EX z. next z = x) --> next y = x) &
        ((x = null | (ALL z. next z ~= x)) --> y = null)";


    invariant "init --> (ALL x. ~(next x = root))";
    invariant "(~init --> root=null & current=null)";

    invariant "(init --> (root ~= null & (current=null |
        rtrancl (lambda v1 v2. next v1 = v2)
                (next root) current)))";

    invariant "ALL x. x ~= null &
        ~(rtrancl (lambda v1 v2. next v1 = v2) root x) -->
        ~(EX e. e ~= null & next e = x) & (next x = null)";

    procs init, add, remove, removeFirst, getFirst,
          isEmpty, openIter, nextIter, isLastIter,
          closeIter, contains, removeAtIter;
  }
  use plugin "flags" for {
    procs clear;
  }
}
```

Figure 4-2: Example List Abstraction Module

4.2.1 Example: Flag abstraction module

The `Board` module of our minesweeper example maintains the overall state of the minesweeper game board and coordinates with the data structures which are responsible for maintaining sets of exposed and unexposed cells. Figure 4-3 presents an abstraction section for the `Board` module. The Hob system verifies this module using the flags analysis, which allows developers to assign membership in abstract sets based on an object's concrete field values. Line 1 of the example states that an abstraction section for the `Board` module follows; lines 1–9 will provide definitions for the set and boolean variables used in the specification section of the `Board` module in terms of that module's concrete program state. This module does not contain any invariants. The `use plugin` declaration on line 2 states that the `Board` module should be analyzed by Hob's flags plugin. Line 3 defines the set `U` (for Universe) as the set of all `Cell` objects in the concrete heap which have the field `init` set to true. All other sets in this module are defined as intersections with the `U` set (the `cap` operator denotes intersection). Lines 4 through 7 define the `MarkedCells`, `ExposedCells`, `UnexposedCells` and `MinedCells` sets as *derived* sets. Line 4, for instance, states that the `MarkedCell` set contains those objects that are members of the `U` set and have their `isMarked` field set to `true`; the other set definitions are similar. Finally, line 8 declares that the `gameOver`, `init` and `peeking` boolean variables from the implementation are visible as specification-level boolean variables.

Protecting Sets from External Changes. The Hob implementation language definition specifies that new `Cell` objects always `isExposed` set to `false`, the default initial value for boolean fields. If we define the `UnexposedCells` set to contain all objects whose field `isExposed` is set to `false`, then this set would gain a new element whenever any part of the program instantiates a new `Cell` object. In such a situation, it would be very difficult to reason modularly about the `UnexposedCells` set: any part of the program could modify this set! The implication would be that any plugin that wished to soundly analyze a procedure call would need to analyze all potential callees from that site. Any modular analysis technique must, of course, somehow avoid the analysis of all of a procedure's transitive callees. The Hob stationarity condition avoids this potential barrier to modular analysis by requiring plugins to prevent such pathological set definitions. The set `U` satisfies the stationarity condition, since it contains those objects with field `init` set to true, and new objects have `init` set to false. Therefore, the subset `UnexposedCells` of `U`, as we've defined it, also satisfies the stationarity condition.

Initial Program State. In general, developers may not define sets that contain newly-initialized objects—objects that hold the initial field values assigned by the Hob implementation language. Chapter 2 stated that in the Hob implementation language, integer fields are initially initialized to 0, while boolean fields are initialized to `false`. The flag plugin uses this property of the implementation language to enforce the constraint that named sets must always defined to be initially empty. The format construct guarantees that these sets remain empty until a flags module

```
1 abst module Board {
2     use plugin "flags";
3     U = { x : Cell | "x.init = true" };
4     MarkedCells = U cap { x : Cell | "x.isMarked = true" };
5     ExposedCells = U cap { x : Cell | "x.isExposed = true" };
6     UnexposedCells = U cap { x : Cell | "x.isExposed = false" };
7     MinedCells = U cap { x : Cell | "x.isMined = true" };
8     predvar gameOver; predvar init; predvar peeking;
9 }
```

Figure 4-3: Example Flag Abstraction Module

executes: due to the format construct, no other module may modify an object's flags, as long as modules only define sets using the fields that they have contributed to a type. Our flags plugin ensures that a module's set definitions use only the object fields that the module has contributed.

4.2.2 Loop invariant inference

Loops are typically problematic for static analyses, as they introduce a unbounded number of execution paths that need to be analyzed. A standard approach for dealing with loops is by using loop invariants. Loop invariants state a condition that holds regardless of the number of times that the loop iterate. Loop invariants tame the verification task by eliminating the need to reason about an unbounded number of execution paths. Because the Hob flags plugin analyzes procedures by propagating formulas in the boolean algebra of sets, it can use loop invariants expressed in that logic to verify properties of loops. In particular, Hob's flags plugin can either verify developer-provided loop invariants or synthesize loop invariants from the program source code and specifications. The loop invariant synthesis algorithm is a novel contribution of this thesis.

Explicit Loop Invariants. If the developer provides an explicit loop invariant, the flags plugin verifies that the loop invariant: 1) holds on entry to the loop; and 2) is preserved by the loop body. At the exit of the loop, the loop invariant conjoined with the loop exit condition characterizes the post-loop program state.

Our loop invariant verification algorithm uses information from the loop's context to automatically augment the explicit loop invariant with properties that are known to be invariant over the loop. In particular, the loop's containing procedure will have a `requires` clause, which states the procedure precondition. This clause involves only the initial values of sets at the beginning of the procedure (which appear as unprimed set variables in our set specification language). Therefore, the clause holds throughout the procedure's execution, and this clearly includes the interior of the loop body. We also use the containing procedure's implementation, as well as its

`modifies` clause, to identify all non-modified sets, and construct a conjunct which states that these non-modified sets are preserved by the loop[1]. We then conjoin both the original procedure precondition and clauses guaranteeing the preservation of non-modified sets to all explicit loop invariants. Developers therefore need not provide these two pieces of redundant information, which helps to make explicit invariants more concise and easier to understand.

Inferred Loop Invariants. If the developer does not provide an explicit loop invariant, the flag analysis plugin attempts to automatically synthesize one. The synthesis starts with the boolean algebra formula characterizing the program state at the entry of the loop and weakens the formula by iterating the analysis of the loop until it reaches a fixpoint. We next present an example of the algorithm in action and discuss some properties of the algorithm.

Loop Invariant Inference Example. Figure 4-4 presents the `clear` procedure, which iterates through a set, removing each element until the set is empty. We use this procedure to illustrate our loop inference technique. In this procedure, each execution of the loop body removes an element from the `Content` set. Because the precondition of the `removeFirst` procedure must hold prior to its invocation, the loop body cannot execute successfully unless the `Content` set is non-empty, i.e. `card(Content') >= 1`. Furthermore, to be useful in practice, loop invariants must be strong enough to enable the verification of the procedures which contain them. In this case, the postcondition of the `clear` procedure is `card(Content') = 0`. A valid loop invariant must therefore ensure that executing the loop body in a state satisfying the invariant 1) does not violate the precondition of `removeFirst`, and 2) leads to a state that satisfies the loop invariant. A loop invariant that enables the analysis of `clear` must also ensure that, upon termination of the loop, the postcondition of `clear` holds (since `clear` does not contain any statements after the loop).

One possible loop invariant that satisfies these criteria is

$$I_p : e' \Leftrightarrow \mathsf{card}(\texttt{Content'}) = 0,$$

where e' is the return value from the `isEmpty()` procedure; it is `true` iff `Content'` is empty. Since e' is always false when execution enters the top of the loop body, I_p expresses the condition that the set is non-empty, thereby guaranteeing that the loop body can execute correctly; and since e' is always true when execution exits the loop, I_p implies that the set is empty at the end of the procedure, satisfying the procedure postcondition.

[1]Using the procedure's `modifies` clause alone results in an overly-conservative estimate of modified *private* sets in the presence of scopes, because scope-public procedures do not declare modifications of scope-private sets. Our use of the `modifies` clause in conjunction with the procedure implementation (to identify modifications to scope-private sets), on the other hand, allows the developer to state more detailed information about *public* sets than our modified-set inference algorithm could deduce.

```
specvar Content : Element set;

proc clear()    // specification
  requires true
  modifies Content
  ensures card(Content') = 0;

proc clear() {  // implementation
    pre:  bool e; e = isEmpty();
    head: while (!e) {
    body:    Entry q = removeFirst();
             e = isEmpty();
          }
    post: return;
}
```

Figure 4-4: Procedure containing a loop

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

proc removeFirst() returns e : Element
  requires card(Content)>0
  modifies Content
  ensures (card(e')=1) & (e' in Content) &
          (Content' = Content - e');
```

Figure 4-5: Procedures called within the loop

The flags analysis plugin analyzes the `clear()` procedure by starting with the procedure precondition (in this case, `true`) and successively computing an approximation of the strongest postcondition over the statements in the procedure. Eventually, the analysis reaches the `while()` statement containing the loop (labelled `head`), with the intermediate analysis result f. By construction, f holds for all reachable states at program counter `head` that the analysis has explored up to this point. In our example, f is the formula:

$$f = (\exists e_3.\ \neg e_3) \wedge q' = \emptyset \ \wedge\ (e' \Leftrightarrow \neg \mathsf{card}(\texttt{Content'}) \geq 1) \ \wedge\ \texttt{Content} = \texttt{Content'}$$

The formula f states that: 1) at some intermediate stage (represented by e_3), the variable e was false (in this case, e was initially `false`); 2) the variable q points to `null`; 3) `e'` is true iff the `Content` set is nonempty; and 4) the `Content` set is unchanged from its value on entry to the procedure. Note that e_3 is only defined and never accessed in the formula. This variable arises from the initial value `false` for local variable e, which is never read.

Our inference algorithm next strengthens f by conjoining the loop condition, producing a formula f_0 which holds at the start of the loop at the label `body` after zero loop iterations. For our example, f_0 is $f \wedge \neg e'$:

$$f_0 = (\exists e_3.\ \neg e_3) \wedge q' = \emptyset \wedge (e' \Leftrightarrow \neg \mathsf{card}(\texttt{Content'}) \geq 1) \wedge \texttt{Content} = \texttt{Content'} \wedge \neg e'$$

Since any loop invariant I must hold for all such states, it must be the case that $f_0 \Rightarrow I$. However, f_0 is unlikely to be the desired loop invariant, since it does not take the effect of the loop body into account. In particular, f_0 is probably too strong. Our algorithm therefore computes the strongest postcondition over the loop body, starting with f_0 at the top of the loop body, to obtain f_0'. The formula f_0' holds for the set of states that are reachable at the loop entry after executing exactly one loop iteration. Any acceptable loop invariant I must satisfy the constraints $f_0 \Rightarrow I$ and $f_0' \Rightarrow I$. For our example:

$$\begin{aligned} f_0' = \quad & (\exists e_3.\ \neg e_3) \wedge (e' \Leftrightarrow \neg \mathsf{card}(\texttt{Content'}) \geq 1) \\ & \wedge\ (\exists e_5.\ \neg e_5 \wedge (e_5 \Leftrightarrow \mathsf{card}(\texttt{Content}) = 1)) \\ & \wedge\ \texttt{Content'} = \texttt{Content} \setminus q' \wedge \mathsf{card}(q') = 1 \wedge q' \in \texttt{Content} \wedge e' \end{aligned}$$

The formula f_0' states that the set `Content'` is equal to the set `Content` minus q', which points to an object in the heap (since $\mathsf{card}(q') = 1$). The formula f_0' also states that at some previous program state, the variable e was true iff the set `Content` had cardinality 1. (Note that e_5 was formerly e' at the top of the loop; when composing formulas to take the effects of statements into account, our analysis renames e' to the existentially quantified e_5.) Finally, f_0' states that at some previous program state, the variable e was false, and that at the present state, e is true iff the `Content'` set is empty. Note that these final two conjuncts are common to f_0 and f_0'.

Building Potential Invariants. The formula f_0 summarizes the program state after zero iterations of the loop body, while f_0' summarizes the state after one iteration.

Our goal is to produce a logical formula which holds after an arbitrary number of loop iterations. We can start by producing a formula which holds after either zero or one loop iterations. We take conjuncts from f_0 which are implied by f_0', as well as conjuncts from f_0' which are implied by f_0. Any such conjuncts will then hold after both zero and one iterations of the loop body. We conjoin these conjuncts to produce the formula f_1:

$$\begin{aligned} f_1 \;=\; & (\exists e_3.\ \neg e_3) \wedge (e' \Leftrightarrow \neg \mathsf{card}(\texttt{Content'}) \geq 1) \\ & \qquad \wedge\ \texttt{Content'} = \texttt{Content} \setminus q' \wedge q' \in \texttt{Content} \end{aligned}$$

In formula f_1, we dropped the intermediate state e_5 and the constraint $\mathsf{card}(q') = 1$. We dropped the intermediate state e_5 because it does not exist after zero iterations of the loop, and we dropped the cardinality constraint because q' is the empty set in f_0 and known to be nonempty in f_1; no cardinality constraint in our analysis representation satisfies both of these conditions. Dropping the cardinality constraint allows q' to contain an arbitrary number of heap objects; it is no longer required to point to a single location in the heap.

Our technique then checks whether f_1 is a loop invariant, using the technique described above for verifying explicit loop invariants. In our example, f_1 is not a loop invariant: it contains the conjunct $\texttt{Content'} = \texttt{Content} \setminus q'$, where q' is a free variable; that is, in all iterations of the loop, Content' is equal to Content minus the set q', for all values of q'. Here, q' is only constrained to be a subset of Content). While this conjunct holds for the zeroth and first iterations of the loop, it does not hold for all iterations of the loop, because q' is free. Therefore, we iterate again, computing f_1', the strongest postcondition of f_1 over the loop body. We combine conjuncts from f_1 which are implied by f_1' with conjuncts from f_1' which are implied by f_1, yielding the next estimate f_2.

The formula f_2 summarizes the program state after zero, one and two iterations. It contains the clause $\texttt{Content'} = \texttt{Content} \setminus q_8 \setminus q'$. Because q_8 is existentially-quantified (rather than free), and because q_8 does not carry any cardinality constraints, the set q_8 can be interpreted to represent the difference between the initial Content set and the intermediate Content' set after any number of loop iterations. The analysis tests f_2 and finds that it is a loop invariant.

$$\begin{aligned} f_1' = \quad & \exists e_9.\ (\neg e_9 \wedge \exists q_8.\ (q_8 \in \texttt{Content} \wedge q' \in \texttt{Content} \setminus q_8 \\ & \qquad \wedge\ \texttt{Content'} = \texttt{Content} \setminus q_8 \setminus q') \\ & \qquad \wedge\ (\neg e_9 \Leftrightarrow \mathsf{card}(\texttt{Content} \setminus q_8) = 1)) \\ & \wedge\ (\exists e_3.\ \neg e_3) \wedge \mathsf{card}(\mathrm{q'}) = 1 \wedge (e' \Leftrightarrow \neg \mathsf{card}(\texttt{Content'}) \geq 1) \end{aligned}$$

$$\begin{aligned} f_2 = \quad & \exists q_8.\ (q_8 \in \texttt{Content} \wedge q' \in \texttt{Content} \setminus q_8 \\ & \qquad \wedge\ \texttt{Content'} = \texttt{Content} \setminus q_8 \setminus q') \\ & \wedge\ (\exists e_3.\ \neg e_3) \wedge q' \in \texttt{Content} \wedge (e' \Leftrightarrow \neg \mathsf{card}(\texttt{Content'}) \geq 1) \end{aligned}$$

The general loop invariant inference problem consists of finding a formula that summarizes all of the possible number of executions of the loop body. This formula is

a fixed point (hence "invariant") that is preserved by executing the loop body. One way to find such a formula is by starting with a formula that is stronger than the desired invariant and then weakening it. Formulas may be weakened by using disjunction; this is how we treat control-flow merges. Disjunction preserves information; it summarizes what is happening if the loop might execute n times or $n+1$ times. On its own, disjunction will never find a fixed point: without somehow weakening the formula, disjunction could only summarize the execution of a finite number of executions. Our approach to weakening formulas is ad-hoc: we drop conjuncts until we do reach a fixed point. We have designed our approach so that it is guaranteed to terminate, but the remaining formula might not be strong enough to enable the execution of the loop body. However, in our experience, our approach found all loop invariants needed by our benchmarks.

Existential Quantifiers. In our exposition so far, we have ignored the internal structure of the conjuncts in our formulas, and treated each top-level conjunct as an atomic unit. However, we found it necessary in practice to decompose top-level conjuncts, retaining only the parts of the conjunct which are true. In particular, our algorithm can infer stronger invariants by examining the internal structure of existentially quantified clauses instead of dropping entire clauses at a time. For instance, in the formula above, if c_j is of the form $\exists e.\ \bigwedge c_k^j$, then our algorithm drops sub-conjuncts c_k^j that are not implied by f_i'. Note, however, that even if some set of sub-conjuncts K such that $c_k^j \in K$ are individually implied by f_i', it does not necessarily follow that $f_i' \Rightarrow \bigwedge K$: in the presence of existential quantifiers, two sub-conjuncts may conspire to contradict the antecedent. If we do construct such a K which fails to imply f_i', then we drop those conjuncts of K that mention e and try again.

Comparing our inferred loop invariant f_2 with the invariant I_p, we can observe that f_2 has a number of extraneous clauses (e.g. $q' \in \texttt{Content} \wedge (\exists e_3.\ \neg e_3)$, and also the clause containing q_8) which are not required to verify the loop or the procedure in general. We have found no simple way to automatically produce smaller invariants. One possible heuristic is to eliminate those conjuncts from an inferred loop invariant which are not required for the analysis of the loop body to go through. In our experience, this strategy generates invariants that are sound, but too weak to prove the postconditions of some procedures, so we do not apply it.

Enforcing Termination. As presented above, our algorithm for generating and checking trial loop invariants is not guaranteed to terminate; we can construct contrived examples on which our algorithm does not terminate. In practice, we are able to infer all loop invariants in our example programs in at most three iterations.

A small change to the algorithm presented above ensures termination in all cases where it is possible to construct a loop invariant. We limit the number of iterations that the original algorithm may execute. Once the limit is reached, the algorithm subsequently drops any non-preserved conjuncts and does not introduce any new ones;

$$\begin{array}{rcl} n & ::= & \mathsf{flags} \\ D_n & ::= & x.f{=}c \end{array}$$

Figure 4-6: Grammar for Flag Abstraction Modules

that is,

$$f_{i+1} = \bigwedge_j \{c_j \mid f'_i \Rightarrow c_j\}.$$

This phase is guaranteed to terminate because it operates on a finite number of conjuncts; no new conjuncts are added. If no conjuncts are dropped in a given iteration, then the algorithm has found a loop invariant and terminates. Otherwise, the size of the formula strictly decreases at each step.

Our algorithm, as amended, is guaranteed to never loop with an infinite sequence of potential invariants that are too strong. On the other hand, it is possible to construct an example where our algorithm produces an invariant that is not strong enough for verifying the loop body. If a loop invariant exists, the developer can provide a hint to the inference algorithm by inserting the pair of statements `assert C; assume C;` inside the loop body.

Implications of Loop Invariant Inference. Hob's analysis approach relies on procedure summaries to enable the modular analysis of call sites. Analysis plugins must somehow analyze loops which occur in module implementations; one way to analyze loops is by verifying that the loops preserve loop invariants. Like procedure summaries, loop invariants provide useful information for code understanding; however, unlike procedure summaries, they are not essential for modular analysis. Our loop invariant inference technique therefore reduces the annotation burden on developers, which simplifies the tasks of developing programs and verifying relevant data structure consistency properties for these programs. We found that our loop invariant inference technique was successful in inferring all loop invariants in our benchmarks.

4.2.3 Using the flag analysis plugin

Figure 4-6 presents the grammar for flag abstraction modules. The flags plugin accepts base set definitions of the form `x.f = c` and derived set definitions which combine such base set definitions. The set definition `S = { x : T | x.f = c }` denotes all objects of type `T` in the concrete heap with field `f` equal to integer or boolean value `c`.

A module M's set definitions must satisfy the following conditions:

- Any field `f` used in a set definition must be defined by module M.
- Named sets may not contain uninitialized (*i.e.* newly-instantiated) objects.

These conditions allow the flags plugin to satisfy the stationarity condition. Note that the second condition is needed to ensure that arbitrary external modules cannot modify a flags module's sets by executing `new` statements.

Because the flags plugin analyzes procedures by propagating formulas in the boolean algebra of sets, it would not gain any additional expressive power by supporting invariants at the level of abstraction modules: Hob's specification module invariants have the same expressive power as would plugin-specific flags abstraction module invariants. We therefore do not provide a syntax for flags analysis abstraction module invariants.

To use the flags plugin, a developer must give relevant flag set definitions and ensure that the implementation always ensures the specified postconditions. When the postcondition for a flags procedure ranges only over sets defined in that procedure's module, the flags analysis plugin starts with the procedure precondition and computes the strongest postcondition of the implementation by tracking changes in flag values. The analysis then uses the MONA decision procedure to verify that the implementation's strongest postcondition implies the procedure's specified postcondition. Otherwise, the flags plugin relies on the design of the Hob system and incorporates the postconditions of called procedures into its computed strongest postcondition to verify that flags procedures satisfy their postconditions. We found the flags plugin to be especially useful for analyzing modules that delegate data structure manipulations to worker modules—coordination modules—due to its loop invariant inference algorithm; in the case of a coordination module, the developer only needs to specify the preconditions and postconditions of procedures in that module (as well as the information needed to verify the worker modules) to verify data structure consistency properties.

4.3 Bohne Abstraction Module Language

Shape analysis is a family of static analysis techniques for showing consistency of linked data structures [71, 38, 84]. Shape analysis is a promising technique for general data structure consistency checking, because it can reason about statically unbounded sets of objects and relations between them. As a result, shape analysis has great potential for improving software reliability. Unfortunately, precise shape analyses tend to lack scalability, greatly limiting their impact, despite significant recent progress in improving shape analysis efficiency [95, 69, 70].

Hob's Bohne plugin enables developers to use field constraint analysis [93], a particular instantiation of shape analysis, for verifying consistency properties of linked data structures. Field constraint analysis supports recursively-defined data structures which have a tree-like backbone plus nondeterministic field constraints. Bohne can handle a range of data structures, from singly-linked lists to two-level skip lists. Like the flags plugin, the Bohne plugin can also use developer-supplied loop invariants or infer them itself. Bohne uses symbolic shape analysis based on boolean heaps to deduce loop invariants.

Field constraint analysis uses set definitions stated in the monadic second-order logic over trees augmented with nondeterministic field constraints, which allow developers to constrain non-tree fields in the heap. Second-order logic permits quantification over predicates, namely functions and relations, as well as base objects in the

logic; monadic second-order logic restricts quantification over predicates to quantification over one-place predicates (*i.e.* sets). Due to this restriction on quantification, monadic second-order logic is decidable.

Because Bohne's underlying logic is second-order, it is sufficiently powerful to express the concept of transitive closure, which enables predicates in the logic to describe tree backbones; nondeterministic field constraints (stated in terms of invariants) then enable developers to describe properties of non-tree edges in the heap whose structure is constrained by the tree backbone.

4.3.1 Example: Bohne abstraction module

We designed the flags plugin to analyze relatively simple modules which rely on other modules to carry out sophisticated data structure manipulations. Hob's Bohne plugin, on the other hand, uses shape analysis techniques to statically analyze the worker modules that actually carry out data structure manipulations. Such worker modules often do not contain procedure calls. The Bohne plugin therefore supports only leaf procedures—procedures which do not call other procedures in turn.

