Design Conformance 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.

We present a new system that enforces the connection between abstract sets of design objects and concrete 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 constraints. We have implemented our system and used it to develop several applications; our experience shows that it can effectively enable developers to explicitly state and verify that the applications correctly implement the design.

1 Introduction

Sets of objects are a primary concept in many object modeling formalisms. Within this context, using sets of objects as a conceptual tool enables developers to productively articulate and explore key aspects of the system's design. This articulation and exploration can help developers discover and correct problematic design flaws early when the cost of the correction is still relatively tractable. And sets of objects provide an invaluable communication medium that allows team members to precisely and easily communicate their ideas about the system to others.

Object-oriented languages such as Smalltalk and Java enable developers to structure the state of the system in terms of objects. It therefore seems natural for the sets of objects in the implemented system to parallel the sets of objects in the design model. In practice, however, implementation concerns often cause the sets of objects in the implementation to diverge from the sets of objects in the design. All too often, the result is a poor correspondence between the objects in the design and the objects in the implementation, with the gap growing during the development and maintenance phases as the program evolves under the pressure of implementation challenges and new functionality demands. The absence of a design that accurately reflects the reality of the implementation can cause the overall structure of the implementation to degrade and deprive developers of a useful high-level perspective on the program.

This paper presents a new specification language and associated verification system that can establish a guaranteed connection between the sets of objects in the design and corresponding objects in the implementation. A developer using our Hob system first declares the abstract sets of objects in the design along with abstraction maps that establish the correspondence between the abstract sets of objects in the design and the concrete objects in the implementation. With this correspondence in place, the developer can then use the abstract sets in conjunction with the specification language to specify 1) design-level invariants that the sets of objects must satisfy, 2) how the operations of the program affect the movement of objects between the abstract sets, and 3) preconditions (in terms of membership in abstract sets of objects) that objects must satisfy to participate in different operations.

1.1 Basic Structure

A developer using our system structures the program as a collection of (instantiable) modules. Modules encapsulate data structures and/or related computations. Each module may also export one or more abstract sets, with membership in abstract sets defined by one or more abstraction maps. So, for example, a system might contain a list module that encapsulates a list of objects. The module would export an abstract set; the abstraction map would specify that the set contains the objects in the list. The system could then instantiate the module multiple times, once for each list in the program, providing a different name for the exported set each time. The developer would then be able to use these sets to express verified design-level constraints that involve multiple lists (for example, that two lists contain disjoint sets of objects).

The list module would also encapsulate the implementations of the operations that access and update the list. Such operations have preconditions and postconditions; the specification language enables the developer to use abstract sets of objects to precisely state many of these properties. For example, the list removal operation might require the removed element to be in the list before the removal and ensure its absence after the removal. The corresponding precondition would require the object to be a member of the abstract set of objects in the list before the removal; the corresponding postcondition would guarantee the abstract set of elements after the removal to be the set from before the removal minus the removed object. The verification system would then verify that the implementation satisfies all needed preconditions and postconditions. In this way, our system enables developers to confidently reason about the effects of the actions of the system at the design level rather than at the implementation level, secure in the knowledge that the implementation correctly implements the stated design.

1.2 Implementation

We have developed a system, Hob, that embodies the concepts in this approach. Hob contains multiple analyses and is capable of deploying these analyses in a coordinated way to verify our targeted design-level properties. We have used Hob successfully to design, implement, and verify several complete applications.

1.3 Benefits

This 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 effectively 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:** Abstraction maps can discard objects that are required for the implementation but irrelevant and distracting for high-level design purposes. By defining appropriate sets of objects, abstraction maps can eliminate the clutter and excess implementation detail that would otherwise obscure key elements of the design.

- **Design Conformance:** Developers use an abstract set specification logic to specify key invariants that the sets of objects must satisfy and preconditions that objects must satisfy to participate in each computation. These invariants and preconditions capture key design constraints. The fact that our analysis system verifies the invariants and preconditions 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 key 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.

1.4 Contributions

This paper makes the following contributions:

- **Rationale:** It discusses the rationale behind the design of the specification language. In contrast to general specification frameworks such as JML and Z, which aspire to enable the specification of a broad range of program correctness and consistency properties, we focus on properties that relate the implementation and design of the system. This different goal causes our specification language to differ significantly from other approaches.

- **Experience:** We have used our system to specify and verify several programs. We discuss the kinds of properties we chose to specify and verify, how we expressed the specification, how we integrated the specification and verification into the development process, and our overall experience using our system in this context.

