Lightweight Object Specification with Typestates

Kevin Bierhoff
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
kevin.bierhoff @ cs.cmu.edu

Jonathan Aldrich
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
jonathan.aldrich @ cs.cmu.edu

ABSTRACT

Previous work has proven typestates to be useful for modeling protocols in object-oriented languages. We build on this work by addressing substitutability of subtypes as well as improving precision and conciseness of specifications. We propose a specification technique for objects based on abstract states that incorporates state refinement, method refinement, and orthogonal state dimensions. Union and intersection types form the underlying semantics of method specifications. The approach guarantees substitutability and behavioral subtyping. We designed a dynamic analysis to check existing object-oriented software for protocol conformance and validated our approach by specifying two standard Java libraries. We provide preliminary evidence for the usefulness of our approach.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.2 [**Software Engineering**]: Design Tools and Techniques; F.3.1 [**Theory of Computation**]: Specifying and Verifying and Reasoning about Programs

General Terms

Design, Verification, Languages, Reliability

Keywords

Typestate, refinement, substitutability, behavioral subtyping, union and intersection types

1. INTRODUCTION

In object-oriented software systems, objects routinely store data. The data can only be accessed and modified through the methods defined in the objects' interfaces. This is the essence of information hiding [19]. The engineering of object-oriented software includes specifying these interfaces, implementing them with classes, and verifying design and implementation with formal methods and testing. Thus improving interface specifications can greatly facilitate all these engineering tasks.

An interface specification defines a *type* of object. At the very least, an interface lists available methods. But frequently the specification also defines a *protocol* of allowed method call sequences. For instance, an input stream would require that it can be read only before it is closed. A method can be specified even more precisely with a *contract* between the interface and its clients. A contract describes the "duties" of both parties involved in a method call.

In particular in the presence of a protocol, implementation invariants are frequently disjunctions of predicates. Which predicate is true at a given point in the object's lifetime is often hard to tell. This can for example result in passing the contents of a field that is currently *null* to a method, which might accidentally violate that method's contract.

Mainstream object-oriented languages offer only informal documentation to capture protocols, contracts, and invariants. Eiffel [18] was the first language to support *pre- and post-conditions* for formalizing contracts. Contracts are arbitrary boolean expressions that Eiffel checks at run time. Careless pre- and post-conditions can cause side-effects or directly refer to fields in the implementation and thus violate information hiding. Pre- and post-conditions can formalize an individual method's contract very precisely. However, they capture protocols only implicitly. Therefore very precise conditions can obfuscate the underlying protocol rather than making it clear. They also impose high specification overhead.

Typestates [22] are an alternative to pre- and post-conditions. Typestates were proposed to capture the intuition that the set of operations that can be performed on an object frequently not only depends on its fixed type but also its changing *state*. For example, the input stream from above can be open or closed (figure 1). Although contracts based on typestates are less precise than pre-/post-conditions, they capture protocols in a more lightweight and direct way (figure 3).

The only existing object-oriented typestate system, Fugue [7], demonstrated the usefulness of typestates for object-oriented programming. We propose to employ typestates more broadly for a lightweight technique of specifying object behavior based on abstract states. We build on Fugue's approach and address challenges of object-oriented software related to subtyping and inheritance with the following contributions.

- In previous work, substitutability of subtypes for supertypes can be violated. We propose hierarchical *state refinement* into substates to resolve this problem. State refinement allows a subtype to define more fine-grained states. For instance, a buffered stream can refine *open* to indicate that its buffer can be *filled* or *empty*.
- Objects frequently change state along multiple conceptual "axes". Distinguishing all state combinations can lead to state explosion problems. We avoid state explosion using *state dimensions* which build on AND-states from Statecharts [15]. State dimensions also help capture multiple "roles" an object plays in a system.
- In previous work, once a method is defined, subclasses cannot change its specification. We introduce *method refinement* in subtypes. Refined methods can accept more inputs or yield more specific results. Methods can also be specified more precisely based on more fine-grained state refinements.

Thus our approach not only addresses technical challenges but also captures important semantic extensions to existing work. The approach guarantees behavioral subtyping [16] even with method refinement. Code expecting a supertype will therefore always work with an object of a subtype. This eliminates common hierarchy violations in existing pre- and post-condition systems [11].

Behavioral subtyping results from using union and intersection types [9] in method specifications. They can express non-determinism and case-by-case behavior, increasing expressiveness and precision over specifications in Fugue. An alternative approach by Butkevich et al. uses labelled transition systems to specify protocols with non-determinism that can be checked statically for hierarchy violations [2].

Sound static checking of conformance between specification and implementation would require aliasing restrictions as in Fugue [7]. However, our goal is to explore the benefits of typestate specifications without any aliasing restrictions, even on objects that change state. To this end we designed and implemented a dynamic analysis to check Java programs. It can be applied to programs which Fugue would rule out. It also is a testbed to explore the usefulness of more powerful specifications than we could currently statically check. It will therefore guide future efforts in developing static conformance checking for typestate specifications.

Our analysis is similar to traditional pre- and post-condition systems [18, 10] in that it flags specification violations at run time. However, it provides better information hiding than most of these systems by strictly separating abstract typestates and underlying implementation invariants.

We performed case studies on the usefulness of our approach. In this paper we report on our experiences with specifying parts of the standard Java I/O and SQL libraries. We could capture interesting properties from informal documentation with moderately low overhead. We provide preliminary evidence that the novel properties of our system are useful in practice.

The remainder of this paper is organized as follows. Section 2 introduces our typestate specification approach. A more formal description follows in section 3. Section 4 presents our dynamic typestate analysis. We report on case studies in section 5. Section 6 summarizes related work and section 7 concludes.

```
interface InputStream extends Object {
    [open, closed] refines alive;

    void close();
    boolean isClosed();
    int read();
}

interface BufferedInputStream extends InputStream {
    [empty, filled] refines open;
}
```

Figure 1: State refinement in the stream example

2. INTERFACE SPECIFICATION

The fundamental idea of our approach is to model object behavior using abstract states that can change over the lifetime of the object. The approach augments standard interface specifications as in Java with state-based information. The following subsections introduce state refinement, method specifications, and state dimensions. Finally, we discuss the expressiveness of our approach.