Figure 4-7 presents the Bohne abstraction body for the doubly-linked list module with header (`DLL`) used in the minesweeper example. The Hob system verifies the `DLL` module with the `Bohne decaf` and `flags` plugins. "Bohne decaf" refers to a variant of the Bohne [93] shape analysis which relies on developer-provided loop invariants; the full Bohne plugin infers loop invariants using predicate abstraction. In this thesis, we discuss only the Bohne decaf plugin.

Linked list set definitions

The first part of the Bohne abstraction sub-module for the `DLL` module contains set definitions. The Bohne plugin allows developers to specify set contents using monadic second-order logic (MSOL). Lines 2–3 use MSOL to define the `Content` set, while lines 4–5 define the `Iter` set. In Bohne's interpretation of MSOL, fields in the heap are represented as relations between objects, so that `next x y` is true iff the field `x.next` points to `y`. Therefore, the central lambda-expression `lambda v1 v2. next v1 = v2` is a predicate which relates its formal parameter `v1` (a heap object) with formal parameter `v2`, a candidate linked-list successor; that is, the lambda expression is true when `v1.next = v2`. Next, Bohne's built-in `rtrancl` higher-order function takes a function and returns its `r`eflexive `tran`sitive `cl`osure. This causes the `lambda` predicate to be true for those objects `v2` reachable from `v1` by following zero or more `next` fields. Finally, we supply arguments to the `rtrancl lambda` expression: `Content` is the set of all objects `n` reachable through `next` fields from `root.next`, and `Iter` is the set of all objects `n` reachable from `current`.

Linked list invariants

Lines 7 through 22 state invariants for the doubly-linked list. These invariants must hold initially, are assumed to hold upon entry to linked list procedures and verified

upon exit. They enable the Bohne analysis to focus its attention only on reachable concrete states; otherwise, pathological (and unreachable) program states would prevent Bohne from successfully verifying the linked list. We next present all of the invariants of this module.

Field constraints. The invariant on lines 7–10 is an example of a *field constraint* on the `prev` field. In this case, the constraint on the `prev` field states that if `x.prev` points to `y` for `x` non-null, and if there exists another object `z` such that `z.next` points to `x`, then it must be the case that `y.next` points to `x`. In other words, `prev` is the inverse of `next` whenever `next` has an inverse. The constraint also states that `x.prev` must be null if `x` is null or if no object points to `x` through its `next` field.

Field constraints enable developers to give interpretations for derived fields. Derived fields are important because monadic second-order logic can only support tree-like heap structures; even a doubly-linked list is not a tree structure due to the `prev` fields. Our definitions of the `Content` and `Iter` sets fall within the monadic second-order logic over trees because they only discuss the subgraph of the heap which consists of the heap objects and the tree-structured `next` fields. In the presence of field constraints, the Bohne shape analysis tool must verify 1) that mutations to the `Content` and `Iter` sets are consistent with their specifications, and 2) that the `prev` field continues to satisfy the appropriate field constraints. In return, the field constraint tells Bohne how to interpret references to `prev` in the implementation. Note that this particular field constraint is deterministic, since `prev` is a deterministic function of `next`. The Pointer Assertion Logic Engine [71] supports deterministic field constraints. The Bohne system also supports nondeterministic field constraints [93], which state (partial) conditions that must hold on derived fields; nondeterministic field constraints enable developers to express data structures like two-level skip lists.

Only one pointer to root. The invariant on line 13 states that, if the module has been initialized (*i.e.* `init` is true), then for all objects in the heap, no object has a `next` field pointing to `root`. No static analysis could conclude that the `next` backbone remains acyclic upon addition to the list without using some form of this invariant.

Sets initially empty. The invariant on line 14 ensures that `Content` and `Init` are both empty if `init` is false. This invariant holds by the definition of the Hob implementation language: variables are initialized to `null` or `false`, as appropriate.

Constraints on variable values. Lines 16–18 give more well-formedness constraints on initialized concrete states. First, `root` must always be non-null if the program has been initialized. Furthermore, either `current` is null or it is reachable from `root` through the `next field`.

Orphan objects. Finally, lines 20–22 constrain objects that are not in the `Content` set. If an object `x` is not in `Content`, then it must not be reachable through the `next`

```
use plugin "Bohne decaf" for {
  Content = { n : Node |
      "rtrancl (lambda v1 v2. next v1 = v2) (next root) n" };
  Iter = { n : Node |
      "rtrancl (lambda v1 v2. next v1 = v2) current n" };

  invariant "ALL x y.
      prev x = y --> (x ~= null &
                      (EX z. next z = x) --> next y = x) &
      ((x = null | (ALL z. next z ~= x)) --> y = null)";


  invariant "init --> (ALL x. ~(next x = root))";
  invariant "(~init --> root=null & current=null)";

  invariant "(init --> (root ~= null & (current=null |
      rtrancl (lambda v1 v2. next v1 = v2)
              (next root) current)))";

  invariant "ALL x. x ~= null &
      ~(rtrancl (lambda v1 v2. next v1 = v2) root x) -->
      ~(EX e. e ~= null & next e = x) & (next x = null)";

  procs init, add, remove, removeFirst, getFirst,
        isEmpty, openIter, nextIter, isLastIter,
        closeIter, contains, removeAtIter;
}
```

Figure 4-7: Bohne abstraction body for doubly-linked list

field. Furthermore, `x.next` must be `null`. This invariant enables Bohne to conclude, for instance, that adding an object to the linked list adds precisely that object to the list, and no others.

4.3.2 Using the Bohne analysis plugin

Figure 4-8 presents the grammar for the Bohne abstraction language. As with other analysis plugins, Bohne abstraction modules contain set definitions and invariants. In the case of the Bohne plugin, set definitions must be expressed in monadic second-order logic over trees. The Bohne plugin uses a subset of the Isabelle abstraction language as its abstraction language; this design decision allowed us to leverage our pre-existing parser for the Isabelle abstraction language.

The Bohne plugin natively supports the `rtrancl` higher-order function for reflex-

ive transitive closure. A typical Bohne set definition uses the

```
S = { n : Node | rtrancl (lambda v1 v2. f v1 v2) r n };
```

formulation to denote the set of objects starting at module variable `r` and reachable through the `f` field. In principle, Bohne also supports more sophisticated set definitions in the monadic second-order logic over trees; however, in our work, we have focussed on exploring applications of the richer properties expressible as Bohne invariants rather than on exploring applications of more sophisticated Bohne set definitions.

Note that because the Isabelle grammar is quite general, `rtrancl` does not need to appear explicitly in the Bohne abstraction module language's grammar; during parsing, `rtrancl` is treated as an uninterpreted function. `rtrancl` is given the proper interpretation in the Bohne verification stage.

Nondeterministic field constraints enable developers to state properties of fields which do not belong to a data structure's tree backbone. Field constraints are a specific kind of invariant which enable the verification of implementations which traverse non-tree fields by stating the relationship between the non-tree fields and the tree backbone fields which occur in set definitions. As with invariants in general, nondeterministic field constraints are also useful for stating properties of non-tree fields that the developer expects to hold upon exit from all procedures (assuming that these properties hold upon entry). A nondeterministic field constraint has the form

$$\mathsf{FC}_n(x, y) = \texttt{ALL x y.}\ \ \texttt{n x y -> f(x, y)}$$

where `n` is the constrained field; this constraint states that property $\mathsf{FC}_n(x, y)$ holds whenever `x.n` points to heap object `y`. We have implemented an elimination algorithm for converting modules which use nondeterministic field constraints into modules which use formulas expressible in the monadic second-order logic over trees; the idea is to replace the occurrence $G(f(x))$ by the implication $\forall y.\ G(f(x)) \Rightarrow \mathsf{FC}_f(x, y)$. Our elimination algorithm is sound in all cases and complete when field constraints are nondeterministic; please refer to [93] for further details on field constraint analysis.

Bohne also supports general monadic second-order formulas as invariants. Invariants enable modular analysis by identifying certain concrete states as being unreachable by the implementation; without invariants, the analysis must assume the worst at each procedure entry point, and this may include program states which are too pathological for verification to succeed. In general, the developer must provide sufficiently strong invariants to enable the analysis to conclude that the tree-like backbone remains tree-like after executing each implementation procedure. Such invariants usually include prohibitions on pointers to the root of the data structure and prohibitions of pointers to and from objects not in the data structure; in our doubly-linked list example, these invariants were on lines 13–22.

Additionally, if a developer intends to maintain relationships between a module's concrete data structures, Bohne can verify that these relationships are, in fact, preserved by verifying that developer-provided invariants always hold upon exit from

$$
\begin{array}{rcl}
n & ::= & \mathsf{Bohne} \mid \mathsf{Bohne\ decaf} \\
D_n & ::= & F \\
I_n & ::= & F \\
F & ::= & \mathsf{ALL\ T}.F \mid \mathsf{EX\ T}.F \mid \mathsf{lambda\ T}.F \mid G \\
G & ::= & A \mid A^* \mid \sim G \mid G \wedge G \mid G \vee G \mid G \Rightarrow G \mid G \Leftrightarrow G \mid G = G \mid G \neq G \\
A & ::= & (F[,F]^*) \mid id \mid id[F] \mid \mathsf{null} \mid \mathsf{true} \mid \mathsf{false} \mid \emptyset \mid \{id : F\} \mid [|F[;F]^*|] \\
\mathrm{T} & ::= & id^* \mid (id :: Y)^* \\
Y & ::= & Y \rightarrow Y \mid Y\ \mathsf{set} \mid id\ \mathsf{ref} \mid \mathsf{void} \mid \mathsf{universe} \mid id
\end{array}
$$

Figure 4-8: Grammar for Bohne Abstraction Modules

the module's procedures. These invariants—stated at the abstraction module level—enable developers to state low-level properties of the concrete state. The Hob system also accepts high-level relationships between data structures. Such relationships are expressed in the set specification language and given in a module's specification section.

For the Bohne analysis, a module M's invariants may only use fields that module M contributes to a format. Note how formats contribute to modular verification: it is safe for other modules to remain oblivious of M's invariants, since they cannot possibly violate them.

4.4 Theorem Proving Abstraction Module Language

The Hob system enables the use of arbitrarily powerful static analysis techniques for reasoning about module implementations. Shape analysis, for instance, is one of the most precise static analysis techniques known today. However, sometimes developers reason about programs using techniques that lie beyond the capabilities of the current state of the art in automated program analysis. We believe that if a developer is willing to expend the effort needed to formally prove a particular data structure consistency property, then the Hob system should seamlessly accept such proofs in its program verification methodology. Theorem proving techniques can in principle verify arbitrarily complicated consistency properties; interactive theorem provers such as Isabelle [81] and Athena [4] allow writing general mathematical statements about program state. The difficulty in using theorem proving tools is that their application may require manual effort and familiarity with their behaviour. Because manual effort is expensive, theorem proving is effective only if it is focused on relevant parts of a program; the assumptions used during theorem proving must then be guaranteed by the rest of the program. Hob's theorem proving plugin [99] shows how it is possible to apply interactive theorem proving technology to the verification of data structure consistency properties. Using this plugin, we verified implementations of a set in terms of a linear array, as well as a partial specification of a priority queue (heap) implemented as a binary search tree stored in an array.

```
1 abst module Arrayset {
2     use plugin "vcgen";
3     Content = { x : Node | "exists j. 0 <= j & j < s & x : d[j]"};
4     predvar setInit;
5     invariant "0 < s";
6
7 }
```

Figure 4-9: Example Theorem Proving Abstraction Section

The theorem proving plugin takes set definitions and invariants in the Isabelle formula syntax. It then converts procedure specifications into Isabelle and computes weakest preconditions from procedure implementations. The theorem proving plugin splits the resulting proof obligations into subgoals, which it attempts to prove automatically using Isabelle. It then saves the proof obligations that cannot be proven automatically for the developer to discharge manually. Essentially, a user of the theorem proving plugin must show that the procedure's precondition (plus invariants) implies the weakest precondition needed to imply that the procedure's postcondition (plus invariants) holds at the end of the procedure. Note that, unlike the other plugins we have described, the theorem proving plugin does not include a loop invariant inference algorithm. Instead, the developer must always supply explicit loop invariants in code to be verified with the theorem proving plugin, which then generates the appropriate verification conditions for these annotated loops.

4.4.1 Example: Theorem proving abstraction module

Figure 4-9 presents an abstraction module for `Arrayset`, one of the set implementations used in the minesweeper example. The Hob system uses the theorem proving (`vcgen`) plugin to analyze the `Arrayset` module; currently, the theorem proving plugin is the only Hob plugin that can analyze properties of array-based data structures. The theorem proving plugin generates verification conditions in Isabelle. Once a developer discharges the relevant verification conditions, the module is known to satisfy the specified data structure consistency properties. A key point of the Hob system is that clients of this module, or any module in general, do not need to understand how the module is verified. The effort of verifying a module can be amortized over all potential uses of the module.

Line 3 of the abstraction module gives the definition of the `Content` set. It first states that the `Content` set consists of the objects `x` of type `Node` for which there exists some integer `j` between 0 and s, the array's upper bound, such that `d[j]` contains `x`. Line 4 states that the `setInit` boolean variable is visible in specifications. Finally, line 6 states that the implementation-level variable s is always non-negative.

4.4.2 Using the theorem proving analysis plugin

All analysis plugins must first conceptually compute weakest preconditions from a module's implementations, specifications, set definitions, and invariants; plugins then verify that procedure preconditions imply the computed weakest preconditions[2]. Hob's theorem proving plugin conforms to the general Hob analysis plugin scheme by computing weakest preconditions. However, the theorem proving plugin differs from other plugins because it does not promise to discharge the resulting proof obligations (that is, it is not complete): when using the theorem proving plugin, the developer is ultimately responsible for guiding the theorem prover to the appropriate proofs.

To use the theorem proving plugin, the developer must first provide set definitions and invariants for the module under verification. Our current implementation of the theorem proving plugin supports Isabelle/HOL, so developers may express set definitions and invariants in terms of Isabelle/HOL clauses. Figure 4-10 presents the concrete grammar for the theorem proving plugin's abstraction language.

Given implementation, specification, and abstraction parts of a module, the theorem proving plugin computes weakest preconditions for the procedures in that module. Each procedure's weakest precondition takes the form of a set of conjuncts. The theorem proving plugin then attempts to verify each conjunct in turn. First, it verifies if a conjunct belongs to the library of proved lemmas; if not, it attempts to discharge the conjunct using proof hints included (with `assert` statements) in the procedure code; finally, if that verification fails, it attempts to prove the conjunct using Isabelle's built-in simplifier and classical reasoner with array axioms.

In our experience, most generated verification-condition conjuncts are discharged automatically using array axioms. For the remaining conjuncts, the fully automated verification fails, and the plugin reports that these conjuncts are "not known to be true". After the developer interactively proves these difficult cases in Isabelle, our system stores these cases in its library of verified lemmas and subsequent verification attempts pass successfully without assistance. Our system compares conjuncts against the library of proved lemmas by comparing abstract syntax trees of formulas, taking into account some basic properties of logical operations. This enables the reuse of existing lemmas even when the verification conditions have changed slightly.

4.4.3 Expressive power of the theorem proving plugin

The theorem proving plugin allows developers to state and prove set definitions and invariants by writing higher-order logic predicates for the Isabelle/HOL theorem proving system. In general, higher-order logic is more powerful than the first-order logic used in our common set specification language [64]. In our examples, we have used second-order logic, which allows quantification over relations. Second-order logic is necessary for naturally expressing the transitive closure relation, which enables reasoning about heap reachability (as needed for linked data structures). More generally,

[2]Recall that the flags plugin actually computes strongest postconditions rather than weakest preconditions. Nevertheless, the flags plugin satisfies the general contract of an analysis plugin; it just uses a different analysis technique to do so.

$$
\begin{array}{rcl}
n & ::= & \mathsf{vcgen} \\
D_n & ::= & F \\
I_n & ::= & F \\
F & ::= & \mathsf{ALL}\ \mathrm{T}.F \mid \mathsf{EX}\ \mathrm{T}.F \mid \mathsf{lambda}\ \mathrm{T}.F \mid G \\
G & ::= & A \mid A^* \mid \sim G \mid G \wedge G \mid G \vee G \mid G \Rightarrow G \mid G \Leftrightarrow G \mid G = G \mid G \neq G \\
 & \mid & G < G \mid G \leq G \mid G > G \mid G \geq G \mid G : G \mid G \sim: G \mid G \cup G \mid G \cap G \\
 & \mid & G + G \mid G - G \mid G \times G \mid G \div G \mid G :: G \\
A & ::= & \mathsf{arrayread} \mid \mathsf{arraywrite} \mid \mathsf{newarray} \mid \mathsf{arraysize} \mid \mathsf{fieldread} \mid \mathsf{fieldwrite} \mid (F[,F]^*) \\
 & \mid & id \mid id[F] \mid \mathsf{null} \mid \mathsf{true} \mid \mathsf{false} \mid nat \mid \emptyset \mid \{id : F\} \mid [|F[;F]^*|] \\
\mathrm{T} & ::= & id^* \mid (id :: Y)^* \\
Y & ::= & Y \rightarrow Y \mid Y\ \mathsf{list} \mid Y\ \mathsf{set} \mid Y\ \mathsf{array} \mid id\ \mathsf{ref} \mid \mathsf{bool} \mid \mathsf{int} \mid \mathsf{void} \mid \mathsf{universe} \mid id
\end{array}
$$

Figure 4-10: Grammar for Theorem Proving Abstraction Modules

second-order logic enables the user to define structures which are constrained to having a finite number of elements.

Our use of Isabelle/HOL also enables developers to state internal constraints which rely on integer (or potentially floating-point) values. For instance, a developer could define a set which contains all elements of an array at prime indices. Because the theory of integers with addition and multiplication is undecidable, we chose to not include integer constraints in our common set specification language.

The Hob approach enables developers to combine arbitrarily expressive theorem proving invariants with more tractable logics for more straightforward parts of the program. When using the theorem proving plugin, developers may use basically arbitrarily expressive invariants and set definitions. But procedure preconditions and postconditions must always be given using Hob's set specification language. Upon exit from any procedure, Hob must verify that the program state satisfies that procedure's postcondition conjoined with any applicable invariants. Because Hob ensures that procedure postconditions always hold upon exit, the analysis of a module may rely on the validity of other modules' high-level set specifications without needing to see how these specifications are verified.

Limits of Isabelle/HOL's expressive power. Isabelle/HOL allows users to write any logical statement for which it can compute the type; in particular, it allows quantification over relations. Such quantification appears to be sufficient for expressing a large number of concepts used in modern mathematics. Isabelle/HOLCF [73] is an extension to Isabelle/HOL which adds support for domain theory, thereby aiding the reasoning process for functional programs. Isabelle/HOLCF does not increase the expressive power of Isabelle/HOL, but it does make some definitions and proofs easier to write. Finally, Isabelle/HOLZF supports the full axiom of choice, unlike Isabelle/HOL. Isabelle/HOL only supports a restricted form of the axiom of choice (and this, of course, appears to have no impact on its usefulness).

Comparing the theorem proving plugin and the Bohne plugin. The theorem proving plugin might appear to be quite similar to the Bohne shape analysis plugin. Indeed, the Bohne plugin accepts a subset of the theorem proving's abstraction module syntax, and both the theorem proving plugin and the Bohne plugin use the semantics of the implementation language to produce weakest preconditions from the source module.

The primary difference between these plugins is that, after generating verification conditions, the Bohne plugin applies the MONA decision procedure to automatically verify these verification conditions. The theorem proving plugin subsumes the Bohne plugin in terms of expressive power, since it supports a superset of Bohne's abstraction module syntax. However, because we designed it to accept a restricted input language, the Bohne plugin will generate a restricted domain of verification conditions. This domain is decidable. That is, procedures which are specified using Bohne can be shown to satisfy (or not) their specifications without user intervention[3]. Contrast the two-part Bohne plugin—it generates verification conditions (for MONA to process), then decides them—with the theorem proving plugin, which just generates the verification conditions (for Isabelle/HOL). While the Isabelle/HOL theorem prover might successfully prove some parts of the proof obligation resulting from the verification condition, users of Isabelle have no guarantees. Any user of the theorem proving plugin is obliged to prove any subgoals that Isabelle cannot prove automatically.

In any case, once the developer manually verifies the needed verification conditions, the Hob system enables the developer to productively use the analysis results. The broader implication of the theorem proving plugin is that it allows the composition of verification results obtained through theorem proving with verification results obtained from static analysis techniques. We have successfully used the Hob system to establish global data structure consistency properties by combining these different verification results.

4.5 How Abstraction Modules Enable Checking of Global Properties

The Hob system allows developers to state and verify global data structure consistency properties using the scopes and defaults mechanisms. Figure 4-11 presents a scope used in our minesweeper example. The scope invariant states that, outside the scope, the set `Board.ExposedCells` is always equal to the set `ExposedList.Content`; similarly, `Board.UnexposedCells` is equal to `UnexposedList.Content`. But the `Board` module is analyzed with the flags plugin, while the `ExposedList` and `UnexposedList` modules are both analyzed with the Bohne plugin. Hence the `Board` sets and the `ExposedList` sets are defined using completely different formalisms and verified using different static analysis techniques; despite this, the Hob system can successfully verify a statement that relates the two different kinds of sets.

[3]The developer does have to specify loop invariants for Bohne if the loop invariant inference fails, however.

```
scope Model
{
    modules Board, ExposedList, UnexposedList, List, Arrayset;
    exports Board;
    invariant (Board.ExposedCells = ExposedList.Content) &
              (Board.UnexposedCells = UnexposedList.Content) &
              (Board.init => ExposedList.setInit) &
              (Board.peeking | (card(UnexposedList.Iter) = 0));
}
```

Figure 4-11: Model scope from Minesweeper example

The Hob framework manages to divide the verification task among analysis plugins by using abstraction functions throughout the analysis task. Due to the use of abstraction functions, analysis plugins may safely assume that implementations of procedures in other modules implement their contracts, as expressed in the set specification language. Analysis plugins therefore never need to inspect implementations or abstraction functions of other modules. In the context of global program properties, the Hob approach enables the overall program verification task to guarantee that, for instance, the `Board.UnexposedCells` set always equals the `UnexposedList.Content` set, without requiring the `flags` plugin used for the `Board` module to read the code for the `UnexposedList` module. Note that the analysis of the `UnexposedList` module does not require the specifications for the `Board` module, because the `UnexposedList` does not call the `Board`. Figure 4-12 illustrates this situation: it shows the modules that the `flags` and `Bohne` analyses see in the context of verifying the `Board` and `UnexposedList` modules.

