# Lightweight Analysis of Object Interactions

Daniel Jackson
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts, USA
dnj@mit.edu

Alan Fekete
Basser Dept. of Computer Science
University of Sydney
Sydney, Australia
fekete@cs.usyd.edu.au

*Abstract*. The state of the practice in object-oriented software development has moved beyond reuse of code to reuse of conceptual structures such as design patterns. This paper draws attention to some difficulties that need to be solved if this style of development is to be supported by formal methods. In particular, the centrality of object interactions in many designs makes traditional reasoning less useful, since classes cannot be treated fruitfully in isolation from one another. We propose some ideas towards dealing with these issues: a relational model of heap structure capable of expressing sharing and mutual influence between objects; a declarative specification style that works in the presence of collaboration; and a tool-supported constraint analysis to expose problems in a diagram that captures, at a design level, a pattern of interaction. We illustrate these ideas with an example taken from a program used in the formatting of this paper.

## 1 Motivations

The last decade has seen significant changes in the way high-quality software is developed. One important trend has been the wide adoption of practices that guide designers in the reuse not only of code but also of larger conceptual structures. The 'design patterns' movement has identified a range of problems that arise repeatedly in object-oriented design, and has named and articulated standard solutions to these problems. The movement has spawned a book [2] that is widely acknowledged as one of the most important contributions to software development practice in the last decade; on-going conferences extend this work. Some common structures (less general than patterns, but still applicable to a diversity of specific situations) have been expressed in 'frameworks' [1] whose classes express common aspects of a domain, while leaving 'template methods' with no behavior or only a default behavior; the details are then filled in by subclasses for each particular application.

Despite advances in formal methods, such practices still lack good support for reasoning at the design level. Does a design correctly express a pattern? When two patterns overlap, does one pattern undermine key properties supposedly ensured by the other? What obligations must be placed on a programmer who fills in a template method so that the framework properties are maintained? These kinds of questions might benefit from the ability to describe designs precisely, and analyze them automatically. Unfortunately, much of the theoretical work inspired by object-oriented programming has focused on providing tools to help language designers or compiler writers, leaving the software developer out in the cold.

To help developers reason about modern object-oriented software designs

involving patterns and frameworks, several difficult issues must be addressed. One is collaboration: the interaction of two or more objects designed to work in tandem. Almost all patterns achieve flexibility and decoupling by having several objects (of different classes) continually interacting with one another, delegating operations back and forth. This means that reasoning cannot treat each class in isolation as in classical abstract-data-type reasoning [4]; there are global invariants that need to be maintained. For example, the Observer pattern is central to systems built with a Model-View-Controller approach: a 'subject' class maintains some state, and notifies a range of different 'observers' whenever its state changes: each observer in turn queries the subject to obtain an up-to-date value for the aspect of the state it cares about.

Another difficulty arises from the fact that patterns and frameworks express only partial information about the final system. A class that plays Subject in an Observer pattern may also be a Component in a Composite pattern, all in addition to its application-specific roles. We need to reason about some aspect of the behavior of a class, while still capturing frame conditions so that our reasoning is not invalidated by other behaviors not currently of interest. Similarly, a template method in a framework is not known completely when we reason about the framework's contribution to an overall design, but some aspects of the method's properties must be expressed in order to establish properties of the framework as a whole.

## 2 Example

Java's iterators illustrate some of the complexities of object interaction. To traverse a collection sequentially, one obtains an iterator object (by calling a method on the collection), and then makes calls on the iterator itself to yield the objects of the collection one by one. To avoid copying, the iterator is usually implemented as a cursor that references the internal data structure of the collection. If the collection is mutated in the lifetime of the iterator, subsequent calls on the iterator can fail. In the standard Java collection framework, iterators are 'fail-fast', and if such a mutation occurs, the iterator will throw an exception rather than fail unpredictably at a later point. Avoiding such a failure has been dubbed the *comodification problem*.

To make sense of Java iterators, an iterator and its underlying collection must be considered together, as a single interacting unit. The objects are not aliased in the traditional sense – they have different types – but a mutation of one can affect the other. In fact, because one often wants to delete objects during iteration, a Java iterator provides a method that safely removes the object last yielded from the underlying collection. So the interaction goes both ways: calls on the iterator can affect the collection, and vice versa. It is also possible to create several iterators simultaneously active on the same collection; in this case, there is a worry that a removal via one iterator will cause another iterator to fail.

Object-oriented programs are rife with these kinds of interactions, and they account for much of their complexity (and perhaps many of their bugs). Mutations to a hash key can invalidate the invariant of a hash table, so that a lookup on the key will fail even though it is present. Streams attached to network sockets are interde-

pendent in Java; closing an input stream can cause an output stream on the same socket to be closed too. And the relationships between listeners, events and user interface components can be a major source of frustration even for experienced programmers.

For the rest of our paper, we will study the comodification problem in the context of a small text-processing program written recently by the first author (and used in the production of this paper). The program takes source text marked up with some simple tags that associate style names with paragraphs, name special characters symbolically, indicate regions of the text to be treated as mathematical formulas, and so on. It produces text marked up with more elaborate tags, in the import language of a typographic layout program such as Quark Xpress or Adobe InDesign. These import languages require more intrusive and low-level tagging than the source language; a special character (eg, †) is specified by font name and index, for example, rather than symbolically (\dagger). They are also application-specific.

The program works as follows. The source text is broken into tokens corresponding to commands, sequences of alphabetic characters, punctuation marks, etc. Tokens are classified into types, and a list of actions is associated in a registry with each type. When a token is read, the actions associated with its type are performed. For example, when the token for a special character command is read, the symbolic name is looked up in a table giving the font name and index, and the appropriate tagged output is generated. One possible effect of an action is to change the registry itself, by creating or deleting associations of actions with types. A command to enter mathematical mode, for example, produces a token whose action attaches an italicizing action to the token type corresponding to sequences of alphabetic characters, so that only letters (and not numbers or symbols) are affected.