2.1 State Refinement

Subtypes are usually required to be substitutable for a supertype. That guarantees that subtype objects can *always* be treated as objects of the supertype. Fugue [7] violates this principle because subtypes can define new states that exist alongside the ones defined in the supertype. For instance, an *InputStream* can be "open" or "closed". A subtype like *BufferedInputStream* defines an additional state for the case where its buffer is "empty". Consequently, a *BufferedInputStream* can end up in a state unknown to its supertype. In this situation Fugue *has to prohibit* any implicit or explicit upcasting of the object into the supertype. This has the effect that a *BufferedInputStream* is only sometimes an *InputStream*. Substitutability is violated.

On the other hand, it seems necessary to allow subtypes to define their own states. The only way to allow this and preserve substitutability is through *state refinement* which defines a set of states $s_1, \ldots, s_n$ as special cases of an existing state s. In a specification we write:

$$[s_1, \ldots, s_n] \text{ refines } s$$

The s_i are *substates* of s, written $s_i \leq s$, and whenever an object is in a state s_i, it is also in s. For the moment we assume that s has not been refined yet. We define a unique root state *alive* in the root type *Object* from which the refinements build a tree of substates. Figure 1 shows some state refinements for our streams.

2.2 Method Specification and Refinement

A classic approach to specifying methods is to define pre- and post-conditions directly in terms of implementation variables [18]. Specification languages for objects such as JML [3] can capture pre- and post-conditions more abstractly with specification variables which are independent from a concrete implementation. This better preserves information hiding. Nonetheless, it can be quite tedious to write specifications in this way. And in both approaches the actual *protocol* that needs to be enforced is implicit.

$$
\begin{array}{rcl}
M & ::= & C \mid M \wedge M \\
C & ::= & P \rightarrow U \\
U & ::= & P \mid U \vee U \\
P & ::= & (T, \ldots, T) \\
T & ::= & S \mid T \wedge T \mid T \vee T \\
S & ::= & s \mid n \mid op\ n \mid true \mid false \mid null \\
s & ::= & statename \\
n & ::= & integer \\
op & ::= & == \mid != \mid < \mid <= \mid > \mid >
\end{array}
$$

Figure 2: Method specification language

Fugue [7] took the approach of specifying methods with a single state transition of the receiver. It also allows defining a state requirement for method arguments. However, arguments cannot change state, and state transitions have to be deterministic. We overcome these limitations by adapting type refinements with union and intersection types [9] to our setting. The basic notation is as follows:

- We refer to individual states by their name.
- We annotate a method with $A \rightarrow A'$ to express that the method requires the *source state* A and produces the *destination state* A'. Thus, it implements a *state transition* $A \rightarrow A'$. For instance, the method *close* will translate a stream from "open" to "closed".
- *Intersections* can relate source and destination states on a case-by-case basis. A method annotated with $A \rightarrow A' \wedge B \rightarrow B'$ will translate A to A' and B to B'. If both A and B are satisfied, both cases apply. For example, *isClosed* returns *true* when the stream is closed and *false* when it is open.
- *Unions* express non-determinism with $A \vee B$. For instance, a call to *read* can return a character *or* -1 to signal the end of the file.
- Finally, we use a *product* notation $(T_1, \ldots, T_n)$ to group the states of receiver, arguments, and method result together.

Unions of the form $A \rightarrow A' \vee B \rightarrow B'$ are primarily useful in a setting with first-class functions. Since most object-oriented systems don't have them, we exclude unions of state transitions and state transitions within products. Intersections of state transitions essentially list the different *cases* of behavior for a method. The developer can omit intersections between cases in our concrete syntax.

Figure 2 gives the exact syntax we use for specifying methods.[1] A method annotation is an intersection of cases. Each case has an arrow as its outermost operator. The arrow's domain is a product defining the source states of receiver and arguments (in their order in the signature). The range can be a union of products that will have one more element for the state of the method result. Each element of a product can be an arbitrary combination of basic states defined

[1]In the tradition of Davies' intersection types for ML [4] we write "&" for intersection and "|" for union in actual state annotations. We use $\wedge$ and $\vee$ in formal definitions to avoid confusion with the symbol | for separating grammar options.

```
interface InputStream extends Object {
    [open, closed] refines alive;

    (open) -> (closed)
    void close();

    (open) -> (open, false)
    (closed)->(closed, true)
    boolean isClosed();

    (open) -> (open, -1 | (>= 0 & <= 255))
    int read();
}

interface BufferedInputStream extends InputStream {
    [empty, filled] refines open;

    (filled) -> (open, >= 0 & <= 255)
    int read();
}
```

Figure 3: Method specifications for stream example

for the respective object (receiver, argument, or result) using unions and intersections. We introduce special states for integer values (n), integer ranges ($op\ n$) and boolean constants. For arguments and method results we allow the "pseudo-state" *null.* A "real" state (*alive* or any substate) implies that the reference is valid, i.e. not *null.*

Figure 3 illustrates the specification of our streams (we write each case in a new line). Maybe the most interesting method in this example is *read.* We *refine* its specification in *BufferedInputStream* to show that a read will always yield a character if the buffer is filled. This is consistent with the specification in *InputStream* but guarantees more precise behavior for a particular situation. In this case the possibility to return -1 is ruled out. The next section will include an example that illustrates the benefit of unions between products.

Fugue does not allow methods to be respecified in a subtype. This rules out two important ways of changing method behavior in a subtype: extending it to cover more cases (effectively relaxing its precondition) or requiring it to produce a more specific outcome (tightening its postcondition) [16].