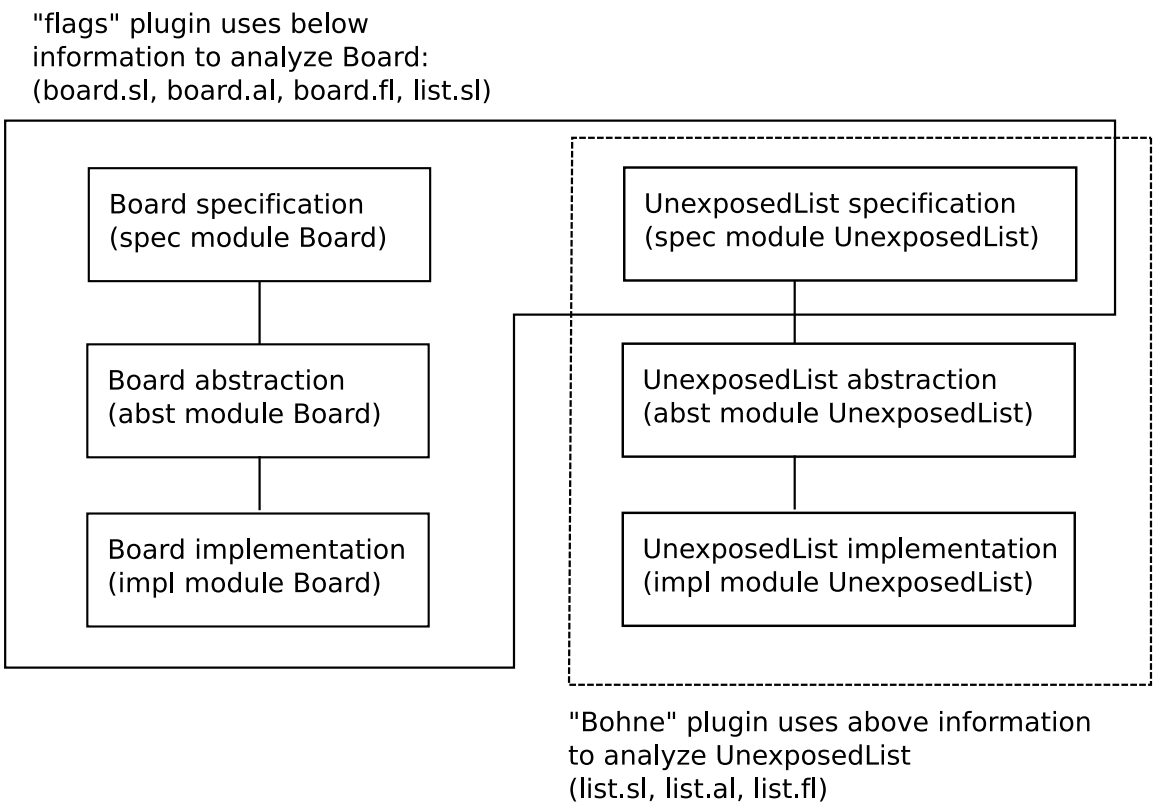

Figure 4-12: Module visibility by various analysis plugins

Chapter 5

Ensuring Consistency Properties

The Hob system verifies two broad classes of data structure consistency properties: local properties and global properties. Developers use local properties to establish the validity of Hob's set abstraction by guaranteeing that data structure implementations conform to their set interfaces, and then use global properties—expressed in terms of abstract sets—to guarantee that domain-specific consistency properties hold. Because Chapter 3 has already described how Hob converts global properties into local properties, it remains only to verify local data structure consistency properties.

The Hob system uses a suite of *analysis plugins* to ensure that various implementations conform to their interfaces. Each plugin is especially designed to verify data structure consistency properties for a particular class of implementations. It is the developer's responsibility to select an analysis plugin which can verify the desired data structure consistency properties. Section 5.1 explains the general contract of Hob analysis plugins; Chapter 6 presents one plugin, our Hob *flags* plugin, in detail. Figure 5-1 presents a schematic diagram illustrating what analysis plugins do. Briefly, analysis plugins read the implementation, specification and abstraction sections of a module M as well as the specifications for any modules that M calls, and decide whether the module's implementation conforms to its specification or not.

Global consistency properties, unlike local properties, are not necessarily related to any particular program module. Developers must therefore inform the Hob system about the complete set of global consistency properties to get sound analysis results. Section 5.3 describes our verification driver, which ensures that all necessary external module declarations and scope declarations are included in the analysis of any given module, and also ensures that Hob verifies all of the modules in a program.

5.1 Analysis Plugin Responsibilities

Each Hob analysis plugin is responsible for verifying that some target class of procedures conform to their specifications. To verify that a procedure implementation conforms to its specification, modular program verification tools—including Hob—typically assume that the procedure's precondition holds upon entry to the procedure and attempt to show that the postcondition holds upon exit from the procedure.

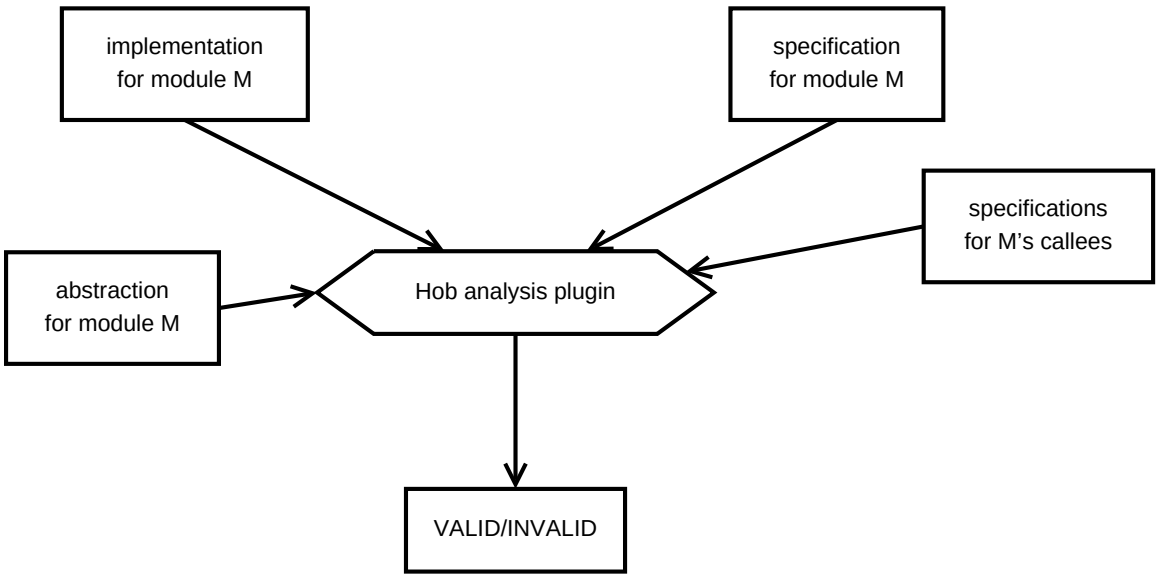

Figure 5-1: Overview flowchart for generic analysis plugin. Boxes represent data. Hexagons represent actions.

In the Hob approach, procedure specifications contain preconditions (requires clauses) and postconditions (ensures clauses) expressed in the boolean algebra of sets, which we presented in Chapter 3. Hob implementations are written in the Hob implementation language. This language is formally defined by its operational semantics, which we presented in Chapter 2. Abstraction modules, discussed in Chapter 4, mediate the relationship between the concrete states of the operational semantics and the abstract set-based specifications.

Hob analysis plugins therefore use a module's abstraction module to convert procedure preconditions from the boolean algebra of sets into a suitable internal representation. Plugins then construct a summary of the possible program states upon exit from the procedure (which are defined by reference to the Hob implementation language's operational semantics). Finally, plugins must verify that each of the possible states upon exit imply the procedure postcondition. Figure 5-2 summarizes this textual description by presenting a more detailed view of the internal workings of analysis plugins.

The Hob system includes the flags, Bohne and theorem proving plugins. Chapter 6 describes the Hob flags analysis plugin. The flags analysis plugin supports abstraction modules which assign set membership based on field values; because it can infer loop invariants, it is also useful for analyzing high-level coordination modules. Coordination modules call upon other modules to manipulate data structures but do not directly maintain any data structures themselves. The Bohne plugin allows developers to use shape analysis techniques to reason about program properties in the presence of pointer-linked heap data structures. Specifically, the Bohne plugin implements field constraint analysis [93], a particular instantiation of shape analysis. The theorem proving plugin enables developers to state and prove arbitrary program properties—

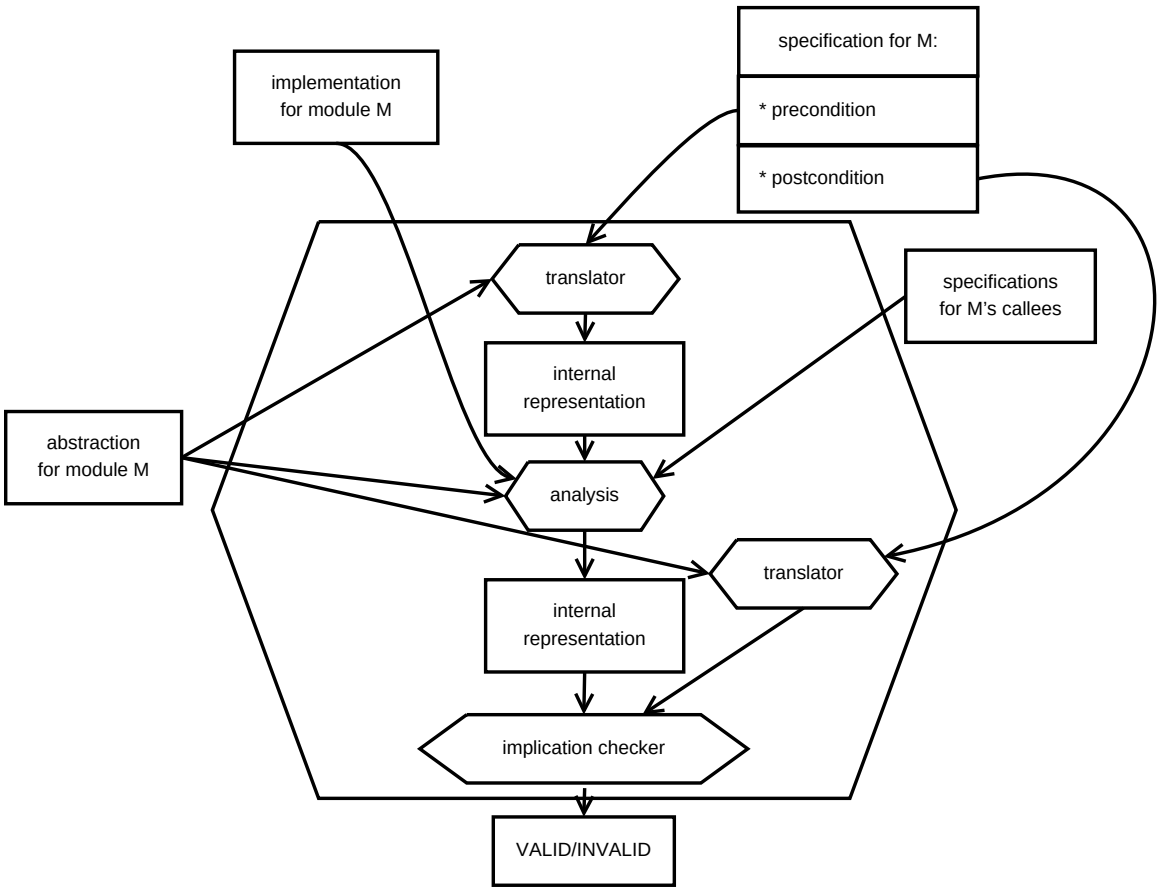

Figure 5-2: Detailed flowchart for generic analysis plugin. Boxes represent data. Hexagons represent actions.

including those that are beyond the reach of current static analysis techniques—by constructing weakest preconditions from the implementation and relying on the developer to discharge the resulting verification conditions using the Isabelle theorem proving system [99].

5.2 Developing New Analysis Plugins

A key design goal of the Hob framework was to support the development of a variety of analysis plugins. We next explain how to extend Hob with new analysis plugins.

Hob analysis plugins are responsible for verifying procedure postconditions. Because developers write these postconditions using the common set-based specification language, analysis plugins must implement a mapping between the specification-level set-based abstract state and the implementation-level concrete state. The first step in developing a new analysis plugin is therefore to choose a family of abstraction mappings for the plugin; for instance, the Bohne shape analysis plugin enables its users to map pointer-linked heap data structures (*e.g.* linked lists) to abstract sets. Analysis plugins may also support implementation-level invariants, which help make the analysis problem more tractable by constraining the set of possible concrete heap states. Because different analyses require markedly different types of abstraction mappings and invariants, it is the responsibility of each analysis plugin to translate abstraction mappings and invariants from strings into some suitable internal representation.

The designer of a Hob analysis plugin should next decide whether or not to handle procedure calls. Some Hob plugins, such as the Bohne plugin, are designed for leaf procedures, and do not handle procedure calls. We believe that many program designs modularize intricate data structure manipulations rather than intermingling such manipulations with procedure calls. Analysis plugins may therefore decline to handle procedure calls, saving some implementation effort. Note that all of the machinery for handling procedure calls will be present (in some form) in any analysis plugin: to handle procedure calls, an analysis plugin needs to integrate the precondition and postcondition of the called procedure. But any analysis plugin must already integrate the precondition and postcondition of the procedure under analysis. Handling procedure calls is therefore just an issue of hooking up the appropriate machinery at the appropriate program points. Nevertheless, in our experience, it was not necessary for all plugins to handle procedure calls.

Most analysis plugins include some provision for handling loops. The key challenge in supporting loops is in handling the potentially unbounded number of execution paths through the loop; many analyses use loop invariants to summarize the possible effects of these paths. Existing Hob plugins support both developer-supplied loop invariants and (in some cases) loop invariant inference. Loop invariant inference makes it easier for developers to verify programs at the cost of plugin development effort. Even if a plugin supports invariant inference, the fact that inference may be computationally expensive (and possibly an open question, depending on the analysis plugin's internal representation) implies that it is almost always useful for analysis plugins to support developer-supplied loop invariants. A plugin developer might

choose to support developer-supplied loop invariants written in either, or both, the set specification language and the plugin's concrete invariant notation. The Hob framework passes any provided loop invariants to the plugin as a string. If invariants contain set specifications, the plugin may call back into the Hob framework to parse the set specifications into abstract syntax trees.

Having made these design decisions, a developer must next implement the analysis plugin. The Hob framework provides the plugin with abstract syntax trees (ASTs) for the module's implementation, specification, and abstraction sections. Whenever the Hob framework cannot provide an abstract syntax tree because the interpretation of the input depends on the analysis plugin (*e.g.* abstraction mappings, assertions), an analysis plugin developer must instead parse the strings into a suitable format inside the plugin itself.

The analysis plugin must accept the provided ASTs and decide whether, given the provided implementation, the postcondition is guaranteed to hold at all procedure exits (assuming that procedure preconditions hold upon procedure entry). Recall that the Hob framework has processed the preconditions and postconditions to include any necessary global consistency conditions and the effect of the procedure's `modifies` clause; at this point, the provided preconditions and postconditions can be verified without reference to any other part of the program. The Hob framework has also arranged for all implementation-level invariants to hold at entry points for public procedures; the analysis is responsible for ensuring that these invariants hold upon exit.

The Hob framework does not impose any particular methodology for the core verification task. Existing plugins have taken a number of different approaches. Many existing plugins translate the procedure precondition into an internal representation and perform some kind of verification condition generation, passing an implication to a decision procedure for each procedure exit point (and call site, if appropriate). The PALE plugin, however, translates an entire procedure (both its specification and implementation) into a notation suitable for the PALE tool and delegates the verification task to the PALE tool.

Once a plugin has decided whether or not an implementation conforms to its specification, the plugin must report success or failure to the analysis tool. Analysis plugins are also encouraged to provide meaningful error messages in the event of failure.

5.3 Hob Analysis Driver

To verify a program module M, the Hob system clearly needs the implementation, specification and abstraction modules for M. However, this does not suffice: Hob also needs specifications for M's dependencies—the modules that M calls, as well as scope definitions for scopes that M belongs to. Note that overlooking scope definitions can result in soundness problems, because scopes impose additional requirements for modules to satisfy (in the form of scope invariants). This section describes how the Hob analysis driver ensures that Hob's analyses see all needed components when

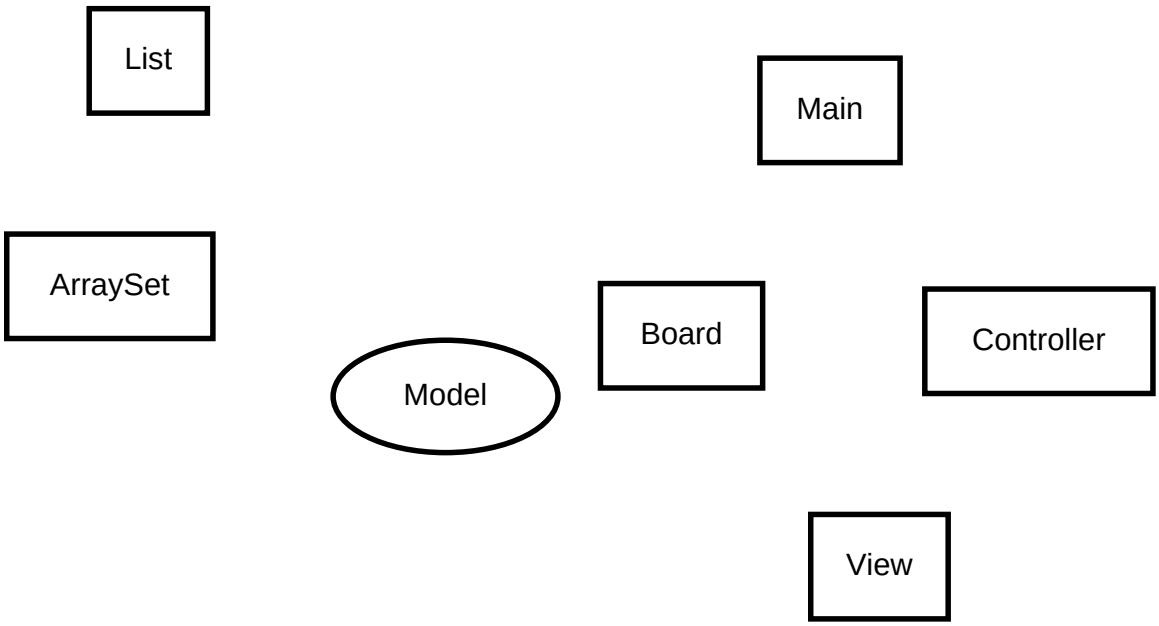

Figure 5-3: Hob analysis driver state after parsing `minesweeper` files. Boxes represent implementation/specification/abstraction triples. Ovals represent scopes.

analyzing a module. It also presents a sample run of the Hob analysis driver on our `minesweeper` example.

Parse all files. The Hob analysis driver first parses all Hob abstraction, implementation, and specification files in a directory, as well as all scope declarations. Once the Hob analysis driver has parsed all relevant files, it can compute inter-file dependencies. Figure 5-3 presents the state of the Hob analysis driver after parsing the modules in our `minesweeper` example.

Instantiate modules. The Hob analysis driver next expands static module instantiations (as described in Chapters 2 and 3), since modules may have instantiated modules as dependencies. Figure 5-4 presents the state of the Hob analysis driver after instantiating the `List` module as `UnexposedList` and `ArraySet` as `ExposedSet`.

Add dependencies. The Hob analysis driver must next add dependencies between different program components. The analysis driver first adds dependencies between scopes and their contained modules. Figure 5-5 illustrates the state of the Hob analysis driver after adding dependencies from scopes to their contained modules. Next, the analysis driver adds dependencies between modules and their callees. Figure 5-6 presents the state of the Hob analysis driver after adding inter-module dependencies.

Topological sort and command generation. Having computed all of the dependencies, the Hob analysis driver performs a topological sort to determine 1) a

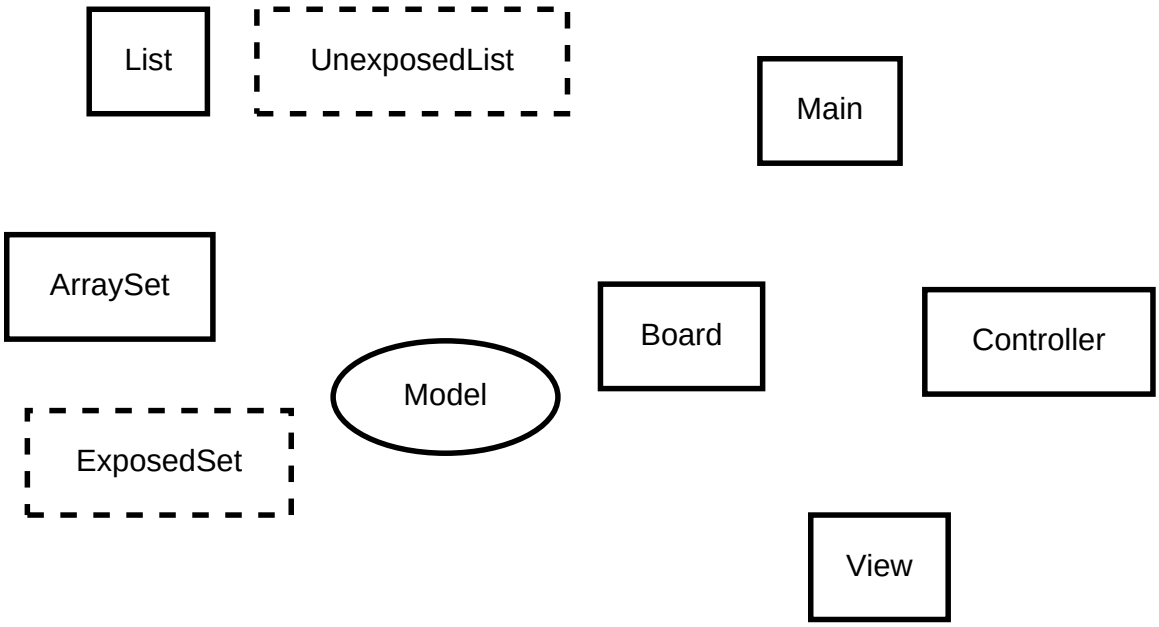

Figure 5-4: Hob analysis driver state after processing `minesweeper` static module instantiations. Solid boxes represent implementation/specification/abstraction triples. Dashed boxes represent instantiated modules. Ovals represent scopes.

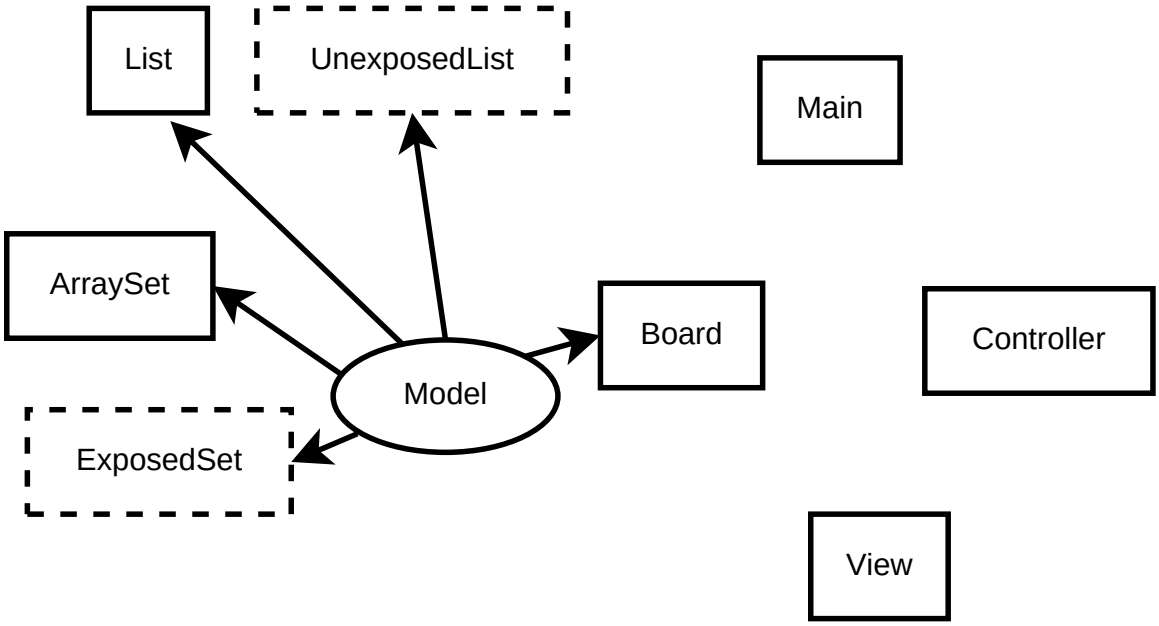

Figure 5-5: Hob analysis driver state after adding `minesweeper` scope dependencies. Solid boxes represent implementation/specification/abstraction triples. Dashed boxes represent instantiated modules. Ovals represent scopes. Lines represent scope containment.