The object model of Figure 1 expresses some of the heap invariants of the program. For readers unfamiliar with object models, here is brief explanation of the diagram's meaning; a full and more formal discussion of object models may be found elsewhere [7]. Each box corresponds to the set of objects belonging to the class (or interface) whose name is the label of the box. An arrow with a closed head from $A$ to $B$ says that the set of objects $A$ is a subset of the set of objects $B$, usually because $A$ is declared as a subclass of $B$. An arrow with an open head labelled $f$ from $A$ to $B$ says that each object in $A$ references objects in $B$ with a variable $f$, where $f$ is usually a field name, but may also be a local variable available only transiently (eg, during a particular method call), or a reference bound by the formation of an inner class. An exclamation mark at the end of a reference arrow says that there is exactly one object obtained by navigating the reference; if absent, there may be any number of objects, usually due to the elision of a collection object such as an array. Finally, an arrow labelled $f[I]$ from $A$ to $B$ indicates a set of indexed references: for each object $i$ in the index set $I$, there is a reference $f[i]$ from $A$ to $B$.

The main class is *Tagger*. Its objects hold references to a *Tokenizer*, which provides tokens by reading from a file through Java's *Reader* interface, and an *Engine*, which consumes tokens. An *Engine* is a registry; for each token type, it references a *LinkedList* whose elements are *Actions*. An *Action* may hold references to one or more *Engines*, to other *Actions*, and to a *Generator*. The interface *Generator* hides dif-

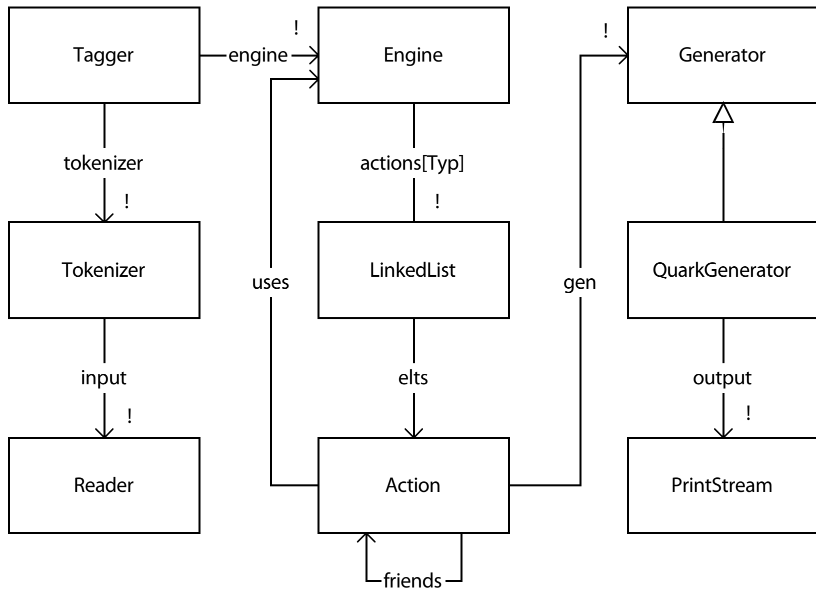


*Figure 1: Object Model for Tagging Program*

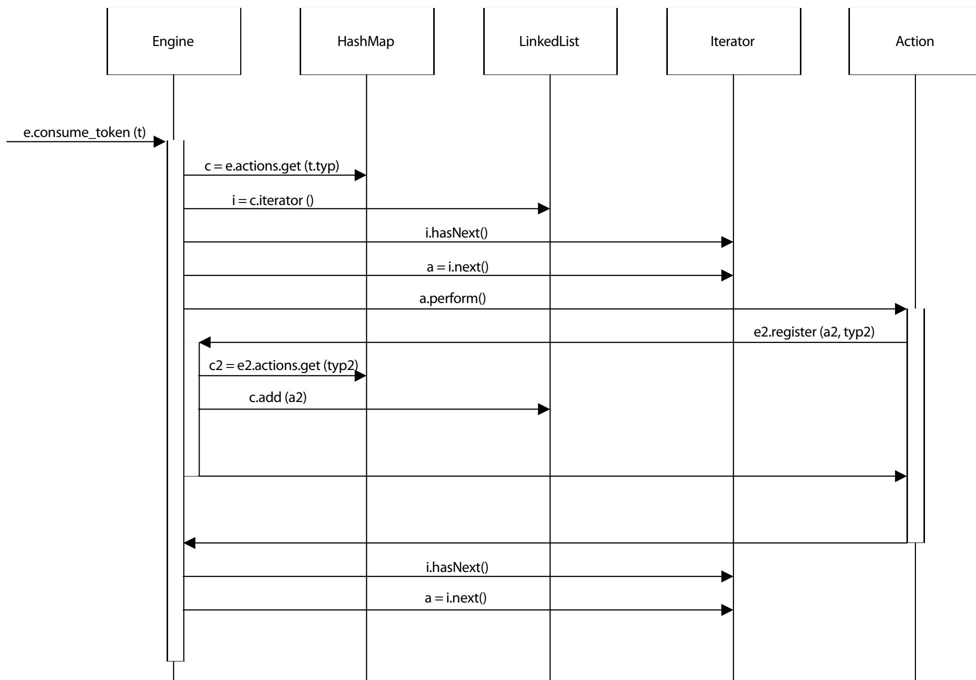


*Figure 2: Interaction Diagram for Tagging Program*

tacs-01.qxd 8/1/2001 11:16 AM Page 5

ferences between the tagging conventions of different layout programs. The class *QuarkGenerator* provides Quark-specific tagging, and holds a reference to a *PrintStream* for its output.

An object model does not have to be uniform in its level of abstraction. We have chosen to make explicit the presence of linked lists that hold actions in the engine, but to treat them abstractly as containers with a reference *elts* to their elements. The hash map that maps token types to lists is not shown, although its mapping is represented in the indexed relation from *Engine* to *LinkedList*. There are a number of subtleties that we shall ignore since their elaboration would add little to our discussion. The actions, for example, are in fact constructed from a variety of inner classes, which differ in what references they hold. A more accurate model would show these inner classes as subsets of *Action*, each with its own distinct references.

