Mechanized Analysis of Behavioral Conformance in the Eiffel Base Libraries

Steffen Helke and Thomas Santen

Technische Universität Berlin
Institut für Kommunikations- und Softwaretechnik
FR 5-6, Franklinstrasse 28/29, D-10587 Berlin
email: helke@cs.tu-berlin.de, santen@acm.org

Abstract. We report on an analysis of the inheritance relationships in the Eiffel Base Libraries, a library of container data structures. If inhertance is behaviorally conforming, then polymorphism can be used safely, and the inheritance hierarchy can be considered part of the interface of the library to its clients. We describe a theory of object-oriented specification in higher-order logic that we used to specify part of the Eiffel Base Libraries. With the theorem prover Isabelle/HOL, we mechanically prove conformance relationships between those specifications. This work allows us to draw conclusions about the design of the Eiffel Base Libraries, and about the feasability of using an interactive theorem prover to apply a strictly formal theory to the specification of a commercial product.

1 Introduction

One of the promises of object-oriented software development is to facilitate the reuse of software components by incremental modification through inheritance, and by decoupling abstract interfaces from concrete implementations through polymorphism. Clients commonly use the components of a library as they are provided. Therefore, the components must have a precisely defined interface. An important information that the interface must make available to the clients are the inheritance relations in the library that the clients may consider part of the interface. Only the inheritance relations of the interface that yield a behaviorally conforming subclass allow clients to use polymorphism *safely*. Therefore, it is worthwhile to study behavioral conformance in a library of components designed for reuse, in particular for inheritance relations that the library designers consider part of the library's interface. Unless clients are sure that abstract classes provide a *behaviorally conforming* interface to more concrete classes of the library, they will pick only a few concrete classes of the library and base their code on the interfaces of those classes. Thus, their code will be more sensitive than necessary to a change of the implementation of a data structure they use. On the other hand, if the clients may trust in the conformance of the inheritance relations in a library, they can design their software to rely on the interfaces to objects provided by abstract classes only. The resulting code will be more abstract and robust to changes of the implementation of objects.

Not all libraries of object-oriented components, however, consider inheritance relations part of their interface to clients. The *Library of Efficient Data Structures and Algorithms* (LEDA) [10], for example, uses inheritance for implementation purposes only. The inheritance "hierarchy" presented to clients basically is flat. Cook [2] investigated behavioral conformance (based on America's [1] notion of behavioral subtyping) between the Smalltalk-80 collection classes. The conformance hierarchy that he derived from the specifications of those classes exhibits multiple relations between the classes. Contrasting the conformance hierarchy with the inheritance hierarchy, he notes that the two are mostly unrelated and in some cases even contradict each other.

In contrast to LEDA and the Smalltalk-80 library, a design goal of the Eiffel Base Libraries [9] was to accomplish a "Linnean reconstruction of the fundamental software structures. ... to devise a coherent overall structure, in which every important variant will have a precise place deduced from the application of a small number of classification criteria" [9]. Therefore, the rich inheritance hierarchy in the Eiffel Base Libraries must be considered part of the interface of the libraries. It is designed to provide various abstractions (or views) of the objects that implement concrete data structures.

Using a theory of object-oriented specification that we have developed with the theorem prover Isabelle/HOL, we have investigated the conformance relations part in the Eiffel Base Libraries. The aim of this work was to provide mechanized tool support for reasoning about software components and to test the feasability of such an approach on a practically used software product.

In Section 2, we give a brief overview of the part of the Eiffel Base Libraries that we worked on. Section 3 discusses why specifying classes is useful and introduces our specification notation. The representation of specifications in Isabelle/HOL is the topic of Section 4. Section 5 introduces behavioral conformance and the way we have defined it in higher-order logic. We analyze the way classes are built by inheritance in the Eiffel Base Libraries in Section 6. Section 7 shows how those constructions can be mimicked at the specification level. Section 8 summarizes the work on the specifications with Isabelle. We conclude in Section 9 with observations about the Eiffel Base Libraries as a software product and with an analysis of our work with Isabelle/HOL.

2 The Eiffel Base Libraries

The several hundred classes of the Eiffel Base Libraries are grouped into five main libraries: the *kernel*, *data structure*, *iteration*, *Lex*, and *Parse* libraries. We concentrate on the data structure library. It is the largest part of the Eiffel Base Libraries and covers fundamental data structures and algorithms, such as lists, arrays, queues, stacks, trees, and hash tables. Because we investigate the data structure library only, we use the name "Eiffel Base Libraries" as a synonym for the data structure library in the following.

The data structures covered by the Eiffel Base Libraries are container structures: their objects are collections of items that may be changed and accessed according to the specific properties of the data structure. Figure 1 shows the part of the Eiffel Base Libraries on which we focus our investigation. The root of the class hierarchy of the data structure library is the class *CONTAINER* shown in Figure 2. Because the type G of the items is irrelevant for a container, the classes of the *CONTAINER* hierarchy are generic in G. The interface of the class *CONTAINER* contains three basic features that any container structure must provide: the test $has(v)$ determines if the container object contains an item v; the attribute *empty* is true if the container does not contain any item;

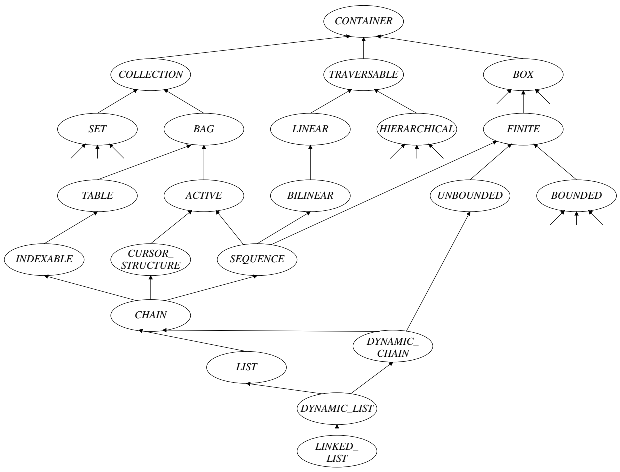

Fig. 1. The *CONTAINER* hierarchy.

the attribute *linear_representation* provides access to the container as a sequence (of type *LINEAR*[*G*]).

The keywords ***require*** and ***ensure*** are part of the assertion language of Eiffel that makes it possible to state preconditions (***require***) and postconditions (***ensure***) of features, and class invariants, and check their validity during execution of an Eiffel program. We will discuss the assertion language in more detail when we consider specifications of Eiffel classes in Section 3.

A major design principle of the Eiffel Base Libraries is to organize the inheritance hierarchy according to a taxonomy of container structures. The three direct heirs of *CONTAINER* represent three criteria – groups of related properties – that serve to describe containers.

– The *access* criterion describes how the clients of a container can access its items. For example, only the top item of a stack is accessible to its clients, whereas the clients of an array can access each of the items of the array by an index. The class *COLLECTION* is the root of the access hierarchy.
– The *traversal* criterion describes how the clients of a container can navigate through the container, accessing each of its items in turn. For example, a particular kind of list may allow traversals only from the front to the back of the list, whereas another kind of list may also allow to traverse it from the back to the front. The class *TRAVERSABLE* is the root of the traversal hierarchy.
– The *storage* criterion describes the cardinality of a container and the ways to change the cardinality. For example, some containers are finite, others are potentially infinite. A finite container may have a fixed size, or its size may be unbounded. The class *BOX* is the root of the storage hierarchy.