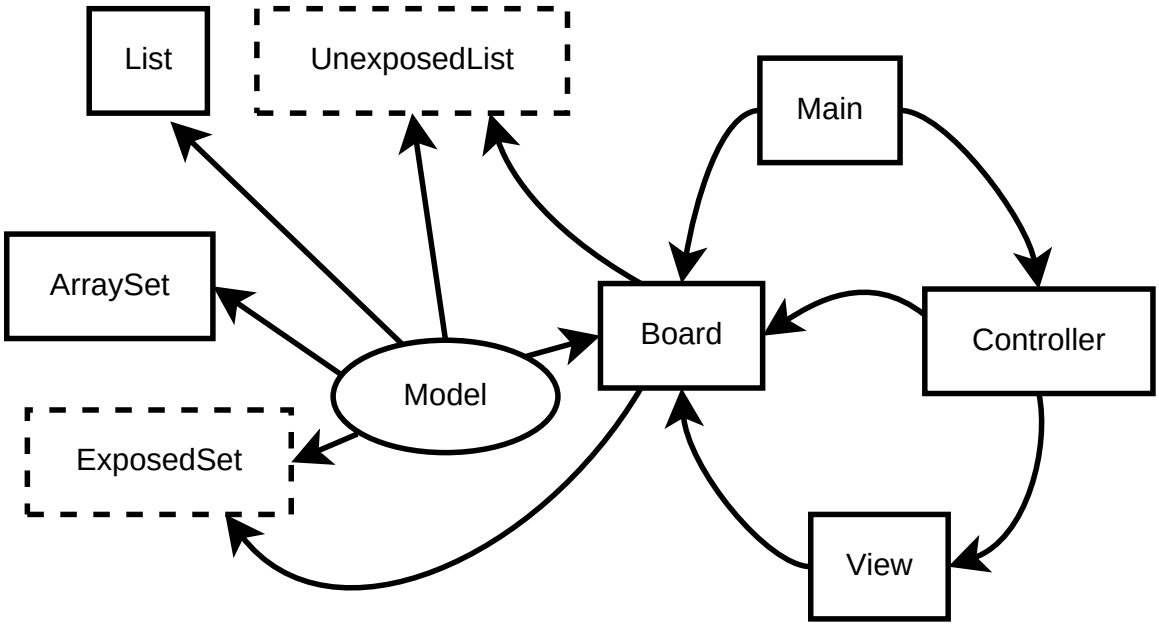

Figure 5-6: Hob analysis driver state after adding `minesweeper` inter-module dependencies. Solid boxes represent implementation/specification/abstraction triples. Dashed boxes represent instantiated modules. Ovals represent scopes. Lines represent scope containment. Curved lines represent module dependencies.

set of invocations of the Hob analysis tool which guarantees that all modules are checked; and 2) the set of relevant files to pass to the Hob analysis tool for each invocation. This set of relevant files includes the implementation, specification, and abstraction sections of a particular module, plus any scopes that the module belongs to, and finally all specification modules for the module's callees. Figure 5-7 presents the analysis tool invocations which, together, verify the `minesweeper` benchmark.

```
$ ../../bin/verify all
Verifying module Arrayset...
 -> analyze ./arrayset.fl ./arrayset.sl ./arrayset.al
Verifying module List...
 -> analyze ./list.fl ./list.sl ./list.al
Verifying module View...
 -> analyze ./view.fl ./view.sl ./view.al ./board.sl
Verifying module Board...
 -> analyze ./board.fl ./board.al ./model.scope ./view.sl
            ./arrayset.sl ./board.sl ./list.sl
Verifying module Controller...
 -> analyze ./controller.fl ./controller.sl ./controller.al
            ./board.sl ./view.sl
Verifying module Main...
 -> analyze ./main.fl ./main.sl ./main.al ./board.sl ./controller.sl
```

Figure 5-7: Commands generated by Hob analysis driver

Chapter 6

Flags Analysis Plugin

The Hob flags analysis plugin verifies modules in which integer or boolean flags indicate abstract set membership. The developer specifies (using the flags abstraction language) the correspondence between the implementation's concrete flag values and the specification's abstract sets, and additionally identifies the concrete boolean variables which also appear as abstract specification-level boolean variables. The flags plugin is also suitable for analyzing coordination modules, which do not maintain any sets themselves, but instead coordinate the sets of other modules; in analyzing such modules, the flags plugin keeps track of set contents for externally-defined sets and updates them at procedure call sites.

Section 4.2 presented the abstraction language for the flags plugin. The abstraction language allows developers to specify *what* properties to verify. This chapter explains *how* the flags plugin verifies properties. The flags plugin uses the MONA decision procedure [51] to verify whether or not procedures satisfy their postconditions. MONA was built to process formulas expressed in monadic second-order logic so by compiling formulas into automata and analyzing these automata. Our flags plugin only emits formulas in the weak monadic second-order theory of 1 successor, a subset of the logic that MONA supports, and our examples verify in dozens of seconds. The weak monadic second-order theory of 1 successor suffices for the flags plugin because this plugin only manipulates statements in first-order logic over uninterpreted sets.

6.1 Flags Analysis Example

Figure 6-1 presents the implementation and specification of a short procedure, as well as the relevant part of its abstraction section. This procedure either adds or removes an object from the `MarkedCells` set by mutating its `isMarked` boolean-valued field. To analyze the procedure, the flags analysis plugin generates boolean formulas for each program point and verifies whether or not the formulas at procedure exit points imply the stated postcondition.

```
impl module Board {
    proc setMarked(c:Cell; v:bool) {
        c.isMarked = v;
    }
}

spec module Board {
    proc setMarked(c:Cell; v:bool)
      requires (c in U) & (card(c)=1)
      modifies MarkedCells
      ensures (v <=> (c in MarkedCells')) &
              (MarkedCells' <= MarkedCells + c);
}

abst module Board {
    use plugin "flags";
    U = { x : Cell | "x.init = true" };
    MarkedCells = U cap { x : Cell | "x.isMarked = true" };
}
```

Figure 6-1: Minesweeper `Board` specifications, implementations, and abstractions

At the start of the procedure, the flags plugin generates the following formula by reiterating the procedure precondition and stating that all sets and variables are unmodified.

$$\forall_2 M.\forall_0 p.\forall_2 M'.\forall_0 p'.\cdots \tag{6.1}$$

$$(M = U \cap M_1) \wedge \exists M_1'.\ M' = U' \cap M_1' \tag{6.2}$$

$$c \subset U \wedge \mathsf{card}(c) = 1 \tag{6.3}$$

$$\wedge \quad M' = M \wedge U' = U \wedge C' = C \wedge p \Leftrightarrow p' \wedge \cdots \tag{6.4}$$

The formula ranges over the set variables and boolean predicates in the program. Procedure parameters occur as free variables of the formula, while the program's abstract state is given in terms of universally quantified variables. Note that this formula is a relation between unprimed (initial) sets and boolean variables and primed (current) sets and boolean variables. (For brevity, we refer to the `MarkedCells` set by the abbreviation M. We also omit unused variables except for `peeking`, abbreviated as p. We chose to leave p in our example to illustrate our treatment of unmodified variables.)

Line 6.1 contains universal quantifiers for abstract variables. $\forall_2$ denotes universal quantification over sets while $\forall_0$ denotes universal quantification over boolean variables. Line 6.2 states definitions for derived sets; M is a derived set because it is defined as the intersection of the universal set U with the base set M_1 of objects with

`isMarked` set to `true`. These definitions are repeated twice, once for unprimed variables and once for primed variables. Line 6.3 states the procedure precondition, which holds throughout the procedure, since it states constraints on unmodifiable unprimed sets. Finally, line 6.4 constrains sets and boolean variables that are unmodified by the procedure. Initially, all sets and variables are unmodified. Each modification of state removes a variable from this line.

The flags plugin next processes the statement `c.isMarked = v`, using the assignment statement transfer function, to obtain the following relation.

$$
\begin{aligned}
&\forall_2 M.\forall_0 p.\forall_2 M'.\forall_0 p'.\cdots && && (6.5)\\
&\qquad && (M = U \cap M_1) \wedge \exists M_1'.\ M' = U' \cap M_1' && (6.6)\\
&\qquad \wedge && ((M_1' = M_1 \cup c) \wedge v) \vee ((M_1' = M_1 \setminus c) \wedge \neg v) && (6.7)\\
&\qquad \wedge && c \subset U \wedge \mathsf{card}(c) = 1 && (6.8)\\
&\qquad \wedge && U' = U \wedge C' = C \wedge p \Leftrightarrow p' \wedge \cdots && (6.9)\\
& && && (6.10)
\end{aligned}
$$

The transfer function updates the value of the implicit base M_1 set by adding the object c iff the v variable is true (line 6.7).

Having reached the end of the procedure, the flags plugin then generates the following formula to submit to the MONA decision procedure.

$$
\begin{aligned}
&\forall_2 M.\forall_0 p.\forall_2 M'.\forall_0 p'.\cdots && && (6.11)\\
&\qquad && (M = U \cap M_1) \wedge \exists M_1'.\ M' = U' \cap M_1' && (6.12)\\
&\qquad \wedge && ((M_1' = M_1 \cup c) \wedge v) \vee ((M_1' = M_1 \setminus c) \wedge \neg v) && (6.13)\\
&\qquad \wedge && c \subset U \wedge \mathsf{card}(c) = 1 && (6.14)\\
&\qquad \wedge && U' = U \wedge C' = C \wedge p \Leftrightarrow p' \wedge \cdots && (6.15)\\
&\qquad \Longrightarrow && && (6.16)\\
&\qquad && C' = C \wedge p \Leftrightarrow p' && (6.17)\\
&\qquad \wedge && ((v \Leftrightarrow c \subset M') \wedge M' \subset M \cup c) && (6.18)
\end{aligned}
$$

The formula contains two parts. Lines 6.11 through 6.15 specify the program state after symbolic execution of the procedure, while lines 6.17 and 6.18 state the requirements on the program state needed by the procedure's postcondition. To verify that the procedure satisfies its specification, MONA's decision procedure must prove that lines 6.11 through 6.15 imply lines 6.17 and 6.18. The known state at procedure exit (lines 6.11 through 6.15) simply contain the relation that the transfer function computes; this relation captures the effect of the assignment to the `isMarked` field,

Lines 6.17 and 6.18 contain the requirements that the flags plugin must ensure. No executions of the procedure's implementation may modify any sets that are not declared to be modified, as stated in line 6.17. Also, the procedure's implementation must cause its postcondition to hold; line 6.18 states that constraint.

Once the flags plugin generates the appropriate formula, it passes the formula on to the MONA tool. In this case, the verification succeeds because the antecedent is

sufficiently strong. The flags plugin may therefore conclude that the procedure indeed implements its specification.

6.2 Flags Analysis Algorithm

To verify a procedure, the flags analysis performs abstract interpretation [20], using the space of boolean formulas as the abstract domain. It attempts to show that procedure postconditions are implied by the analysis domain element computed for each procedure exit points. Figure 6-2 illustrates the operation of the flags analysis algorithm. Starting with the procedure precondition, the analysis's transfer functions manipulate boolean formulas and modify these formulas following assignment statements and procedure calls. The analysis treats loops by using developer-provided loop invariants or by inferring the invariants itself. We call the key technique for manipulating formulas *incorporation*. This technique updates a boolean algebra formula by incorporating the effect of a second boolean algebra formula. Whenever the analysis creates a new formula (mostly during incorporation), it also applies some simple optimizations to the formula before it is created. We found that these optimizations were crucial to the successful verification of our benchmark programs.

More formally, our analysis associates a quantified boolean formula F with each program point. A formula F is a relation between two collections of variables. Unprimed set variables S (or boolean variables b) denote initial values of sets (or booleans) at the entry point of the procedure, while primed set variables S' (or primed boolean variables b') denote the values of these sets (or booleans) at the current program point. In general, set and boolean variables are defined in their containing module's abstraction sections; Section 4.2 described how developers may define set and boolean variables for the flags plugin. The use of primed and unprimed variables allows the flags analysis to represent, for each program point p, a binary relation on states that overapproximates the reachability relation between procedure entry and point p [48, 19, 86].

The flags analysis also tracks (object-typed) local variables using sets. For each local variable, the corresponding set contains the object to which the local variable refers; such a set comes with a cardinality constraint that restricts the set to have cardinality at most one (null references are represented by the empty set). This approach automatically disambiguates some local variable and object field accesses; if a formula contains a constraint stating that two local variables are disjoint, then these variables are unaliased. Other static analyses often rely on a separate pointer analysis to provide this information.

The initial dataflow fact at the start of a procedure is the precondition for that procedure, transformed into a relation by conjoining $S' = S$ for all relevant sets and $b' \Leftrightarrow b$ for all relevant boolean variables. Clearly, at the beginning of a procedure, all sets and boolean variables have their initial values. At merge points, the analysis combines boolean formulas with disjunction. The analysis also performs loop invariant verification and inference if necessary (Section 6.6). After running the dataflow analysis, our analysis checks that the procedure conforms to its specification by checking

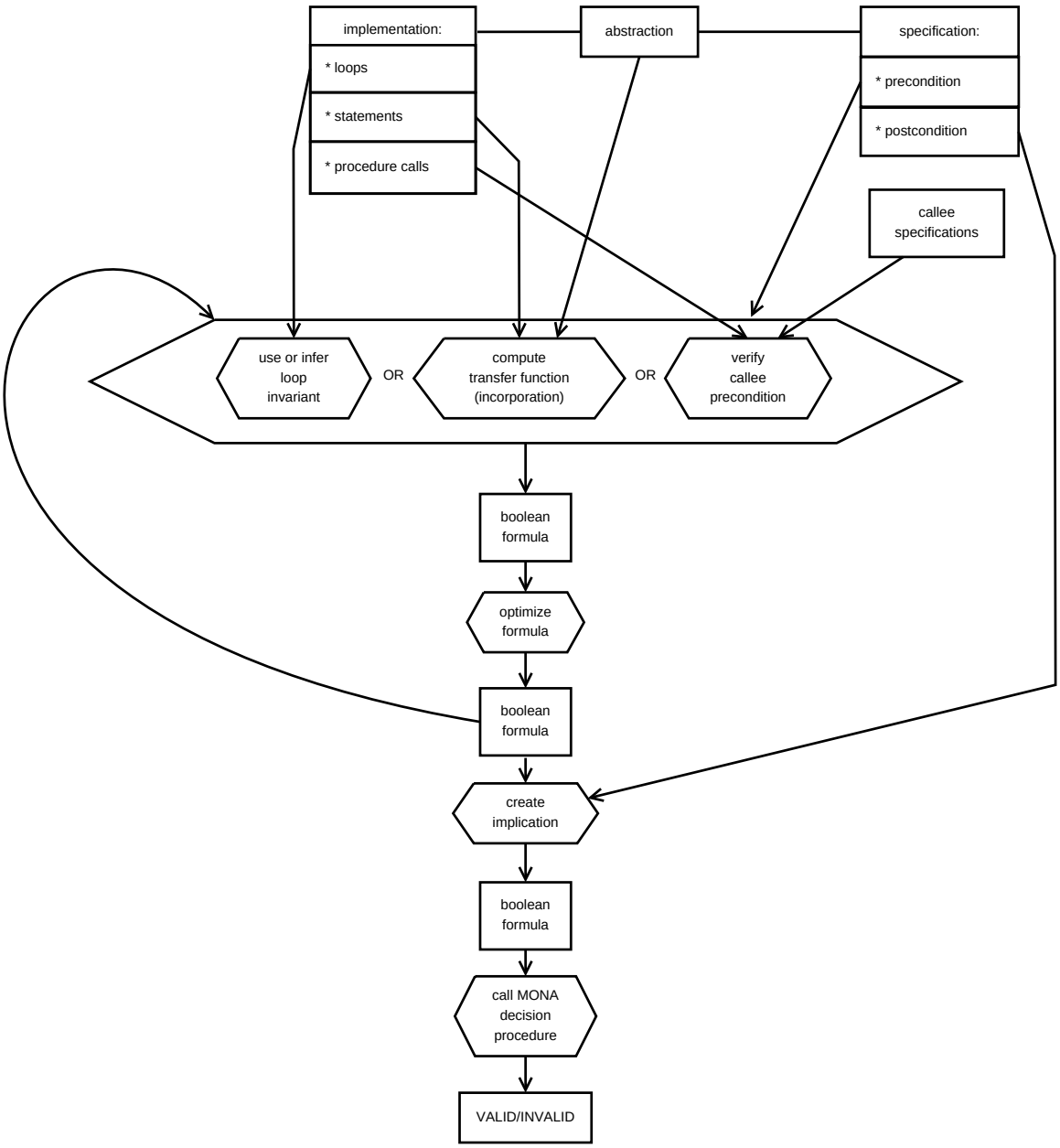

Figure 6-2: Flowchart for flags analysis plugin. Boxes represent data. Hexagons represent actions.

that the effective postcondition (which includes the `ensures` clause and any required representation or global invariants) holds at all exit points of the procedure. In particular, the flags analysis checks that for each exit point e, the computed formula B_e implies the procedure's postcondition.

6.3 Incorporation

The transfer functions in the dataflow analysis update boolean formulas to reflect the effect of each statement. Recall that the dataflow facts for the flags analysis are boolean formulas B which denote a relation between the state at procedure entry and the state at the current program point. Let B_s be the boolean formula describing the effect of statement s. Our flag analysis uses the incorporation operation to update B with the effect of B_s. The incorporation operation $B \circ B_s$ computes the composition of the relations defined by the formulas B and B_s.

Incorporation example. Let $B \equiv y' = y \wedge x' = x \wedge S' = S \wedge S = x$. We explain how the flags plugin abstractly executes the statement `s: y = x`. To execute this statement, the plugin must incorporate $B_s \equiv y' = x \wedge x' = x \wedge S' = S$ (representing the effect of `s`) into B (the state before `s` executed). Incorporation proceeds by quantifying over hatted sets $\hat{v}$, substituting $\hat{v}$ for v' in B and $\hat{v}$ for v in B_s and applying quantifier elimination. This gives the formula

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

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

Definition of incorporation. The flags plugin computes $B \circ B_s$ by applying equivalence-preserving simplifications to the formula

$$\exists \hat{S_1}, \ldots, \hat{S_n}, \hat{b_1}, \ldots, \hat{b_j}.\ B[S_i' \mapsto \hat{S_i}, b_j' \mapsto \hat{b_j}] \wedge B_s[S_i \mapsto \hat{S_i}, b_j \mapsto \hat{b_j}]$$

Incorporation computes the abstract state after executing s in state B for the following reason. The desired abstract state is the relation between the sets upon entry to the procedure (expressed in terms of unprimed, unhatted sets and booleans S_i and b_j) and after s has executed (expressed in terms of primed sets and booleans S_i' and b_j'). Incorporation creates (using existential quantification) the hatted sets $\hat{S_i}$ and hatted booleans $\hat{b_i}$, and uses them to represent the abstract state after B by substituting primed variables of B by hatted variables. Since B_s describes the relation between the program's abstract state before executing s (represented in B_s by unprimed variables) and after executing s (represented by primed variables), incorporation substitutes the unprimed variables of B_s with hatted variables. Conjoining the substituted B and B_s formulae therefore gives a relation which expresses the program's abstract state after executing s from state B.

6.4 Transition Relations

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

$$[\![\text{st}]\!](B) := B \circ \mathcal{F}(\text{st}),$$

where $\mathcal{F}(\text{st})$ is the formula symbolically representing the transition relation for the statement st, as expressed in terms of abstract sets.

Frame condition generator. Before providing transfer functions for implementation language statements, we define a generic frame condition generator. This frame condition generator will show up in most of our transfer functions. The generator creates a boolean formula which states that a particular variable may potentially be modified, but that all other sets and booleans are unmodified. Let

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

where S ranges over sets and b over boolean variables.

Recall that the set specification language enables developers to define base sets and derived sets. A base set definition has the form $\{\texttt{x:T} \mid \ldots\}$. Base set definitions may be named ($S = \{\texttt{x:T} \mid \ldots\}$) or anonymous (when a base set definition occurs as part of a larger derived set definition). Derived set definitions combine named sets and anonymous set definitions using set operations.

Note that our definition of the frame condition explicitly omits derived sets. Instead, the flags analysis creates a formula stating that the anonymous base sets used in the derived set definitions are preserved and conjoins derived set definitions before applying the decision procedure. This treatment automatically works for derived sets and helps avoid inconsistency: as long as the base sets making up a derived set are preserved, then the derived set is preserved as well.

We continue by presenting transition relations for the statements in our implementation language.

Assignment statements. Our flags analysis tracks values of boolean variables:

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

where $f^+(e)$ is the result of evaluating e, defined below in our analysis of conditionals.

The analysis also tracks local variable object references:

$$\begin{array}{rcl}\mathcal{F}(\mathtt{x}=\mathtt{y}) & := & (\mathtt{x}'=\mathtt{y}) \wedge \mathsf{frame}_x \\ \mathcal{F}(\mathtt{x}=\mathsf{null}) & := & (\mathtt{x}'=\emptyset) \wedge \mathsf{frame}_x \\ \mathcal{F}(\mathtt{x}=\mathsf{new}\ t) & := & \neg(\mathtt{x}'=\emptyset) \wedge \bigwedge_S(\mathtt{x}' \cap S = \emptyset) \wedge \mathsf{frame}_x\end{array}$$

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

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

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

The rules for reads and writes of boolean fields are more detailed than those for field variables because our analysis tracks the flow of boolean values:

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

where $B^+ = \{x : T \mid x.\mathtt{f} = \mathsf{true}\}$ and $B^- = \{x : T \mid x.\mathtt{f} = \mathsf{false}\}$.

The rules presented above do not overlap in their applicability. However, they do not cover all statements in the Hob implementation language. We therefore use a pair of default rules to conservatively account for expressions not otherwise handled,

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

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

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

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

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

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

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

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

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

$$\begin{array}{rcl@{\qquad}rcl} f^+(x.f) & := & x \subseteq B & f^-(x.f) & = & x \not\subseteq B \\ f^+(x.f\mathsf{==false}) & := & x \not\subseteq B & f^-(x.f\mathsf{==false}) & := & x \subseteq B \end{array}$$

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

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

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

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

Finally, we have a catch-all condition,

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

which conservatively captures the effect of unknown conditions.

Assertions and Assume Statements. We analyze a statement s of the form `assert A` by verifying that the formula for the program point s implies `A`. Assertions allow developers to check that a given set-based property holds at an intermediate point of a procedure. `assume` statements enable the developer to specify properties that are known to be true, but which have not been shown to hold by the analysis. Our analysis prints out a warning message when it processes `assume` statements, and conjoins the assumption to the current dataflow fact. Assume statements have proven to be valuable in understanding analysis outcomes during the debugging of procedure specifications and implementations. Assume statements may also be used to communicate properties of the implementation that go beyond the abstract representation used by the analysis.

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

6.5 Verifying Implication of Dataflow Facts

Our flags analysis verifies implication when it encounters an assertion, procedure call, or procedure postcondition. In these situations, the analysis generates a formula of the form $B \Rightarrow A$ where B is the current dataflow fact and A is the claim to be verified[1]. The implication to be verified, $B \Rightarrow A$, is a formula in the boolean algebra of sets. We use the MONA decision procedure to check its validity [51].