The events that occur when a token is consumed by the engine are shown in the interaction diagram of Figure 2. In short, the diagram is read as follows: time runs top to bottom; vertical boxes represent the activations of method calls; solid horizontal arrows represent method invocations; method returns are usually implicit, but when shown are drawn dashed. This diagram shows a particular trace. Invoking the *consume_token* method causes the engine to obtain the action list for the token's type by a lookup on its hash map. It then calls the *iterator* method of the list to create an iterator object. It then loops over the elements of the list: it checks that further elements are available, by calling *hasNext*; obtains the next element; calls the *perform* method on the action; and then repeats until the second call to *next*. The *perform* method registers an action with the engine; in response, the engine, obtains the appropriate action list and adds the action to it.

This trace demonstrates a potential flaw of the design: a possibility of comodification. If the list to which the action is added is the same list on which the iterator is based, the second call to *next* will fail.

## 3 An Approach

No comprehensive method for describing and reasoning about realistic object-oriented designs has been developed that can handle the kinds of object interaction due to collaboration, sharing, and overlapping patterns. Below we offer some fragmentary ideas we hope may contribute towards such a method. Before introducing them, let us summarize the main requirements a method will need to satisfy to be workable in practice.

Formal description must be lightweight [11]; this means that a software developer should not have to express everything about the system being developed, but can instead target formal reasoning at those aspects of the system that are especially difficult or risky. Furthermore, it must be possible to vary the level of detail applied to each part of the program or each aspect of its behaviour in a non-uniform manner. The developer will want to have some part of the design represented with attention to all the objects and the exact sequence of operations, while other parts are summarized by a declarative pre- and postcondition for each method, and still others are simply expressed in frame conditions saying that nothing significant happens.

Automated tool support is essential. Relying solely on the mathematical insight of software developers has not been successful. At the simplest level, a tool can check the syntax of a formal representation; at the other extreme, a tool can extract formal representations from code without user involvement and check them against predefined properties. We believe that developers will get most assistance from a tool that allows the developer to explore the behavior of a design, and that determines automatically whether some specified property holds. Some tools have concentrated on certifying that a property does hold by generating (or checking) a logical proof; our experience has led to an emphasis on tools that generate counterexamples when the property does not hold. This is based on the value of formal methods in finding mistakes in designs early in the life-cycle, and it also reflects our belief that a developer passes through many wrong designs before converging on a good one.

The elements of the approach we propose are:

- *A relational model of the heap*. We show how a simple model of the heap can account for the fundamental notions of object identity and sharing. Unlike previous models we have proposed [5], and the models used in shape analyses (such as [23,19], this model represents dereferencing in two steps: with a relation that maps an object reference to an abstract mathematical value, and then a relation that projects the value to one or more object references. This allows frame conditions to be expressed more systematically and succinctly, and leads to a more natural specification style.
- *A declarative notation for specifying methods*. We show how, using this model, it is easy to construct succinct specifications of methods. Specifications are not only the yardstick against which code is measured, but are also proxies for missing code. Our notation, being a slight extension of first-order logic, is *declarative*, and expresses the behaviour of a method as a formula relating the pre- and post-states. Declarativeness is the lynchpin of partiality; it allows details of behaviour to be omitted, because they are not relevant to the analysis in hand, or because a program depends only on an interface of a component and not its particular implementation.
- *An extraction strategy*. To analyze some aspect of the program's behaviour, an abstract skeleton of the program is extracted from its code. This skeleton includes only those parts of the program relevant to the analysis, and represents methods by their specifications when possible. Aspects of behaviour that are not fully determined by the analysis may be left open using the declarative facilities of the specification notation. We have yet to investigate how this extraction might be done; a later section gives some initial thoughts about what kind of analysis might be required. For the purpose of this paper, we assume that an interaction diagram has been extracted, and we explain how such a diagram might be analyzed.
- *A constraint solving technique*. We use a technology developed previously [6] to analyze the skeleton extracted from the code. The idea, in short, is to reduce the analysis to finding models of a logical formula, so that any model found is a counterexample to the property being checked. A sequence of actions is represented as a conjunction of formulas for individual actions, with different variables explicitly representing the states between the actions. The formula is solved by translation

to a propositional boolean formula, and then by application of an off-the-shelf SAT solver.

In presenting our example, we use the new version of our Alloy notation [8], whose tool is under construction. To check the example, we translated the description to an older version of the notation [7] so that we could use our existing tool [9]. The key difference between the new and old versions is that the new one has no built-in notion of state machines, and is thus more flexible in its application. Indeed, our example here is precisely the kind that motivated the design of the new language.

### 3.1 A Relational Heap Model

In Section 1, we identified the importance of knowing how objects collaborate, and being able to reason about cases in which changes to one object affect the state of another. In the case of our tagging program, any reasoning about the system requires a representation that captures which iterators may be invalidated by comodification; this means we must know which lists exist, and how different actions may belong to different lists.

To accomplish this, we propose an explicit representation of the heap structure of the system (or rather, of the heap structure of relevant parts of the system). In traditional abstract-data-type reasoning, each object instance is represented in the formal model by an abstract value from a mathematical domain: for example, a list might be represented as a mathematical sequence (that is a function from a prefix of the natural numbers). The only change we make to this view is that the primitive values from which an abstract value is constructed become object references, and a global relation that maps references to their abstract values is included in the program state. Thus a set of sets of integers, which would be represented traditionally as a mathematical set of sets, becomes a set whose elements are object references, each being a reference to a set object, which itself contains references to integers. For immutable types, it may be desirable to remove the indirection, but the loss of uniformity seems undesirable (and not compatible with Java, whose equality operator distinguishes, for example, two string objects containing the same character sequence).