Fortunately, our notion of state refinement enables us to allow both. A method can be annotated in a subtype again. This *refinement* does not replace but *intersect with* the specification for the method that is inherited from the supertype. As we will see in section 3, this does not only produce a sensible specification, but it also suffices to guarantee behavioral subtyping.

In summary, our technique gives both theoretical and practical leverage over existing work. We can express arbitrary state transitions with non-determinism for receivers and arguments. We can also refine specifications in subtypes.

2.3 Orthogonal States

So far, our method has a state explosion problem. As an example, consider the definition of a result set in the standard Java SQL library (figure 4). A result set iterates over the result of a database query row by row. Initially,

```
interface ResultSet extends Object {
    default = [open, closed] refines alive;
    position = [start, row, end] refines open;
    direction = [forward, reverse] refines open;

    (start | row) -> (row, true) | (end, false)
    boolean next();

    (open) -> (closed)
    void close();

    (open, 1000) -> (forward, 1000)
    (open, 1001) -> (reverse, 1001)
    void setFetchDirection(int direction);
}
```

Figure 4: State dimensions in a SQL result set

no row is selected. *next* moves to the next available row. It returns *false* when no more rows are available. We may identify three distinct states: "start", "row", and "end".

At any given moment the client can change the *fetch direction*.[2] Now our three states from above build a cross product with the possible fetch directions: all six combinations make sense. Obviously, it quickly becomes infeasible to enumerate all combinations — not impossible, but tedious. Moreover, the two *dimensions* of the result set's behavior are largely unrelated. Thus a compound state like "rowForward" makes little sense.

To resolve this we allow any state to be refined at any time. Refining an already-refined state defines a new *state dimension* that is orthogonal to the state's existing refinements. Thus whenever a result set is open (figure 4), it will be in one of the "position" and one of the "direction" states. (For practical considerations each dimension is named.)

As an additional benefit, state dimensions enable us to naturally cover multiple interface inheritance. The states defined in each inherited interface simply define orthogonal dimensions — unless they were introduced in a common super-interface. ("alive" therefore remains the unique root state.)

2.4 Expressiveness

State and method refinement make our specification language strictly more expressive than Fugue's. We can encode any state space definable in Fugue with exactly one state refinement per type that includes a pseudo-state to represent any additional states that are defined in subtypes. Fugue's restriction essentially means that subtypes have to refine this pseudo-state. Our technique not only allows arbitrary state refinements but also includes state dimensions for conciseness.

Union and intersection types increase the expressiveness of our method specifications in various ways as discussed in section 2.2. A Fugue method specification can be encoded with one method case without unions or intersections that is not refined in subclasses. Unions, intersections, and refinements let the developer express her intention more precisely.

We want to highlight here that our specifications can express non-determinism, both explicitly with unions as in *next* and implicitly with imprecise states as in *setFetchDirection* (figure 4). In the latter example, "reverse" certainly implies "open", but nothing more. This is the desired semantics. In fact, the method will sometimes translate the result set from "end" to "start" or vice versa. The example of *next* shows how the state of the returned boolean can be used to determine the typestate of the *ResultSet*. This is the reason we use the product notation in method specifications rather than give a separate state transition for each argument and receiver.

Specifications often have to rely or choose to rely on non-determinism. Subtypes can add precision as demonstrated for the *BufferedInputStream* to complement the more fine-grained states a subtype can have. Thus our technique provides incremental benefit: the more detail is put into a specification, the more precise it gets. The developer can choose the appropriate level of abstraction for her specification purposes.

[2]The integer values are constants defined in the library.

$$\begin{array}{rcl} \sigma & ::= & \emptyset \mid \{s = (d_1, \ldots, d_k)\} \cup \sigma \\ d & ::= & [s_1, \ldots, s_n] \\ s & ::= & \mathit{statename} \end{array}$$

Figure 5: Grammar to define state spaces

3. FORMAL PROPERTIES

This section first formally defines state refinement and proves substitutability. Then we introduce the union/intersection type system we use for method specifications. We finally prove behavioral subtyping. The section relies on type-theoretic foundations and notations (Pierce gives a nice introduction in [20]).

3.1 Defining State Refinement

We assume a nominal type system for interfaces, meaning that a name uniquely identifies an interface. Each interface C in this paper defines a *state space* written $\sigma(C)$. Figure 5 defines a grammar for state spaces.

A state space is a set of state definitions, exactly one for each state in the state space. A state is defined with a *product* containing zero or more refinement dimensions. (The empty product is called *unit* and marks leaf states.) Each dimension is defined as a *variant* (written with square brackets). It lists the set of refining states in a dimension.

Definition 1 (State Spaces). *σ is a state space if it can be derived with the following rules.*

$$\frac{}{\{\rho = ()\}\ \mathit{space}}\ \text{P-Root}$$

$$\frac{\sigma\ \mathit{space} \quad s = (d_1, \ldots, d_k) \in \sigma \quad d = [s_1, \ldots, s_n] \quad s_1, \ldots, s_n \notin \mathit{dom}(\sigma)}{\sigma \otimes \{s = (d_1, \ldots, d_k, d),\ s_1 = (), \ldots, s_n = ()\}\ \mathit{space}}\ \text{P-Refine}$$