The remainder of the paper is structured as follows. Section 2 presents an example that illustrates our approach. Sections 3, 4, 5 and 6 briefly present key parts of the Hob system. Section 7 presents our experience using our system to develop two applications, a Minesweeper game and a web server. Section 8 presents related work and section 9 concludes.

2 Example

Our example is inspired by the `fontconfig` library for font customization and configuration. Consider a situation in which program, e.g. a word processor, needs to draw text on the screen. Before the program may do so, it must select an appropriate font to render this text; the font must belong to the set of fonts available on a particular system. Often, more than one font may be acceptable — when prompting the user to pick a font, for example, the client program selects one font from the set of all fonts satisfying some partial constraints. Our solution to this problem takes as input the set of available fonts on the system, as well as some matching criteria, and returns as output a suitable set of fonts to the client program.

Our solution contains several modules. The `Matcher` module provides the public interface to the matching code; it takes a target set of font attributes and uses the system state to compute a set of fonts which match the desired attributes. The `Patterns` module is used by the `Matcher` module (and others) to access the set of available fonts. This module instantiates a doubly-linked list twice to represent two sets: the set of fonts available to the system, and the set of selected (matching) fonts.

To enable developers to communicate meaningfully about the program, we have given names to the sets described above. Our system includes the set of active fonts, `Active.Content`, and the set of selected fonts, `Selected.Content`. At the design level, the task then reduces to populating the set `Selected.Content` with the appropriate fonts from the set `Active.Content` of active fonts. The following invariant (which states that all selected fonts are also active):

```
Selected.Content in Active.Content
```

should hold throughout the program's execution, and especially upon return from the `Matcher`.

Note that the `Selected.Content` and `Active.Content` sets do not exist as such in the implementation — because our implementation stores the selected and active fonts in linked lists, the the `Active.Content` set consists of all objects reachable through the `next` field from a reference in the `Active` module (and similarly for the selected set). Expressing this constraint directly in terms of the reachability at the implementation level would inappropriately inject implementation-level concerns into design-level information. As discussed below, our solution therefore factors this constraint into two parts: an abstraction map uses a transitive closure operation on the implementation-level `next` fields to define the meaning of the abstract sets, and a specification invariant (which operates entirely at the level of abstract sets) states the crucial inclusion relationship. This separation enables designers and developers to reason about design-level information in isolation from the complexity of the implementation, all the while secure in the knowledge that the verification system will ensure that the implementation correctly reflects the design.

To realize this approach, each module in our system contains three parts: a specification (which contains the declarations of the exported abstract sets and the preconditions and postconditions of the exported procedures), an implementation (which contains the concrete data structures and code that implements the module), and an abstraction map, which defines the meaning of the abstract sets. Constraints (such as the one discussed above) are typically stated separately (facilitating the expression of constraints involving sets from multiple modules). Our use of specifications, in terms of preconditions and postconditions, enables the use of assume/guarantee reasoning; one unique aspect of the Hob system is that it exploits this program structure by deploying multiple analyses, each of which is tailored to a specific class of modules. These analyses then cooperate to verify that the program correctly preserves the constraints.

We next describe the structure of the implementation, specification, and abstractions used in our solution to the font matching problem. We also briefly describe how we ensure that the implementation conforms to the design properties included in our specifications.

List Implementation. Our example is centred around two sets: the set of available fonts and the set of selected fonts. We use a list data structure to implement both of these sets. In our approach, we expect that common library modules such as linked lists will be implemented and verified once and for all. Our support for instantiable modules allows developers to reuse both implementation and verification effort.

We next describe how Hob uses one of its analyses (the Bohne analysis [11]) to verify the relevant list module properties for our example. The Bohne analysis enables developers to verify properties involving abstract sets implemented by linked data structures in the heap. In particular, given a procedure implementation and specification, the Bohne analysis computes a weakest precondition for the procedure's implementation (starting from the postcondition) in terms of monadic second-order logic formulas, and then uses the decision procedure implemented in the MONA tool to verify that the given precondition (as translated into monadic second-order logic) implies the computed weakest precondition.

Figure 1 presents our implementation of a doubly-linked list add procedure, while Figure 2 presents the corresponding specification; we have omitted the other procedures in this module. Observe that the List.add specification states that the abstract set Content' of objects in the list—the prime indicates the Content set after the procedure finishes executing—gains exactly the object n passed as a parameter to add upon completion. The implementation carries out this specification by adding the given object to the head of the linked list. If the list was not previously empty, then the implementation connects the old root's prev field to the new element.