Each class in the *CONTAINER* hierarchy provides a view to containers with a particular combination of properties. The (deferred) classes near the root of the hierarchy

```
class CONTAINER [G]
  feature has (v : G) : BOOLEAN is
    deferred
    ensure not_found_in_empty : Result implies not empty
  end;
  feature empty : BOOLEAN is
    deferred
  end;
  feature linear_representation : LINEAR [G] is
    deferred
  end;
  feature object_comparison : BOOLEAN
    ⋮
end
```

```
class COLLECTION [G]
  inherit CONTAINER [G]
  feature extendible : BOOLEAN is
    deferred
  end;
  feature prunable : BOOLEAN is
    deferred
  end;
  feature put, extend (v : G) is
    require extendible : extendible
    deferred
    ensure item_inserted : has(v)
  end;
  feature fill (other : CONTAINER [G]) is
    do
    ...
  end;
  feature prune (v : G) is
    require prunable : prunable
    deferred
  end;
  feature prune_all (v : G) is
    require prunable : prunable
    do
    ...
    ensure no_more_occurrences : not has(v)
  end;
  feature wipe_out is
    require prunable
    deferred
    ensure wiped_out : empty
  end;
end
```

```
class BAG [G]
  inherit COLLECTION [G]
    redefine extend
    end
  feature occurrences (v : G) : INTEGER is
    deferred
    ensure not_negative_occurrences : Result >= 0
  end;
  feature extend (v : G) is
    deferred
    ensure then one_more_occurrence : occurrences(v) = old(occurrences(v)) + 1
end
```

Fig. 2. Container classes.

```
class ACTIVE [G]
  inherit BAG [G]
  feature item : G is
    require readable : readable
    deferred
  end;
  feature readable : BOOLEAN is
    deferred
  end;
  feature writable : BOOLEAN is
    deferred
  end;
  feature replace (v : G) is
    require writable : writable
    deferred
    ensure item_replaced : item = v
  end;
  feature remove is
    require prunable : prunable;
            writable : writable
    deferred
  end;
  invariant
    writable_constraint : writable implies readable;
    empty_constraint : empty implies (not readable) and (not writable)
end
```

Fig. 3. "Active" containers.

provide very abstract views of containers. For example the class *BAG* shown in Figure 2 adds only one feature to the ones it inherits from *COLLECTION*: the feature *occurrence*(v) returns the number of occurrences of the item v in the container. Thus, the class *BAG* implicitly represents the property that a bag may store an item more than once. The leaves of the container hierarchy, such as *LINKED_LIST* at the bottom of Figure 1, are effective classes implementing a specific kind of container structure.

The siblings that directly inherit from the same class in some cases represent alternative properties of the class of containers their parent class represents. For example, the classes *SET* and *BAG* represent such a "choice point". The feature *occurrence*(v) provided by *BAG* is useless for sets, because a set may contain an item v at most once. The class *SET* declares an attribute *count*, instead, that holds the number of (different) elements in a set. It also strengthens the postcondition of the feature *prune*(v), which it inherits from *COLLECTION* (c.f. Figure 2). The feature *prune*(v) removes an occurrence of v from the container. For a set, which may contain v at most once, *prune*(v) removes *all* occurrences of v, i.e. it ensures ***not*** *has*(v). For a bag, in contrast, *prune_all*(v) establishes that postcondition, but *prune*(v) does not necessarily do so.

Other branches in the inheritance hierarchy represent independent properties, which may be combined further down in the hierarchy by multiple inheritance. For example, the heir *TABLE* of *BAG* describes containers whose items are accessible through keys. Figure 3 shows the class *ACTIVE*, which is another heir of *BAG*. The class *ACTIVE* describes containers that have a (possibly undefined) "current item" (called *item*) on which access and modification features work. Being accessible through keys and having a current item are no contradictory properties of a container. The class *CHAIN*, which is a common abstraction of (non-circular) lists and circular containers, combines those properties by multiple inheritance. Thus, the interfaces of *TABLE* and *ACTIVE* both provide valid "views" of a chain.

Collection[G]

STATE

$$
\begin{array}{l}
c : \mathrm{bag}\ G \\
\mathsf{visible} \\
empty : Bool \\
extendible : Bool \\
prunable : Bool \\
lin\_rep : \mathrm{seq}\ G \\
\hline
empty = true \Leftrightarrow c = [\![\,]\!] \\
prunable = true \Rightarrow empty = false \\
items\ lin\_rep = c
\end{array}
$$

fill

$$
\begin{array}{l}
\Delta(c, empty, extendible, prunable, lin\_rep) \\
other? : Collection[G] \\
\hline
c' = c \uplus items\,(other?.lin\_rep)
\end{array}
$$

INIT

$$
c = [\![\,]\!]
$$

has

$$
\begin{array}{l}
v? : G \\
result! : Bool \\
\hline
result! = true \Leftrightarrow v? \sqsubseteq c
\end{array}
$$

extend

$$
\begin{array}{l}
\Delta(c, empty, extendible, \\
\quad prunable, lin\_rep) \\
v? : G \\
\hline
extendible = true \\
v? \sqsubseteq c' \\
\{v?\} \mathbin{⩤} c = \{v?\} \mathbin{⩤} c'
\end{array}
$$

prune_all

$$
\begin{array}{l}
\Delta(c, empty, extendible, prunable, lin\_rep) \\
v? : G \\
\hline
prunable = true \\
v? \sqsubseteq c \Rightarrow c' = \{v?\} \mathbin{⩤} c \\
\neg\ v? \sqsubseteq c \Rightarrow (c' = c \wedge empty' = empty \wedge \\
\qquad\qquad extendible' = extendible \wedge \\
\qquad\qquad prunable' = prunable \wedge \\
\qquad\qquad lin\_rep = lin\_rep')
\end{array}
$$

prune

$$
\begin{array}{l}
\Delta(c, empty, extendible, prunable, lin\_rep) \\
v? : G \\
\hline
\{v?\} \mathbin{⩤} c' = \{v?\} \mathbin{⩤} c \\
c \sharp v? = 1 \Rightarrow c' = c \\
c \sharp v? > 1 \Rightarrow c' \sharp v? < c \sharp v?
\end{array}
$$

Fig. 4. Specification of *COLLECTION*.

3 Specification of Library Classes

Eiffel is one of the few programming languages that support *assertions*. Assertions are Boolean expressions that are evaluated at run-time (if the program is compiled with the appropriate compiler option). An assertion that evaluates to *false* raises an exception. Assertions thus provide a valuable tool for debugging programs.

The assertion language is similar to an algebraic specification language that allows one to specify relations between the functions (i.e. the features) of a given signature (i.e. a class). Nevertheless, assertions are too weak to state all relevant properties of a class. In particular for abstract – deferred – classes, it often is impossible to state assertions that describe the intended functionality of a feature. For example, there are no assertions about the attribute *empty* of *CONTAINER* (c.f. Figure 2), because the features available in *CONTAINER* are not expressive enough to state a postcondition on *empty* that captures the intended meaning of that attribute. Similarly, there are no constraints on the attribute *linear_representation* of *CONTAINER*, because it is impossible to state that the items in *linear_representation* are exactly the items in the container object.