The judgment σ *space* defines valid state spaces. First, the unrefined root state ρ is a valid state space. Second, if we have a valid state space that contains a definition for a state s then we can refine s along a new dimension d into $[s_1, \ldots, s_n]$ if these state names are not used in σ yet. The

$$\frac{\Gamma \vdash e \in A \quad \Gamma \vdash e \in B}{\Gamma \vdash e \in A \wedge B} \wedge I \qquad \frac{\Gamma \vdash e \in A \wedge B}{\Gamma \vdash e \in A} \wedge E_L \qquad \frac{\Gamma \vdash e \in A \wedge B}{\Gamma \vdash e \in B} \wedge E_R$$

$$\frac{\Gamma \vdash e \in T \quad T \leq S}{\Gamma \vdash e \in S} \text{T-SUB} \qquad \frac{\Gamma \vdash e_1 \in A \rightarrow B \quad \Gamma \vdash e_2 \in A}{\Gamma \vdash e_1\ e_2 \in B} \text{T-APPLY} \qquad \frac{\Gamma \vdash e_i \in A_i}{\Gamma \vdash (e_1, \ldots, e_l) \in A_1 \times \ldots \times A_l} \text{T-PROD}$$

$$\frac{A \leq B_1 \quad A \leq B_2}{A \leq B_1 \wedge B_2} \wedge R \qquad \frac{A_1 \leq B}{A_1 \wedge A_2 \leq B} \wedge L_1 \qquad \frac{A_2 \leq B}{A_1 \wedge A_2 \leq B} \wedge L_2$$

$$\frac{A_1 \leq B \quad A_2 \leq B}{A_1 \vee A_2 \leq B} \vee L \qquad \frac{A \leq B_1}{A \leq B_1 \vee B_2} \vee R_1 \qquad \frac{A \leq B_2}{A \leq B_1 \vee B_2} \vee R_2$$

$$\frac{B_1 \leq A_1 \quad A_2 \leq B_2}{A_1 \rightarrow A_2 \leq B_1 \rightarrow B_2} \text{S-ARROW} \qquad \frac{A_1 \leq B_1 \ldots A_k \leq B_k \quad (0 \leq k \leq l)}{A_1 \times \ldots \times A_l \leq B_1 \times \ldots \times A_k} \text{S-PROD}$$

Figure 6: Typing and substate judgments for method specifications with unions and intersections

notation $\sigma \otimes S$ is borrowed from Z [1] and leaves σ untouched except that definitions in S override competing ones in σ.

The state space of a type C is derived by applying P-REFINE zero or more times to the state space $\sigma(B)$ of C's direct supertype B. The root object's state space is $\sigma(Object) = \{\rho = ()\}$.

Next we define the substate relation between individual states already introduced informally in section 2.1. We write a single state refinement with rule P-REFINE as $\sigma' \ \overrightarrow{s \mapsto s_1, \ldots, s_n}\ \sigma$.

DEFINITION 2 (SUBSTATES). *Let σ be a state space such that σ space and $s, t, u \in dom(\sigma)$. The substate relation $s \leq_\sigma t$ between states is defined by the following rules*

$$\frac{\sigma' \ \overrightarrow{s \mapsto s_1, \ldots, s_n}\ \sigma}{s_i \leq_\sigma s \quad (i \in 1, \ldots, n)}$$

$$\frac{}{s \leq_\sigma s} \qquad \frac{s \leq_\sigma u \quad u \leq_\sigma t}{s \leq_\sigma t}$$

As alluded to a number of times, a major consequence of our approach is that it preserves substitutability of subtypes. This is in contrast to previous work.

PROPERTY 1 (SUBSTITUTABILITY). *If C is a subtype of B then*

$$\forall s \in \sigma(C).\ \exists s' \in \sigma(B).\ s \leq_{\sigma(C)} s'$$

PROOF. *Immediate from the fact that C's state space is a refinement of B's.* $\square$

3.2 Specifying Methods

As mentioned in section 2, we base our method specifications on refinement types, in particular on *datasort refinements* with union and intersection types [9]. Refinement types add information to an underlying conventional type system in order to increase its precision. Datasort refinements build on uninterpreted atomic refinements of the underlying types. Refinements are assumed to have a partial ordering. Atomic refinements and partial order are given by the states defined in a given state space σ and its substate relation $\leq_\sigma$. We will write $\leq$ where it is unambiguous.

Refinements refine expressions. We write $e \in A$ to indicate that A refines expression e or, equivalently, that e typechecks with refinement A. From atomic refinements we can build unions and intersections. In particular, an intersection refinement $e \in A \wedge B$ means $e \in A$ *and* $e \in B$.

Figure 6 defines typing rules and the substate relation for our setting. Our system follows [9]. The only exception is that we can type arbitrary expressions with intersections. We need this to type function applications more precisely. We can omit typing rules for unions because we do not support them between arrows. We do need to cover both unions and intersections in the substate relation. Finally note that nested products and arrows or arrows within products do not apply to our setting.

Given the method specification and the states of concrete receiver and argument objects, we want to compute the objects' destination states. Unfortunately the tridirectional typechecking system in [9] is non-deterministic and requires partial typing annotations. However, for the simplified situation of our dynamic analysis (cf. section 4), we developed a deterministic algorithm to type function applications (figure 7, starting from rule T-APPLY).

The analysis always "knows" what states each object is in. It builds intersections of these states for all objects involved in a method call and invokes the algorithm to compute their expected destination states. With this simplification we can prove soundness and completeness of our algorithm with respect to figure 6. The theorem implies that we always generate the most precise possible type.

THEOREM 1 (SOUNDNESS AND COMPLETENESS). *For a given method body m and argument $e = (r, a_1, \ldots, a_n)$, where r is the receiver and a_i are the method arguments, let $m \in A$ be m's specification following the restrictions in figure 2 and $e \in B$ a valid refinement for e containing no unions. Then*
(1) If $A \cdot B = C$ then $\Gamma \vdash m\ e \in C$ according to figure 6.
(2) If C' is a sort refinement such that $\Gamma \vdash m\ e \in C'$ according to figure 6, then $A \cdot B = C$ such that $C \leq C'$.

