Static Verification of Design Constraints and Software Correctness Properties in the Hob System

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

Abstract

Sets of objects are an intuitive foundation for many object-oriented design formalisms, serving as a key concept for describing elements of the design and promoting communication between members of the development team. It may be natural for the sets of the objects in the design to correspond to the sets of objects in the implementation. In practice, however, the object structure of the implementation is much more complex than that of the design. Moreover, the lack of an enforced connection between the implementation and the design enables the implementation to diverge from the design, rendering the design unreliable as a source of information about the implementation.

Hob allows developers to express and verify the connection between abstract sets of design objects and concrete sets of implementation objects. Abstraction maps define the meaning of the design sets in terms of the objects in the implementation, enabling the elimination of implementation complexity not relevant to the design. An abstract set specification language enables the developer to state important relationships (such as inclusion and disjointness) between abstract sets of objects; our verification system statically checks that the implementation correctly preserves these design-level correctness properties. We have implemented Hob and used it to develop several software systems. Our experience shows that Hob enables the effective expression and verification of precise design constraints that manifest themselves as important correctness properties that the implemented system is guaranteed to preserve.

1 Introduction

Sets of objects are a primary concept in many object modeling formalisms — using sets of objects as a conceptual tool enables developers to articulate and explore key aspects of the system's design. When these aspects are made formal (i.e., expressed in a suitable formal specification language), the result is a set of key correctness properties that the resulting software system must preserve.

One of the goals of the Hob project is to develop a new specification language and associated verification system that can establish a guaranteed connection between the design and the implementation, with sets of objects in the design corresponding to sets of selected objects within the implementation and key design properties formalized as key system correctness properties. Given these correctness properties, the Hob system then analyzes the system before it executes to determine if it may ever violate one of the correctness properties. Hob is sound: if its analysis reports no violations, then the system is guaranteed never to violate the correctness properties.

The Hob system currently contains multiple analyses and deploys these analyses in a coordinated way to verify targeted design-level properties. We have used Hob to design, implement, and verify several applications. As part of our experience, we have found that the Hob approach provides several significant benefits:

- **Effective Naming:** One of the key reasons that object models can be so effective as a communication medium is that they provide a single set of names that developers can use to identify different kinds of objects independent of the particular context at hand. Our specification language preserves this naming approach — each system has a collection of abstract sets, each with its own name. Developers therefore have a single unified set of names and concepts that supports quick and easy communication across different contexts. In particular, this approach enables developers working on different parts of the system to communicate without first establishing a translation between names and concepts from different contexts. It also supports the effective expression of high-level design constraints that cut across different parts of the system.
- **Appropriate Abstraction:** To support the expression of a formal connection between the implementation and the design, Hob supports *abstraction maps*, which allow the developer to specify the sets of objects in the implementation that make up each abstract set in the design. Because abstraction maps can discard objects that are required for the implementation but are irrelevant and distracting for high-level design purposes, they can eliminate the clutter and excess implementation detail that would otherwise obscure key elements of the design.
- **Design Conformance:** The fact that our analysis sys-

tem verifies correctness of the specifications ensures the conformance of the design to the implementation. Developers can therefore rely on the specification to provide accurate information about the design. The design can therefore become a source of useful information about the program (especially its high-level structure) throughout its entire lifetime. One especially important advantage is that the presence of an accurate design makes poor design decisions much more obvious and therefore much less likely to occur as the system evolves in response to changing goals and requirements.

- **Documentation:** One of the benefits of design conformance is that the abstract sets and the specification language precisely document the state, invariants, and preconditions of the software system. The elimination of low-level object clutter provides an appropriate level of abstraction for documentation; the guaranteed correspondence with the implementation eliminates the possibility that the documentation may be out of date or simply wrong.

2 Implementation Language

Hob's implementation language is a simple imperative language with modules, procedures, object references, and dynamic object allocation. Each implementation module may contain format declarations, module variables (which correspond to global variables in standard languages), and procedures. Each format declaration describes the fields that the module contributes to objects of the specified type [3,10]. Formats therefore provide a form of distributed field declarations—instead of centralizing field declarations in a single type declaration, the declarations of object fields are distributed across the modules that access objects of that type. Modules therefore encapsulate data structures *and not objects*. A program might have an object that simultaneously participates in a list module and a tree module, with the fields that implement the list encapsulated in the list module and the fields that implement the tree encapsulated in the tree module.