We use a notation similar to Object-Z [16] to specify the intended behavior of classes in a "model-based" way. Refering to a (hidden) mathematical model of containers, we can easily express the effect of features on a container. Figure 4 shows the specification of the class *COLLECTION*, whose Eiffel code is sketched in Figure 2. The semantics of our notation is given by representing it in higher-order logic – a topic that we address in Section 4.

A collection is a mutable container structure storing a number of items of a set G. It can be tested whether a collection is empty and whether it contains a given item. A collection can be extended by a single item (*extend*) or the items of another collection (*fill*). Conversely, duplicate items can be removed from a collection, but once an item with a particular value has been added to the collection, at least one item with an equal value must remain in the collection. The attribute *lin_rep* gives access to a representation of the collection as a sequence.

The constituents of the class specification are gathered in a *class schema* called $Collection[G]$. The parameter indicates that the class is generic in a set G of items. The first component of the class schema is the state schema *STATE*. It describes the valid states of the objects of the class, including attributes of the objects that are observable by other objects. The first component c of the state is the mathematical model of a collection, a bag.

Bags are defined relationally by partial functions from the set of items G to the positive natural numbers, $\mathrm{bag}\, G == G \rightarrow\!\!\!\!\!+ \mathbb{N}_1$. Bags are formed using double brackets, $[\![v_1, \ldots, v_n]\!]$, the membership test for bags is $v \sqsubseteq c$, $b \uplus d$ is the union of two bags, and $b \sharp v$ is the number of v's in b. The function *items* maps sequences to bags. The domain anti-restriction $D \lhd R$ restricts the domain of relation R to items not in D.

In contrast to c, which is not accessible from outside of the class, the state components below the keyword visible are externally visible attributes. Visible state components are a notational abbreviation for methods returning values that depend only on the internal state of an object, so-called *observer methods*. The predicate below the horizontal line of the state schema is the *state invariant*. It describes the relationship between the state components of valid objects.

The schema *INIT* describes the valid initial states of objects: the container c is initially empty, and, because the state invariant is implicitly required by *INIT*, $empty = true$ and $lin\_rep = \langle\rangle$.

The schemas *has*, *extend*, *fill*, *prune*, and *prune_all* specify the methods of the class. Like in Object-Z, input and output variables are decorated with question marks and exclamation marks, respectively. Undecorated variables refer to the state of the object before executing the method (*pre-state*), primed variables to the state after executing the method (*post-state*). The method schemas of a class *implicitly* refer to the state schema, and the notation $\Delta(x, y, z)$ is used to indicate that the method may change only the state components x, y, and z.

The observation that the assertions of a class are insufficient to specify the intended behavior of its objects properly is also valid for the classes further down in the container hierarchy, which have a richer structure. The assertions of the class *ACTIVE* (c.f. Figure 3), for example, do not at all clarify the purpose of the new features it declares. The postcondition of $replace(v)$, which shall replace the parameter v for the current item of the container, ensures that the attribute *item* equals v, but it cannot state the effect of

$ACTIVE[G]$

$BAG[G]$

$STATE$

visible

$item : G$

$readable : Bool$

$writable : Bool$

$readable = true \Rightarrow item \sqsubseteq c$

$empty = true \Rightarrow (readable = false \land writable = false)$

$writable = true \Rightarrow readable = true$

$replace$

$\Delta(c, item, readable, writable, linear\_representation)$

$v? : G$

$writable = true$

$item = v? \Rightarrow c' = c$

$$item \neq v? \Rightarrow count(c') = count(c) \oplus \{item \mapsto (c(item) - 1)\} \oplus \{v? \mapsto (c \sharp v? + 1)\}$$

$item' = v?$

$remove$

$\Delta(item, c, readable, writable, extendible, prunable, linear\_representation, empty)$

$prunable = true$

$writable = true$

$count(c') = count(c) \oplus \{item \mapsto (c(item) - 1)\}$

Fig. 5. Specification of *ACTIVE*.

$replace(v)$ on the container. The method could just set the attribute that holds the value of the current item to v, and leave the container data untouched.

The specification of *ACTIVE* in Figure 5, in contrast, clarifies the effect of $replace(v)$ on the internal data of the container, which, for this class, is a bag $c : \mathrm{bag}\, G$. It relates the value of *item* to the elements of c and requires c to change accordingly. The function $count(c)$ maps a bag c to a total function of the same type that maps all items not in the domain of c to 0.

The preceding examples do not question the use of assertions as a documentation of Eiffel code and as a tool for effective debugging, but they show that specifications in a more powerful language than the assertion language of Eiffel can extend the information provided by assertions to include information about the intended functionality of features. That information is not only useful as a documentation for users of the Eiffel Base Libraries, but it also is indispensable for a machine assisted analysis of the library.

4 Representation of Class Specifications in Isabelle/HOL

Since our aim is to come up with tool support for reasoning about class specifications, we must map the Object-Z-like specifications of classes to a logical formalism for which a proof tool exists. To this end, we use Isabelle/HOL, the implementation of higher-order logic (HOL) in the generic theorem prover Isabelle [11]. Our representation of class specifications builds on the representation $\mathcal{HOL}$-$\mathcal{Z}$ [7] of plain Z in Isabelle/HOL, which we extend by definitions of object-oriented concepts. We have derived an extensive theory about those concepts using Isabelle, and we have implemented a number of tailor-made proof procedures that provide efficient proof support to work with concrete specifications such as the ones describing the Eiffel Base Libraries.

In the following, we can only very briefly introduce the general approach of representing class specifications in HOL and reasoning about them using Isabelle. For a more detailed description of the theory and its implementation, we refer to [7, 12]. $\mathcal{HOL}$-$\mathcal{Z}$ and the complete theory of object-oriented specification in HOL are described in [15].

4.1 The Type of Classes

In $\mathcal{HOL}$-$\mathcal{Z}$, the schemas of Z are represented by predicates in HOL, i.e. a schema definition of Z is mapped to a function definition in HOL. The defined function maps the tuple of the signature components of the schema to the Boolean values (in HOL, the truth values are just ordinary values). This leads to a so-called shallow embedding of Z in HOL, where the expressions of Z are considered as a concrete syntax for certain expression of HOL. The strong similarity of the semantics of Z and HOL justifies that view [13].

To represent object-oriented specifications, we extend $\mathcal{HOL}$-$\mathcal{Z}$ by defining a type of classes in HOL. For technical reasons, we distinguish between the constant and the mutable part of the state of an object in that definition: Given that α is the type of method identifiers, the types κ and σ are the types of the immutable and mutable parts of the object state, and the methods of the class have the input type ι and the output type ω, then the type $(\alpha, \kappa, \sigma, \iota, \omega)\mathit{Class}$ consists of all class specifications whose components have the respective types.

$$
\begin{aligned}
(\alpha,\kappa,\sigma,\iota,\omega)\,\mathit{Class} \stackrel{\mathsf{typ}}{=} \{&(C,S,I,\mathit{Mths},H) \mid \\
&(C :: \kappa\ \mathit{Const}) \\
&(S :: (\kappa,\sigma)\,\mathit{State}) \\
&(I :: (\kappa,\sigma)\,\mathit{Init}) \\
&(\mathit{Mths} :: (\alpha,\kappa,\sigma,\iota,\omega)\,\mathit{Methods}) \\
&(H :: (\kappa,\sigma)\,\mathit{History}). \\
&\mathit{Cls}^{\checkmark}\ C\,S\,I\,\mathit{Mths}\,H\}
\end{aligned}
\tag{1}
$$

According to that type definition, a class specification is represented by a quintuple $(C, S, I, \mathit{Mths}, H)$. The components C and S are schemas describing invariants over the constant and mutable part of the objects' state. The predicate I represents the initialization schema. The method suite *Mths* is a (finite) function mapping method identifiers to representations of operation schemas specifying the methods of the class. The predicate H, the *history constraint*, is a relation between before and after states of the objects of the class.

