Specifying Reusable Components*

Nadia Polikarpova, Carlo A. Furia, and Bertrand Meyer

Chair of Software Engineering, ETH Zurich, Switzerland
{nadia.polikarpova,carlo.furia,bertrand.meyer}@inf.ethz.ch

Abstract. Reusable software components need expressive specifications. This paper outlines a rigorous foundation of *model-based contracts*, a method to equip classes with strong contracts that support accurate design, implementation, and formal verification of reusable components. Model-based contracts conservatively extend the classic Design by Contract approach with a notion of model, which underpins the precise definitions of such concepts as abstract object equivalence and specification completeness. Experiments applying model-based contracts to libraries of data structures suggest that the method enables accurate specification of practical software.

1 Introduction

The rationale for precise software specifications involves several well-known arguments; in particular, specifications help understand the problem before building a solution, and they are necessary for verifying implementations. In the context of reusable software components, there is another essential application of specifications: providing client programmers with an accurate description of the API. Design by Contract techniques [9] enable authors of reusable modules to equip them with specification elements known as "contracts" (routine pre and postconditions, class invariants).

While specification methods primarily intended for formal verification typically use notations based on mathematics, Design by Contract approaches, such as Eiffel [9], JML [8] and Spec$^{\#}$ [2], rely instead on an assertion language embedded in the programming language. This adds a significant benefit: assertions can be *evaluated* during execution. As a consequence, contracts have played a major role in testing, especially for Eiffel, where an advanced testing environment, AutoTest [10], takes advantage of executable specifications for automatic test generation. More generally, Eiffel programmers routinely rely on runtime contract evaluation for testing and debugging. Another practical benefit of Design by Contract is approachability: programmers do not need to learn a separate notation for specifications.

These advantages of contracts have traditionally come at a price: expressiveness. The lack of an advanced mathematical notation makes it harder to express the full specification of components (see examples in Section 2). An extensive study [3] indicates that in practice Eiffel classes contain many contracts, but they cover only part of the intended functional properties.

Can we get all the advanced benefits of expressive formal specifications while retaining an executable specification language that does not introduce complex notation?

* This research has been funded in part by Hasler Stiftung, ManCom project, grant # 2146.

The present paper proposes a positive answer, based on the idea of *models*. Specifications, in this approach, are expressed in terms of the abstract model of a class, defined through one or more *model queries*. Model queries return instances of *model classes*: direct translations of mathematical concepts (such as sets or sequences) into the programming language.

The idea of using model classes and model queries in contracts is not new; our previous work [13, 12] and, among others, JML [8] introduced the concepts and provided libraries of model classes. Two main contributions of the present paper are developing a *rigorous and systematic approach* to model-based specifications and confirming the applicability of the approach through two realistic *case studies*.

Section 3 shows how the interface of a class defines unambiguously a notion of *abstract object space* that determines the class model. Section 3 also outlines precise guidelines for writing contracts that refer to model queries. The guidelines come with a definition of specification *completeness* (with respect to the model). The definition is formal, yet amenable to informal reasoning; it is practically useful in assessing whether a contract is sufficiently detailed or is likely omitting some important details.

Section 4 describes two case studies applying model-based contracts to EiffelBase [4], Eiffel's standard collection of fundamental data structures, and to the development of EiffelBase2, intended to replace EiffelBase and to contribute to the Verified Software Repository [14]. The results show that the method is successful in delivering well-engineered components with expressive — usually complete — specifications. Most advantages of standard Design by Contract are retained, while pushing a more accurate evaluation of design choices and an impeccable definition of interfaces. The executability of most model classes supports the reuse of Eiffel's AutoTest infrastructure with more expressive contracts, which boosts the effectiveness of automated testing in finding defects in production software.

For lack of space, the rest of the paper omits some examples and references, which are available in an extended version [11].

2 Motivation and Overview

Design by Contract uses the same notation for expressions in the implementation and in the specification. This restriction ultimately impedes the formalization and verification of full functional correctness, as demonstrated below on two examples from the EiffelBase library [4].