The Alloy notation [8] gives a direct and reasonably succinct expression to these notions, and is amenable to automated analysis [6]. We hope that the notation is simple enough that an informal explanation of the example will suffice. There is one essential, and unconventional, idea underlying Alloy: the reduction of all data structures to sets and relations. Every expression denotes a set or a relation (and in fact a set can be viewed as a degenerate relation). There are no constructors for tuples, sequences, or records. Instead, every structure is objectified, so that a tuple, for example, is represented as an atomic individual, along with some relations that act as projections so that the elements of the tuple can be obtained (much like Java itself). Likewise, the states of a program are regarded as atomic individuals. There is no scalar type; a single individual is represented as the singleton set containing it.

#### 3.1.1 Generic Heap Structure

The generic structure of the heap is expressed by the following declarations:

```
sig Ref {}
sig Object {}
sig State {
    refs: set Ref,
    obj: refs ->? Object
    }
static disj sig NullRef extends Ref {}
fact {
    all s: State, r: Ref | r = NullRef <=> no r.s::obj
    }
```

Each *signature* introduced by the keyword *sig* declares a set of atomic individuals. *Ref* is a set of references to objects; *Object* is a set of abstract object values; and *State* is a set of program states. These sets have global scope, and should be regarded as constants. The declaration of the signature *NullRef* indicates that it extends *Ref*; this means little more than that its elements are a subset of the elements of *Ref*. Signatures with no extends clause represent basic types, and are disjoint from one another; so no individual can be at once a reference and an object, for example. The keyword *static* indicates that the set *NullRef* is a singleton: there is only one null reference.

The *fields* of the *State* signature are relations with global scope. Thus *refs*, for example, is a relation from *State* to *Ref*, and is intended to associate with each state the references that are live in that state. Given an expression *s* denoting a state (that is, a singleton set containing one element of the set *State*), the expression *s.refs* (in which the dot is relational image) denotes this set of references. The relation *obj* is ternary; *s.obj* will denote a relation from references in *s.refs* to objects. The question mark is a 'multiplicity symbol' that makes each such relation a partial function. Thus *r*.(*s.obj*) will denote a set containing one or no objects when *r* denotes one reference. For convenience, we allow this expression to be written equivalently as *r.s::obj*, in which the double colon has the same meaning as the dot, but binds more tightly.

The *fact* is a formula constraining the sets (corresponding to the signatures) and the relations (corresponding to their fields). It states that for every state *s* and reference *r*, the expression *r.s::obj* denotes a set of no elements exactly when *r* is the null reference. In other words, we associate an object with every reference unless null.

It will be convenient to have a name for the empty set of references:

```
sig NoRef extends Ref {}
fact {no NoRef}
```

### 3.1.2 Abstract Specifications of Classes

Particular Java classes or interfaces are represented in a similar way, with one signature for references and one for objects. For example, we might represent collections like this:

```
disj sig CollectionRef extends Ref {}
disj sig Collection extends Object {
    elts: set Ref
    }
```

where the field *elts* associates a set of references with each collection, abstracting away any ordering, and we might represent iterators over such collections like this:

```
disj sig IteratorRef extends Ref {}
disj sig Iterator extends Object {
    on: CollectionRef,
    left: set Ref,
    last: Ref
    }
```

The fields of *Iterator* map an iterator object to, respectively: a reference to the collection object on which the iterator is based (*on*); the set of references that have yet to be yielded (*left*); and the last reference yielded (*last*).

Subsignatures of a common signature, unlike Java subclasses, are not implicitly disjoint. The keyword *disj* on several subsignatures of a common signature makes them mutually disjoint. Thus *Collection* and *Iterator* are disjoint, and so are *CollectionRef*, *IteratorRef* and *NullRef*. We are therefore assuming that no object is both a *Collection* and an *Iterator*, and that the *on* field of an iterator is non-null, although the elements of a *Collection* and the *last* element of an *Iterator* may be null references.

We record the type constraints of references, so that an *IteratorRef* is indeed a reference to an *Iterator*, and a *CollectionRef* is a reference to a *Collection*:

```
fact {
    all s: State | IteratorRef.obj in Iterator
    all s: State | CollectionRef.obj in Collection
    }
```

A specification may be introduced for any class in this manner, and used in the analysis in the place of its code. This allows the analysis to be conducted in terms that are familiar to the user, without cluttering the analysis with irrelevant implementation details. In our example, we would likely provide a specification for *HashMap*, with a signature such as

```
sig HashMap extends Object {
    val: (Ref - NullRef) ->! Ref
    }
```

where the field *val* is a total function that maps (non-null) references to (possibly null) references. We will treat the *LinkedList* class as a *Collection*, since our analysis does not rely on any special properties of lists over other collections.

### 3.1.3 Extracted Specifications of Classes

For classes whose code is to be reasoned about, it will be appropriate to use a signa-

ture obtained directly from the code. From the *Engine* class, for example, we might obtain:

```
sig Engine extends Object {
    actions: Typ ->! LinkedListRef
    }
```

or

```
sig Engine extends Object {
    actions: HashMapRef
    }
```

if we want to expose the hash map. For some classes, we may want to apply some rudimentary abstraction manually to the extracted signature. For example, as the object model of Figure 1 suggests, we might collapse all references an *Action* has to other *Action* objects into a field *friends*, and all references to *Engine* objects into a field *uses*.

### 3.2 Declarative Specification

Methods are specified as parameterized formulas. For each method, we give a precondition, a postcondition and frame condition. The *add* method of *Collection*, for example, might be specified as follows:

```
fun add-pre (s: State, this: CollectionRef, e: Ref) {
    this != NullRef
    }
fun add-post (pre, post: State, this: CollectionRef, e: Ref) {
    this.post::obj.elts = this.pre::obj.elts + e
    }
fun add-modifies (pre, post: State, this: CollectionRef, e: Ref) {
    modifies (pre, post, this)
    }
```

The precondition requires that the reference to the receiver object (*this*) be non-null. The postcondition says that the set of elements of the collection object in the post-state is the set in the pre-state with the addition of the reference *e*. (The plus denotes set union). The frame condition says that the receiver object is the only object modified; it uses a generic formula

```
fun modifies (pre, post: State, rs: set Ref) {
    all r: pre.refs - rs | r.pre::obj = r.post::obj
    }
```

that constrains a pre-state, post-state and a set of references *rs* so that the values of all references live in the pre-state except those in *rs* are unchanged. (The minus denotes set difference.)