Finally, the predicate $(\mathit{Cls}^{\checkmark}\ C\,S\,I\,\mathit{Mths}\,H)$ describes the *internal consistency* of the class. It is defined by four conditions, which ensure that the constituents of a class refer to each other in a way that conforms to our intuition of a class specification:

$$
\begin{aligned}
\mathit{Collection}\,G \stackrel{\mathsf{def}}{=}\; & (\mathit{basicclass}\,(\mathit{CollConst}\,G)\,(\mathit{CollState}\,G)\,(\mathit{CollInit}\,G)\,(\mathit{CollHist}\,G))\\
& \boxdot\,(''\mathit{empty}'', \mathit{empty}\,G)\\
& \boxdot\,(''\mathit{extendible}'', \mathit{extendible}\,G)\\
& \boxdot\,(''\mathit{prunable}'', \mathit{prunable}\,G)\\
& \boxdot\,(''\mathit{lin\_rep}'', \mathit{lin\_rep}\,G)\\
& \boxplus\,(''\mathit{has}'', \mathit{has}\,G)\\
& \boxplus\,(''\mathit{extend}'', \mathit{extend}\,G)\\
& \boxplus\,(''\mathit{prune}'', \mathit{prune}\,G)\\
& \boxplus\,(''\mathit{prune\_all}'', \mathit{prune\_all}\,G)\\
& \boxplus\,(''\mathit{fill}'', \mathit{fill}\,G)
\end{aligned}
$$

Fig. 6. Representation of *COLLECTION* in Isabelle/HOL.

1. The constant of an object's state can be chosen independently of the mutable part of its state.
2. The initialization establishes a state that satisfies the state invariant.
3. The history constraint relates only states satisfying the state invariant.
4. Each operation schema specifying a method of the class relates only states satisfying the invariant, and it respects the history constraint.

Internal consistency establishes a semantic relation between the components of a class that is in part indicated by the Object-Z-like notation we use to denote class specifications (c.f. Figure 4). For example, it is understood that the methods of a class implicitly refer to the properties of its state schema and constant declarations. The logical representation, however, must make these conditions explicit, because much of the theory about classes derived in Isabelle/HOL relies on them.

4.2 Class Constructors

The type definition (1) abstractly introduces the concept of a class in HOL. Thus, we can define HOL functions that construct class specifications. Those functions take as parameters the representations of Z schemas provided by $\mathcal{HOL}\text{-}\mathcal{Z}$ of the component specifications of a class. For example, Figure 6 shows the representation of the specification of *COLLECTION* given in Figure 4.

The function *basicclass* constructs a class specification without methods, the functions $\boxdot$ and $\boxplus$ add the specification of an observer or of an ordinary method to a class specification. Thus, the definition of *Collection G* in Figure 6 first builds a class without methods from the schemas describing the invariant and the initialization of the class. For example, $(\mathit{CollInit}\,G)$ is the HOL predicate representing *INIT* of Figure 4. Then, the definition successively adds the specifications of observers and the other methods of *COLLECTION*, where *extend G*, for example, is the representation of the schema *extend* of Figure 4. The string $''\mathit{extend}''$ is the identifier of *extend G* in *Collection G*. We can use it to select the method specification from the class specification, e.g. to model method invocations. The set $\mathit{ids}\,(\mathit{Collection}\,G)$ is the set of method identifiers of *Collection G*.

To be able to work with a class specification in HOL, we must show that its components are internally consistent, i.e. that they satisfy the predicate $Cls^{\checkmark}$. A tactic implemented in Isabelle accomplishes that proof fully automatically for specifications adhering to the usual style of specifying in Z.

5 Behavioral Conformance

Polymorphic assignments in object-oriented languages allow us to express the substitution of (an object of) a data type for another in a program at the level of the programming language. Thus, polymorphism is a means of the object-language to express the meta-language substitution of programs for programs in the context of a client program. We may therefore apply the concept of data type refinement to answer the question whether a variable can safely be bound to an object of a proper subtype of the variable's static type. America [1] coined the term *behavioral subtyping*[1] for this property. Using Hoare-triples to specify the behavior of methods of an object type, he requires that the specification of a subtype *data refines* the specification of a supertype (using a function from subtype to supertype as the simulation relation). He uses the well-known forward simulation rule of data refinement with a retrieve function as it is commonly used for Z (see, e.g., [17]).

5.1 Extra Methods

Rephrased in object-oriented terminology, America's definition of behavioral subtyping requires that no program can distinguish whether a particular variable is bound to an object of its static type or to an object of a proper subtype of that type – provided that the program uses the interface of the supertype only. This restriction on the observing programs is inadequate for most (imperative) object-oriented programming languages, because it is common practice to access the same object via different interfaces.

If we allow sharing of object references in observing programs, we must take the effect of the extra methods of a subtype into account in the definition of behavioral subtyping. The *extra methods* of a subtype with respect to a supertype are the methods of the subtype that do not directly correspond to methods of the supertype. Liskov and Wing [8] proposed two modifications of America's definition of subtyping to account for extra methods. We use the relation between the classes $CONTAINER[G]$ and $COLLECTION[G]$ shown in Figure 2 to illustrate Liskov and Wing's propositions. The purpose of $COLLECTION[G]$ (and many other classes of the Eiffel Base Libraries) is to extend *CONTAINER*[G] by several new methods, leaving the inherited methods untouched. Even this small example shows that considering extra methods is indispensable for reasoning about object-oriented component libraries.

[1] The term "behavioral subtyping" is misleading, because the logical relationship that America [1] defines may hold between classes that are not related by inheritance. We call that relationship *behavioral conformance* throughout this paper. Only in the remainder of the present section, where we refer to the terminology of others, will we stay with the term "behavioral subtyping".

Extension Rule. One approach to address subtyping in the presence of extra methods is to interpret a class specification as a complete description of the possible behaviors of all objects of the class – including objects of subclasses. Liskov and Wing [8] call that the *extension rule* of subtyping. In this interpretation, the methods of a class induce the behavior of its objects. All changes of an object's state are caused by the methods defined by its class, and we can identify the objects of a class with the inductively defined set of traces whose starting points are initial objects and which are extended by method applications. Then, extra methods can but mimic the behavior that is already possible without using extra methods.

Consider the relationship between the classes *CONTAINER* and *COLLECTION* shown in Figure 2. The class *CONTAINER* defines the observers *has*, *empty*, and *linear_representation*, but it does not provide a method to change an object's state. The class *COLLECTION* inherits from *CONTAINER* and introduces new methods to add items to a container and remove items from it. According to the extension rule, *COLLECTION* is *not* a behavioral subtype of *CONTAINER*, because the mutators *extend*, *prune*, etc. cannot be explained by the methods inherited from *CONTAINER* (because the latter are observers only).