Lines 1–13 in Figure 1 show a portion of class LINKED_LIST. Features (members) *count* and *index* record respectively the number of elements stored in the list and the position of the internal cursor. The routine *put_right* inserts an element, *v*, to the right of the cursor. The precondition of the routine (**require**) demands that the cursor not be after the last element. The postcondition (**ensure**) asserts that inserting an element increments *count* by one but does not change *index*. This is correct, but it does not capture the essence of the insertion semantics: in particular, it doesn't prevent the implementation from changing elements that were in the list before.

Expressing such complex properties is impossible or exceedingly complicated with the standard assertion language; as a result most specifications are *incomplete* in the

```
1  class LINKED_LIST [G]
2    put_right (v: G)
3      -- Add 'v' to the right of cursor.
4      require  0 ≤ index ≤ count
5      do  ...
6      ensure
7        count = old count + 1
8        index = old index
9      end
10
11   count: INTEGER -- Number of elements
12   index: INTEGER -- Current cursor position
13 end
14
15 class TABLE [G, K]
16   put (v: G ; k: K)
17     -- Associate value 'v' with key 'k'.
18   require  ...
19   deferred
20   end
21 end
```

Fig. 1. Snippets from the EiffelBase classes LINKED_LIST (lines 1–13) and TABLE (lines 15–21).

sense that they fail to capture precisely the semantics of routines. Specification weakness hinders formal verification in two ways. First, establishing weak postconditions is simple, but confidence in the full functional correctness of a verified routine will be low. Second, weak contracts affect negatively verification modularity: it is impossible to establish what a routine r achieves, if r calls another routine s whose contract is not strong enough to document its effect within r precisely.

Weak assertions limit the potential of many other applications of Design by Contract. Specifications, for example, should document the abstract semantics of operations in deferred (abstract) classes. Incomplete contracts cannot fully do so; as a result, programmers have fewer safeguards to prevent inconsistencies in the design and fewer chances to make deferred classes useful to clients through polymorphism.

The feature *put* in class TABLE (lines 16–20 in Figure 1) is an example of such a phenomenon. It is unclear how to express the abstract semantics of *put* with standard contracts. In particular, the absence of a postcondition leaves it undefined what should happen when an element is inserted with a key that is already present: should *put* replace the existing element with the new one or leave the table unchanged? Indeed, some child classes of TABLE, such as class ARRAY, implement *put* with a replacement semantics, while others, such as class HASH_TABLE, disallow overriding of preexisting mappings with *put*. HASH_TABLE even introduces another feature *force* that implements the replacement semantics. This obscures the behavior of routines to clients and makes it questionable whether *put* has been introduced at the right point in the inheritance hierarchy.

Enhancing Design by Contract with Models. This paper presents an extension of Design by Contract that addresses the aforementioned problems. The extension conservatively enhances the standard approach with *model classes*: immutable classes representing mathematical concepts that provide for more expressive specifications. Wrapping mathematical entities with classes supports richer contracts without any need to extend the Design by Contract notation, familiar to programmers. Contracts using model classes are called *model-based contracts*.

Figure 2 shows an extensions of the examples in Figure 1 with model-based contracts. LINKED_LIST is augmented with a query *sequence* that returns an instance of class MML_SEQUENCE, a model class representing a mathematical sequence; the implementation, omitted for brevity, builds *sequence* according to the actual content of the

```
1  note model: sequence, index
2  class LINKED_LIST [G]
3    sequence: MML_SEQUENCE [G]
4      -- Sequence of elements
5      do ... end
6
7    index: INTEGER -- Current cursor position
8
9    put_right (v: G)
10     -- Add 'v' to the right of cursor.
11     require  0 ≤ index ≤ count
12     do ...
13     ensure
14       sequence = old ( sequence.front (index).
15         extended (v) + sequence.tail (index + 1) )
16       index = old index
17     end
18 end
```

```
19 note model: map
20 class TABLE [G, K]
21   map: MML_MAP [G, K]
22     -- Map of keys to values
23   deferred end
24
25   put (v: G ; k: K)
26     -- Associate value 'v' with key 'k'.
27     require map.domain [k]
28     deferred
29     ensure
30       map = old map.replaced_at (k, v)
31     end
32 end
```