To verify that the implementation conforms to the specification, we use the following abstraction map:

```
Content = {x : Node | "(lambda v1 v2: v1.next = v2)* root x"};
```

This abstraction mapping states that the Content set consists of all objects reachable from the root reference through next fields. This abstraction map establishes the connection between the concrete list data structure and the abstract set of elements in the list. By giving meaning to the specification in terms of the implementation, it lays the foundation for the verification of the implementation. Figure 3 presents the full abstraction module, which includes, in addition to the abstraction map, a specification of the internal representation invariants required to ensure that the data structure consistency. Bohne uses these invariants as it analyzes the operations to ensure that they operate correctly; it verifies that if each invariant holds upon entry to each procedure, it also holds on exit. Given this set of module sections (the implementation, specification, and abstraction module sections), Hob is able to deploy the Bohne analysis to verify that the List module implementation correctly satisfies its implementation.

List Instantiations. When developing programs for the Hob system, we expect that developers will make use of implemented, specified and verified library modules like the above List module. To use these library modules, developers must *instantiate* the modules. The instantiation operation creates a fresh copy of the List module with a different name and operating on a different type. In our fontconfig example, we need a set of available fonts and a set of selected fonts. Figure 4 presents the instantiation operation: it instantiates the List module twice, substituting all references to the Node type with references to the Pattern type. After instantiation, the Patterns module is then free to use the Available and Selected modules as if they were List modules.

Scopes. The key design invariant in our example is that Selected.Content in Active.Content. Note that this invariant involves sets from different modules. Note also that, in general, such invariants may

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

  reference root : Node;
  proc add(n : Node) {
    if (root==null) {
      root = n;
      n.next = null; n.prev = null;
    } else {
      n.next = root; root.prev = n;
      n.prev = null; root = n;
    }
  }
}
```

Figure 1: Implementation of List.add procedure.

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

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

Figure 2: Specification of List.add procedure.

```
abst module List
{
    use plugin "Bohne decaf";

    Content = {x : Node | "(lambda v1 v2: v1.next = v2)* root x"};

    invariant "ALL x. (x.next = root) --> root = null";

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

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

Figure 3: Abstraction module for List module.

```
impl module Available = List with Node <- Pattern;
impl module Selected = List with Node <- Pattern;

spec module Available = List with Node <- Pattern;
spec module Selected = List with Node <- Pattern;
```

Figure 4: Instantiation of the Available and Selected sets.

```
scope Fontconfig {
    invariant "Selected.Content in Active.Content";
    modules Matcher, Patterns, Selected, Active;
    exports Matcher, Patterns;
}
```

Figure 5: Scope declarations for font matching example

```
impl module Matcher
{
    proc select(target:Pattern) {
        bool b;
        Patterns.openAvailableIter();
        b = Patterns.hasAvailableNext();
        while (b) {
            Pattern p = Patterns.getAvailableNext();
            if (Matcher.matchAny (target, p))
                Patterns.addSelected (p);
            b = Patterns.hasAvailableNext();
        }
    }
}
```

Figure 6: Implementation of outer loop for font matching.

be temporarily violated as the program updates the data structures that define these sets. The developer therefore uses the `scope` construct to identify the invariant and a set of modules within which the invariant may be temporarily violated. Figure 5 contains the `scope` construct in our example. This construct identifies the invariant and the set of modules (`Matcher, Patterns, Selected,` and `Active` within which the invariant may be violated. The program must properly coordinate operations that affect the `Selected.Content` and `Active.Content` to ensure that the invariant holds; in our example the `Matcher` and `Patterns` modules perform this coordination. These modules must therefore contain the only invocations of procedures in the `Selected` and `Active` modules. The scope construct in Figure 5 accomplishes this goal by exporting only the `Matcher` and `Patterns`.

The analysis engine verifies the invariant by assuming it upon entry to procedures in the `Matcher` and `Patterns` modules and verifying that these modules properly restore the invariant upon exit.

Matching Fonts. The main task for this example consists of iterating through the set of available fonts and choosing the fonts that fit the specified criteria. In terms of sets, the main loop therefore adds all fonts from the `Available.Content` set that match the given criteria to the `Selected.Content` set. Since the matcher and the set of fonts are in different modules, it is reasonable to expect that these modules should not share implementation details; instead, the matcher calls upon procedures in the `Pattern` module to navigate the `Active.Content` set and to add members to the `Selected.Content` set. In turn, the `Pattern` module delegates tasks to the `Selected` and `Active` modules, which instantiate the standard doubly-linked list seen above.