Constraint Rule. The designers of the Eiffel Base Libraries clearly intended the class *CONTAINER*, which is the root of the inheritance hierarchy of the libraries, to provide the most abstract interface to any object of a container type. A heir such as *COLLECTION* should be accepted as a behavioral subtype of *CONTAINER*, but the principal assumption underlying the extension rule – that the methods of a class induce all possible behavior of its objects – is obviously not satisfied for that inheritance hierarchy. We must assume that the designers of the library have a more liberal view on subtyping: the description of *CONTAINER* does not forbid that objects change, therefore a subclass may introduce mutators.

The *constraint rule* of subtyping [8] reflects this liberal view: any behavior of an object is admissible unless the specification of its class explicitly disallows it. Liskov and Wing augment class specifications by *history constraints* to restrict the admissible behavior of objects. History constraints are relations of object states describing admissible state changes of objects independently of the changes that the methods make possible. The constraint rule requires that the history constraint of a subclass is sufficient for the one of the superclass. The rule poses the same restrictions on methods as America's definition of subtyping, allowing for weakening of preconditions. Weakening the precondition of a method does not lead to subtyping relations contradicting the intuition that the history constraint describes the possible behaviors of an object, because Liskov and Wing require all methods of a class to satisfy the history constraint. We captured that requirement in the definition of internal consistency.

In the example of Figure 2, we assume an unconstrained history (the full relation on valid states of objects) for *CONTAINER*. With that specification of *CONTAINER*, the constraint rule establishes that *COLLECTION* is a behavioral subtype of *CONTAINER*.

5.2 Behavioral Conformance in Isabelle/HOL

We have defined the analogon of Liskov and Wing's constraint rule for our specifications in Isabelle/HOL. That definition is quite lengthy and its exact technical phrasing is not important for this paper. In the following, we therefore paraphrase the definition in a condensed form using Z schema calculus. For more details, we refer the reader to [15].

The predicate *conforms* defines behavioral conformance in HOL. It is declared as follows:

$$\begin{aligned} \mathit{conforms} :: [\,&(\alpha, (\iota, \iota', \omega, \omega')\,\mathit{IOconv})\mathit{finmap}, (\alpha, \alpha')\mathit{finmap}, (\kappa, \kappa', \sigma, \sigma')\,\mathit{retrv},\\ &(\alpha, \kappa, \sigma, \iota, \omega)\,\mathit{Class}, (\alpha', \kappa', \sigma', \iota', \omega')\,\mathit{Class}] \to \mathit{bool} \end{aligned}$$

For two class specifications $A = (\mathit{AConst}, \mathit{AState}, \mathit{AInit}, \mathit{AMeth}, \mathit{AHist})$ and $C = (\mathit{CConst}, \mathit{CState}, \mathit{CInit}, \mathit{CMeths}, \mathit{CHist})$, the proposition $\mathit{conforms}\,\Theta\,\phi\,R\,A\,C$ is true if C behaviorally conforms to A with the retrieve relation R, the signature morphism ϕ, and the conversion Θ of input/output types between the methods of A and the methods of C.

The relation R relates the states of A to the states of C. Unlike America (and data refinement in VDM [5]) we do not require R to be a function, because it turns out that some conformance relationships in the Eiffel Base Libraries require that more liberal view of data refinement, which is also the usual one for Z [17].

The signature morphism ϕ maps the names of the methods in *AMeths* to the names of the methods in *CMeths*. For each method of A it thus identifies the method of C that simulates it.

We need the type conversion Θ for technical reasons: HOL is strongly typed. Therefore, we need to inject the different input and output types of the operation schemas specifying the methods of a class into a common type that we can use as *the* type of inputs and outputs, respectively, of the method suite of the class. As a consequence of that representation of classes, the input/output types of different classes, in general, are different. To relate single methods of two classes, we need to map a subset of the input/output type of one class to an isomorphic subset of the input/output type of the other class. The conversion Θ is a family of isomorphisms [14] (indexed by the method identifiers of A) that accomplishes that task.

Paraphrased without the technical overhead we just discussed, $\mathit{conforms}\,\Theta\,\phi\,R\,A\,C$ requires the following conditions to hold:

1. The signature morphism ϕ maps all methods of A to some of C:

   $$\mathsf{dom}\,\phi = \mathit{ids}\,A \wedge \mathsf{ran}\,\phi \subseteq \mathit{ids}\,\mathit{CCls}$$

2. All concrete initial states represent abstract initial states:

   $$\forall\, \mathit{CConst};\ \mathit{CState} \bullet \mathit{CInit} \Rightarrow \exists\, \mathit{AConst};\ \mathit{AState} \bullet \mathit{AInit} \wedge R$$

3. The preconditions of the abstract methods imply the preconditions of the concrete methods:

   $$\begin{aligned} \forall\, n : \mathsf{dom}\,\phi;\ \mathit{AConst};\ \mathit{AState};\ \mathit{CConst};\ \mathit{CState};\ \mathit{inp}_n? : X_n \bullet&\\ \mathsf{pre}\,\mathit{AMeths}(n) \wedge R \Rightarrow \mathsf{pre}\,\mathit{CMeths}(\phi(n))& \end{aligned}$$

4. The concrete methods simulate the abstract methods:

$$\forall\, n : \mathsf{dom}\,\phi;\ AConst;\ AState;\ CConst;\ CState;\ CState';\ inp_n? : X_n;\ out_n! : Y_n\ \bullet$$
$$\mathsf{pre}\, AMeths(n) \land R \land CMeths(\phi(n)) \Rightarrow \exists\, AState' \bullet R' \land AMeths(n)$$

5. The concrete history is sufficient for the abstract history:

$$\forall\, CConst;\ CState \bullet CHist \Rightarrow \exists\, AConst;\ AState \bullet AHist \land R \land R'$$

Those conditions are just the conditions of forward simulation rephrased to classes and augmented with the constraint rule that handles extra methods.

5.3 Transitivity of Conformance

Our definition of behavioral conformance in HOL is strictly formal. Therefore, we can derive properties of conformance mechanically within higher order logic using Isabelle/HOL. One important property that we have derived in that way is the transitivity of conformance: Given three classes A, B, and C, and conformance relationships between A and B, and B and C, we know that C also conforms to A:

$$\frac{conforms\,\Theta_{ab}\,\phi_{ab}\,R_{ab}\,A\,B \quad conforms\,\Theta_{bc}\,\phi_{bc}\,R_{bc}\,B\,C \quad \Theta_{ac} = \ldots}{conforms\,\Theta_{ac}\,(\phi_{ab}\ \text{⨾}_m\ \phi_{bc})\,(R_{ab}\ \text{⨾}_\ddagger\ R_{bc})\,A\,C} \tag{2}$$

6 Inheritance in the Eiffel Base Libraries

In the following, we identify patterns of inheritance in the Eiffel Base Libraries. Each of the patterns corresponds to an operation that constructs a class schema from another one. Those specification building operations do not necessarily produce a behaviorally conforming subclass of their parameter class. Therefore, it would be inadequate to call one of them “the inheritance operation” of our specification language.