We have separated the method specification into these parts anticipating that in practice a syntactic sugar would be used. Traditionally, the state and receiver argu-

ments are not declared explicitly, and each of the three parts is written as a clause of a single paragraph. A convenient shorthand would allow *e.pre*::*obj* to be written *e*^, and *e.post*::*obj* to be written *e*′. For a language such as Java, it would be appropriate to regard a declaration of type *Object* as if it were actually a declaration of type *Ref*. Our specification might then be written:

```
class Collection {
    …
    void add (Object e)
        pre: this != NullRef
        modifies: this
        post: this'.elts = this^.elts + e
    …
    }
```

The specification of the method as a whole is interpreted as an implication, stating that the post-condition and modifies clause are only required to hold when the pre-condition is satisfied:

```
fun add (pre, post: State, this: CollectionRef, e: Ref) {
    add-pre (pre, this, e) =>
        add-post (pre, post, this, e) && add-modifies (pre, post, this, e)
    }
```

We shall assume that all method specifications are assembled from their parts in this manner, and subsequently show only the formulas for the individual clauses.

### 3.2.1 Linking Objects

The *iterator* method of the collection returns an iterator object that is based on the collection, and its specification illustrates how a link between objects is established:

```
fun iterator-pre (s: State, this: CollectionRef) {
    this != NullRef
    }
fun iterator-post (pre, post: State, this: CollectionRef, result: IteratorRef) {
    result.post::obj.on = this
    result.post::obj.left = this.pre::obj.elts
    no result.post::obj.last
    fresh (pre, result)
    }
fun iterator-modifies (pre, post: State, this: CollectionRef) {
    modifies (pre, post, NoRef)
    }
```

The *fresh* formula, used in the postcondition, states that a set of references *rs* has no intersection with the references of the pre-state *s*:

```
fun fresh (s: State, rs: set Ref) {
    no rs & s.refs
    }
```

### 3.2.2 Non-determinism

The specification of the *next* method of the iterator illustrates the use of non-determinism to hide details – in this case the order in which elements are stored and yielded. Its precondition says that there are some elements left to yield; its postcondition says that some arbitrary element is yielded; and the frame condition says that only the iterator is modified:

```
fun next-pre (s: State, this: IteratorRef) {
    this != NullRef
    some this.pre::obj.left
    }
fun next-post (pre, post: State, this: IteratorRef, result: Ref) {
    this.post::obj.on = this.pre::obj.on
    this.post::obj.left = this.pre::obj.left - result
    this.post::obj.last = result
    }
fun next-modifies (pre, post: State, this: IteratorRef) {
    modifies (pre, post, this)
    }
```

### 3.2.3 Mutation at a Distance

The specification of the *remove* method of the iterator illustrates an interaction between objects, encapsulated by specification in a single method. Its precondition says that there is a last element; its postcondition says that the last element yielded is deleted from the underlying collection, and the *last* field is made null; and its frame condition allows only the iterator and the underlying collection to change:

```
fun remove-pre (s: State, this: IteratorRef) {
    this != NullRef
    this.pre::obj.last != NullRef
    }
fun remove-post (pre, post: State, this: IteratorRef) {
    this.post::obj.on = this.pre::obj.on
    this.post::obj.left = this.pre::obj.left
    this.pre::obj.on.post::obj.elts =this.pre::obj.on.pre::obj.elts - this.pre::obj.last
    this.post::obj.last = NullRef
    }
fun remove-modifies (pre, post: State, this: IteratorRef) {
    modifies (pre, post, this + this.pre::obj.on)
    }
```

The key constraint of the postcondition is a little daunting written out in full, but using the shorthand it would become

```
(this^.on)'.elts = (this^.on)^.elts - this^.last
```

Note how our notation allows us to mix dereferencing in the pre- and post-states;

expressions as a whole are not interpreted in one state or the other. Thus (*this*^.*on*)′.*elts* denotes the value of the *elts* field in the post-state of the object obtained by dereferencing the *on* field of *this* in the pre-state.

### 3.2.4 Comodification

Finally, we consider how to say that comodifications should not occur by extending our specifications. Using a trick taken from [19], we associate with each collection and iterator object an abstract version:

```
sig Version extends Object {}
sig VersionRef extends Ref {}
sig Collection extends Object {
    …
    cver: VersionRef
    }
sig Iterator extends Object  {
    …
    iver: VersionRef
    }
```

The *iterator* method gets a new constraint saying that the iterator's version matches the collection's (which the frame condition assures cannot have changed):

```
fun iterator-post (pre, post: State, this: CollectionRef, result: IteratorRef) {
    …
    result.post::obj.iver = this.post::obj.cver
    }
```

No alterations are made to mutators of the collection, such as *add*. The omission of a constraint will allow the collection's version to take on any value in the post-state.
The *remove* method includes a similar constraint as that of the *iterator* method:

```
fun remove-post (pre, post: State, this: IteratorRef) {
    …
    this.post::obj.iver = this.pre::obj.on.post::obj.cver
    }
```

but since both objects may be modified, this allows new versions to be obtained, so that after a *remove*, although this iterator will be consistent with its underlying collection, another iterator on the same collection will not be.
Finally, the precondition of the *next* method says that versions should match:

```
fun next-pre (s: State, this: IteratorRef) {
    …
    this.pre::obj.iver = this.pre::obj.on.pre::obj.cver
    }
```

We have chosen to specify a behaviour slightly different from that of Java iterators. While our iterators are *allowed* to fail when a comodification occurs, Java iterators

are *required* to. Our specification models exactly what happens when a key of a hashmap is mutated in such a way that the hashmap's invariant is undermined. In this case, it is impractical to demand that an exception be raised.