Figure 6 presents the main loop for font matching. It iterates on the set of available patterns and selects those patterns that satisfy `Matcher.matchAny`. Our loop invariant inference algorithm [9] automatically deduces the invariant `Patterns.Selected' in Patterns.Active'` from this implementation, and Hob successfully verifies this postcondition.

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

Figure 7: Grammar for Implementation Language

3 Implementation Language

The implementation language is a simple imperative language with procedures, object references, and dynamic object allocation.

Figure 7 presents the syntax for the implementation language. 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 [2]. 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 persistent 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.

This implementation 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 in a module. This structure harmonizes with the abstract set approach — in most cases, each data

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

Figure 8: Grammar for Module Specification Language

structure will be rooted at a given module variable and will implement one or more abstract sets.

4 Specification Language

Figure 8 presents the syntax for the module specification language. Abstract set declarations identify the 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 B over abstract sets to specify these effects. The `modifies` clause bounds the collection of sets directly modified by a procedure; note that `modifies` clauses need not declare sets modified by the transitive callees of a procedure.

The expressive power of boolean clauses B is the first-order theory of boolean algebras, which is decidable. It is therefore possible to develop an algorithm that automatically synthesizes loop invariants, to implement the transfer function for each statement, and to check implication when verifying ensures clauses. Our implemented flag typestate analysis uses such an algorithm [9]; we expect other analyses to exploit this decidability property in similar ways.

5 Scopes

Developers use boolean clauses B (from the specification language in Figure 8) 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 [6] 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* (Figure 9 presents the syntax of scope declarations) 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.

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

Figure 9: Grammar for Scope Declarations

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 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.

6 Abstraction Maps

At this point we have defined an implementation language that uses standard concrete programming language elements such as object references and fields to implement the full behavior of the program. We also have a specification language that uses abstract sets of objects to state key design properties that the program should satisfy. By the very nature of design, the properties are partial — they do not completely specify the behavior of the program (and any attempt to do so would obscure the high-level design properties in a mass of detail).

The remaining step is to establish a verified connection between the design and the implementation. Abstraction maps provide the connection; module analyses provide the verification. A central feature of the Hob framework is the ability to deploy multiple analyses, each of which is specialized to analyze a specific kind of module. Because different analyses use different techniques, it becomes possible to select the most appropriate an analysis for each module, depending on the expressiveness and the scalability required to analyse that particular 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 [9] assigns set membership according to the values of integer fields (similar to an enumeration in standard programming languages). The Bohne analysis [11] performs field constraint analysis, a generalization of shape anal-

$$\begin{array}{rcl}
M & ::= & \textsf{abst module } m \ \{U\ D^*\ I^*\ P^*\} \\
U & ::= & \textsf{use plugin ``flags''}; \\
D & ::= & \textsf{id}{=}D_r; \\
D_r & ::= & D_r \cup D_r \mid D_r \cap D_r \mid \textsf{id} \mid \{x : T \mid x.f{=}c\} \\
I & ::= & \textsf{invariant } A; \\
A & ::= & \neg A \mid A \wedge A \mid A \vee A \mid B \\
P & ::= & \textsf{predvar } p;
\end{array}$$

Figure 10: Grammar for Flag Abstraction Modules

ysis [4], and assigns set membership according to heap reachability properties. Finally, our theorem proving analysis [13] assigns set membership using Isabelle predicates; it is useful, for instance, in reasoning about complex data structures that require the power of theorem provers to verify.

Because plugins may support a wide variety of different abstraction maps, the Hob framework does not have a single syntax for specifying abstract set membership. Each analysis is free 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.

Figure 10 presents the syntax for abstraction modules for the flags analysis. Our flag analysis 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 B. In that case, the flag analysis assigns internal names to anonymous sets and tracks their values to compute the values of derived sets.

We use the flag analysis in our Minesweeper example, described in Section 7.2. 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.

7 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.