Each module variable contains a reference to an object; references serve as roots of data structures. Each procedure contains a sequence of imperative statements that manipulate references and objects. The type checker ensures that any well-typed program accesses only fields that exist and are visible to the executing module. In particular, it checks that the types of the actual and formal parameters match at call sites and that each field access refers to a field declared in a format declaration from the enclosing module.

Our language is designed to support programs with an unbounded number of objects that participate in a bounded (at compile time) number of data structures, with each data structure encapsulated within a module. This structure harmonizes with the abstract set approach—in most cases, each data structure will be rooted at a given module variable and will implement one or more abstract sets.

3 Specification Language

Hob's specification language uses abstract sets to represent program state [10]. Abstract set declarations identify each module's abstract sets. Procedure specifications use these sets to identify the effects of each procedure in the module. The `requires` and `ensures` clauses in the procedure specifications use arbitrary boolean clauses over abstract sets to specify these effects. The `modifies` clause bounds the collection of sets directly modified by a procedure.

The expressive power of boolean clauses is the first-order theory of boolean algebras, which is decidable, implying that there exist algorithms that analyze each statement and computes its effect on the boolean algebra formulas, automatically synthesize loop invariants, and check implication when verifying ensures clauses. Our flags analysis uses such an algorithm [13]; we expect other analyses to exploit this decidability property in similar ways.

4 Scopes

Developers use boolean clauses to specify the properties that the abstract sets must satisfy during the execution. These boolean clauses often involve sets from different modules, thereby capturing high-level design properties that cut across the module structure. Conceptually, such clauses are invariants that hold everywhere during the execution of the program. In practice, however, such clauses may be temporarily violated as the program updates its data structures (with object membership in the corresponding abstract sets reflecting these changes). Scopes [10, 11] make it possible to identify the regions of the program in which the clause may be temporarily and legitimately violated. Each clause is therefore embedded in a *scope declaration* that identifies: 1) a boolean clause, 2) the modules within which the clause may be legitimately violated, and 3) the publicly-available modules that may be invoked from outside the scope.

The analysis engine ensures that the clause holds everywhere outside the modules in the scope by first assuming that the clause holds on entry to each publicly-available module, then verifying that the clause always holds upon exit from that module. The analysis system can then assume that the clause holds everywhere outside the module.

The Hob analysis engine uses an interprocedural link-time analysis to check that the developer has marked all reentrant calls. A reentrant call is a call from within the scope—where the clause is potentially violated—that invokes a publicly-available module that assumes that the clause holds. In cases where a module within the scope must invoke a module outside the scope that assumes the clause, the "reentrant" label explicitly instructs the analysis to verify that the clause holds before the invocation point.

5 Abstraction Maps

Abstraction maps provide the connection between the concrete objects in the implementation and the abstract sets of objects in the design. A central feature of the Hob framework is the ability to deploy multiple analyses, each specialized to analyze a specific kind of module. Because different analyses use different techniques, it becomes possible to select the most appropriate analysis for each module, depending on the expressiveness and the scalability required to analyse that module.

To evaluate the feasibility of using different analyses for different modules, we have developed and used a number of different analyses. The flags analysis [13] assigns set membership according to the values of integer fields. The Bohne analysis [19] performs field constraint analysis, a generalization of shape analysis [9], and assigns set membership according to heap reachability properties. Finally, the theorem proving analysis [20] assigns set membership using Isabelle predicates; it is useful for reasoning about complex data structures that require theorem provers for verification.

Because plugins may support a variety of different abstraction maps, the Hob framework allows each analysis to accept the syntax which is most convenient for its purposes. In general, however, each analysis typically accepts one or more abstraction maps (one for each abstract set in the module) defined in a language suitable for the properties it is designed to analyze. It may also accept a specification of the internal representation invariants that any analyzed data structures satisfy. Once again, these invariants are expressed in a language suitable for the verified properties.

Our flag analysis plugin verifies that modules implement set specifications in which integer or boolean flags indicate abstract set membership. Using these abstraction modules, the developer may specify the correspondence between concrete flag values and abstract sets from the specification. This abstraction language defines abstract sets in two ways: (1) directly, by stating a base set; or (2) indirectly, as a set-algebraic combination of sets. *Base sets* have the form $B = \{x : T \mid x.f{=}c\}$ and include precisely the objects of type T whose field `f` has value `c`, where `c` is an integer or boolean constant; the analysis converts mutations of the field `f` into set-algebraic modifications of the set.