### 3.2.5 Subtypes and Subclasses

We have modelled Java's type hierarchy by associating a set of objects with each class or interface, and treating subclassing or extension simply as subset. This approach fails when objects of a subclass do not behave according to the superclass specification. The notion of equality is also problematic. In specifying the *get* method of *HashMap*, for example, we might write:

```
fun get-post (pre, post: State, this: HashMap, k: Ref, result: Ref) {
    result = k.(this.pre::obj.val)
    }
```

in which keys are matched by reference equality, but sometimes a more elaborate treatment of equality will be needed.

## 3.3 Extraction and Constraint Solving

Suppose we have extracted from the code the interaction diagram of Figure 2, representing the behaviour of the method *consume_token* as far as the second call to *next*. Although widely used, interaction diagrams have no standard meaning, and most attempts to provide semantics have focused on concurrency and not accounted for dynamic allocation and linking of objects. We are not able to give a formal semantics here, although our translation into Alloy is a starting point.

We shall treat the boxes that run along the top not as individual objects, but as sets of objects; thus *Engine* denotes the set of objects belonging to the *Engine* class. Each label on a method call explicitly identifies its receiver, and we shall assume a single global scope for the variables in the diagram. Method calls whose constituent steps are not shown – call these *basic calls* – will be modelled by their specifications; the remaining calls are merely transfers of control and can be ignored.

We now associate distinct states with each point on the vertical scale at which a horizontal arrow representing a basic call is present. We then construct a formula that constrains these states with the specifications of the calls, whose models will be the possible execution traces associated with the interaction diagram:

```
{   get (s0, s1, e.s0::obj.actions, t.s0::obj.typ, c)
    iterator (s1, s2, c, i)
    hasNext (s2, s3, i, True)
    next (s3, s4, i, a)
    get (s4, s5, e2.s4::obj.actions, typ2, c2)
    add (s5, s6, c2, a2)
    hasNext (s6, s7, i, True)
    next (s7, s8, i, a)
}
```

The *hasNext* method – the only one not specified above – has arguments for the pre-

and post-states, the iterator, and a boolean result. Because the *hasNext* tests are assumed both to have succeeded in this trace, the result argument is instantiated as *True*. A sequence of formulas enclosed in braces is a conjunction in Alloy; this formula represents an execution in which *get* is called in state *s0*, resulting in state *s1*, then iterator is called in state *s1* resulting in state *s2*, and so on. The arguments to the parameterized formulas are Alloy expressions using variables whose declarations will be given below.

### 3.3.1 Checking Comodification

To check the comodification constraint, we assert that in any such sequence, the precondition of the final call to *next* is met:

```
assert {
    all s0, s1, s2, s3, s4, s5, s6, s7, s8: State,
        e, e2: EngineRef, actions: HashMapRef,
        t: TokenRef, c, c2: CollectionRef,
        i: IteratorRef, a, a2: ActionRef, typ2: TypRef |
                {get (s0, s1, e.s0::obj.actions, t.s0::obj.typ, c)
                iterator (s1, s2, c, i)
                hasNext (s2, s3, i, True)
                next (s3, s4, i, a)
                get (s4, s5, e2.s4::obj.actions, typ2, c2)
                add (s5, s6, c2, a2)
                hasNext (s6, s7, i, True)
                next (s7, s8, i, a)
                } => next-pre (s7, i, a)
        }
```

An assertion in Alloy is checked by searching for a counterexample – that is, a model of the negation. In this case, the analyzer finds (in a few seconds) the expected scenario in which *c* and *c2* refer to the same collection. This can be displayed textually or graphically, with a snapshot for each state showing its heap structure. The textual output includes, for example:

```
c = Ref0
i = Ref1
c2 = Ref0
e = Ref5
e2 = Ref5
s5 = Sta5
s6 = Sta6
s7 = Sta6
…
State = {Sta1, …, Sta5, Sta6}
…
CollectionRef = {Ref0}
Collection = {Obj0, Obj1}
```

```
…
Sta5 = {obj: Ref0 -> Obj0, Ref1 -> Obj2, …}
Sta6 = {obj: Ref0 -> Obj1, Ref1 -> Obj2, …}
…
Obj0 = {cver: Ref3}
Obj1 = {cver: Ref4}
Obj2 = {iver: Ref3}
…
```

which shows first the bindings of values to variable names, then the contents of sets, then the values of fields. Both *Collection* variables *c* and *c2* have the same reference value *Ref0* – that is, they are aliased – as do both *Engine* objects. The value of the state variable *s5*, called *Sta5*, maps *Ref0* to a collection object *Obj0* whose version field has the value *Ref3*. But the state after the *add* method, *Sta6*, maps it to the object *Obj1* with a different version. The version of the iterator object *Obj2* associated with *Ref1* remains unchanged. The pre-state of the final call to *next*, *s7*, has the same value as the state just after the *add*, *Sta6*, and because of the mismatch of versions, violates the precondition. (We performed the analyses discussed here using a tool that handles a previous version of the Alloy language. The form of the output is different but the content is essentially the same.)

Alloy's analysis is incomplete. A formula may have a model (or counterexample) that the tool fails to find, since it only considers assignments in a 'scope' set by the user that bounds the size of the basic sets. Our hypothesis is that a high proportion of bugs can be found in a small scope.

### 3.3.2 Extending the Correctness Argument

We anticipate using the tool interactively. Having found a counterexample, we consider whether some invariant is needed to establish the property. In this case, it's clear that there may be a comodification unless we make some attempt to ensure that actions do not affect the collection to which they belong.