7.1 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) in memory and serving results from the cache. The Hob webserver contains 14 modules, 1229 lines of implementation, and 335 lines of specification. Figure 11 presents a module dependency diagram for our web server. 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

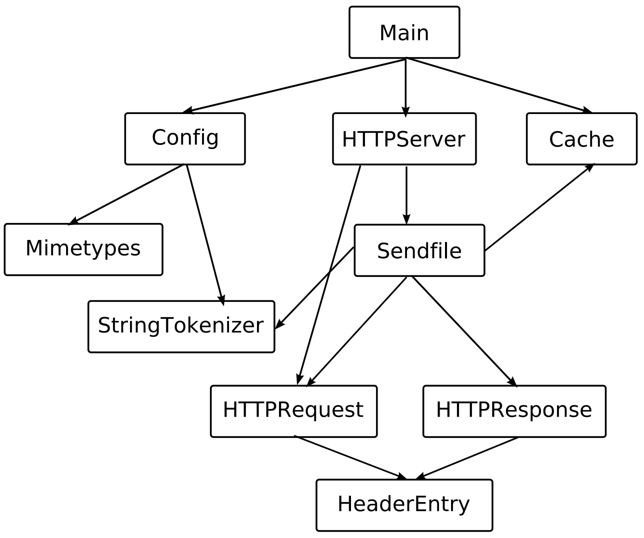

Figure 11: Module dependency diagram for web server

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 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 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, 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 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`. It turns out that these sets are defined by instantiating 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;
```

Note that this specification 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, with the Hob analysis system verifying that the implementation correctly conforms to this design.

7.2 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.

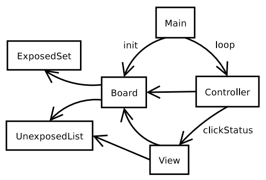

Figure 12: Module dependency diagram for Minesweeper

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.

8 Related Work

We compare the Hob approach to the Z notation and to JML (as verified in the ESC/Java2 project).

Comparison with Z. The Z notation [10, 12] allows system designers and implementers to express properties of their systems, including system specifications. Z is based on first-order predicate logic and typed set theory; Z specifications are undecidable in general. This implies that there is no algorithm which can check (in general) that Z specifications are logically consistent; nor can the developer compute (in all cases) whether a certain statement is implied by a system's specifications. There are, however, many tools which typecheck, model check and animate Z specifications, increasing the developer's confidence that a system's specifications are meaningful. Because of the power of the Z notation, it can completely specify system properties, so that, in principle, any system can be specified completely, even down to the implementation level, if desired.

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. One of the payoffs for this partial specification approach is that Hob can effectively verify the design. Moreover, we believe that our set-based specification language facilitates the task of providing the design information and rewards an iterative specification process. In particular, developers can specify only a certain subset of properties of interest and verify just those properties. Later on, when different properties become important, the developer can add the additional properties to the specification and check these properties independently. Partiality is also important in our approach because our goal is to make the design accessible — the level of detail in a complete specification would obscure the design decisions we wish to make easily available to developers and designers.

Comparison with JML. The ESC/Java2 [3] tool verifies partial JML [1] 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 then the boolean algebra over sets and boolean flags. Unlike JML specifications, which support implementation-level constructs such as strings, integers, or floating-point values, Hob's specification language is focused on a particular set of properties that we believe is important and relevant for the design. We find this approach productive in that it focuses the attention of the designers and developers on the important core aspects of the design and facilitates the effective verification of those aspects.

9 Conclusion

The divergence of design and implementation can significantly complicate the development and maintenance of software systems. Potential issues include the unreliability (and therefore decreased utility) of the design as a source of information about the implementation, the potential for the development team to lose its awareness of the underlying structure of the system as expressed in the design, with the resulting potential for poor design decisions to creep in a nd degrade the system structure.

The Hob system addresses this problem by enabling developers to define abstract sets of objects and use these sets to state important design constraints. Abstraction maps enable these abstract sets to avoid the inevitable (and for designs, irrelevant) detail that implementations require while still maintaining a verified correspondence between the implementation and the design. Our experience using Hob to implement and verify several programs indicates that this approach can facilitate the expression and verification of high-level design information and that this information can make key design properties easily stand out from the complexity of the implementation. Eliminating the lack of correspondence between design and implementation can dramatically increase the transparency of key design decisions, enabling these decisions to be either preserved or intelligently reworked (as appropriate) and providing developers with key documentation that helps them to better understand their system.

References

[1] 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.

[2] 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.

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

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

[5] P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking using set interfaces and pluggable analyses. *SIGPLAN Notices*, 39:46–55, March 2004.

[6] 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.

[7] 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.

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

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

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

[11] T. Wies, V. Kuncak, P. Lam, and M. Rinard. Field constraint analysis. In *7th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2006)*, in preparation.

[12] *Information technology—Z formal specification notation—Syntax, type system and semantics.* ISO Standard ISO/IEC 13568:2002.

[13] 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.