PROOF. *(1) By induction on the derivation of $A \cdot B = C$.*
(2) By induction on the derivation of $\Gamma \vdash m\ e \in C'$. $\square$

$$\frac{\Gamma \vdash e_1 \in A \quad \Gamma \vdash e_2 \in B \quad A \cdot B = C}{\Gamma \vdash e_1\ e_2 \in C} \text{ T-APPLY}$$

$$\frac{C = \bigwedge_i \{C_i \mid A_i \cdot B = C_i\} \quad (\exists i.\ A_i \cdot B = C_i)}{(\bigwedge_{i \in 1,2} A_i) \cdot B = C} \text{ A-INTER}$$

$$\frac{B \leq A}{(A \to C) \cdot B = C} \text{ A-ARROW}$$

Figure 7: Algorithm to compute destination states

3.3 Behavioral Subtyping

Method refinement allows adding information to a method specification in a subtype. Changing a method specification in a subtype is potentially dangerous because the new specification could contradict the supertype specification. (Imagine a call to *BufferedInputStream.read* would close the stream, see figure 3.)

Liskov and Wing formalized sensible pre- and post-conditions for subtypes in their work on behavioral subtyping [16]. Whenever an object is used where a supertype is expected, behavioral subtyping guarantees that it reacts to method calls in a way that is compatible to the supertype's specification.

Behavioral subtyping requires the supertype's pre-condition to imply the subtype's, and the subtype's post-condition to imply the supertype's. The different implication directions result from functions being contravariant in the domain and covariant in the range. The behavioral subtyping requirements are therefore equivalent to substate tests on method refinements.

THEOREM 2 (BEHAVIORAL SUBTYPING). *If C is a subtype of B then for all methods m defined in B the following holds: Let $B.m \in S$ be the refinement for m in B and $C.m \in T$ the refinement for m in C. Then $T \leq S$.*

PROOF. *If C is a subtype of B and $B.m \in S$ then $C.m \in S \wedge T'$, where T' is the additional information given for m in C. According to figure 6, $T = S \wedge T' \leq S$ always holds.* $\square$

Findler calls it a "hierarchy violation" if an overriding method violates behavioral subtyping. This happens if the pre-condition is strengthened or the post-condition is weakened. Butkevich et al. can detect hierarchy violations statically for their protocols [2]. Pre- and post-condition systems in general cannot detect hierarchy violations statically. Moreover, if at runtime a predicate violation results from a faulty specification, most systems flag a normal condition violation [11].

The theorem above implies that *any* method specification T in a subtype validates the behavioral subtyping conditions. In other words, hierarchy violations are impossible with our approach; we always guarantee behavioral subtyping. This pleasant result naturally falls out because of the way we designed our specifications.

4. DYNAMIC STATE ANALYSIS

This section describes the dynamic analysis we developed for enforcing the interface specification technique presented in the preceding sections. We first describe the annotations needed to analyze implementations. Then we introduce our dynamic checking technique. We give an intuition of how to detect unsatisfiable specifications and briefly discuss the discovery of dangling resources. Finally, details of our analysis implementation for Java are described.

$$\begin{array}{rcl}
D & ::= & s := P \\
P & ::= & A \mid P \;\&\&\; P \mid P \;||\; P \mid !P \\
A & ::= & f\ \mathit{instate}\ S \mid f\ \mathit{substateof}\ f \mid f\ op\ F \mid F \\
F & ::= & S \mid f \\
S & ::= & s \mid n \mid \mathit{true} \mid \mathit{false} \mid \mathit{null} \\
f & ::= & \mathit{fieldname} \mid \mathit{super} \\
s & ::= & \mathit{statename} \\
n & ::= & \mathit{integer} \\
op & ::= & == \mid != \mid < \mid <= \mid > \mid >
\end{array}$$

Figure 8: State invariant definition language

Specifying State Invariants. Our interface specifications define abstract states and state transitions for each method. An interface is implemented with a class that defines fields and method bodies. We map the interface's abstract states onto the fields of the implementing class by defining a predicate for each state, called a *state invariant* in Fugue. This idea extends the concept of *class invariants* in object-oriented specification methods [8] in a principled way.

The language for defining a predicate is given in figure 8. Each state is assigned a predicate which can be any boolean combination of atomic predicates. Atomic predicates include state tests, state comparison, integer comparisons, and boolean constants and fields.

Each class can define its own predicates for all states. This is a major advantage of our approach because it allows programmers to freely reimplement and extend functionality in subclasses. An object can be seen as a stack of *frames*. Each frame corresponds to a class in the inheritance hierarchy and holds the object's fields defined in that class. Using each class's predicates we can determine the state of each frame in an object.

An extension to Fugue's predicates is the special variable *super*. Because of information hiding subclasses typically cannot directly access fields in superclasses. *super* allows a class to make its states dependent on the abstract state of the immediate superclass. We do *not* impose any restrictions on what the state dependencies look like. This is an important extension of Fugue's predicate language — we benefited from it a number of times in our case studies.

Nonetheless our predicates are somewhat limited compared to traditional pre- and post-conditions like Findler's [10] because we do not allow method calls in them. This choice has a number of advantages including potential static checking and freedom from side-effects. Also, extending our predicates with method calls would be straightforward. Thus our approach can be seen as a framework for more lightweight pre- and post-condition specifications.

Dynamic Predicate Checking. To test whether an object is in a particular state, we evaluate the state's predicate and all predicates of superstates. Thus state refinements build a hierarchy of predicates that allows the programmer to define state invariants very concisely. The default predicate for each state is *true*.

The analysis tracks sets of states for each object frame. When the analysis first encounters an object frame it tests all known states and "remembers" the set of applicable states (whose predicates evaluated to *true*). Afterwards, usually only a few states are retested at a time. For each method execution we perform the following steps.

1. Before a method is executed, build a product $e_2 = (o, a_1, \ldots, a_n)$ to represent the receiver object o and argument objects $a_1, \ldots, a_n$ involved in the method call. Determine the refinement $e \in A$ of the called method.
2. Compute the refinement $e\ e_2 \in E$ with our *apply* algorithm (figure 7) to get the expected post-condition E. To test $b \le s$ where b is an object in e_2 and s a state, simply look up whether b is remembered to be in s.[3]
3. If E cannot be computed, flag a pre-condition violation.
4. Execute the method body.
5. After method execution, build a product e_2' similar to e_2 that contains receiver, arguments, and the additional result object, if any.
6. For each object in e_2', collect all states it can be in according to E. Test these possible states and remember the ones that are applicable.[4]
7. Test $e_2' \le E$ the same way as in step 3.
8. If the test yields false, flag a post-condition violation.