There are five ways the classes in the hierarchy[2] of Figure 1 are constructed by inheritance, none of which, in general, yield behaviorally conforming subclasses:

1. *add new methods*: extend the parent by new “extra” methods;
2. *redefine methods*: change the specification or implementation of methods inherited from the parent class;
3. *add new attributes*: extend the parent by new attributes, thus extending the state of the object;
4. *hide a feature*: export the feature to *NONE*, thus excluding it from the interface of the new class;

[2] This hierarchy includes just the inheritance relations that we consider part of the interface of the Eiffel Base Libraries; the figure does not show inheritance that is used for implementation purposes only.

5. *combine classes by multiple inheritance*: combine several classes into one, selecting a definition from one class for each feature whose name appears in several of the inherited classes.

Most inheritance relations of the container hierarchy are combinations of those inheritance patterns. For example, the class *BAG* (c.f. Figure 2) extends *COLLECTION* by the new attribute *occurrences*, and it redefines the method *extend*.

We use the following specification building operations to construct specifications according the inheritance patterns: The constructor $\boxplus$ of class schemas (c.f. Section 4.2) extends a class schema by a new method. We can also use it to model redefinition, because if $n \in ids\, Cls$, then $Cls \boxplus (n, OP)$ overrides the definition of n in Cls.

Introducing new attributes involves two modifications of the specification: first, it modifies the state schema to include the new attribute and describe its invariant relations to the other state components; second, it adds a new observer method to the class schema. Modifying the state schema to include a new attribute is a special case of changing the internal model of a class schema such that, under certain conditions, the resulting class schema behaviorally conforms to the original one. We define an operation on class schemas to change the internal model of a class schema and characterize the conditions under which it yields a behaviorally conforming subclass in the following Section 7.

7 Subclass Calculation

Just as it is much more productive to build subclasses from classes by inheritance in Eiffel, it is necessary to construct specifications of subclasses from the specifications of their superclasses. It would be infeasable to specify each class in the Eiffel Base Libraries from scratch, because the resulting specifications would become very large and incomprehensible.

In the preceeding section, we showed that most of the constructions in the Eiffel Base Libraries can be mimicked on the specification level with the class constructors of Section 4. In this section, we define another class constructor, the *subclass calculator*, that allows us to change the mathematical model of a container when constructing the specification from a subclass from the specification of a class. Under certain conditions, the resulting specification behaviorally conforms to the original one.

7.1 The Subclass Calculator

The subclass calculator works dually to a posit-and-prove approach to behavioral conformance. Instead of defining two classes schemas and exhibiting a retrieve relation to establish behavioral conformance, the subclass calculator maps a class schema and a retrieve relation to a new class schema. The calculator uses the conditions of behavioral conformance to specify the components of the resulting concrete class in terms of the given abstract class and the retrieve relation. Roughly, the calculator conjoins the components of the abstract class with the retrieve relation, and existentially quantifies over the abstract constants and states in the resulting formula. The construction for methods relies on the the "data refinement calculator" that Josephs [6] defines to construct a concrete operation schema as a data refinement of an abstract one in Z.

In HOL, the function *SimBy* maps a given class and a retrieve relation to a new class.

$$SimBy :: [\,(\alpha, \kappa, \sigma, \iota, \omega)\, Class, (\kappa, \kappa', \sigma, \sigma')\, retrv\,] \rightarrow (\alpha, \kappa', \sigma', \iota, \omega)\, Class$$

The components of *SimBy Cls R* are defined as follows:

- The constant and state schemas of the calculated subclass are defined in terms of the conjunction of the constant and state schemas of *Cls* and the retrieve relation *R*, and hiding the abstract entities.
- For each method M of *Cls*, there is a method M' of the calculated subclass. The precondition of M' contains all pairs of constants and states for which abstract constants and states exist that satisfy the precondition of M. For those states, the concrete operation relates the pre- and post-states for which abstract ones exist that satisfy M.

$$\begin{aligned} M' \stackrel{\mathsf{def}}{=} \; & (\exists\; c\, s.\; \mathsf{pre}\; M\, c\, s\, inp \wedge R\, c\, cc\, s\, cs) \wedge \\ & \quad (\forall\; c\, s.\; \mathsf{pre}\; M\, c\, s\, inp \wedge R\, c\, cc\, s\, cs \Rightarrow \\ & \qquad (\exists\; s'.\; R\, c\, cc\, s'\, cs' \wedge M\, c\, s\, s'\, inp\, out))) \end{aligned}$$

- The history schema of the calculated subclass is defined similarly to the constant and state schemas. The retrieve relation *R* establishes correspondences between the abstract and the concrete pre- and post-states.

Josephs [6] already notes that the data refinement calculator for Z does not always yield a refinement. For general retrieve relations *R*, it is still necessary to check the applicability condition relating the preconditions of abstract and concrete operations. (c.f. Section 5.2). If, however, the retrieve relation *R* is a function from the concrete onto the abstract states, then the calculated subclass *SimBy Cls R* behaviorally conforms to *Cls*. That condition on *R* corresponds to the adequacy condition of data reification in VDM [5], which admits functional abstractions only. The predicate *is_surj_fun* defines that condition formally.

$$\frac{is\_surj\_fun\; R\; (cma\;\; Cls)\; (sma\;\; Cls) \quad CVs = IdentList^{io}_{\cong}\;\; Cls \quad M = id_m\; (ids\;\; Cls)}{conforms\; CVs\; M\; R\; Cls\; (SimBy\; Cls\; R)} \tag{3}$$

Theorem (3) is of great practical importance. It establishes once and for all the – relatively simple – conditions under which subclass calculation implies behavioral conformance. As the next section will show, subclass calculation is suitable to define the specifications of most classes of the Eiffel Base Libraries. Theorem (3) helps a lot to simplify proofs of behavioral conformance between those specifications – compared to proofs against the definition of behavioral conformance. In particular, the theorem does not require the proof to consider the method suites of the two classes, which otherwise is the major work in a proof of behavioral conformance.

7.2 An Alternative Representation of *Collection*

Using the subclass calculator, we can build a specification of *Collection* from the specification of *Container*. The internal model of *Container* is a set of the items in the container. The internal model we use in *Collection*, however, is a bag (c.f. Figure 4). Given a relation R_{Coll}^{Cont} that equates the internal model of *Container* to the domain of c in *Collection*, we can construct the specification of *Collection* by calculating a subclass of *Container* according to R_{Coll}^{Cont} and augmenting the resulting class with the extra methods of *Collection*:

$$
\begin{aligned}
Collection\, G \stackrel{\mathsf{def}}{=}\ & (SimBy\,(Container\, G)\, R_{Coll}^{Cont}) \\
& \boxplus\, (''prunable'', Prunable\, G) \\
& \boxplus\, \ldots \boxplus\, (''prune\_all'', Prune\_all\, G)
\end{aligned}
$$

We used that way of constructing specifications to economically specify the classes of the Eiffel Base Libraries shown in Figure 1.