We use the flag analysis in our Minesweeper example, described in Section 6.2. In many other examples, we use the flag analysis without defining any sets to analyze "coordination" modules. These coordination modules simply call upon other modules to manipulate set membership. The flag analysis is appropriate for analyzing such modules because it does not impose any overhead when no sets are defined.

6 Experience

In this section, we describe our experience using the Hob system to implement and specify design information for two programs. The first program implements an HTTP 1.1 server, the second implements the popular Minesweeper game. 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. For both programs we describe how the set specifications allow designers and developers to state, communicate, and enforce design-level information about the programs.

6.1 Case study: HTTP server

The HTTP 1.1 server implements the basic HTTP 1.1 protocol. This server hosts the Hob project homepage (see `http://hob.csail.mit.edu`).

Description. Our web server reads configuration data from a file and then listens for HTTP requests on the port specified in the configuration file. It serves these requests by transmitting the appropriate headers and content to the client. If the client's headers indicate that it supports compression, the server uses a gzip library to compress the data, and sends the compressed version to the client. Furthermore, we optimized our HTTP server by caching the results of previous requests (both uncompressed and compressed) and serving results from the cache. The Hob webserver contains 14 modules, 1229 lines of implementation, and 335 lines of specification. The server contains the following abstract sets of objects:

- `HTTPRequest.Headers` — the set of HTTP request headers.
- `HTTPRequest.Entity`, `HTTPRequest.Request`, `HTTPRequest.General`. The three different kinds of HTTP request headers. Together, these sets partition `HTTPRequest.Headers`.
- `HTTPResponse.C` — the set of HTTP response headers
- `CacheSet.Content`, `CacheBlacklist.Content`. These two sets capture information about the request content cache. `CacheSet.Content` is the set of objects in the cache; `CacheBlacklist.Content` is a set of objects that must not be placed in the cache (typically because they are too large).

Serving a request. When serving an HTTP request, the server first needs to capture information about what data the client is prepared to accept. To do this, it builds the set `HTTPRequest.Headers` and partitions it by header kind into the sets `HTTPRequest.General`, `HTTPRequest.Request` and `HTTPRequest.Entity`, for general, request and entity-headers, respectively. These headers affect the response which the server will transmit back to the client; for instance, the presence of the appropriate request header allows the server to transmit compressed data to the client. 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.

Response headers. The usual structure of an HTTP response occurs in two parts: a 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, which can send itself 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. Before serving any HTTP request, we always emit a basic header, which contains mandatory fields like the `Date` field; we can therefore guarantee that the `HTTPResponse.C` set is non-empty. 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`. We ensure that this precondition always holds by restoring it upon exit to `sendFile`; in particular, we ensure that `card(HTTPResponse.C') = 0`.

Note that this specification does not constrain the cardinality of `C` during the execution of the procedure. In fact, the `HTTPResponse.emit` procedure requires `C` to be non-empty; clearly, it is inconsistent with this design to transmit empty responses. A different (and in our opinion inferior) design might populate the set `C` only 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 decision 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, using the cache if applicable. 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 code.

Our approach makes this design decision explicit and much more accessible. We declare the sets `CacheSet.Content` and `CacheBlacklist.Content`. It turns out that these sets are defined by instantiating linked lists, and Hob's ability to combine 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. The specification for the `sendEntry` procedure makes it clear that the content to be transmitted will either be in the `CacheSet.Content` or `CacheBlacklist.Content` sets. Hob establishes the precondition for this procedure by 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 Hob verifies that the implementation conforms to this design.

6.2 Case study: Minesweeper

We have ported an implementation of the popular Minesweeper game to the Hob system and verified design conformance properties for this program. We structured our version of minesweeper using several modules: a game board module (which represents the game state), a controller module (which responds to user input), a view module (which produces the game's output), an exposed cell module (which stores the exposed cells in an array), and an unexposed cell module (which stores the unexposed cells in an instantiated linked list). There are 787 non-blank lines of implementation code in the 6 implementation modules and 328 non-blank lines of design information in specification and abstraction modules.

Minesweeper uses the standard model-view-controller (MVC) design pattern. The `board` module implements the model part of the MVC pattern. We have chosen to represent game state using an array of `Cell` objects. Each `Cell` object may be mined, exposed or marked. Abstractly, we define the sets `MarkedCells`, `MinedCells`, `ExposedCells`, `UnexposedCells`, and `U` (for Universe) to 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.