Unsatisfiable Specifications. We say that a method specification is *unsatisfiable* if for a valid pre-condition the expected post-condition cannot be true. This post-condition will most likely contain an intersection of mutually exclusive states like $open \wedge closed$. This can in particular occur in the presence of method refinement. If a method specification $S \rightarrow T$ is refined with $S' \rightarrow T'$, the resulting specification is $S \rightarrow T \wedge S' \rightarrow T'$. Applying this e.g. to $S \wedge S'$ will yield $T \wedge T'$, and the specification is unsatisfiable if T and T' happen to be mutually exclusive.

Static implementation checking can discover this problem because it is impossible to write a terminating implementation for an unsatisfiable specification. But even ignoring the implementation we can check statically if a specification is satisfiable. Because this check is not currently implemented, we only give an intuition here.

For a given method, list all possible combinations of receiver and argument states. This list is always finite. Then exhaustively simulate the possible method calls based on this list and compute the expected destination states. Test whether all expected destination states are satisfiable, i.e. are free of intersections of mutually exclusive states. This procedure can be fully automated. The intersection with the inherited specification for method refinements must be used in order to cover the important case we discussed above.

[3]We in fact retest the predicates of remembered states. If some state is no longer valid, we recompute all applicable states and flag a warning.

[4]If none of the possible states are applicable for an object, then typically the state invariants are not properly defined and an error is flagged.

Detecting Dangling Resources. One of the most important applications of Fugue was to find dangling resource bugs such as files and SQL connections that were not properly closed [6]. Just like Fugue, Butkevich et al. mark valid end states and check that they are reached [2]. We suggest an alternative approach that works better in the presence of state refinements.

We refine our root state *alive* into two states *collect* and *bound*, where the latter indicates that the object should not become available for garbage collection. For instance, an open stream is bound. *collect* implies in contrast that the object can be collected safely, like a closed stream.

All three states *alive*, *bound*, and *collect* can be refined. We consider it good practice to normally extend *collect* unless resources are going to be bound in the refining states.

The dynamic analysis can now check all objects that become available for garbage collection for potential dangling resources. Whenever an object is found in *bound* or one of its substates, a warning is issued.

Implementation in Java. Our dynamic state analysis is implemented for Java and can monitor method executions in a running program with AspectJ. We use Java 5 annotations (JSR-175) to put our state specifications directly into Java source files. We will see examples of their usage in the following section.

- `@Refine` refines a state using three fields *parent*, *dimension*, and *states*.
- `@Case` defines a method case with the grammar in figure 2.
- `@Pred` defines a state invariant with a string that is parsed according to the grammar in figure 8.
- `@States`, `@Cases`, and `@Predicates` are aggregates to overcome the JSR-175 limitation of one annotation of a particular type per element.

AspectJ lets us intercept all method calls of interest and perform the dynamic predicate checking algorithm described above as an *around* advice. We do not produce custom checking code for each state predicate but rather implemented a generic strategy to evaluate predicates using Java's reflection mechanism. Atomic predicates essentially trigger substate tests on fields. Boolean combinations of predicates are evaluated in standard left-to-right order, omitting predicates that cannot change the result.

5. CASE STUDIES

In section 3 we saw that our state-based specification technique has desirable theoretical properties. In this section we report on case studies with the Java JDK 1.5_01. We modeled the standard libraries for stream-based I/O and for accessing SQL databases. We report in this order.

5.1 Modeling Java I/O

The Java I/O library (`java.io`) provides subclasses of *InputStream* to read byte streams from various sources. We saw part of the stream interface in section 2.1. Subclasses of *OutputStream* can be used to write byte streams. Streams turned out to be a rich source of interesting observations. We will mostly discuss pipes because of their relatively high complexity.

Formalizing documentation. Java's I/O package is very well documented. Most methods, even private ones, include extensive usage information. Here is part of the documentation for method `int read(byte b[], int off, int len)` in *InputStream*.

Reads up to len *bytes of data from the input stream into an array of bytes... If* len *is zero, then no bytes are read and* 0 *is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at end of file, the value* -1 *is returned; otherwise, at least one byte is read and stored into* b*.*

To model this, we refined our *open* state into *within* and *eof* (end of file).[5] The method annotation looks as follows. Notice that the first case guarantees at least one byte, as required. The second case applies when *len* is 0.

```
@Cases( {
  @Case("(within, alive, >= 0, > 0)" +
    "-> (within|eof, alive, >= 0, > 0, > 0)"),
  @Case("(open, alive, >= 0, 0)" +
    "-> (open, alive, >= 0, 0, 0)"),
  @Case("(eof, alive, >= 0, > 0)" +
    "-> (eof, alive, >=0, > 0, -1)")  } )
```

This demonstrates the usefulness of specifying a method with multiple cases, which are a benefit of using intersection types. Complex methods like *read* tend to behave differently depending on the situation. Method cases proved ideal to express this behavior precisely and succinctly.

Most methods in Java I/O have documentation like the one above, validating our hypothesis that programmers have to informally document contracts and protocols. We can formalize these to a large extent — and in a succinct way: We needed only 4 lines of annotations as opposed to 62 lines of comments for this method.

State refinement. The state refinements in *InputStream* (described above) demonstrate that refinements are an intuitive way to specify the state space of a class. But initially they were motivated by subclasses needing additional states. We did indeed use state refinements quite frequently in subclasses, e.g. to distinguish interesting states of buffers (section 2.2).