8 Conformance Analysis of the Eiffel Base Libraries

We followed a five step procedure to specify classes and analyze behavioral conformance with Isabelle/HOL (see [4] for the technical details):

1. Specify a class in Object-Z-like notation; represent the specification as a class schema in HOL. For the representation, decide whether to construct the class schema independently of other class schemas, or to use the subclass calculator to construct the class schema from another one. In the latter case, determine an appropriate retrieve relation.
2. Prove the internal consistency of the new class schema, and derive unconditional equations to select the components of the class schema.
3. Derive the preconditions of the methods of the new class schema.
4. Determine retrieve relations between the new class schema and the specifications of the immediate ancestors of the class in the inheritance hierarchy, if such relations exist. Analyze whether or not the retrieve relations are surjective functions.
5. Prove behavioral conformance of the new class with its immediate ancestors.

Step 2 is strictly necessary to be able to prove non-trivial propositions about the specific class schema. Deriving the preconditions in Step 3, however, mainly served to validate the specifications, because the retrieve relations we considered all were surjective functions such that we could apply Theorem (3) to prove behavioral conformance in Step 5. If a class schema is constructed by subclass calculation in Step 1, then Step 4 only needs to analyze the function property of the relation used in Step 1.

Figure 7 shows the subgraph of the *CONTAINER* hierarchy that we considered. We represented the classes in HOL that are shaded gray in the figure. The class schemas of the *COLLECTION* and *TRAVERSABLE* hierarchies are defined in terms of their immediate ancestors in those hierarchies. The class schema *BAG* is a simple extension of

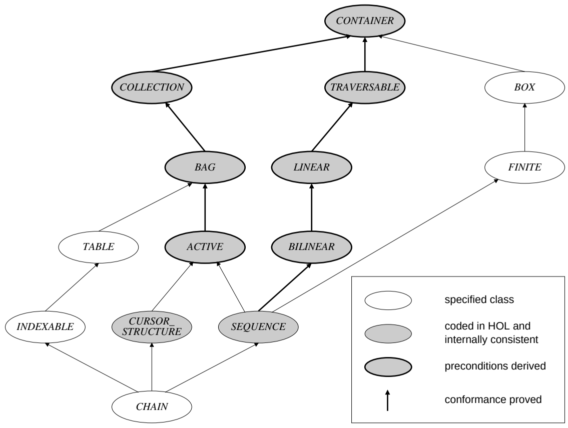

Fig. 7. The specified and analyzed subgraph of the *CONTAINER* hierarchy.

COLLECTION by new methods (using ⊞). All other constructions in the two hierarchies are subclass calculations that are extended by new methods and observers for new attributes. The transitions from *CONTAINER* to *COLLECTION* and to *TRAVERSABLE* involve changes of the mathematical model of the container: in *CONTAINER*, it is a set, whereas it is a bag in *COLLECTION* and a sequence in *TRAVERSABLE*. The constructions of the other classes only extend the state with new attributes. The retrieve relations for those constructions are therefore projections from the extended concrete state to the abstract state.

To derive the internal consistency (Step 2) of the class schemas shaded gray in the figure, we used a tailor-made tactic (c.f. Section 4.2). Deriving the preconditions (Step 3) of the methods of all classes except *CURSOR_STRUCTURE* and *SEQUENCE*, served to validate the specifications. To derive the preconditions, we used a tactic to apply the one-point-rule in a sophisticated way, which included handling subclass calculations. All retrieve relations used in the subclass calculations, and also the one used to analyze the inheritance of *ACTIVE* by *SEQUENCE*, are surjective functions (Step 4). This allowed us to rely on Theorem (3) when proving behavioral conformance relations (Step 5).

To achieve a high degree of automation, we implemented two tactics to prove conformance for the typical construction used in the library. Given an abstract class schema *ACls* that construction consists of two steps: first, we defined an intermediate class schema

$$BCls = SimBy\, ACls\, R$$

by subclass calculation; second, we extended the class schema *BCls* by a number of methods (possibly overriding method definitions of *ACls*):

$$CCls = BCls \boxplus (n_1, Op_1) \ldots \boxplus (n_k, Op_k)$$

Our tactics consider conformance proofs for subclass calculation and extension by methods, respectively. They automatically synthesize appropriate I/O conversions and handle all proof obligations related to class schema constructions. Using the theorems about subclass calculation of Section 7 and similar theorems about extension of a class schema by a method, they reduce a conformance proposition to verification conditions on (the $\mathcal{HOL}$-$\mathcal{Z}$ representations of) the Z schemas that make up the specification of the class. They present only those verification conditions to the user of Isabelle to prove them interactively. Thus, the two tactics establish behavioral conformance of *BCls* to *ACls*, and of *CCls* to *BCls*. An application of Theorem (2), the transitivity of behavioral conformance, completes the proof of conformance of *CCls* to *ACls*.

9 Conclusions

We conclude with an evaluation of our work on the Eiffel Base Libraries. First, we discuss our experience with the libraries and their specification.. Then we draw some conclusions about the suitability of our framework for applying it to the libraries.

The Eiffel Base Libraries. Setting up the specifications of classes of the library and analyzing their relationships required a detailed examination of their informal description, their specification by assertions, and their implementation code, too. Often, Meyer's [9] informal description of the purpose of classes and their features helped to clarify their functionality, but occasionally it was necessary to inspect the code not only of the class in question but also of its descendants, in particular the effective ones at the leaves of the inheritance hierarchy, to determine the precise function of a feature in a class. This was especially necessary to distinguish between similar features, such as *extend*, *put*, and *replace*, or *prune* and *remove*.

Class specifications concisely describe the possible behavior of objects. The resulting specifications capture not only the information that is directly present in the class description (by assertions and code), which is quite sparse for the deferred classes near the root of the class hierarchy. Capturing information about the descendants of a class, too, the specifications describe the possible "evolution" of features (by late binding) when moving down in the inheritance hierarchy toward effective classes. Thus, the specifications disambiguate the functionality of features and describe – in a single specification – not just a single class but the cluster of classes that inherit from that class. A class specification concisely captures the possible behavior of the objects of a class. Behaviorally conforming subclasses assumed, the specification of a method completely describes the possible effects of an invocation of the method at the presence of late binding and polymorphism. It is not necessary to inspect the different implementations of the method in the descendants of the class.

Inheritance induces behavioral conformance. All inheritance relations in the part of the libraries that we analyzed induce behavioral conformance. Therefore, it is safe to use objects of those classes polymorphically, accessing them through an abstract interface, as it is provided by the deferred classes in the library. This justifies to consider the entire

Table 1. Analysis of Class Specifications.

Class	Number of methods		Proof time (in min.)	
	new	total	Internal consistency	Precondition analysis
CONTAINER	3	3	2	1
COLLECTION	8	11	17	10
BAG	2	12	2	4
ACTIVE	5	17	13	41
CURSOR_STRUCTURE	11	28	125	—
TRAVERSABLE	3	6	3	5
LINEAR	8	14	20	50
BILINEAR	2	16	10	25
SEQUENCE	14	30	130	—

Performance measured on a SUN-Enterprise 3000
with 4 UltraSparc-I processors (167 MHz) and 4 GB main memory.

inheritance graph shown in Figure 7, not just the interfaces of the classes, as part of the interface of the Eiffel Base Libraries to their clients.

Retrieve relations are simple. The retrieve relations that relate immediate descendants to their ancestors are relatively simple. Overriding the specifications of inherited methods is rarely necessary. These observations indicate that the individual classes encapsulate few well-defined "properties" or "design decisions" rather than merging many in one class.