6.6 Loop Invariant Inference

Section 4.2.2 described our loop invariant inference algorithm; we next describe its implementation. The synthesis starts with the formula characterizing the transition relation at the entry of the loop and weakens the formula by iterating the analysis of the loop until it reaches a fixpoint. Figure 6-3 presents pseudocode for the algorithm. COMPUTE-POSTCONDITION is the algorithm that we have presented in the preceding section. This algorithm takes a boolean formula f and a statement s and outputs the boolean formula corresponding to the program state after executing s, if f was the state before executing s. The top-level function INFER-LOOP-INVARIANT therefore attempts to find invariants by taking those conjuncts which are common to both the pre-state f and the post-state f' of the loop (technically, it identifies conjuncts c which are implied by both f and f'). If loop invariant inference takes too long, then our algorithm enforces termination by dropping conjuncts from f'. The GET-IMPLIED-CONJUNCTS subroutine finds those conjuncts of its first parameter f_1 which are implied by the second parameter f_2, while the HANDLE-EXISTENTIAL subroutine handles existential quantifiers by dropping sub-conjuncts (underneath the existential quantifier) that are not implied by the source formula.

6.7 Boolean Algebra Formula Transformations

In our experience, applying several formula transformations drastically reduced the size of the formulas emitted by the flags analysis, as well as the time needed to determine their validity using an external decision procedure; in fact, some benchmarks could only be verified with the formula transformations enabled. This section describes a number of useful transformations that we discovered.

Smart Constructors. The constructors for creating boolean algebra formulas apply peephole transformations as they create the formulas. Constant folding is the simplest peephole transformation: for instance, optimizing $B \wedge \mathsf{true}$ gives B. Our

[1]Note that B may be unsatisfiable. This often indicates a problem in a procedure precondition. The flags analysis can, optionally, check whether B is unsatisfiable every time it invokes the decision procedure, and emit a warning if it is. This check enabled us to identify errors in preconditions sooner; of course, it also slowed down the flags analysis by a factor of 2. Without such a check, unsatisfiable preconditions become visible only at calls to affected preconditions, which are analyzed separately—and possibly much later—due to modular verification

```
INFER-LOOP-INVARIANT(f_0, loop-condition, loop-body, max-iterations)
 1  i ← 0
 2  f ← f_0
 3  f' ← COMPUTE-POSTCONDITION(f ∧ loop-condition, loop-body)
 4  while i < max-iterations and f' ⇏ f
 5      do f ← GET-IMPLIED-CONJUNCTS(f, f', []) ∧ GET-IMPLIED-CONJUNCTS(f', f, [])
 6         f' ← COMPUTE-POSTCONDITION(f ∧ loop-condition, loop-body)
 7         i ← i + 1
 8  if i ≥ max-iterations
 9      then while f' ⇏ f
10                do f ← GET-IMPLIED-CONJUNCTS(f, f', [])
11                   f' ← COMPUTE-POSTCONDITION(f ∧ loop-condition, loop-body)
12  return f

GET-IMPLIED-CONJUNCTS(f_1, f_2, [x_0, ..., x_n])
 1  result ← True
 2  foreach c in CONJUNCTS(f_1)
 3      if f_2 ⇒ ∃x_0, ..., x_n.c
 4          then result ← c ∧ result
 5      else if c has the form ∃x.e
 6          then result ← HANDLE-EXISTENTIAL(e, f_2, [x_0, ..., x_n, x]) ∧ result
 7  return result

HANDLE-EXISTENTIAL(e, f, [x_0, ..., x_n])
 1  g ← GET-IMPLIED-CONJUNCTS(e, f, [x_0, ..., x_n])
 2  if f ⇒ ∃x_0, ..., x_n.g
 3      then return ∃x_n.g
 4  g ← True
 5  foreach c in CONJUNCTS(e)
 6      if c does not contain x_n
 7          then g ← c ∧ g
 8  return GET-IMPLIED-CONJUNCTS(g, f, [x_0, ..., x_{n-1}])
```

Figure 6-3: Pseudo-code for Loop Invariant Inference Algorithm

constructors fold constants in implications, conjunctions, disjunctions, and negations. Similarly, when there is a quantification over a variable that is not subsequently used, we simply drop the quantifier: $\exists x.F$ becomes just F as long as x does not occur free in F. Most interestingly, we factor common conjuncts out of disjunctions: $(A \wedge B) \vee (A \wedge C)$ is optimized to $A \wedge (B \vee C)$. Conjunct factoring greatly reduces the size of formulas tracked after control-flow merges, since most conjuncts are shared on both control-flow branches following a conditional. The effects of the conjunct factoring transformation appears to be similar to the effects of SSA form conversion in weakest precondition computation [37, 63].

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

It is instructive to compare our technique to weakest precondition computation [37] and forward symbolic execution [16]. These techniques are optimized for the common case of assignment statements and perform relation composition and quantifier elimination in one step. Our technique—using incorporation and then performing a range of ad-hoc formula optimizations—achieves the same result in practice, but is easier to implement and also enables the optimization of general boolean formulas. Our technique can therefore also take advantage of equalities in transfer functions that are not a result of analyzing assignment statements, but are given by explicit formulas in `ensures` clauses of procedure specifications. Such transfer functions may specify more general equalities such as $A = A' \cup x \wedge B' = B \cup x$ which do not reduce to simple backward or forward substitution.

Leveraging Quantifier Elimination in Implications The flags analysis rewrites $\forall x.f \Rightarrow g$ as $\neg(\exists x.f \wedge \neg g)$. Once the analysis expresses implications this way, the quantifier-elimination optimization applies to the existential quantifier inside the negation, which can greatly reduce the size of the formulas that need to be verified. Since formulas with explicit implications are easier to understand, we have added a runtime flag which specifically disables this optimization for debugging purposes.

Quantifier Nesting. We have experimentally observed that the MONA decision procedure works substantially faster when each quantifier is applied to the smallest scope possible. We have therefore implemented a quantifier nesting step that reduces the scope of each quantifier to the smallest possible subformula that contains all free

variables in the scope of the quantifier. For example, our transformation replaces the formula $\forall x.\ \forall y.\ (f(x) \Rightarrow g(y))$ with $(\exists x.\ f(x)) \Rightarrow (\forall y.\ g(y))$.

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

6.8 Evaluating Formula Optimization Impact

We analyzed our benchmarks on a 2.80GHz Pentium 4, running Linux, with 2 gigabytes of RAM. Table 6.1 summarizes the results of our formula transformation optimizations. Each line summarizes a specific benchmark with a specific optimization configuration. A ✓in the "Smart Constructors" column indicates that the smart constructors optimization is turned on; a × indicates that it is turned off. Similarly, a ✓in the "Optimizations" column indicates that all other optimizations are turned on; a × indicates that they are turned off. The "Number of nodes" column reports the sizes (in terms of AST node counts) of the resulting boolean algebra formulas. Our results indicate that the formula transformations reduce the formula size by 2 to 60 times (often with greater reductions for larger formulas); the Optimization Ratio column presents the reduction obtained in formula size. The "MONA time" column presents the time spent in the MONA decision procedure (up to 73 seconds after optimization); the "Flags time" column presents the time spent in the flags analysis, excluding the decision procedure (up to 477 seconds after optimization). Without optimization, MONA could not successfully check the formulas for the compiler, board, view, ensemble and h2o modules because of an out of memory error.

	Optimizations	Smart Constructors	Number of nodes	Optimization ratio	MONA time (s)	Flags time (s)
prodcons	✓	✓,×	12306	2.46	0.17	0.03
	×	✓,×	30338	1.00	0.27	0.04
compiler	✓	✓	15854	32.06	0.45	5.10
	✓	×	28003	18.15	0.60	6.19
	×	✓,×	508375	1.00	N/A	60.27
scheduler	✓	✓,×	442	2.44	0.05	0.04
	×	✓,×	1082	1.00	0.12	0.14
ctas	✓	✓,×	2874	3.18	0.21	0.12
	×	✓,×	9141	1.00	12.79	0.33
board	✓	✓	28658	41.43	1.92	18.89
	✓	×	106550	11.14	11.45	29.27
	×	✓	926321	1.28	N/A	134.94
	×	×	1187379	1.00	N/A	151.46
controller	✓	✓	6759	4.23	0.41	0.18
	✓	×	7101	4.02	0.41	0.18
	×	✓,×	28594	1.00	3.08	0.54
view	✓	✓	15878	59.08	1.07	12.38
	✓	×	53925	17.39	1.45	18.88
	×	✓,×	938000	1.00	N/A	263.15
atom	✓	✓	9677	3.14	0.53	0.13
	✓	×	10244	2.97	0.54	0.13
	×	✓,×	30447	1.00	40.95	0.43
ensemble	✓	✓	120279	20.60	50.90	34.15
	✓	×	148748	16.66	105.59	47.06
	×	✓,×	2478004	1.00	N/A	464.52
h2o	✓	✓	205933	4.32	73.80	477.01
	✓	×	206167	4.31	81.85	475.86
	×	✓,×	889637	1.00	N/A	1917.99

Table 6.1: Formula sizes before and after transformation. The entry ✓, × in a Smart Constructors column indicates that the smart constructors did not affect the results in that row.

Chapter 7

Experience

We have implemented our modular pluggable analysis system, populated it with several analyses (including the flags, Bohne shape analysis, and theorem prover plugins), and used the system to develop several benchmark programs and applications.

7.1 Data Structure Implementations

We have verified a number of data structure implementations using the Hob system. Our experience confirms the hypothesis that the Hob system can successfully verify that data structure implementations preserve local invariants and conform to their interfaces. A data structure implementation conforms to its interface when all of the procedures in the implementation satisfy their postconditions upon exit, assuming that those procedures' preconditions held upon entry. As we have described earlier, abstraction functions mediate between the concrete heap operations of the implementations and the abstract set operations of the interfaces.

Using the Bohne shape analysis plugin, we have successfully verified singly-linked lists, doubly-linked lists with and without iterators and header nodes, and two-level skip lists. Section 4.3 explained how developers specify and verify properties with the Bohne plugin. We have also verified properties of queues, stacks, trees and priority queues using the PALE shape analysis plugin, a forerunner to the Bohne plugin. When the developer supplies loop invariants, Bohne verifies data structure consistency properties in times ranging from 1.7 seconds (for the doubly-linked list) to 8 seconds (for insertion into a tree). Bohne automatically infers loop invariants for insertion and lookup in the two-level skip list in 30 minutes total.

7.1.1 Tree data structure

We have used the Bohne plugin to verify insertion into a binary search tree. This tree maintains an abstract set `S` of objects representing the contents of the tree data structure. The following definition gives the translation of the concrete heap state into the abstract set `S`. In words, it states that `S` contains the set of objects reachable from the `root` module-level variable through `left` and `right` fields.

```
S = {x : Entry |
      "rtrancl (lambda v1 v2. left v1 = v2 | right v1 = v2) root x"};
```

The Bohne plugin automatically verifies that the tree's backbone is acyclic and that the backbone forms a tree along the `left` and `right` edges. In addition, we explicitly instruct Bohne to verify the following two invariants:

```
invariant "ALL x. x ~= null &
        ~(rtrancl (lambda v1 v2. left v1 = v2 | right v1 = v2) root x) -->
                ~(EX y. y ~= null & (left y = x | right y = x)) &
                (left x = null) & (right x = null)";

invariant "ALL x y. parent x = y -->
              (x ~= null & (EX z. left z = x | right z = x) -->
                          (left y = x | right y = x))
            & (((ALL z. left z ~= x & right z ~= x) | x = null)
                    --> y = null)";
```

The first invariant states that that all heap objects `x` (except `null`) that do not belong to the tree are not pointed to by any other object `y` in the heap[1], and additionally that such objects `x` have `left` and `right` fields set to null. This invariant ensures, in particular, that there are no "loose" tree fragments (in terms of this module's `left` and `right` fields) in the heap that exist independent of the main tree. Such fragments are potentially problematic because they may cause insertions to add unanticipated extra objects to the tree.

The second invariant is a field constraint on the `parent` field, which is a derived field—that is, the `parent` field can be defined in terms of the `left` and `right` fields. In particular, this field constraint states that the `parent` field is the inverse of the `left` and `right` fields. More precisely, if a heap object `x`'s parent pointer points to `y` (again, for `x` non-null), and if there is some object `z` which has `x` as a child, then `y` has `x` as a child. The second invariant additionally states that if `x` has no parent (quantifying over the entire heap), then `x`'s `parent` field must be set to `null`.

Note that these two invariants describe the pointer structure of the tree, and do not discuss any sortedness properties for tree elements. Sortedness properties, which are properties of integers, are beyond the scope of the Bohne shape analysis plugin. The Hob analysis approach enables developers to state and verify *partial* properties of data structures and programs. Developers who are concerned with the sortedness of the tree could invent and use a specialized plugin that could reason about properties of integers. Alternatively, developers could use the theorem proving plugin, which can handle arbitrarily complicated properties at the cost of developer effort, to verify the desired sortedness properties.

One drawback of verifying partial properties is that such partial properties might not, by themselves, be strong enough to enable the verification of other desired properties. The two invariants stated above are too weak to enable the verification of any

[1]Note that this reachability relation is defined by only the `left` and `right` edges. Hob's format mechanism enables the Bohne plugin to safely ignore all other fields in the heap.

interface for a `remove` procedure that states that the procedure removes its parameter from the tree. The issue is that any efficient implementation of a `remove` procedure—which would remove its parameter from the tree, assuming that the parameter is in the proper position—must rely on the ordering of elements in the tree. If the concrete tree in the heap were to contain improperly sorted elements, then efficient implementations of `remove` would not be able to correctly remove the requested object, which would make it impossible to guarantee the procedure's desired postcondition. Note that a stronger invariant language would enable the verification of `remove`.

We can, however, verify the `add` procedure for trees. This procedure's specification states that the procedure adds its parameter `e` to the set `S`.

```
proc add(e:Entry; v:int) requires card(e) = 1 & not (e in S)
                         modifies S
                         ensures S' = S + e;
```

Figure 7-1 presents the complete implementation of the `add` procedure. This procedure implements a standard search for `e` in the tree, removing it if present. The procedure is remarkable only for its loop invariant. Bohne is, in principle, capable of inferring this loop invariant given suitable abstraction predicates. However, the verification finishes in dozens of seconds rather than dozens of minutes if the developer supplies the invariant explicitly. The invariant for `add` states the following properties:

- The parameter `e` is non-null and remains unchanged.
- The objects `e`, `n` and `p` are all reachable in the tree.
- If local variable `p` is non-null, then `n` is the child of `p`.
- If `p` and `n` are both null (indicating an empty tree), then `root` also contains `null`.
- The definition of the set `S` continues to hold: an object `x` belongs to the abstract set `'S`[2] if and only if `x` is reachable from the root through `left` and `right` fields.
- For all non-null objects `x` in the heap that do not belong to the tree, no non-null object `y` points to `x`, and that `x`'s `left` and `right` fields are `null`.

Note that we restate the module's invariants within the loop invariant. In general, module invariants must be stated explicitly in the loop invariant because they might be temporarily violated during the execution of the `add` procedure; stating them explicitly guarantees that they are not violated across loop iterations.

[2]The prime indicates that the statement is about the current value of `S`; Bohne expects the prime before the set name, rather than after it, as in the general Hob convention.

```
1  proc add(e:Entry; v:int) {
2   e.v = v;
3   e.left = null; e.right = null; e.parent = null;
4   Entry n = root, p = null;
5   bool wentLeft;
6   while "e ~= null & e = 'e &
7          ~(rtrancl (lambda v1 v2. left v1 = v2 | right v1 = v2) root e) &
8          rtrancl (lambda v1 v2. left v1 = v2 | right v1 = v2) root n &
9          rtrancl (lambda v1 v2. left v1 = v2 | right v1 = v2) root p &
10         (p ~= null --> (left p = n & wentLeft | right p = n & ~wentLeft)) &
11         (p = null & n = null --> root = null) &
12         (ALL x. (x : 'S) <=>
13            rtrancl (lambda v1 v2. left v1 = v2 | right v1 = v2) root x) &
14         (ALL x. x ~= null &
15          ~(rtrancl (lambda v1 v2. left v1 = v2 | right v1 = v2) root x) -->
16              ~(EX y. y ~= null & (left y = x | right y = x)) &
17              (left x = null) & (right x = null))"
18    (n != null) {
19    p = n;
20    wentLeft = (v < n.v);
21    if (wentLeft)
22      n = n.left;
23    else
24      n = n.right;
25  }
26  if (p == null) {
27    root = e;
28  } else {
29    e.parent = p;
30    if (wentLeft) {
31      p.left = e;
32    } else {
33      p.right = e;
34    }
35  }
36 }
```

Figure 7-1: Implementation of `TreeSet` insert procedure

7.1.2 Stack data structure

We next describe one aspect of our experience verifying a stack implemented as a doubly-linked list. The PALE shape analysis plugin (a predecessor to our current Bohne shape analysis plugin) discovered an invariant violation in the course of verifying the stack's implementation. Like our tree, our stack data structure maintains an abstract set `S` representing the contents of the stack. The Hob system verifies that stack insertions actually insert the given object into the stack (the `insert` procedure ensures that `S' = S + e`), and that removals actually remove an object from the stack, if possible (the `removeFirst` procedure ensures that `card(S) = 0 | (exists e:Entry. (S' = S - e) & card(e) = 1)`).

Hob's shape analysis plugins use developer-provided invariants to check that objects that belong to a set have consistent values for navigational fields (e.g. `next`, `prev`), and that objects that do not belong to the set have their navigational fields set to `null`. Our experience suggests that is not difficult to write implementations that inadvertently violate these invariants. Our initial implementation for the `removeFirst` procedure was as follows:

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

where the `pragma` statement indicates to the PALE analysis plugin that it is verifying a set removal. We found that the analysis reports an error while verifying this implementation. Careful inspection of the above code reveals that the removed object, `res`, retains a reference to an object in the stack even after its removal. Such an implementation violates the invariant that objects not belonging to the data structure must have their `next` and `prev` fields set to `null`. Unexpected field values for "orphan" heap objects may in turn lead to non-list structures appearing in the heap. Adding `res.next = null` to this procedure satisfies the PALE plugin: setting each object's `next` field to `null` on exit enables PALE to verify the invariant that all objects passed in to the `insert` procedure will have their `next` field set to `null`.

7.2 Water

Having described the verification of a few data structures using the Hob analysis system, we continue by describing applications of Hob to verifying complete applications. Our first application is water, a port of the Perfect Club benchmark MDG [10].

Benchmark description. The water benchmark evaluates forces and potentials in a system of water molecules in the liquid state using a predictor/corrector method. The central loop of the computation performs a time step simulation. Each step

predicts the state of the simulation, uses the predicted state to compute the forces acting on each molecule, uses the computed forces to correct the prediction and obtain a new simulation state, then uses the new simulation state to compute the potential and kinetic energy of the system.

Our implementation of the water benchmark includes the `simparm`, `atom`, `H2O`, `ensemble`, and `main` modules, as well as a number of helper modules. Figure 7-2 presents the module dependency diagram for the water benchmark, with an arrow between the box for module `A` and the box for module `B` indicating that module `A` calls module `B`. These modules contain 2000 lines of implementation and 500 lines of specification.

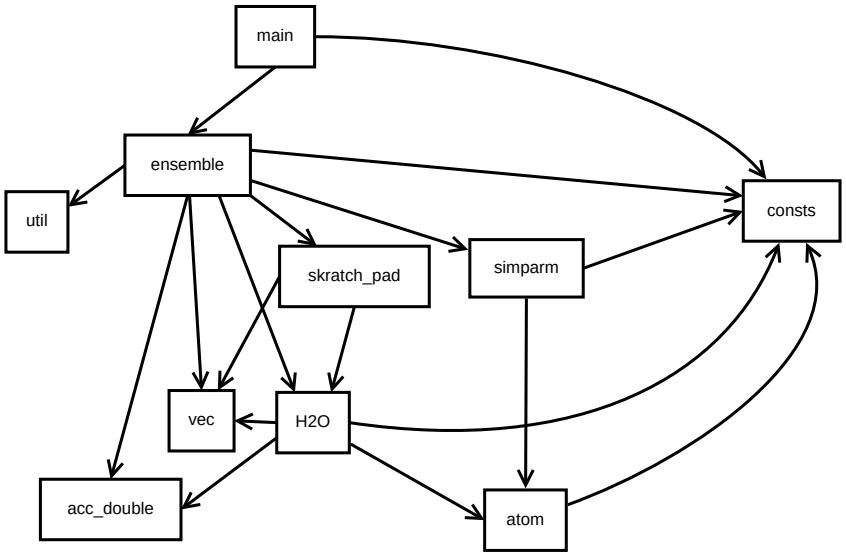

Figure 7-2: Module dependency diagram for water benchmark

The `main` module initializes the program state, calls the main loop (which is in the `ensemble` module), and prints out the final program state. The `ensemble` module captures the state of the entire computation and calls the other modules to actually carry out the computation. The `simparm` module stores inputs to the computation and values that are computed once at the beginning of the computation, while the related `consts` module stores physical constants used by the simulation. The `H2O` module stores collections of atoms, which are tracked by the `atom` module. The helper modules `skratch_pad`, `vec`, `acc_double` and `util` perform auxiliary tasks.

Consistency properties. We verified the following properties for the water benchmark. These properties can be characterized mainly as typestate properties; for the most part, they do not describe data structure properties, since the water benchmark does not maintain specific relationships between different objects in the heap.

- Space for simulation parameters is always allocated and the parameters always loaded before accesses to the simulation parameters.

- The `predic` and `correc` actions on `atom` objects are always interleaved: no atom is corrected unless it has just been predicted, and vice-versa.
- Actions on molecules are properly sequenced: for instance, a molecule always has its kinetic energy calculated before the boundary box is applied to it.
- Global computation state transitions are consistent with the transition order declared by the developer.

The Hob system verifies that the program does not load simulation parameters before it allocates arrays for holding these parameters, and that the program does not access the simulation parameters until they have been loaded from the disk and stored into the arrays. The `simparm` module is responsible for storing simulation parameters, which are loaded from a text file at the start of the computation. To track the current state, this module defines two boolean variables, `Init` and `ParmsLoaded`. If `Init` is true, then the module has been initialized, *i.e.* the appropriate arrays have been allocated on the heap. If, additionally, the variable `ParmsLoaded` is true, then the simulation parameters have been loaded from disk and written into these arrays.

One important property of the main computation concerns atoms (handled by the `atom` module); atoms are the fundamental unit of this simulation. Atoms cycle between the *predicted* and *corrected* states, which are distinguished by values of the `predic` and `correc` flags on atoms. The `predic` and `correc` procedures perform the computations necessary to effect these state changes. Only atoms in the "corrected" state may have their position predicted, and only atoms in the "predicted" state may have their position corrected. To enforce this property, we define two sets, `Predic` and `Correc`, and populate them with predicted and corrected atoms, respectively. The `correc` procedure operates on a single atom; its precondition requires this atom to be a member of the `Predic` set. The `correc` procedure's postcondition ensures that, upon exit, the atom is no longer in the post-state of the `Predic` set, but is instead in the post-state of the `Correc` set. The `predic` procedure has the corresponding symmetric specification.

The next step up from the atom is the molecule. Molecules (handled by the `H2O` module) contain three atoms, tracking their position and velocity. We verify that when a molecule is in the predicted or corrected state, the atoms in the molecule are also in the same state. Molecule states indicate not only whether the program has predicted or corrected the position of the molecule's atoms, but also whether the program has applied intra-molecule force corrections, whether it has scaled the forces acting on the molecule, and other similar properties. The interface of the `H2O` module can therefore ensure that the program performs the operations on each molecule in the correct order—for example, the `bndry` procedure may only be called with molecules in the `Kineti` set, which have had their kinetic energy calculated by the `kineti` procedure.