State dimensions. We included state dimensions for scalability, i.e. to allow concise state spaces and short method specifications. Java's pipe implementation shows the validity of this hypothesis. Pipes are implemented with two classes. A producer pushes data into a *PipedOutputStream* (writer), which will forward it to a buffer in *PipedInputStream* (reader), from where a consumer can poll the data. An open reader has to distinguish between its buffer being empty, having data available, and being full. It also has to keep track of the writer being connected or closed: Only a connected writer can put data into its reader's buffer (the exact protocol follows).

It turns out that most methods in a class only care about one of the dimensions. Thus dimensions simplified the specification a great deal and prevented state explosion problems. For instance, *PipedInputStream* distinguishes 13 distinct states which can be combined in 52 possible combinations.

[5]We omitted this refinement in section 2.1.

Usage protocols. The two sides of a pipe communicate with a relatively complex protocol. As long as the pipe is connected, the writer pushes data to the reader through a (package-private) method *receive*. This method calls *checkStateForReceive* that implements the very same state tests that our dynamic analysis performs to verify that the pipe is intact. When a writer is closed, it calls *receivedLast* to transition its reader from "writer connected" to "writer closed" (discussed in the preceding section). The reader can still provide data from its buffer. Conversely, closing the reader breaks the pipe immediately by closing the writer as well.

State and method refinements allowed us to formalize the pipe protocol almost completely.[6] The specification is compatible with the much simpler protocol set forth in the base classes. This shows in practice that our technique guarantees substitutability and yet is flexible enough to express interesting changes to protocols in subclasses.

Method specifications. We found several examples like the one given in section 2.2 of method refinements to strengthen the post-condition. We didn't see cases of relaxed preconditions. This might be more common in less carefully designed libraries. Nonetheless the examples suggest that method refinement is a useful technique to include.

Many arguments in Java I/O are immutable simple types. But we did find an example where argument states change through a method invocation. *PipedOutputStream* has a method *connect* that takes a *PipedInputStream* and hooks the receiver to the argument. That changes both objects' states, which is not expressible in Fugue.

Predicate identification. Deriving predicates was mostly straightforward. Comparing integer field values came in handy a number of times while we never used the possibility to compare states of two objects. Consider the following quite typical description in *BufferedInputStream* for the integer field *pos*. (It refers to another integer *count* and the actual buffer array *buf*.)

... This is the index of the next character to be read from the buf *array. This value is always in the range* 0 *through* count*. If it is less than* count*, then* buf[pos] *is the next byte to be supplied as input; if it is equal to* count*, then the next* read *or* skip *operation will require more bytes to be read from the contained input stream.*

BufferedInputStream inherits from *FilterInputStream*, which is a proxy [13] for an underlying *InputStream*. The filter's states are entirely defined by that underlying stream. But a buffered stream does not automatically reach *eof* when the superclass does: the buffer can still have data. In contrast to Fugue [7] we can express this invariant with our "pseudo-field" *super*.

```
@Pred("eof := super instate eof && pos >= count")
@Pred("within := super instate within || " +
  "(super instate eof && pos < count)")
```

Annotation overhead. Annotating the byte stream classes of `java.io` took about one day. We could annotate all methods with at most three cases. Each case fit easily into one line. The two base classes require 3 and 5 state refinements (where 2 each are used to rename "collect" into "closed"

[6]The implementation additionally detects dead threads.

and "bound" into "open"). Subclasses add between 0 and 3 refinements. Input streams in general have more states than output streams, but even the most complex class *PipedInputStream* only needs 13 declared leaf states and 11 predicates. Because of ubiquitous use of state dimensions, the number of state combinations and thus of atomic invariants is much larger with 52 possible combinations. Each of these would require an individual predicate in Fugue, not to mention the complexity of method specifications. This validates the utility of state dimensions both for concise interface specifications and state invariant definitions.

What we cannot express. An obvious shortcoming of our current implementation for low-level libraries is its handling of arrays. Arrays are treated as normal objects. A straightforward extension would be to consider the length of an array as its (immutable) state. That would allow more exact state predicates for *PipedInputStream* as well as requirements on the length of buffers passed to *read* and *write* methods.

We saw in the example of *read* that we sometimes want to relate states of different arguments. We allow this for predicates already. We are working on a compelling syntax for method specifications. For *read* and *write* this could ensure that the array is long enough to hold the number of bytes requested by separate parameters. But these were the only interesting applications of this feature we found.

Finally, our technique does not deal with exceptional control flow. This is definitely future work. For now, our analysis implementation expects an exception to be thrown if a pre-condition is violated.

5.2 Modeling JDBC

JDBC is the Java standard for accessing relational databases supporting SQL. It is specified in the package `java.sql`. A database is accessed in three steps: A *DriverManager* is asked for a *Connection* to the database. From there a *Statement* can be acquired to execute SQL commands. For each query a *ResultSet* is created that represents the rows retrieved from the database.

DriverManager is a class. The other three types are specified with Java interfaces. Each database vendor implements them to support its database. Most observations made above for Java I/O also apply to JDBC. Again we could capture important invariants from the documentation.

State dimensions were tremendously useful for modeling JDBC, and we include here some observations about statements and result sets. We took a glimpse at result sets in section 2.3 (figure 4). Besides the ones already seen, there are two more state dimensions we want to mention.

- Ability to scroll. Some result set instances can only go linearly through the rows while others can "scroll" to any row at any time. Scrolling can be sensitive or insensitive to concurrent changes.
- Concurrent access can be read-only or updating.

The states in these dimensions are fixed upon the result set's creation. We model this by giving statements two dimensions corresponding to the ones in *ResultSet*. The *Statement* method to create a new result with the signature `ResultSet executeQuery (String sqlQuery)` has the following cases (we omit `@Case` for space reasons).

```
(bound, alive) -> (noResultsAvailable, alive, start)
(rsLinear, alive) -> (rsLinear, alive, linear)
(rsInsensitive, alive)
      -> (rsInsensitive, alive, insensitive)
(rsSensitive, alive) -> (rsSensitive, alive, sensitive)
(rsReadOnly, alive) -> (rsReadOnly, alive, readOnly)
(rsUpdatable, alive) -> (rsUpdatable, alive, updatable)
```

Imagine the specification of this method without state dimensions! Moreover, notice that method cases work very well to specify behavior along multiple dimensions, as shown in this example. Each state combination will "trigger" three of the six cases listed above.