Note that the sets of exposed and unexposed cells in the `board` are implicit: in fact, they are defined by fields of the `Cell` objects. Our implementation also maintains explicit copies of these sets in the `ExposedSet` and `UnexposedList` modules; the set-based copies point to the same `Cell` objects as those in the board, but permit access to and reasoning about these subsets of the board in a more direct fashion. Our set specifications document the fact that the `board` and the `ExposedSet`/`UnexposedList` always have identical memberships, and permit the Hob analysis engine to verify that this equality holds throughout the program's maintenance life-cycle.

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

- The sets of exposed and unexposed cells are disjoint; unless the game is over, the sets of mined and exposed cells are also disjoint.
- The set of unexposed cells maintained in the `board` module is identical to the set of unexposed cells maintained in the `UnexposedList` list.
- The set of exposed cells maintained in the `board` module is identical to the set of exposed cells maintained in the `ExposedSet` array.
- At the end of the game, all cells are revealed; *i.e.* the set of unexposed cells is empty.

These properties illustrate how Hob enables designers and developers to state application-level design properties, establish a connection between these properties and the implementation, then verify the properties to provide a guarantee that the implementation conforms to the design.

Enforcing Set Consistency Properties. In our set specification language, the set consistency property equating sets in the `board` with sets in the `ExposedSet` and `UnexposedList` modules is expressed as follows:

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

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

6.3 Discussion

We found that the presence of abstract sets affected how we structured the our implementations: as we coded the implementations, our underlying set-based view of the program state forced us to think about the system at a deeper and more abstract level. In particular, we had to think about which invariants the system would satisfy (which helped us further structure our implementation). The process of encoding the relevant invariants and postconditions in terms of Hob's set specification language served to record this design documentation and to bind it to the implementation.

Hob's verification of design properties complements software testing. While developing our benchmarks, we found testing to be quite good at finding errors in programs, especially those encountered by typical program executions. However, testing has a number of shortcomings: while static analysis considers all executions of the program, testing's effectiveness is limited by the test suite. More importantly, testing cannot neither record design information nor enforce design constraints. Conventional regression testing only detects drift in the program's behaviour, not the program's design. In particular, testing does not guard implementators from changes that falsify needed invariants unless the changes alter the program's output[1].

The sequencing of our implementation and specification process differs from the traditional "specification first" approach. We believe that our process shares a number of the benefits of the traditional approach: because developers have already designed the program's abstract set layout and have thought about the necessary invariants, the program will have more structure than in a specification-free approach. On the other hand, the fact that the specifications are only formalized relatively late preserves code flexibility; it is easy to refactor procedures and redesign the implementation's structure during the development process. Once the program is relatively well-developed, we can set down the relevant design information in the form of specifications and verify that the implementation conforms to the specification. The verified design properties thus obtained will continue to be useful throughout a program's development lifecycle: they can serve as program documentation, and the Hob system guarantees that the implementation correctly implements the design.

Verifying design properties helps to avoid design drift. After we encoded our design properties in the form of specifications, the fact that Hob can verify these specifications made it possible for us to, in some sense, regression-test the design properties. As long as Hob certifies that the implementation continues to conform to the specification, the design properties are still valid on the current version of the implementation. We believe that Hob's automated verification approach is critical for avoiding design drift.

Hob's support for an iterative specification process helped us adopt an incremental specification development process. We believe that such a process is a useful way to add specifications to pre-existing systems, which often start without any design documentation information at all. The prospect of documenting a system's design is often daunting, and the ability to do so piecemeal rewards developers continuously as they add additional documentation. Furthermore, an incremental process works well when different aspects of a system's design become important over time. For instance, when a developer wishes to augment a subsystem's functionality, it could be useful to verify invariants and postconditions for the initial code—which was previously not worth specifying—before adding new functionality to the subsystem. That way, after carrying out the changes, the developer could ensure that the invariants and postconditions continue to hold.

7 Related Work

We compare the Hob system with several specification systems and tools for verifying design conformance. Specification systems can express sophisticated program properties, but typically do not include support for automatically proving that implementations satisfy these properties. Tools for verifying design conformance do not support the detailed static analyses that Hob employs.

The Z Notation. The Z notation [18] allows system designers and implementers to express properties of their systems. The power of the Z notation enables it to completely specify system properties, so that—in principle—any desired property of the system could be specified down to the implementation level.