Finally, the `ensemble` module manages the collection of molecule objects. This module stages the entire simulation by iterating over all molecules and computing their positions and velocities over time. The ensemble module uses boolean predicates to track the state of the computation as a whole. When the ensemble's boolean

predicate `INTERF` is true, for example, then the program has completed the interforce computation for all molecules in the simulation. By encoding allowable state transitions into procedure preconditions and postconditions, our analysis verifies that the program's state progresses in only the following order:

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

For example, the benchmark has a procedure `INTRAF`. This procedure requires that the boolean flags `INITIA` and `PREDIC` be true upon entry, and ensures that the flag `INTRAF'` is true upon exit.

Unsoundness. Hob's successful verification of the water benchmark depends on an implication from the simulation's global boolean predicates to properties ranging over the collection of molecule objects. We do not currently verify this particular implication; instead, we currently use `assume` statements to let the verification go through. In the short term, the developer can manually verify all of a program's `assume` statements by inspecting the code. We foresee two possible longer-term solutions: the developer may use a theorem prover to verify the properties that are beyond the reach of Hob's current analysis plugins, or a new analysis plugin that can verify the relevant properties could become available.

Discussion. The properties that we verify for the water benchmark ensure that the computation's phases execute in the correct order; such properties are especially valuable in the maintenance phase of a program's life, when the original designer, if available, may have long since forgotten the program's phase ordering constraints. Incidentally, Hob specifications' set cardinality constraints also prevent empty sets (and null pointers) from being passed to procedures that expect non-empty sets or non-null pointers.

7.3 HTTP Server

The HTTP 1.1 server implements a server which responds to requests for web pages. We have used this server to host the Hob project homepage.

Benchmark description. Our web server reads configuration data from disk and then listens for HTTP requests on the port specified in the configuration file. It serves responses to these requests by transmitting the appropriate headers and content to the client. If the client's request indicate that it supports compression, the server uses library routines to compress the data using the gzip algorithm, and then sends the compressed version to the client. Furthermore, we optimized our HTTP server by caching the results of previous requests (both uncompressed and compressed) in memory and serving results from the cache whenever possible.

Figure 7-3 presents a module dependency diagram for our web server. The `HTTPServer` module receives connections and sends responses to the client. It uses the

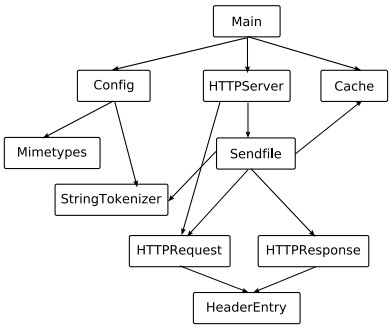

Figure 7-3: Module dependency diagram for web server

`HTTPRequest` and `Sendfile` modules to process the request and send the response, respectively. The `Sendfile` module takes a filename and a connection and serves the appropriate response to the client, using the `Cache` module to store file contents in memory for later requests. In all, the Hob webserver contains 14 modules, 1229 lines of implementation, and 335 lines of specification.

The HTTP server includes the following sets of objects. `HTTPRequest.Headers` stores a set of HTTP request headers. The related `HTTPResponse.C` set stores HTTP response headers. The cache uses a pair of sets, `CacheSet.Content` and `CacheBlacklist.Content`, to store past requests and (if appropriate) their corresponding responses. The `CacheSet.Content` set stores objects that point to responses to certain requests, while the `CacheBlacklist.Content` set contains information about objects that must not be placed in the cache (typically because they are too large).

Data structure consistency properties. Our implementation of the web server maintains the following consistency properties. Some of these properties constrain the heap and prevent corruption in the program's heap-based data structures. Other, more interesting, properties, summarize design decisions that we made during our implementation of the web server.

- The linked list making up the cache set maintains its list invariants (*e.g.*, the linked list `prev` field is the inverse of its `next` field).
- The server configuration is loaded before any requests are served.
- Response headers are always cleared between requests.
- Responses are always either served from the cache or blacklisted from the cache.

Serving a request. When serving an HTTP request, the server first reads data from the client describing the request and the form of response that the client is expecting. The server then creates an HTTP response header and populates the set `HTTPResponse.C` with the proper header entries. Next, it searches the cache

blacklist `CacheBlacklist.Content` and the cache content `CacheSet.Content` for cached versions of the response; if no cached content is available, and the content is not blacklisted, then it adds the content to the cache. The program then consults the `sendHead` and `sendBody` procedure parameters (which depend on the request) to determine whether it should serve the header and content, and serves the relevant parts of the response to the client.

Response headers. The usual structure of an HTTP response occurs in two parts: response header and content. A response header is a list of colon-separated strings, each string containing a key and a value. In our implementation, we build up an HTTP response in the `HTTPResponse` module. The `HTTPResponse` module also contains a procedure which sends the response, as constructed, over the network to a client.

Our use of sets allows us to document and statically enforce the usage pattern of the HTTP response module: we represent the current response header as a set, `HTTPResponse.C`, and add header entries to this set. Since we do not wish to emit stale header information from previous requests, the precondition of the `sendFile` procedure includes the condition that `card(HTTPResponse.C) = 0`. When serving any HTTP request, the web server always emits a basic header, including mandatory fields like the `Date` field; such fields enable us to guarantee that the `HTTPResponse.C` set is non-empty. We ensure that this precondition always holds by restoring it upon exit from `sendFile`; in particular, we ensure that `card(HTTPResponse.C') = 0`.

Note that this specification does not constrain the membership of `C` during the execution of the procedure. In fact, the `HTTPResponse.emit` procedure requires that `C` be non-empty; clearly, it is inconsistent with this particular design to transmit empty responses. A different (and in our opinion inferior) design might only populate the set `C` if the client had requested that headers be transmitted. Our specifications clearly document the design decision that we took in this particular implementation and prevent maintainers from inadvertently violating this design in the maintenance phase of the program's lifecycle.

Transmitting files to clients. The `sendFile` procedure coordinates the task of sending a file to a client, serving the file from the cache if possible. Content is generally stored in the cache before being served. To avoid undesirable cache effects, however, our server blacklists cache entities that are too large (greater than 1 megabyte in our current implementation). To simplify the implementation, we chose to have our web server always load the content into the cache and then serve the content from the cache, as long as the content is not blacklisted. Our implementation reflects this design decision. In the absence of any reliable information about the design, the developer would have to glean this design decision from the implementation, in particular by locating and understanding the following code in the `sendFile` procedure:

```
if (!Cache.hasEntry (c)) {
    /* ... [load content into t_array] ... */
    Cache.setEntryContent (c, t_array);
```

```
    if (!blacklist)
        Cache.addEntry (c);
}
else
    Cache.loadEntryContent (c);
/* ... */
Cache.sendEntry(oc, c);
```

and observing that the entry `c` is always loaded from the cache or populated from disk and, if not blacklisted, added to the cache.

Our approach makes this design decision explicit and much more accessible. We declare the sets `CacheSet.Content` and `CacheBlacklist.Content`. We defined these sets using instantiated linked lists, and Hob's ability to combine the shape analysis for the cache sets with the simpler typestate analysis used for this module is crucial for obtaining a global design conformance result. The `sendEntry` procedure, which transmits an entry to the client, relies on membership information for these two sets. This membership information propagates from postconditions of calls to the mediating `Cache` module. The specification for the `sendEntry` procedure therefore reads as follows.

```
private proc sendEntry (oc:out_channel; n:Entry) returns c:int
    requires (n in CacheSet.Content) |
             (n in CacheBlacklist.Content)
    ensures true;
```

Discussion. The specification of the `sendEntry` procedure makes it absolutely clear that the content to be transmitted will either be in the `CacheSet.Content` or `CacheBlacklist.Content` sets. The Hob analysis engine establishes the precondition for the `sendEntry` procedure by inspecting the rest of the `sendFile` procedure and observing that either the entry is already in the cache or newly added to the cache, so that `n in CacheSet.Content`; or the entry is blacklisted, in which case `n in CacheBlacklist.Content`. In this way, the `sendEntry` specification clearly and accessibly documents this design decision, and the Hob analysis system automatically verifies that the implementation correctly conforms to this design.

7.4 Minesweeper

Our next benchmark, minesweeper, shows how Hob can verify data structure consistency properties that span multiple modules.

Benchmark description. The minesweeper benchmark implements the standard model-view-controller (MVC) design pattern. Figure 7-4 presents a module dependency diagram containing the modules which make up the minesweeper implementation. The game board module `(Board)` represents the game state and plays the role of the "model" part of the MVC pattern; the controller module `(Controller)` responds

to user input; the view module (View) produces the game's output; the exposed cell module (ExposedSet) uses an array to store the cells that the player has exposed in the course of the current game; and the unexposed cell module (UnexposedList) instantiates a linked list to store the set of cells that have not yet been exposed. There are 750 non-blank lines of implementation code in the 6 implementation sections of minesweeper and 236 non-blank lines in its specification and abstraction sections.

The Board module stores one representation of the game state. (Game state information is also stored in the ExposedSet and UnexposedList modules, which must remain consistent with the Board.) At an abstract level, the board's sets MarkedCells, MinedCells, ExposedCells, UnexposedCells, and U (for Universe) represent sets of cells with various properties; the U set contains all cells known to the board. The board also uses a global boolean variable gameOver, which it sets to true when the game ends. Concretely, the Board stores an array of Cell objects and the global boolean variable. The Board module represents state information for each Cell using the isMined, isExposed and isMarked fields of Cell objects.

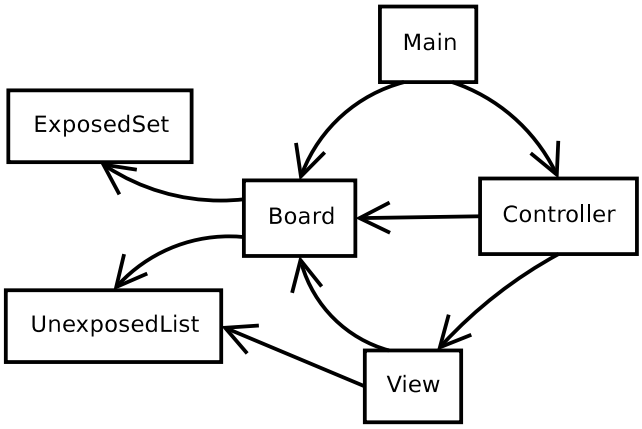

Figure 7-4: Module dependency diagram for Minesweeper implementation

Data structure consistency properties. The minesweeper application uses a variety of data structures and verifies a range of important consistency properties both within and between these data structures. Among the data structure consistency properties that the Hob system verifies are the following:

1. The set of unexposed cells in the UnexposedList module form an acyclic doubly-linked list with all prev references being inverses of next references.

2. The iterator pointer of the UnexposedList module is either null or points inside the list.

3. If the board is initialized, then the ExposedSet module storing the exposed cells is also initialized.

4. The set of unexposed cells maintained in the `Board` module (using flags) is identical to the set of unexposed cells maintained in the linked `UnexposedList` data structure.

5. The set of exposed cells maintained in the `Board` module (using flags) is identical to the set of exposed cells maintained in the `ExposedSet` array.

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

7. The sets of exposed and unexposed cells are disjoint.

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

Notice that the list of minesweeper properties contains two different kinds of properties: i) data structure consistency properties that involve the implementation of a single data structure, such as Property 1, and ii) more abstract properties involving relationships between objects stored in multiple data structures, such as Properties 4, 5, 6, and 8. One somewhat unusual feature of these abstract properties is that they are outward-looking: they capture important features of the system that are directly meaningful to the users of the system, and not just the implementors. To the best of our knowledge, the Hob system is the only currently existing system that supports and promotes the explicit identification and guaranteed checking of these kinds of outward-looking, application-oriented properties.

Verifying data structure use. Our minesweeper implementation uses iterators to process the list of unexposed cells in two contexts; both of these contexts are shown in Figure 7-5. One use of iteration is the `revealAllUnexposed` procedure, which is executed at the end of the game. This procedure causes the implementation to expose all of the `Board` cells. The second use is in a `peek` procedure which we added to our minesweeper implementation. The "peek" command allows the player to peek at all unexposed cells. We implemented this command by iterating twice over the set of unexposed cells, first exposing them, then hiding them.

Figure 7-5 contains loop invariants for our examples. These invariants help to explain how the flags analysis can analyze each of these examples. It turns out that our flags analysis plugin can successfully infer these loop invariants [59], thereby eliminating a potential source of annotation burden on the programmer. Furthermore, this invariant inference executes relatively quickly, in a number of seconds. We believe that one reason for the success of our loop invariant inference technique is that the technique operates at the level of abstract set variables.

Note that users of the linked list module always use the list through its interface; such users cannot directly manipulate the list itself. In other words, users of the linked list do not have access to the `next` and `prev` pointers making up the linked list structure. In general, verifying consistent interface use is simpler than verifying consistency of data structure operations, and our Hob system therefore uses the simpler but more efficient flags plugin to verify the consistency of data structure uses. In

```
// in Board specification

proc setExposed(c:Cell; v:bool) returns causedGameOver:bool
  ...
ensures (v => (ExposedCells' = ExposedCells + c)
         & (UnexposedCells' = UnexposedCells - c)
         & (UnexposedList.Iter' = UnexposedList.Iter - c))
     & ((not v) => ((ExposedCells' = ExposedCells - c)
                & (UnexposedCells' = UnexposedCells + c)))
     & ...

proc revealAllUnexposed()
  requires gameOver
  modifies ExposedCells, UnexposedCells
  ensures card(UnexposedCells') = 0;

// in Board implementation

proc peek() {
  peeking = true;
  Cell c;
  UnexposedList.openIter();
  bool b = UnexposedList.isLastIter();
  while "(b' <=> (UnexposedList.Iter' = {})) & peeking'"
  (!b) {
    c = UnexposedList.nextIter();
    View.drawCellEnd(c);
    b = UnexposedList.isLastIter();
  }
  // ... wait for key press ...
  UnexposedList.openIter();
  b = UnexposedList.isLastIter();
  while "(b' <=> (UnexposedList.Iter' = {})) & peeking'"
   (!b) {
    c = UnexposedList.nextIter();
    View.drawCell(c);
    b = UnexposedList.isLastIter();
  }
  peeking = false;
}

proc revealAllUnexposed() {
  UnexposedList.openIter();
  bool b = UnexposedList.isLastIter();
  // loop invariant in quotes below:
  while "... & (b' <=> (UnexposedList.Iter' = {})) &
   (UnexposedList.Iter' = UnexposedList.Content')" (!b) {
    Cell c = UnexposedList.nextIter();
    setExposed(c, true);
    b = UnexposedList.isLastIter();
  }
}
```

Figure 7-5: Doubly-Linked List Client. An optional loop invariant appears in quotes after the `while` keyword.

this example, we use the flags plugin to verify that the precondition for `nextIter` — namely, that the `Iter` set is nonempty—is always satisfied before calls to `nextIter`. Our implementations satisfy this constraint by first calling the `isLastIter` procedure and ensuring that it returns `false`.

The `peek` example nondestructively iterates over the `UnexposedList` set without changing the backing `Content` set, whereas the `revealAllUnexposed` procedure removes all elements from the list during iteration. The `revealAllUnexposed` procedure guarantees that the unexposed set is empty at the end of the procedure as follows. The procedure maintains the invariant that the `Iter` set equals the `Content` set during every loop iteration, because `nextIter` removes an element from the `Iter` set and `setExposed` removes the same element from `Content`. Note that the `revealAllUnexposed` loop runs until `isLastIter` returns `true`, which implies that `Iter` is true as well. Because of the equality between `Iter` and `Content`, the flags analysis plugin may conclude that, upon loop exit, `Content` is empty as well.

Hob's set abstraction supports typestate-style reasoning at the level of individual objects (for example, all objects in the `ExposedCells` set can be viewed as having a conceptual typestate `Exposed`). Our system also supports the notion of global typestate. (Note that we have used both of these sorts of typestates—local and global typestates—in the earlier water benchmark as well.) The `Board` module, for example, has a global `gameOver` boolean variable which indicates whether or not the game is over. The Hob system uses this variable and the definitions of relevant sets to ensure the preservation of the following scope invariant,

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

This scope invariant connects a global typestate property—is the game over?—with a object-based typestate state property evaluated on objects in the program—no mined cells are also exposed. As described in Chapter 3, the Hob system asks analysis plugins to verify these scope invariants by conjoining the invariants to procedure preconditions and postconditions. Note that scope invariants must be true in the initial state of the program. If some initializer must execute first to establish an invariant, then the invariant can be guarded by a global typestate variable which the initializer sets to true. Note the similarity between such a initialization guard and the `gameOver` guard that appears above.

A second scope invariant states equalities between sets:

$$\begin{array}{c}\texttt{(Board.ExposedCells = ExposedSet.Content)} \wedge \\ \texttt{(Board.UnexposedCells = UnexposedList.Content)}.\end{array}$$

This property ensures that the state of the board is consistent—in other words, that the `ExposedSet` and `UnexposedList` heap data structures and the `Board` do not contain contradictory information. The Hob system verifies this property by conjoining it to the `ensures` and `requires` clauses of appropriate procedures. In this case, it turns out that the `Board` module becomes responsible for maintaining this invariant.

Yet the analysis of the `Board` module does not, in isolation, have the ability to completely verify the invariant: the flags analysis cannot reason about the concrete state of `ExposedSet.Content` or `UnexposedList.Content` (which are defined in other modules). Instead, relying on the `ensures` clauses of `Board`'s callees, in combination with its own reasoning that tracks membership in the `ExposedCells` set, enables our analysis to verify the invariant (assuming that `ExposedSet` and `UnexposedList` work correctly).

7.5 Implications of Modular Analysis

While the Hob system was designed to verify both that modules preserve internal data structure consistency properties and that modules preserve consistency properties relating data structures, Hob's modular analysis approach approach often allows the two kinds of properties—properties of *coordination* modules and of *leaf* modules—to be verified separately. Coordination modules are those that define few, or no, abstract sets of their own, but instead coordinate the activity of other modules to accomplish tasks. In the minesweeper benchmark, the `View` and `Controller` modules are examples of such modules. The `View` module has no state at all; it simply queries the board for the current game state and calls the system graphics libraries to display the state. Conversely, leaf modules such as `ExposedSet` and `UnexposedList` often implement a single data structure and ensure that the data structure remains in a consistent state. Such modules do not coordinate the actions of other modules and usually state no inter-data structure consistency properties.

Because coordination modules coordinate the actions of other modules—and do not encapsulate any data structures of their own—the analysis of these modules only needs to operate at the level of abstract sets. Our flags analysis is capable of ensuring the validity of these modules since it can track abstract set membership, solve formulas in the boolean algebra of sets, and incorporate the effects of invoked procedures as it analyzes each module. Note that for these modules, our flags analysis need not reason about any correspondence between concrete data structure representations and abstract sets; it instead assumes that the modules which implement the sets properly implement the correspondence between implementations and specifications.

7.6 Summary and Reflections

We have used the Hob system to verify a number of data structures, including those based on linked lists and arrays, using theorem proving and shape analysis techniques. Furthermore, we have verified consistency properties for three complete applications: the water molecule simulation, a web server, and an implementation of the minesweeper game. These implementations include up to 2000 lines of implementation and 500 lines of specification. The specifications that we have checked using Hob include a number of properties that reflect applications' design information, enabling developers to verify that programs conform (and that they continue to conform) to their designs.

Reflections on the Hob specification approach. Our design decision limiting the expressive power of our specification language made this language especially suitable for specifying properties related to a program's design. The fact that Hob's set specifications focus on sets as abstractions of data structures—which are central to a program's operation—implies that such specifications can more effectively expose design information than full functional specifications. As an example, consider again the scope invariant about the disjointness of mined cells and exposed cells:

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

This invariant is remarkably concise. It states that either the `gameOver` boolean flag is true, or that the sets `MinedCells` and `ExposedCells` are disjoint. The invariant therefore constrains the program's state in a highly domain-specific way. Note that this invariant is not a generic property that holds for all programs, but rather a property specialized to this particular application. Additionally, this invariant states a fact that is relevant to end users: users expect that a minesweeper implementation should not expose a mined cell unless the game is over.

Furthermore, this invariant has a precise meaning: given any state of the concrete heap, it is possible to decide whether or not the invariant holds in that state. The Hob system decides whether or not an invariant holds by using the definitions of the `MinedCells` and `ExposedCells` sets. In this particular case, we defined both the `MinedCells` and `ExposedCells` sets using the flags plugin; for instance, the `MinedCells` set consists of the heap objects with fields `init` and `isMined` both set to `true`. Note that these set definitions have been factored out of the invariant itself and into the appropriate abstraction modules (as described in Chapter 4). Developers may therefore swap out set definitions and replace them with different definitions, even definitions which are to be verified using different analysis plugins. Invariants such as this one therefore illustrate how the Hob system enables developers to verify properties of arbitrarily complicated data structures and relationships between such data structures.

If we view invariants as distilled design information, then set definitions are irrelevant to the invariants, and the invariants are better expressed without inline set definitions. Consider the minesweeper invariant above. For the purposes of the minesweeper application's design, it is unimportant that the `ExposedCells` set consists of those `Cell` objects with field `isExposed` set to `true`. It is only important that the `ExposedCells` set and the `MinedCells` sets are disjoint. Of course, developers do need to agree on a common vocabulary before they can communicate using these sets; the need to assign meaningful names to sets is similar to the need to assign meaningful names to procedures and classes.

Our decision to use a set specification language also—unexpectedly—enabled us to deploy a simple loop invariant inference algorithm, which we previously described in Section 6.6. This algorithm worked fairly well in our experience and it contributed to our verification of the minesweeper and web server examples.

Of course, since our specifications are partial and set-based, they do not always

capture all important design decisions. For instance, one property that we would have liked to state and verify for the web server was that the content length, as stated in the response header, always corresponds to the number of bytes that we send to the client. However, this property is inexpressible in Hob's specification language: we chose to omit integers from the specification language to limit the complexity of the required decision procedure and to enable the use of pre-existing tools to decide formulas expressed in the Boolean algebra of sets.

Implementation language design. To evaluate the Hob approach, we had to develop programs for the Hob implementation language and write specifications for them. In our experience, it was inconvenient to port programs to the Hob implementation language, due to its lack of modern programming language features such as dynamic dispatch. In retrospect, we might have chosen to include more features in the programming language, which would have made it slightly more difficult to implement the Hob system, but much easier to write programs for it. Because one of the primary bottlenecks in our research was the availability of benchmarks, it seems that trading increased system development complexity for decreased benchmark development complexity would have been advantageous.

Consistency properties for leaf and coordination modules. Our experience illustrated that it was possible to verify consistency properties for leaf modules (which do not make any calls to other modules) using modular static analysis techniques. The analysis of the linked list using the PALE plugin showed that it is possible to use static analysis to verify properties that go beyond what is possible to verify using testing, since it would be difficult to construct a test case which exposes the problem.

Reasoning about coordination modules that use higher-level set specifications suggests that it is possible to use Hob's set specifications to build more scalable and more automated static analyses which verify design properties. We found that it was possible to verify typestate properties for systems, as we did in the water example. Such properties ensure that the proper operations occur in the correct order, both at a global level and at a per-object level. The minesweeper and web server examples furthermore demonstrated that it was possible to verify properties which related data structures (using their set specifications). In our experience, we found that these properties successfully expressed design-level information about the programs that we were verifying.