Partial conformance can describe the effect of hiding methods. The inheritance relations that are part of the interface of the libraries to clients are designed to ensure behavioral conformance. Hiding features avoids duplicating classes in some places. In a strict sense, hiding destroys conformance, because the signature morphism from the abstract to the concrete class is necessarily partial. But, as Frick et. al. [3] observe, considering all methods of a class often is unnecessary to ensure safe polymorphic uses: the conditions of behavioral conformance need only hold for the methods that a context actually uses. It is, of course, easy to define a weaker notion of conformance in our framework that does not require the signature morphism to be total.

The Framework. Table 1 gives an impression of the size and complexity of the application of our tactics to the classes of Figure 7. It provides data on the sizes of the classes and on the performance of the tactics establishing internal consistency and deriving preconditions. The first two columns show the number of methods that are newly defined in a class and the total number of methods including the ones "inherited" from its parent class (by a subclass calculation). It is common for a class to extend its parent class with approximately ten new methods.

Table 2. Performance for proving behavioral conformance.

Conformance		Proof time in min.
COLLECTION – CONTAINER		2
BAG – COLLECTION		4
ACTIVE – BAG		3
TRAVERSABLE – CONTAINER		4
LINEAR – TRAVERSABLE		4
BILINEAR – LINEAR		4
SEQUENCE – BILINEAR		4
SEQUENCE – ACTIVE	(incomplete)	(20)

New methods determine the effort of internal consistency and precondition analysis. The third column shows performance figures for proving the class schemas internally consistent. The figures increase proportional to the number of *new* methods, because the proofs were carried out starting with *CONTAINER* and moving down in the inheritance hierarchy. Thus, the consistency lemmas for the ancestors of a class were available for proving the internal consistency of the class.

Similarly, the figures for the performance of the precondition analyses indicate that preconditions have been derived for new methods only. Those preconditions served to validate the specifications. The subsequent conformance proofs did not need to consider preconditions, because all retrieve relations were surjective functions, and therefore, the proofs could employ Theorem (3).

Abstract theorems are essential for feasibility. Table 2 shows performance figures for the proofs of behavioral conformance. The conformance proofs required between two and four minutes of computation time. The possibility to apply general theorems, such as Theorem (3) by appropriate tactics helps to keep the effort for conformance proofs low. Isabelle needs two seconds to resolve a proof goal with the transitivity Theorem (2). The theorem about conformance of subclass calculations alone allows for a speed-up of a factor of five in the first conformance proof: Isabelle needs ten minutes to prove that a specification of *COLLECTION* that is *not* based on the one of *CONTAINER* behaviorally conforms to *CONTAINER*.

Non-functional retrieve relations are necessary. There are inheritance relations in the library that require non-functional retrieve relations. An example is the relation between *COLLECTION* and its immediate descendant *SET* (c.f. Figure 1). The model of a container in the specification of *COLLECTION* is a bag, because the specification of *prune* must allow the method to remove some but not all occurrences of the item v. In the specification of *SET*, however, the model of the container is a set (as it is in the specification of *CONTAINER*). To prove behavioral conformance of *SET* to *COLLECTION*, we need a non-functional retrieve relation that relates a set s to all bags b for which

$\mathsf{dom}\ b = s$. That observation also illustrates a difference between specifications of classes and specifications of "classical" abstract data types: in an object-oriented context, deferred classes such as *COLLECTION* do not provide a "complete" interface to their objects as an abstract data type does but they provide a restricted "view" of an object of an effected class that has a much larger interface.

Multiple inheritance is hard. In contrast to the other proofs of conformance, the definitions of *SEQUENCE* and *ACTIVE* are independent. We approached the proof of that conformance relation by simplifying the method schemas of the two classes to obtain "explicit" predicates. The figure at the bottom row of Table 2 shows the computation time to derive some simplified forms of the method schemas of *SEQUENCE*. Like for the precondition analyses in Table 1, the need to eliminate the numerous existential quantifications in the definition of the subclass calculator is responsible for the high cost of those simplifications.

The performance figures of Table 2 indicate that the theory of subclass calculations that we sketched in Section 7 is suitable to handle the patterns of single inheritance that we identified in Section 6. To investigate behavioral conformance for multiple inheritance properly, however, we need a more elaborate theory. A "step-wise" approach that, in the example, would establish relations between *SEQUENCE* and the ancestors of *ACTIVE*, moving down in the inheritance hierarchy, seems promising to prove the behavioral conformance induced by multiple inheritance. General theorems about "pushouts" of conformance relations would support such an approach. Our framework is expressive enough to state and prove such theorems.

We consider our work on the Eiffel Base Libraries as a substantial application of our framework to a mature, practically used software product. The specifications of the part of the Eiffel Base Libraries we worked on total about three thousand lines of Isabelle theory files. The proofs about those specifications consist of another three thousand lines of proof scripts written in ML. Although there is potential for optimization, our work shows the practicality of using a mature theorem prover such as Isabelle to develop an abstract theory of object oriented specification and apply it to reason about the specifications of software components.

References

[1] P. America. Designing an object-oriented programming language with behavioral subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, LNCS 489. Springer-Verlag, 1991.

[2] W. R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *Proc. OOPSLA'92*, volume 27(10) of *ACM SIGPLAN Notices*, pages 1–15, 1992.

[3] A. Frick, R. Neumann, and W. Zimmermann. Eine Methode zur Konstruktion robuster Klassenhierarchien. In *Softwaretechnik 96*, pages 16–23, 1996.

[4] S. Helke. Maschinengestützte Analyse der Untertypbeziehungen zwischen Klassen der Eiffel Datenstrukturbibliothek. Master's thesis, Dept. of Computer Science, Technical University of Berlin, 1998.

[5] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 2nd edition, 1990.

[6] M. B. Josephs. The data refinement calculator for Z specifications. *Information Processing Letters*, 27:29–33, 1988.

[7] Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher-Order Logics*, LNCS 1125, pages 283–298. Springer-Verlag, 1996.

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

[9] B. Meyer. *Reusable Software*. Prentice Hall, 1994.

[10] S. Näher. LEDA manual. version 3.0. Technical Report MPI-I-93-109, Max-Planck-Institut für Informatik, Saarbrücken, February 1993.

[11] L. C. Paulson. *Isabelle – A Generic Theorem Prover.* LNCS 828. Springer-Verlag, 1994.

[12] T. Santen. A theory of structured model-based specifications in Isabelle/HOL. In E. L. Gunter and A. Felty, editors, *Proc. International Conference on Theorem Proving in Higher Order Logics*, LNCS 1275, pages 243–258. Springer-Verlag, 1997.

[13] T. Santen. On the semantic relation of Z and HOL. In J. Bowen and A. Fett, editors, *ZUM'98: The Z Formal Specification Notation*, LNCS 1493, pages 96–115. Springer-Verlag, 1998.

[14] T. Santen. Isomorphisms – a link between the shallow and the deep. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics*, LNCS 1690, pages 37–54. Springer-Verlag, 1999.

[15] T. Santen. *A Mechanized Logical Model of Z and Object-Oriented Specification*. Shaker-Verlag, 2000. Dissertation, Fachbereich Informatik, Technische Universität Berlin, (1999).

[16] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 1999.

[17] J. M. Spivey. *The Z Notation – A Reference Manual.* Prentice Hall, 2nd edition, 1992.