Except for this and another, similar method we again could specify every method with 3 cases or less (each time formalizing 10-20 lines of documentation text). The vast majority of methods have only one (deterministic) case. The specification is huge; e.g. *ResultSet* has about 2400 lines mostly containing documentation. Due to the sheer size it took about one and a half days to annotate the interfaces.

As a final remark, JDBC to some extent relies on aliasing. In particular, *close* is a cascading operation. Thus if a client closes a database connection, all statements and result sets belonging to it also have to be closed. To implement this requirement, a connection could hold references to its statements and call *close* on them when it is closed. Our analysis handles this case just fine. But alternatively statements could hold on to their connection. If the connection is closed, the statements change their state without a call to one of their methods. We call this a *silent state transition*. Handling this case is one of our biggest goals for future work.

6. RELATED WORK

The contributions presented in this paper are founded on research in object-oriented specification and programming techniques as well as fundamental programming languages research. Besides these sources we discuss other areas of research related to ours.

Typestate was initially proposed for imperative languages [22] and has led to powerful linear type systems like Vault [5] that track resource usage. DeLine and Fähndrich developed Fugue [7] which tracks typestates for linear objects. Fugue is the only existing attempt to incorporate typestate into object-oriented languages. It allows subclasses to define additional states and their own state predicates. Fugue contributes in many ways to this paper. In an alternative approach, Butkevich et al. describe protocols as labelled transition systems and can statically check for hierarchy violations [2]. They have no analogue to state refinement or dimensions. Method arguments cannot influence protocols.

Eiffel [18] pioneered the idea of Design by Contract. Methods in Eiffel declare their pre- and post-conditions and check them dynamically. JML is an approach to support this in Java [3]. Behavioral subtyping was proposed by Liskov and Wing to formalize rules for writing subclasses [16]. These rules are based on pre- and post-condition predicates. With these rules, Findler formalizes hierarchy violations [11, 10] which cannot occur in our approach.

State-based specification methods such as Z [1] are successfully used for specifying systems. Object-Z [8] adapts Z to object-oriented systems. It can capture class invariants and supports powerful pre- and post-conditions of methods. Object-Z has no immediate mapping onto an implementation.

Statecharts [15] are used to visually specify reactive systems. We make their concepts of AND- and OR-states suitable for object-oriented languages and base our semantics on union and intersection types.

Dunfield and Pfenning developed a decidable type system for union and intersection type refinements [9]. We propose states as basic units of refinement for objects and develop a deterministic algorithm for typing function applications for our dynamic analysis.

A large body of research is available on logic-based type systems for effective computation. Effective type refinements [17] use linear logic to form a theoretical model for Vault (discussed above) and similar systems. Separation logic [21] is based on Hoare logic.

Metal checks protocols in C programs [14] based on syntax. It defines protocols with states and scales with the number of protocols, just like our approach. ESC [12] verifies properties of programs written in Java an other languages.

7. CONCLUSION

The pervasive use of mutable state in object-oriented software brings about a number of challenges with specifying objects. We propose a novel specification technique based on typestates. It preserves substitutability with state refinement, guarantees precision and behavioral subtyping with union and intersection types, and allows more concise specifications with state dimensions. We developed a dynamic analysis based on state invariants to check specification conformance. Two case studies of annotating standard Java libraries provide evidence for the usefulness of our approach. They each revealed potential for future work.

We see our dynamic analysis as a testbed that helps in exploring various research questions. In future work we plan to extend our model of state spaces and our predicate checking facilities. We also hope to gather empirical data about data sharing policies used in practice. This will guide our effort to devise a static checker for our approach to state-based specifications.

Acknowledgements

We thank Joshua Dunfield, Elissa Newman, George Fairbanks, Mary Shaw, and the anonymous reviewers for comments on this material. This work was supported in part by NASA cooperative agreements NCC-2-1298 and NNA05CS30A, NSF grant CCR-0204047, and the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems".

8. REFERENCES

[1] J.-R. Abrial. *The Specification Language Z: Syntax and Semantics*. Programming Research Group, Oxford University, 1980.

[2] S. Butkevich, M. Renedo, G. Baumgartner, and M. Young. Compiler and tool support for debugging object protocols. In *SIGSOFT Symposium on the Foundations of Software Engineering*, 2000.

[3] Y. Cheon and G. T. Leavens. A runtime assertion checker for the java modeling language (jml). In *International Conference on Software Engineering Research and Practice*, 2002.

[4] R. Davies and F. Pfenning. Intersection types and computational effects. In *ACM International Conference on Functional Programming*, pages 198–208, 2000.

[5] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

[6] R. DeLine and M. Fähndrich. The fugue protocol checker: Is your software baroque? Technical Report MSR-TR-2004-07, Microsoft Research, 2004.

[7] R. DeLine and M. Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*. Springer-Verlag, 2004.

[8] R. Duke, G. Rose, and G. Smith. Object-z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533, 1995.

[9] J. Dunfield and F. Pfenning. Tridirectional typechecking. In *ACM Symposium on Principles of Programming Languages*, 2004.

[10] R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *ACM Conference on Object-Oriented Programming Languages, Systems, and Applications*, pages 1–15, 2001.

[11] R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. In *SIGSOFT Symposium on the Foundations of Software Engineering*, pages 229–236, 2001.

[12] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for java. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[14] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[15] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Programming*, 8:231–274, 1987.

[16] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.

[17] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *SIGPLAN International Conference on Functional Programming*, pages 213–225, 2003.

[18] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[19] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[20] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, 2002.

[21] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[22] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, 1986.