User relevance. We were surprised to find that Hob's set specifications are well-suited for expressing outward-looking user-level constraints on the program's behaviour. Generally, static analysis techniques operate by reading source code and creating models of the program's concrete data structures. It is possible to use these models to constrain permissible program states. However, in general, constraining concrete program states may or may not affect the program's observable behaviour: it is quite difficult to relate the state of a program's internal data structures and a set of desired program outputs.

Hob's set specifications, however, allowed us to express the following user-visible constraints in both the minesweeper and web server benchmarks:

- In the web server benchmark, set specifications ensure that response headers are always for the current request and are never stale.
- In the minesweeper benchmark, set specifications ensure that the mined cells are never exposed unless the game is over.

Hob's developer-provided set definitions enable static analyses to verify properties that directly affect user-relevant concerns (in our above example, contents of the response header and the set of mined cells) by translating them into constraints on the concrete program state (the state of the linked list or array). The abstraction functions that make up Hob set definitions therefore make it possible for our Hob analysis system to statically verify properties that directly affect the program's output. Hob's ability to state and verify properties that are directly relevant to users of the software makes Hob's verification approach especially compelling to developers and valuable to their end users.

Chapter 8

Related Work

The Hob specification language enables developers to succinctly express design properties. One of our primary goals in designing Hob was to make design information relevant, accessible and understandable. The Hob specification language therefore allows developers to specify a selected subset of critical design properties. We believe that our specification language hits a "sweet spot" between expressiveness and verifiability; it is targetted particularly towards expressing data structure properties. We contrast Hob's streamlined specification language to more powerful specification languages such as Z and VDM, which allow developers to specify (but not automatically verify) arbitrary properties of systems, as well as design notations such as UML, which are specialized for design properties (again, without verification support).

Hob relies on static analysis to automatically verify that systems conform to their design properties. We discuss related work on static analysis techniques, including typestate systems, shape analysis, model checking and abstract interpretation. Hob's primary contribution in this area is in integrating different analysis techniques by using the program's module structure and using the combined power of these analysis techniques to verify data structure consistency properties; we compare Hob to related research that combines decision procedures.

8.1 Specification Languages

Program specifications enable modular verification by enabling the verification of program parts—modules—against their interfaces. Most related work in the area of specification languages proposes complete methodologies for better software development. Hob, on the other hand, uses a specification language to enable program verification. Because the design of the specification language influences the types of properties that an associated verification system can ensure, we next survey related work on specifications for software systems.

Specification methodologies typically cover the part of a project's lifecycle between the project's design phase and its implementation and delivery to customers. Some of these methodologies (for instance, Z) provide a general notation which developers may use to express program properties, but still expect developers to carry out

proofs by hand. Most of these methodologies include some tool support in the form of verification condition generators and proof assistants. However, unlike Hob, these methodologies do not leverage current static analysis technologies, such as shape analysis, to automatically verify program properties. In the absence of automatic support for verifying conformance to specifications, design drift—the phenomenon whereby design information becomes outdated and therefore fails to reflect the current capabilities of a software system—inevitably becomes a problem, especially considering that software maintenance typically continues long after the software has been initially delivered to customers.

Origins of specification languages. Parnas was one of the earliest advocates for module specifications in [80]. Many of the ideas proposed in this work have become commonly accepted, at least in principle. The basic proposal is that specifications should enable the developers of a module and the client of that module to communicate effectively. Specifications should hide implementation details but expose usage constraints and guarantees. Parnas acknowledges that specifications can easily be erroneous. Since specifications generally cannot be executed, Parnas suggests that developers should carry out manual symbolic testing of specifications: they should invent a number of predicates which ought to be consequences of their specifications and verify that these predicates do hold.

The original proposal for specifications [80] does not propose a specific specification language[1]. Hob's set-based specifications are especially appropriate for data structure consistency properties. However, many other specification notations exist, and we next discuss some of these notations. We start with notations that are intended to model systems in general, such as the Z notation, and continue with wide-spectrum specification languages and object models. We then explore specification languages that are more specifically targetted towards programs rather than systems, like the Larch and JML specification notations. Like the Larch and JML approaches, the Hob approach embeds specification information directly into the program source code. Hob goes beyond previous approaches: one of Hob's major contributions is in verifying that implementations actually conform to their specifications.

The Z specification language. The Z notation [94, 89] allows system designers and implementers to express properties of their systems. Z was primarily designed as a notation for writing specifications and for manually proving statements about these specifications; it is particularly convenient for writing short proofs about equivalences between Z specifications.

Z is based on first-order predicate logic and typed set theory. Z specifications are therefore undecidable in general; that is, no algorithm can check (in general) that Z specifications are logically consistent, and the developer cannot compute (in all cases) whether a given statement is implied by a system's specifications. A number of tools exist to typecheck, model check and animate (*i.e.* execute on small examples)

[1]Parnas states, in a footnote, that the specification language that he uses in his paper should not be considered in any way to be a model specification language, due to its shortcomings.

Z specifications. These tools can increase a developer's confidence that his system's specifications are meaningful, but cannot provide any guarantees to that effect. The power of the Z notation enables it to completely specify system properties, so that—in principle—any system could be specified completely, even down to the implementation level.

Z specifications have been used to design large industrial systems. One report is [46], which describes the experience of some practitioners at IBM in specifying the CICS transaction processing system. Even without any automatic verification of the specifications or the resulting implementations, they reported that the use of formal specifications led to implementations with fewer errors in general, and to earlier detection.

Because Z and Hob have different design goals—Z enables developers to state properties of systems while Hob enables developers to verify data structure consistency properties—Z and Hob differ in terms of specification language expressiveness. Hob requires developers to specify more specialized properties than Z; Hob's properties are either global or local data structure consistency properties. Global data structure consistency properties state relationships between sets (defined in terms of abstraction functions), while local consistency properties primarily ensure that set implementations maintain the proper invariants. Developers benefit from using Hob specifications because their implementations can be verified against properties specified for the Hob system; this is not true for properties specified in terms of Z designs.

We are aware of one proof assistant for Z specifications, ProofPower, which uses an implementation of higher-order logic (HOL) as its backend and embeds Z into HOL. However, to our knowledge, there are no analogues to the Hob system which can automatically prove that implementations—especially implementations with heap data structures—conform to their Z specifications.

Wide-spectrum specification languages. The wide-spectrum specification language approach attempts to help developers ensure that implementations match their specifications by providing a family of syntactically related languages to both specify and implement systems [49, 34, 22, 2]. Previous work on automatically proving that implementations conform to their specifications has been sparse, and we are not aware of any such research in the context of wide-spectrum specification languages.

Often, developers find deficiencies in specifications while implementing them. When correcting these deficiencies, developers must take care to explicitly update both the original specification and its implementation. In practice, implementations and specifications tend to end up diverging—or drifting—in the absence of tools that automatically verify that an implementation conforms to its specification. We call this phenomenon *design drift*.

The Hob approach does not use a wide-spectrum specification language; we instead provide separate specification and implementation languages, and automatically verify the conformance of an implementation to its specification using the provided abstraction functions. Hob therefore guarantees that a program's implementation

continues to conform to its design throughout its maintenance phase, preventing design drift.

Perhaps the most-used wide-spectrum specification approach is the Vienna Development Method [49]; its successor VDM++ [34] extends VDM with support for object-oriented analysis and design. VDM is quite expressive; it enables developers to write specifications for systems using numbers, sets, maps, sequences, and functions. In fact, it is so expressive that the type-checking problem for VDM is undecidable, because types may depend on conditional VDM expressions. VDM has been extensively used in industry; published examples include models of railway interlock systems, nuclear safety systems, and telephone exchanges [62]. The primary tool supporting the VDM is the VDM++ Toolbox [21], which includes some support for type checking, an interpreter for executable VDM specifications, an verification condition generator for VDM models which generates conditions that ensure that these models are free of run-time errors, a test facility, and an automatic code generator. Other wide-spectrum languages include RAISE [22], which adds support for modular reasoning and concurrency, and the B-method [2], which uses abstract machines to represent the actions of the system. These specification languages generally require manual proofs of refinements between different levels of specifications and implementations. The Hob approach, on the other hand, automatically verifies that implementations conform to their specifications. Hob's approach helps prevent design drift by informing developers immediately when implementations and specifications diverge; it is therefore possible to impose development processes that require developers to immediately correct either the implementation or the specification in case of divergence.

Larch. The Larch project [44] explored the expressive potential of specification languages. In the Larch approach, specifications had two parts: an auxiliary specification and a trait. Traits enable developers to state properties of the mathematical objects that appear in Larch specifications. Using these traits, developers would be able to use appropriate notations for the specification task at hand. Hob, on the other hand (like VDM and Z) takes a strong position on the types of specifications that users may write; we chose set specifications for Hob because we believe that sets are particularly apt for stating data structure consistency properties. Furthermore, Hob specifications are designed primarily to enable verification.

Hob does support extensibility in the following sense: it allows developers to provide user-definable abstraction functions which relate concrete states to abstract states as implemented in analysis plugins; if Hob's set of analysis plugins is insufficient, then developers may write their own analysis plugins. By fixing the specification language to the boolean algebra of sets, we simplify the task of analysis plugins; after all, plugins must consume and produce conditions expressed in the common specification language, and an overly-complicated specification language would impose an excessive burden on writers of analysis plugins. We believe that the choice of a set specification language is a reasonable compromise between expressiveness and tractability in this regard.

We next highlight the differences between Larch and Hob by briefly discussing

the specification of a bounded stack in Larch. Basically, the Larch specification does not abstract away the ordering of the elements in the stack, while Hob represents the contents of the stack as an (unordered) set. When using the Larch specification, the developer must refer to a stack state by writing a sequence of operations, e.g. `push(push(push(empty, S), 2), 3)`, consistent with a world-view based on algebraic specifications of abstract data types. The Hob approach instead allows developers to state set-based properties of the stack's contents. This enables developers to state, for instance, that the stack's contents are disjoint from some other data structure's contents.

One limitation of the algebraic specification methodology is that it is difficult to state global program properties using algebraic specifications. Because the Hob specification language supports global data structures and has set-based specifications, it can easily state global program properties, which will appear as relations between sets.

In general, the Larch system does not have many tools for reasoning about implementations, since it was designed to explore issues associated with specification languages. One implementation-oriented tool is LCLint [29], which performs a limited set of static checks for generic memory-safety properties, guided by some Larch program properties. LCLint verifies that programs never violate abstraction barriers; that they always specify and use all global variables; that modifies clauses are accurate (with some limitations and some unsoundness); that uses occur after definitions; and that macros are properly used.

Note in particular that LCLint does not perform any checks based on `requires` or `ensures` clauses. Contrast this to Hob—one of the key goals of the Hob system is to verify that each procedure's `ensures` clause always holds upon exit from that procedure, as long as the `requires` clause holds upon entry.

Java Modelling Language. The Java Modelling Language enables developers to specify properties for Java programs. JML applies many of the ideas from the Larch project to object-oriented Java programs. JML specifications are typically embedded as comments within Java programs. The specification language mostly contains Java expressions, plus a few extra keywords.

Unlike Larch, Hob is designed to allow designers and developers to express and verify design-level information about a bounded (at compile time) collection of named abstract sets of objects. Hob's specification language is the boolean algebra over sets and boolean variables. Unlike JML specifications, which support implementation-level constructs such as strings, integers, or floating-point values, Hob's set-based specification language is focussed on a particular set of properties that we believe is important and relevant to a system's design.

We find our approach productive in that it focusses the attention of the designers and developers on some important core aspects of the design and facilitates the effective verification of those aspects. In particular, the Hob approach discourages developers from writing specifications that simply reiterate the implementation using specification-level constructs, because we chose to omit the needed constructs from

the Hob specification language.

8.1.1 Expressing design information

The specification languages we have discussed so far are primarily targetted towards expressing general system properties. We next discuss notations that were targetted specifically for expressing design properties, including some that address the issue of design drift by extracting models directly from source code. We believe that such approaches are most likely to succeed in addressing the design drift problem.

Object Models. Object-oriented analysis and design rely on a suite of techniques for specifying (typically software-based) systems. The central technique is object modelling. An object model graphically describes the design of a system with boxes for the different classes in a software system and arrows for the relationships between these classes. The most popular methodology for object-oriented design is the Unified Modelling Language [83]. When following the UML methodology, a developer creates a set of design artifacts which describe the system being designed and implemented. These artifacts are intended to be descriptive, not prescriptive; nothing guarantees that a system conforms to its design. It is the sole responsibility of the developer to ensure that the artifacts produced at each stage remain consistent with each other, without even the notational help provided by wide-spectrum specification languages. The UML approach typically does not include formal verification.

There is rich tool support for object models; some tools, including TogetherJ and Rational Rose, help the developer ensure that the implementation is consistent with the object model by automatically generating a skeleton of the implementation from the object model. Nevertheless, such approaches are still affected by the design drift problem: unless the model is generated from the source, the source and model will diverge in the course of development. We have previously developed the token annotation system, which embeds object modelling metadata into Java source code and can later automatically extract this metadata to generate the model [61]. We believe that such an approach will facilitate the difficult task of keeping design information up-to-date.

Traditional object models do not include constraints on system behaviours; object models are instead intended for specifying the system architecture, *i.e.* the connections between different system components. The Object Constraint Language [79] and the Alloy system [3] provide two ways of specifying system behaviours on top of object models. Alloy also includes tools for visualizing and automatically checking consistency properties of these object models, using bounded model checking. These tools, however, do not verify that implementations of the object models conform to the original models.

Because the Hob approach attempts to verify data structure consistency properties, its focus is quite different from that of object modelling languages. The key features relevant to an object modelling language are the expressive power and the ease-of-use of that language. The Alloy language was also designed to facilitate internal consistency checks for specifications. The Hob specification language supports

more than just object models and internal consistency checks; we designed it in concert with the Hob implementation language and the abstraction languages to enable the static verification of data structure consistency properties.

Design conformance. Reflexion models [74] support the concept of design conformance: they enable developers to developers propose a model of a software system and then compare properties of the actual system to the model. The idea is to use an algorithm to extract a model from the source code; their tool then presents the difference between the proposed model and the extracted model to the developer. Reflexion models may, in principle, use many different kinds of models. In their paper, Murphy et al. propose the following model extraction algorithm: group a number of files together as a module (using wildcards) and use procedure calls to define the inter-module interaction structure. Other model extraction algorithms would also be possible. Like Hob, reflexion models attack the problem of design drift, by identifying differences between the intended design and the actual implementation. While such an approach is quite useful, Hob can encode many design properties that would be difficult to express in terms of graphical reflexion models. Because Hob expresses properties of the program state, it can state (for example) that two sets are always disjoint.

The Pattern-Lint tool [85] uses dynamic analysis and shallow static analysis techniques to verify whether or not software systems conform to desired architectural constraints. The novelty in Pattern-Lint appears to stem from how it decides whether or not systems conform to their designs: Pattern-Lint collects evidence for and against a design property to decide whether or not the implementation conforms to that design properties. Pattern-Lint only uses very simple static analysis techniques: it appears to only inspect method calls and shared global variable accesses.

8.2 Analysis Technologies and Verification Systems

The Hob system verifies set-based specifications by combining various static analysis and theorem proving technologies. We next discuss a number of related approaches to static program verification. A number of these approaches are static analysis approaches; these include typestate systems, shape analysis, and abstract interpretation. We also describe some research which uses model checking. The typical application of model checking verifies that module interfaces are used appropriately (but, unlike Hob, does not verify that the modules are properly implemented). Finally, we discuss theorem proving technology; in our context, theorem provers help construct proofs that indicate that implementations have desired properties.

In general, each of the research projects below presents a single approach to verifying a single class of program properties; there is no effort to integrate results from different analysis approaches. For instance, ESC/Java uses the Simplify theorem prover to discharge all of its verification conditions. We believe that by applying specialized tools—working together—to specialized classes of data structure consistency

properties, the Hob system enables the verification of more sophisticated properties on larger programs than previous research.

Typestate systems. Typestate systems track the conceptual states that each object goes through during its lifetime in the computation [91, 23, 32, 31, 28]. They generalize standard type systems in that the typestate of an object may change during the computation. Aliasing (or more generally, any kind of sharing) is the key problem for typestate systems—if the program uses one reference to change the typestate of an object, the typestate system must ensure that either the declared typestate of the other references is updated to reflect the new typestate or that the new typestate is compatible with the old declared typestate at the other references.

Most typestate systems avoid this problem altogether by eliminating the possibility of aliasing [91]. Generalizations support monotonic typestate changes (which ensure that the new typestate remains compatible with all existing aliases) [32] and enable the developer to temporarily prevent the program from using a set of potential aliases, change the typestate of an object with aliases only in that set, then restore the typestate and reenable the use of the aliases [30]. It is also possible to support object-oriented constructs such as inheritance [24]. Fink *et al.* propose the integration of pointer analysis techniques with typestate property verification for Java programs in [33]. Their technique scales due to the use of a series of abstractions: the simpler abstractions quickly rule out many potential problems and leave more sophisticated properties to more expensive analyses. The role system [54] also integrates pointer analysis techniques with typestate verification. In the role system, however, the declared typestate of each object characterizes all of the references to the object, which enables the typestate system to check that the new typestate is compatible with all remaining aliases after a nonmonotonic typestate change.

In our approach, the typestate of each object is determined by its membership in abstract sets as determined by the values of its encapsulated fields and its participation in encapsulated data structures. Our generalizations of typestate include multiple orthogonal typestates (corresponding to multiple sets), and, most importantly, the ability to verify actual properties associated with the typestate abstraction, as opposed to taking for granted the correctness of interface specifications.

Bierhoff and Aldrich describe a dynamic analysis system for verifying typestate properties in Java programs that correctly handles typestates in the context of subclassing [8]. Like Hob, [8] also supports multiple orthogonal typestates. While a dynamic analysis can prevent programs from executing undesirable actions, typically by terminating a program when it attempts to execute such actions, the advantage of our static approach is that it provides stronger guarantees that programs never violate typestate constraints on any possible execution before actually executing these programs.

Shape analysis. The goal of shape analysis is to verify that programs preserve consistency properties of (potentially-recursive) linked data structures. In [67], Luckham and Suzuki describe an early attempt to verify properties of linked data structures.

They explicitly incorporate reachability and acyclicity into the first-order Stanford Pascal Verifier logic. Their tool deduces that the appropriate shape constraints hold, making use of user guidance throughout the theorem proving process. However, they do not have any notion of shape abstractions as in modern shape analysis, so that the desired program properties are expressed as assertions in Pascal-like expressions augmented with the reachability predicate; it is therefore difficult to modularize their approach.

Since then, researchers have developed many shape analyses and the field remains one of the most active areas in program analysis today [41, 71, 54]. In general, shape analyses focus on extracting and verifying detailed consistency properties of individual data structures.

We explicitly mention TVLA, the Three-Valued Logic Analysis engine [84]. TVLA has some similarity to Hob in that it is not a single analysis, but rather a framework which allows researchers to specify specific abstractions of the heap. The TVLA tool embeds the operational semantics of a particular implementation language and produces abstract interpreters for this language using the specified abstraction. The Hob framework gives developers more flexibility to develop analysis plugins—Hob plugins do not necessarily have to use abstract interpretation, as shown by our theorem proving plugin. We also designed Hob so that it would be able to verify higher-level domain-specific properties, in addition to low-level properties of the concrete heap.

Because shape analyses are very precise, the detail of the properties these analyses must track have limited their scalability. One of our primary research goals is to enable the application of these sophisticated analyses in a modular fashion, with each analysis operating on only that part of the program relevant for the properties that it is designed to verify.

Model Checking Approaches. Model checking is a lightweight approach to program verification that attempts to detect violations of certain specification properties in systems by setting up an abstract model of the program and exhaustively testing the program in that abstract model. Bultan *et al.* describe one application of model checking to modular verification in [7]. Their research focusses on detecting synchronization errors in concurrent programs: they find instances where programs improperly order synchronization operations. Such a model-checking approach can effectively use the information provided in terms of module interfaces, as long as the interfaces have only finite amounts of state (which the model checker can exhaustively explore); this is similar to verifying programs with typestate-like specifications.

Note that, like ESC/Java, Bultan's model-checking approach only verifies the use of the interfaces, and not the underlying implementations of these interfaces. The model checking approach is not, in isolation, well-suited to verifying Hob-style properties of unbounded data structures in the concrete heap, because it is difficult to exhaustively explore an unbounded data structure. Techniques such as symbolic model checking could help, but have not yet been applied to heap data structures.

Abstract interpretation. The ASTREE static analyzer [9] has successfully verified millions of lines of automatically generated C code for the absence of run-time errors. Like Hob, ASTREE combines a number of different static analyses to statically verify program properties; ASTREE uses abstract interpretation over a number of specialized abstract domains. However, the goals of ASTREE differ substantially from our goals. We emphasize two points in particular. First, ASTREE's input language is a subset of C which does not include dynamic memory allocation; we specifically designed Hob to support the inclusion of shape analyses, which reason about the relationships between different dynamically allocated objects. Second, ASTREE verifies that programs never encounter run-time errors such as out-of-bounds array accesses and arithmetic overflows; the set of properties of interest is built into the ASTREE analyzer itself. The Hob system, on the other hand, verifies developer-provided data structure consistency properties. These properties enable the developer to express domain-specific program properties which capture the program's design information.

Stanford Pascal Verifier. The Stanford Pascal Verifier [40, 66] was an early program verification effort. It was surprisingly powerful for its time. Like Hob, the Stanford Verifier attempted to prove that a procedure's postconditions held upon exit if its preconditions held upon entry. However, a key difference between Hob and the Stanford Verifier is that the Stanford Verifier exclusively uses theorem proving to establish program properties, compared to Hob's notion of analysis plugins. In a retrospective evaluation of the Stanford Verifier [68], Luckham writes (in 1981) that "... theorem proving still represents a major bottleneck in verification systems." We believe that, even though theorem proving technology has improved since 1981, the use of static analysis techniques—as in the Hob system—greatly facilitates the verification of many important program properties.

Another key difference between the Stanford Verifier and Hob is in the expected scope of the preconditions and postconditions. The Stanford Verifier ambitiously attempted to prove partial correctness for procedures, rather than our more limited data structure consistency properties. Unfortunately, proving correctness for realistic programs was beyond the capability of both the computer hardware and the theorem proving technology of the time. Note that, due to its design goals, the Stanford Verifier accepts preconditions and postconditions directly stated its underlying logic (which Luckham describes as being "cumbersome"). Unlike Hob, it does not use a set specification language or abstraction functions. Therefore, even though the Stanford Verifier can reason about heap reachability using custom `reach` primitives added on top of its first-order logic [67], heap data structures are quite difficult to reason about in practice, due to the lack of abstraction functions. It is much easier to express properties of sets using their names.

ESC/Java and ESC/Java2. ESC/Java [36] (and its successor ESC/Java2) are program checking tools which aim to identify common errors in programs with the help of program specifications expressed in a subset of the Java Modelling Language [12]. ESC/Java and ESC/Java2 currently use the Simplify theorem prover to verify pro-

gram properties; their design is reminiscent of the Stanford Pascal Verifier's design, updated to use more modern specification languages and theorem provers. The designers of ESC/Java have explicitly stated that, like ASTREE, it was designed to statically identify potential run-time errors, *e.g.* null-pointer exceptions. ESC/Java additionally attempts to establish, at least partially, that preconditions hold at call sites. The Hob system was principally designed to verify program-specific properties, which include preconditions and postconditions, but also global data structure consistency properties. Hob's support for abstraction functions and scopes make data structure consistency properties much easier to express.