A reasonable discipline, which we followed in the construction of our tagging program, is to define for each action the types against which it is ever registered in any engine – call these its *registration types* – and then to require that no action share a registration type with a friend (that is, an action it registers or deregisters). This is easily checked statically, and seems to be liberal enough. We can express this discipline as a fact:

```
fact {
 all a: Action | no a.regtypes & a.friends.regtypes
 }
```

having previously added *regtypes* as a field of *Action*

```
sig Action {
   …
   regtypes: set TypRef
   }
```

along with a fact defining it:

```
fact {
  all a: Action | a.regtypes =
     {t: TypRef | some e: Engine, s, s': State, c: Collection |
          get (s, s', e.actions, t, c) && a in c.elts.s::obj}
 }
```

Running the assertion again (with the new facts now implicitly conjoined to the negated assertion), a counterexample is still found. Studying the output shows a single collection of actions registered against two distinct types. This suggests that *Engine* needs an invariant requiring that the mapping be injective:

```
fact {
     all s, s': State, e: Engine, t1, t2: TypRef, c: CollectionRef |
          get (s, s', e.actions, t1, c) && get (s, s', e.actions, t2, c) => t1 = t2
     }
```

This invariant can be established locally by checking the code of *Engine*, with each method checked by an analysis similar to our analysis of consume_token.

It is a strength of our approach that it can accommodate a programming discipline in the correctness argument. This decouples code and design analysis, so that it may not be necessary always to repeat the design analysis when the code is changed, so long as it still conforms to the discipline. The discipline embodies the essence of the design, and its articulation alone is likely to be valuable. A different discipline that suffices to establish the required property might be liberal enough for the program at hand, but not for anticipated extensions, or for other programs in a family using the same framework. We might, for example, have insisted that an action only register actions in a different engine, but that would be too restrictive.

### 3.3.3 Issues

The construction and analysis of the interaction diagram that we have sketched in outline presents a number of research challenges that will need to be addressed:

- *Extraction*. Given a property of interest – in this case the absence of comodification within *Engine* – an interaction diagram, or a similar kind of behavioural skeleton, is extracted from the code by static analysis. Statements not relevant to the property at hand are eliminated, and possible statement sequences across method calls are collected. The analysis should be simpler than slicing, since it will be acceptable to insert non-deterministic statements (eg, an assignment of an unknown value to a variable): the skeleton that is extracted should include all possible behaviours, but may admit some infeasible ones. How approximate the analysis may be will be determined not by the incidence of spurious counterexamples, but by their quality. It may be reasonable, for example, to ignore control dependences from extraneous data types. We expect that lightweight techniques that have been developed for object model extraction [22] will come in handy, as well as more generic and powerful type-based techniques [18].

- *Namespace*. It may be convenient to flatten the namespace as we did above, using something like SSA form, so that there is no need to express or analyze argument passing.
- *Loops and conditionals*. We considered only a single sequence of statements. Conditional statements might be represented explicitly in the interaction diagram, and then encoded using disjunction for analysis, or it may be better to generate separate diagrams for each path. Loops might also be represented explicitly, so that a loop invariant can be asserted. Alternatively, a loop can be unwound, and an analysis performed up to some finite number of iterations [10]. This will of course entail a further loss of soundness in the analysis.
- *Simplification*. The formula obtained from the diagram may be simplified before analysis. Post-state arguments may be eliminated, for example, from calls to methods that leave the state unchanged. But it is not clear how effective such simplifications would be.

## 4 Related Work

This paper builds on ideas the first author and his students have been developing in the last few years, most notably:

- A simple and flexible notation capable of expressing key properties of software designs in a declarative manner [7,8];
- An analysis whose automation makes it like a model checker, but whose ability to handle relational structure, and to handle arbitrary formulas, makes it more like a theorem prover [6];
- An encoding of program statements as logical formulas, allowing the analysis to be used to find bugs in code [10]; and
- A simple relational view of object-oriented programs [5].

The new ideas in this paper are:

- An improved relational model of the heap, which simplifies the interpretation of frame conditions, and corresponds closely to a style of specification that has been widely taught [16];
- Performing analysis over a design representation such as an interaction diagram that corresponds to code but deals at the level of abstraction at which a designer views a program;
- A simple treatment of sequential composition, in which intermediate states are made explicit and objectified; and
- Making explicit a discipline that decouples code and design analysis, whose accommodation depends on the ability of the analysis to handle declarative specifications.

### 4.1 Object Interaction

'Object-oriented programming' has increasingly come to refer to a style of programming in which interactions between abstract objects is fundamental, and a major focus of the design. Most design patterns [2] involve object interactions, and

many popular frameworks, especially for graphical user interfaces, rely on them heavily. Overuse of object interaction is likely, of course, to make a program incomprehensible, but used carefully, it offers flexibility and a cleaner separation of concerns. In our tagging program, for example, the design allows all the code associated with mathematical mode to be localized. In a more traditional design, it would have been necessary to distribute flags throughout the program to achieve the different behaviour required when within that mode.

The paper that first identified the need for linguistic support for describing object interactions [3] used a largely operational notation, and focused on describing method call sequences. Its focus was primarily on documenting the calls an object is expected to make in order to partipate correctly in an interaction, rather than on analyzing its global effects. Several of the notations of UML [20] are intended for describing object interaction, but they treat objects as processes communicating in a fixed topology, and do not account for allocation or reconfiguration.

### 4.2 Interface Specification Languages

Recently, Leavens and his colleagues have developed a specification language for Java called JML [13]. Like Eiffel [17], it uses expressions of the programming language as terms in specification formulas. This allows pre- and postconditions to be interpreted as runtime assertions; the form *\old(e)* is used in a postcondition to refer to the value of *e* in the prestate. The underlying model of the heap does not involve a notion of object values, but, like [5], treats fields as direct mappings from objects to objects.

The heap model of this paper uses explicit 'state functions' to map object references to object values, as in Larch-C++ [12]. We have chosen this approach for three reasons. First, it gives a simpler semantics to modifies clauses; with fields as relations between objects, the clause *modifies f* for a field *f* is an assertion about all fields except *f*, but a quantification over field names is not expressible in first-order logic. Second, the explicitness of state functions makes it easy to express and give meaning to sequences of method calls (and is also more expressive and more uniform than *old*). Third, the resulting specifications are compatible with a widely taught viewpoint and specification style [16] in which objects rather than locations are viewed as mutable.