Compared to Z, Hob was designed to address the more focused design conformance problem—it restricts developers to design-level properties that they can express using abstract sets of objects. Hob's partial specification approach enables it to effectively verify design conformance properties. Moreover, the set-based specification language facilitates the task of providing design information and rewards an iterative specification process. Partiality is also important because our goal is to make the design accessible: the level of detail in a complete specification could obscure the design decisions we wish to expose.

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 related languages for specifying and implementing systems [1, 6, 8]. Previous work

[1] Assertions can, of course, verify that invariants continue to hold, but they still require a suitable test suite to 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. Wide-spectrum specification languages therefore do not address the issue of design drift: implementations and specifications tend to end up diverging in the absence of tools that automatically verify that an implementation conforms to its specification.

Hob does not use a wide-spectrum specification language; we instead provide separate specification and implementation languages, and automatically verify the conformance of the implementation to the specification with respect to the abstraction language. Hob therefore guarantees that a program's implementation conforms to its design throughout its lifecycle, preventing design drift.

Systems for Verifying Design Conformance. Hob verifies that implementations of software systems conform to their designs. Because techniques that do not inspect source code are always vulnerable to design drift, a number of related efforts also extract design information from source.

The ESC/Java2 [4, 7] tool verifies partial JML [2] specifications in Java programs. The JML specifications that ESC/Java2 verifies are essentially legal Java expressions (with some added keywords). A major difference between the two approaches (ESC/Java2 and Hob) is that Hob takes a stronger position on the kinds of specifications that developers should write. In particular, 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 focused on set-based properties, which we believe to be important and relevant for design information. We believe that the Hob approach focuses the attention of the designers and developers on the important core aspects of the design and facilitates the effective verification of those aspects.

Murphy et al. have proposed reflexion models [16], where developers propose a model of a software system and an algorithm which extracts 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 do not prescribe how models are to extracted. One way to build a model is to assign module membership on a per-file basis and to use procedure calls between these files to determine module interaction. Sefika et al. present the Pattern-Lint tool [17], which explores an approach that is similar in spirit to reflexion models. Their approach combines cursory static analysis and dynamic analysis 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 (by collecting evidence for and against design conformance) rather than the static analysis that it uses, which appears to be limited to inspecting method calls and shared global variable accesses. Lam and Rinard have proposed the token annotation system [14]. The token system automatically extracts design information from the program source code using developer-provided annotations and produces diagrams summarizing direct and indirect (heap-mediated) interactions. Like these systems, Hob attempts to verify that a system's implementation conforms to its design. However, Hob's use of sophisticated static analysis techniques allows it to verify deeper behavioural properties: because Hob's abstraction mappings provide summaries of the concrete heap state, users of Hob can express important program invariants at an abstract, set-based level, and statically verify that these invariants hold in the implementation.

8 Future Work

Hob has demonstrated the feasibility of performing modular verification of sophisticated design and program correctness properties that use abstract sets of objects as their central organizing concept. The next step is to move beyond sets to include relations. Such a step would enable developers to express a wider range of useful program correctness properties. In particular, relations are crucial for expressing designs involving maps between sets of objects; these maps are often implemented in the software system using hash tables or association lists. Successfully generalizing the Hob approach to support relations would enable the expression and verification of such aspect of the design and the corresponding correctness properties.

References

[1] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

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

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

[4] D. R. Cok and J. 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.

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

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

[7] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report 159, COMPAQ Systems Research Center, 1998.

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

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

[10] P. Lam. *The Hob System for Verifying Software Design Properties*. PhD thesis, Massachusetts Institute of Technology, 2006.

[11] P. Lam, V. Kuncak, and M. Rinard. Cross-cutting techniques in program specification and analysis. In *4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, 2005.

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

[13] P. Lam, V. Kuncak, K. Zee, and M. Rinard. Set interfaces for generalized typestate and data structure consistency verification. *Theoretical Computer Science*, submitted.

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

[15] P. G. Larsen and J. Fitzgerald. VDM information: Examples repository, November 2000.

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

[17] M. Sefika, A. Sane, and R. H. Campbell. Monitoring compliance of a software system with its high-level design models. In *ICSE'96*, pages 387–396, 1996.

[18] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., 1992.

[19] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field constraint analysis. In *VMCAI 2006*, 2006.

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