The ESC/Java2 tool [26, 18] extends the original ESC/Java work by supporting current versions of Java and verifying more JML constructs. In particular, ESC/Java2 (as well as ESC/Modula-3 [26]) allows the use of heap abstractions via its support for model fields. Model fields use developer-provided representations. These representations are similar in spirit to the set definitions which appear in Hob's abstraction modules. However, not all model fields are annotated with representations; for instance, the library annotations provided with ESC/Java2 for the `LinkedList` class do not discuss the actual concrete contents of the `LinkedList` as a set of objects in the heap. The first-order logic used by the underlying Simplify theorem prover [25] does not support transitive closure; effective first-order approximations of transitive closure are still active areas of research [76, 65, 56]. ESC/Java2 has therefore not been used to verify the concrete data structure consistency properties that Hob verifies for linked lists, essentially because its logic is not powerful enough. Cok explains how ESC/Java2 handles model fields in [17]; essentially, it treats them as method calls and includes the postconditions of the model fields' representations.

On the other hand, the Hob system enables the developer to use—and verify—implementations which use arbitrary set definitions, as long as an appropriate analysis plugin exists. This enables, for instance, the shape analysis and Isabelle plugins to use logics which go far beyond the expressiveness of the set specification language. The logic used for inter-analysis communication is still the first-order set-based specification language, and we require that each a analysis plugin be capable of reasoning about the first-order set specification language.

Spec$^\#$. The Spec$^\#$ programming system [5] adds ESC/Java2-like features to C$^\#$, including the ability to specify method contracts, frame conditions and class contracts. These contracts may be verified either at run-time or statically. Static verification relies on the Boogie 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; as we've seen, this range includes shape analyses, typestate analyses and interactive theorem provers. Hob's goal of supporting specialized analyses 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 to Hob. 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 objects that that object owns. This hierarchical specification approach is reminiscent of hierarchical access specifications in Jade [82] and hierarchical locking mechanisms in databases [87].

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.

Other theorem provers. We use the Isabelle/HOL interactive theorem prover [81, 78] to discharge the verification conditions generated by our theorem proving analysis plugin. Other interactive theorem provers include Athena [4], which separates computations from deductions in the context of proof presentations and searches; HOL-Light [45], which has an especially small set of base axioms; and CVC Lite [6], which is quite adept at automatically proving theorems about programs with arrays due to its support for integer arithmetic. Users of the ACL2 [50] theorem-proving system have applied theorem-proving techniques as well as term-rewriting techniques to verify properties of large-scale systems, among them software systems [72]. With some engineering work, any of these theorem provers ought to be embeddable into the Hob system as an analysis plugin.

Typical applications of static analysis. Many systems that use static analysis to improve software quality, such as FindBugs [47], search for violations of generic properties that all programs written in a particular programming language must satisfy. Other systems, such as Synergy [43], verify usage properties for system calls (such as locking primitives). While such properties are somewhat domain-dependent, in that they only apply to programs that belong to a certain domain (*e.g.* device drivers), these properties still do not discuss anything specific about the programs themselves. The specification languages that we have discussed could enable the verification of design-level properties. However, to our knowledge, other specification

languages have not been used for verifying such properties. Note that the level of detail in most other specification languages makes it difficult to identify which properties are design properties in a potentially unwieldy specification. We believe the Hob approach is the first approach that gives developers the power to both state and verify truly domain-specific properties which can express aspects of a software system's design.

8.3 Combining Static Analyses

Our research aims to enable the application of multiple analyses that check arbitrarily complicated properties within a single program. This contrasts with most existing approaches, which attempt to develop a single new analysis algorithm or technique. Our system supports the loose integration of analyses where each analysis applies to one procedure or module. The set specification language is key to this integration, as it serves as a *lingua franca* between analysis plugins. Hob's design decisions were taken, in part, to facilitate the incorporation of external tools.

Most other verification systems combine analyses by using a single analysis engine (usually a theorem prover) and combine the decision procedures for different properties using Nelson-Oppen techniques [75] and their generalizations (*e.g.* [96, 97]). Theorem provers based on these principles include Simplify [25], Verifun [35], and CVC [92]. In [13], Chang and Leino explore an approach that proposes a tighter combination of a particular domain (uninterpreted function symbols) with an arbitrary base domain. Their approach would enable the application of static analysis techniques which could reason about the program state using a number of different abstract domains.

Briefly, our approach works well for combining analyses at granularities above the procedure level, while the Nelson-Oppen approach is targeted towards combining analyses below the procedure level. Note also that the two techniques are not mutually exclusive: the Nelson-Oppen technique of combining abstract domains could be incorporated into a Hob analysis plugin. An important design goal of our Hob system was to enable developers to communicate data structure consistency properties to the backend static analysis engines for verification.

Compositional Reasoning Our research has only considered safety properties in the context of sequential programs. In [1], Lamport and Abadi describe a general Composition Principle which examines the circumstances under which it is safe to compose specifications. In the case of Hob-style data structure consistency properties for sequential programs, the Composition Principle basically states that when sequentially composing two procedures, the composition of the procedures requires the precondition of the first procedure and ensures the postcondition of the second procedure, as long as the postcondition of the first procedure implies the precondition of the second procedure. We expect to extend the Hob system to support more general types of software systems in the future, including concurrent and reactive systems, and we expect to use the full Composition Principle for such systems.

Chapter 9

Conclusion

This dissertation has been motivated by the problem of verifying that implementations satisfy stated design properties. In this dissertation, I have presented the Hob system, which can verify that implementations conform to design properties expressed in the form of global data structure consistency properties. Developers may use Hob's set specification language (which contains the boolean algebra of sets) to state design properties and a standard imperative language for implementations. Hob's set specification language also contains the scopes and defaults mechanisms, which enable developers to omit redundant clauses from specifications and therefore to write shorter specifications.

Because implementations and specifications act on different representations of the program state—concrete heap states for implementations versus abstract sets for specifications—the Hob system uses developer-provided abstraction functions to relate implementation and specification states. Abstraction functions mediate between concrete states and abstract states by giving definitions for abstract sets in terms of concrete states. Static analysis techniques use these abstraction functions to verify that implementations correspond to their set-based specification; many precise analyses exist and are capable of verifying some quite sophisticated classes of implementations.

Issues associated with using precise static analyses include scalability limitations and the diversity of important data structure properties, some of which will inevitably elude any single analysis. A key element of the Hob approach is in its use of modular analysis to address these issues: developers may divide the program into modules and verify each procedure belonging to these modules separately, choosing an appropriate analysis technique for each module. To enable modular analysis, Hob modules encapsulate fields (not objects) and data structure implementations; the analysis relies on specifications based on membership in abstract sets; and developers may use sets to express (and enable the verification of) properties that involve multiple data structures in multiple modules analyzed by different analyses. The techniques described in this dissertation will enable the productive application of a variety of precise analyses to verify important software design properties.

This dissertation has described how the Hob framework integrates the flags, Bohne and theorem proving analysis plugins. The flags plugin enables developers to reason

about modules that manipulate sets defined by integer and boolean flag values, as well as modules that coordinate the actions of client modules. The Bohne plugin supports reasoning about linked heap structures by summarizing them in terms of monadic second-order logic. Finally, the theorem proving plugin allows developers to state arbitrarily complicated program properties and to verify them (by directing theorem provers towards proofs of these properties). The Hob system enables developers to combine implementation modules which are analyzed using the flags, Bohne and theorem proving modules. Furthermore, the use of a common set specification language enables developers to verify global consistency properties which depend on results obtained by any plugin in the Hob framework.

Finally, this dissertation has evaluated the feasibility of the Hob approach by applying it to a number of benchmarks, including minesweeper, a web server, and a MIDI file player. I found that the Hob approach was suitable for capturing certain kinds of design information. In particular, the use of set specifications enabled developers to state and verify outward-looking properties. These properties could constrain a program's behaviour and guarantee that it does not misbehave in certain user-visible ways.

9.1 Future Work

I next outline several possible research directions.

Verified data structure library. The Hob system enables the development of a standard library of verified data structures. To date, we have implemented and verified a number of useful data structures, including linked-list and array-based set implementations. I believe that it should be possible to implement all of the data structures in common usage and to verify data structure consistency properties for these data structures. Because Hob's set specification language only supports a targetted class of specification properties, it would not be possible to specify all properties of interest. However, set specifications should be expressive enough to state many useful properties.

A library of verified data structures would be useful in itself, as a toolbox for developers to use, but would also contribute to our understanding of an analysis system's power. Such a library can shed light on which properties a system can verify as well as which properties the system can communicate to the developer.

Specification and implementation inference. Developers using the Hob system must currently state both specifications and implementations; in some sense, the specifications and implementations redundantly state some of the same information. While this redundancy can help to identify errors in both specifications and implementations, the overall research goal of ensuring that implementations conform to their designs does not depend on this redundancy. Note that a specification inference approach would be no worse than the current state of the art in terms of finding program errors, since specifications are currently either nonexistent or—at

best—unchecked. Even though inferred specifications would not immediately find any errors, they would be helpful for identifying design drift: developers could be notified when their implementations change and no longer match previously-inferred specifications.

In principle, it would be possible for analysis plugins to synthesize specifications—or at least initial drafts of specifications—from some classes of implementations, to be polished by the developer later on. The flag analysis plugin would be particularly suitable for inferring specification.

Conversely, it should also be possible to synthesize implementations from specifications, at least for a certain (limited) class of implementations. To synthesize implementations, it would be necessary to deduce the sequence of method calls required to implement a given set of postconditions, a search problem. Such a synthesizer would enable a style of programming which could turn out to be similar to the SETL programming language [27].

Novel specification mechanisms. A related area for future investigation is improved specification languages. Our specification language notions of scopes and defaults arose from an examination of how existing specification language mechanisms could be improved. A general problem with specifications is that developers sometimes write nonsensical specifications which make program properties either vacuously true or unsatisfiable. I believe that the exploration of further specification constructs can yield techniques which would make it easier to create meaningful program specifications.

Handling concurrent programs. Concurrency has become a ubiquitous feature of modern programming environments. It would be productive to explore the issues involved in combining concurrency and the global data structures supported by the Hob system. A major obstacle to developing reliable concurrent programs is the possibility that multiple threads may update the same data structure at the same time. Modular analysis of concurrent programs is particularly difficult because a key assumption in our current modular analysis technique is violated: side effects from concurrent threads may occur at any time in a procedure's execution. In the concurrent setting, it is no longer sufficient to assume the precondition of a procedure and prove the postcondition. One possible solution is to augment the Hob system to support programs in which the developer may choose to implement data structures using either private per-thread instances (which are not subject to changes by concurrent threads) or using atomic transactions to control updates to data which is shared between threads. The current Hob approach ought to be easily adaptable to private data structures. However, novel specification and verification techniques would be needed for transaction-based shared data structures. For example, the current Hob scope invariant mechanism relies on the fact that only one thread is accessing the modules that are working together to maintain the invariant. At the very least, there must be some specification mechanism allowing developers to identify the parts of the state that must be accessed under a transaction context.

This research would enable developers to verify higher-level data structure consistency properties for concurrent systems, including relationships between local and shared data structures. These properties could go beyond the current state of the art for the analysis of concurrent systems, which verify basic properties such as freedom from data races and typestate properties.

Integrating dynamic analysis results. Approaches based on dynamic analysis and testing have proven to be effective for both discovering and enforcing program properties. For instance, Nimmer and Ernst have investigated the dynamic detection of program invariants, as implemented in the Daikon tool, and the static verification of these invariants, using the ESC/Java system; this work is described in [77]. While the Hob system currently uses static analysis approaches exclusively, future work could productively investigate the possible contributions of dynamic analysis in the context of the Hob approach. One approach, which is being currently explored by Zee [98], uses information from program executions to guide loop invariant inference. I believe that because Hob's set specifications focus on high-level relationships between different parts of the program rather than low-level concrete heap properties, the inference of Hob-style set specifications could enable the automatic inference of design information.

9.2 Implications

Until now, developers have been unable to rely on design information to guide them in the course of software development and maintenance: such information is often outdated and inaccurate. Because the Hob system can automatically verify whether or not an implementation matches its design, it enables developers to ensure that a program's design information remains valid throughout the program's lifecycle. In particular, automatic verification enables developers, or their managers, to include program verification in development processes in the same way that unit testing has been incorporated into current development processes.

It has historically been difficult to enforce the use of specifications in software projects. Besides Hob, a number of other recent proposals have also proposed the use of specification information in conjunction with static or dynamic program analysis for verifying this specification information; examples include Bierhoff and Aldrich's dynamic typestate [8] and the Spec$^{\#}$ programming system [5]. I believe that contemporary approaches to the use of specifications will enjoy greater success than past exhortations advocating the importance of specifications. These past exhortations may have been ignored because past approaches failed deliver any concrete benefits from the use of specifications. Because the Hob approach enables developers to verify implementations against specifications, specifications can remain up-to-date, and developers can confidently use these specifications when maintaining software. Furthermore, successful verification (of both low-level data structure consistency properties and global properties that relate different data structures) will augment developers' confidence in the overall quality of their software.

The heap aliasing problem has always been an issue for static analyses; all static analyses must somehow finitize the heap to soundly handle the aliasing problem. Hob's set-based specifications enable developers to state constraints on the heap, including aliasing constraints, and to finitize the heap using a fixed number of sets. Note that Hob's set specification language obviates the need to discuss pointer-based relationships between heap objects at the level of global specifications, hiding the complexities inherent in the underlying concrete state and replacing them with relationships between abstract sets. My experience with set-based specifications suggests that they are a useful abstraction of the heap and that their use ought to ease the development of future static analysis techniques.

Hob's set-based specifications also point out a new approach for researchers to use in making static analysis results accessible to developers. Even though modern static analyses are capable of verifying extremely detailed program properties, these analyses are not useful unless developers can provide program properties to the analyses in the required form. When using the Hob system, a program analysis expert can provide necessary abstraction functions during a module's initial design. Subsequently, developers can apply static analyses to a module's implementation without the assistance of the expert, even as the module evolves, as long as the fundamental representation invariants remain unchanged. Of course, the Hob approach also enables developers to benefit from static analysis in the sense that Hob enables developers to invoke verified procedures, knowing that these procedures satisfy their contracts.

Formal verification of large software systems has long been a goal of the program verification community. I believe that an approach like Hob's modular pluggable analysis approach is most likely to succeed in verifying such systems. There has been a constant tension between analysis power and scalability: more-powerful analyses have historically scaled poorly, yet they are needed to verify important properties of programs. A successful solution to the program verification problem must be able to harness powerful analysis techniques (to obtain needed analysis results) but may only apply them to limited parts of the program (for scalability reasons). Modular analysis enables the productive use of different analyses, of varying power, to potentially verify properties of significantly-sized programs.

Bibliography

[1] Martín Abadi and Leslie Lamport. Composing specifications. *Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

[2] Jean-Raymond Abrial, Matthew K. O. Lee, Dave Neilson, P. N. Scharbach, and Ib Sørensen. The B-method. In *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume 2*, pages 398–405. Springer-Verlag, 1991.

[3] *The Alloy Modelling Language and Analyzer*. Papers and tool available at: `http://alloy.mit.edu`. Software Design Group, Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA.

[4] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin Rinard. Verifying a file system implementation. In *Sixth International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, Seattle, Nov 8-12, 2004 2004.

[5] Mike Barnett, K. Rustan M. Leino, and Wolfram 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.

[6] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the* 16^{th} *International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, July 2004. Boston, Massachusetts.

[7] Aysu Betin-Can, Tevfik Bultan, Mikael Lindvall, Benjamin Lux, and Stefan Topp. Application of design for verification with concurrency controllers to air traffic control software. In *Proceedings of the 20th IEEE International Conference on Automated Software Engineering (ASE 2005)*, pages 14–23, Long Beach, California, November 2005.

[8] Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. In Harald C. Gall, editor, *Proceedings of ESEC-FSE '05*, pages 217–226, September 2005.

[9] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A Static Analyzer for Large Safety-Critical Software. In *ACM PLDI*, San Diego, California, June 2003. ACM.

[10] 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, November 1992.

[11] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *Proc. 30th ACM POPL*, 2003.

[12] 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.

[13] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI'05*, January 2005.

[14] 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.

[15] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proc. 13th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1998.

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

[17] David R. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, September–October 2005.

[18] David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In *CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004.

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

[20] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proc. 6th ACM POPL*, pages 269–282, 1979.

[21] CSK, editor. *VDM++ Toolbox User Manual*. VDMTools, 2005.

[22] Bent Dandanell. Rigorous development using RAISE. In *Proceedings of the conference on Software for citical systems*, pages 29–43. ACM Press, 1991.

[23] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM PLDI*, 2001.

[24] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *Proc. 18th ECOOP*, June 2004.

[25] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories Palo Alto, 2003.

[26] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, COMPAQ Systems Research Center, 1998.

[27] Robert K. Dewar. The SETL programming language. http://birch.eecs.lehigh.edu/∼bacon/setlprog.ps.gz.

[28] 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.

[29] David Evans. Static detection of dynamic memory errors. In *Proc. ACM PLDI*, 1996.

[30] Manuel Fahndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proc. ACM PLDI*, 2002.

[31] 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.

[32] Manuel Fähndrich and K. Rustan M. Leino. Heap monotonic typestates. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2003.

[33] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanual Geay. Effective typestate verification in the presence of aliasing. In *Proc. International Symposium on Software Testing and Analysis*, 2006.

[34] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[35] Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In *CAV*, pages 355–367, 2003.

[36] 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.

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

[38] Pascal Fradet and Daniel Le Métayer. Shape types. In *Proc. 24th ACM POPL*, 1997.

[39] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1994.

[40] Steve M. German. Automating proofs of the absence of common runtime errors. In *POPL '78: Proceedings of the 5th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 105–118, 1978.

[41] Rakesh Ghiya and Laurie Hendren. Is it a tree, a DAG, or a cyclic graph? In *Proc. 23rd ACM POPL*, 1996.

[42] M. J. C. Gordon and T. F. Melham. *Introduction to HOL, a theorem proving environment for higher-order logic*. Cambridge University Press, Cambridge, England, 1993.

[43] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. Synergy: A new algorithm for property checking. In *Proceedings of the 14th Annual Symposium on Foundations of Software Engineering*, Portland, Oregon, November 2006.

[44] John Guttag and James Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[45] John Harrison. HOL Light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *LNCS*, pages 265–269. Springer-Verlag, 1996.

[46] Iain Houston and Steve King. CICS project report: Experiences and results from the use of Z in IBM. In S. Prehn and W. J. Toetenel, editors, *Proceedings of VDM'91*, volume 551 of *Lecture Notes in Computer Science*, pages 588–596. Springer Verlag, 1991.

[47] David Hovemeyer and Willian Pugh. Finding bugs is easy. In *ACM SIGPLAN Notices*, volume 39, pages 92–106, December 2004.

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

[49] Cliff B. Jones. *Systematic Software Development using VDM.* Prentice Hall International (UK) Ltd., 1986.

[50] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Publishers, 2000.

[51] 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.

[52] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. In *Proc. 5th International Conference on Implementation and Application of Automata.* LNCS, 2000.

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

[54] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Proc. 29th POPL*, 2002.

[55] Viktor Kuncak, Hai Huu Nguyen, and Martin Rinard. An algorithm for deciding BAPA: Boolean Algebra with Presburger Arithmetic. In *20th International Conference on Automated Deduction, CADE-20*, Tallinn, Estonia, July 2005.

[56] Shuvendu K. Lahiri and Shaz Qadeer. Verifying properties of well-founded linked lists. In *POPL06*, 2006.

[57] Patrick Lam. A general framework for the flow analysis of concurrent programs. Master's thesis, McGill University, 2000.

[58] Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking using set interfaces and pluggable analyses. *SIGPLAN Notices*, 39:46–55, March 2004.

[59] Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking for data structure consistency. In *6th International Conference on Verification, Model Checking and Abstract Interpretation*, 2005.

[60] Patrick Lam, Viktor Kuncak, and Martin Rinard. Hob: A tool for verifying data structure consistency. In *14th International Conference on Compiler Construction (tool demo)*, April 2005.

[61] Patrick Lam and Martin Rinard. A type system and analysis for the automatic extraction and enforcement of design information. In *Proc. 17th ECOOP*, 2003.

[62] Peter Gorm Larsen and John Fitzgerald. VDM information: Examples repository, November 2000.

[63] K. Rustan M. Leino. Efficient weakest preconditions. KRML114a, 2003.

[64] Daniel Leivant. Higher order logic. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 2: Deduction Methodologies*, pages 229–321. Clarendon Press, Oxford, 1994.

[65] T. Lev-Ami, N. Immerman, T. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *CADE-20*, 2005.

[66] David C. Luckham, Steven M. German, Friedrich W. von Henke, Richard A. Karp, P. W. Milne, Derek C. Oppen, Wolfgang Polak, and William L. Scherlis. Stanford Pascal Verifier user manual. Technical Report CS-TR-79-731, Stanford University, 1979.

[67] David C. Luckham and Norihisa Suzuki. Verification of array, record and pointer operations in Pascal. *Transactions on Programming Languages and Systems*, 1(2):226–244, October 1979.

[68] David C. Luckham and F.W. von Henke. Program verification at Stanford. *Software Engineering Notes*, 6(3):25–27, July 1981.

[69] Roman Manevich, G. Ramalingam, John Field, Deepak Goyal, and Mooly Sagiv. Compactly representing first-order structures for static analysis. In *Proc. 9th International Static Analysis Symposium*, pages 196–212, 2002.

[70] Roman Manevich, Mooly Sagiv, G. Ramalingam, and John Field. Partially disjunctive heap abstraction. In Roberto Giacobazzi, editor, *Proceedings of the 11th International Symposium, SAS 2004*, volume 3148 of *Lecture Notes in Computer Science*, pages 265–279. Springer, August 2004.

[71] Anders Møller and Michael I. Schwartzbach. The Pointer Assertion Logic Engine. In *Programming Language Design and Implementation*, 2001.

[72] J Strother Moore. Proving theorems about Java and the JVM with ACL2. In *Models, Algebras and Logic of Engineering Software*, pages 227–290. IOS Press, 2003.

[73] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[74] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In Gail E. Kaiser, editor, *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 18–28, 1995.

[75] Greg Nelson. Techniques for program verification. Technical report, XEROX Palo Alto Research Center, 1981.

[76] Greg Nelson. Verifying reachability invariants of linked structures. In *POPL*, 1983.

[77] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification*, Paris, France, July 23, 2001.

[78] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[79] Object Management Group (OMG). OCL 2.0 specification, 2005.

[80] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[81] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in LNCS. Springer-Verlag, 1994.

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

[83] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, Reading, Mass., 1999.

[84] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.

[85] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring compliance of a software system with its high-level design models. In *ICSE'96*, pages 387–396, 1996.

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

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

[88] 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.

[89] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., 1992.

[90] Robert E. Strom and Daniel M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Transactions on Software Engineering*, May 1993.

[91] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, January 1986.

[92] A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verification*, 2002.

[93] Thomas Wies, Viktor Kuncak, Patrick Lam, Andreas Podelski, and Martin Rinard. Field constraint analysis. In *Proc. 7th International Conference on Verification, Model Checking and Abstract Interpretation*, 2006.

[94] Jim Woodcock and Jim Davies. *Using Z*. Prentice-Hall, Inc., 1996.

[95] Eran Yahav and Ganesan Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *Proc. ACM PLDI*, 2004.

[96] Calogero G. Zarba. *The Combination Problem in Automated Reasoning*. PhD thesis, Stanford University, 2004.

[97] Calogero G. Zarba. A quantifier elimination algorithm for a fragment of set theory involving the cardinality operator. In *18th International Workshop on Unification*, 2004.

[98] Karen Zee. Personal communication, 2006.

[99] Karen Zee, Patrick Lam, Viktor Kuncak, and Martin Rinard. Combining theorem proving with static analysis for data structure consistency. In *International Workshop on Software Verification and Validation (SVV 2004)*, Seattle, November 2004.