ESC/Java [15] is a tool that checks code for basic properties such as lack of null dereferences and array bounds overflow. Its annotation language is a subset of JML. Unlike our approach, it aims at a uniform level of analysis across a large program, rather than an analysis focused on some aspect of object interaction. Modularity of reasoning has therefore been a central concern, and the approach takes advantage of a technique developed by Leino [14] for interpreting modifies clauses correctly in the presence of hidden components.

### 4.3 Model Extraction

A number of projects are focusing on verifying code by applying an analysis to an abstract model extracted from the code. The Bandera project [24] aims to extract finite-state machines that can be presented to off-the-shelf model checkers. The

SLAM project [25] translates code into 'boolean programs' that are amenable to analysis with a specialized model checker. Both projects are more concerned with event sequencing issues than object configurations and interactions.

### 4.4 Static Analysis

A recent paper [19] shows the application of Sagiv et al's shape analysis framework [21] to the comodification problem. Our use of abstract version objects was taken from that paper. Unlike ours, the analysis is conservative, although, being operational rather than declarative, it requires consideration of the entire program. Our approach allows us to omit details of the behaviour of *Action* objects, for example, and admit any kind of registration behaviour satisfying some discipline. In their approach, however, heap shapes are only considered when code is present to construct them. This work is part of the Canvas Project at IBM Research and Tel-Aviv University, whose aim is to provide a conformance check of client code against criteria specified by a component designer, and which has also investigated protocol analyses based on state machines.

Our work is also part of a larger project to find ways to check conformance of code to a high-level design. As part of this effort, Martin Rinard and his students have recently developed a static analysis that distinguishes objects according to their relationships with other objects. We hope to exploit this analysis in our model extraction.

## Acknowledgments

This work benefited from discussions early on with Michael Ernst, Nancy Lynch and Butler Lampson on how to model sharing in the heap. It was funded in part by ITR grant #0086154 from the National Science Foundation, by a grant from NASA, and by an endowment from Doug and Pat Ross.

The first author dedicates this rather imperfect paper to the fond memory of his father-in-law, Professor Joseph Marbach z″l, a prolific researcher, paper-writer and perfectionist, who died during its preparation, many years before his time.

## References

[1] M. Fayad and D. Schmid, Object-Oriented Application Frameworks, Special Issue, *Communications of the ACM*, 40(10), October 1997.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison Wesley, 1995

[3] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object Oriented Systems. *European Conference on Object-Oriented Programming / ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Ottawa, Canada, Vol. 1, June 1990, 169–180.

[4] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–81, 1972.

[5] Daniel Jackson. Object Models as Heap Invariants. In: Carroll Morgan and Annabelle McIver (eds.), *Essays on Programming Methodology*, Springer Verlag, 2000.

[6] Daniel Jackson. Automating first-order relational logic. *ACM SIGSOFT Conference on Foundations of Software Engineering*, San Diego, California, November 2000.

[7] Daniel Jackson. Alloy: A Lightweight Object Modelling Notation. To appear, *ACM Transactions on Software Engineering and Methodology*, October 2001.

[8] Daniel Jackson, Ilya Shlyakhter and Manu Sridharan. A Micromodularity Mechanism. *ACM SIGSOFT Conference on Foundations of Software Engineering / European Software Engineering Conference*, Vienna, Austria, September 2001.

[9] Daniel Jackson, Ian Schechter and Ilya Shlyakhter. Alcoa: the Alloy Constraint Analyzer. *International Conference on Software Engineering*, Limerick, Ireland, June 2000.

[10] Daniel Jackson & Mandana Vaziri. Finding Bugs with a Constraint Solver. *International Symposium on Software Testing and Analysis*, Portland, Oregon, August 2000.

[11] Daniel Jackson and Jeannette Wing. Lightweight Formal Methods. In: H. Saiedian (ed.), *An Invitation to Formal Methods*, IEEE Computer, 29(4):16–30, April 1996.

[12] Gary T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In: H. Kilov and W. Harvey (eds.), *Specification of Behavioral Semantics in ObjectOriented Information Modeling*, Amsterdam, Holland, 1996, Kluwer, pp.121–142.

[13] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Notation for Detailed Design. In: Haim Kilov, Bernhard Rumpe, and Ian Simmonds (eds.), *Behavioral Specifications of Businesses and Systems*, Kluwer, 1999, Chapter 12, pp. 175–188.

[14] K. Rustan M. Leino. *Toward Reliable Modular Programs*. Ph.D. thesis, California Institute of Technology, January 1995.

[15] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. *ESC/Java User's Manual*. Technical Note 2000-002, Compaq Systems Research Center, October 2000.

[16] Barbara Liskov with John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2001.

[17] Bertrand Meyer. *Object-Oriented Software Construction* (2nd ed.). Prentice Hall, 1997.

[18] Robert O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis. Technical Report CMU-CS-01-124, School of Computer Science, Carnegie mellon University, Pittsburgh, PA, May 2001.

[19] G.Ramalingam, Alex Warshavsky, John Field and Mooly Sagiv. *Toward Effective Component Verification: Deriving an Efficient Program Analysis from a High-level Specification*. Unpublished manuscript.

[20] James Rumbaugh, Ivar Jacobson and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[21] M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, Jan. 20–22, 1999, ACM, New York, NY, 1999.

[22] Allison Waingold. *Lightweight Extraction of Object Models from Bytecode*. Masters thesis, Department of Electrical Engineering and Computer Science, MIT, May 2001.

[23] John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs. *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, November 1999.

[24] The Bandera Project. Kansas State University. http://www.cis.ksu.edu/santos/bandera/.

[25] The SLAM Project. Microsoft Research. http://www.research.microsoft.com/projects/slam/.