Fig. 2. Classes LINKED_LIST (left) and TABLE (right) with model-based contracts.

list. The meta-annotation **note** declares the two features *sequence* and *index* as the model of the class; all contracts will be written in terms of them. In particular, the postcondition of *put_right* can precisely describe the effect of the routine: the new *sequence* is the concatenation of the **old** *sequence* up to *index*, extended with element *v*, with the tail of the **old** *sequence* starting after *index*. We can assert that the new postcondition — including the clause about *index* — is *complete* with respect to the model of the class, because it defines the effect of *put_right* on the abstract model unambiguously. This notion of completeness is a powerful guide to writing accurate specification that makes for well-defined interfaces and verifiable classes.

The mathematical notion of a *map* — encapsulated by the model class MML_MAP — is a natural model for the class TABLE. Feature *map* cannot have an implementation yet, because TABLE is deferred and hence it is not committed to any representation of data. Nonetheless, availability of the model makes it possible to write a complete postcondition of *put* already at this abstract level, which in turn requires to commit to a specific semantics for insertion. The example in Figure 2 chooses the replacement semantics; correspondingly, all children of TABLE will have to conform to this semantics, guaranteeing a coherent reuse of TABLE throughout the class hierarchy.

3 Foundations of Model-Based Contracts

3.1 Specifying Classes with Models

Interfaces, References, and Objects. A class C denotes a collection of objects. Expressions such as $o : C$ define o as a reference to an object of class C; the notation is overloaded for conciseness, so that occurrences of o can denote the object it references or the reference itself, according to the context. Each class C defines a notion of *reference* equality $\equiv_C$ and of *object* equality $\stackrel{\circ}{=}_C$; both are equivalence relations. Two references $o_1, o_2 : C$ can be *reference equal* (written $o_1 \equiv_C o_2$) or *object equal* (written

```
1  note model: sequence, index
2  class LINKED_LIST [G]
3  ...
4    item: G
5        -- Value at cursor position
6      require sequence.domain [index]
7      do ...
8      ensure
9        Result = sequence [index]
10     end
11   duplicate (n: INTEGER): LINKED_LIST [G]
12       -- A copy of at most 'n' elements
13       -- starting at cursor position
14     require n ≥ 0
15     do ...
16     ensure
17       Result.sequence = sequence.interval (index, index + n − 1)
18       Result.index = 0
19     end
20 end
```

Fig. 3. Snippets of class LINKED_LIST with model-based contracts (continued from Figure 2).

$o_1 \triangleq_C o_2$). Reference equality is meant to capture whether o_1 and o_2 are aliases for the same memory location, whereas object equality is meant to hold for (possibly) distinct copies of the same actual content.

The principle of information hiding prescribes that each class define an interface [9]. It is good practice to partition features into queries and commands; queries are functions of the object state, whereas commands modify the object state but do not return any value. $I_C = Q_C \cup M_C$ denotes the interface of a class C partitioned in queries Q_C and commands M_C.[1] It is convenient to partition all queries into *value-bound* queries and *reference-bound* queries. Value-bound queries create fresh objects to return (or more generally objects that were unknown to the client before calling the query), whereas reference-bound queries give the client direct access, through a reference, to preexisting objects. In other words, clients of a value-bound query should not rely on the identity of its result. The classification in value-bound and reference-bound extends naturally to *arguments* of features.

Example 1. Query *item* of class LINKED_LIST (Figure 3) is reference-bound, as the client receives a reference to the same memory location that was earlier inserted in the list. Query *duplicate* is instead value-bound, as it returns a copy of a portion of the list.

Abstract Object Space. The interface I_C induces an equivalence relation $\asymp_C$ over objects of class C called *abstract equality* and defined as follows: $o_1 \asymp_C o_2$ holds for $o_1, o_2 : C$ iff for any applicable sequence of calls to commands $m_1, m_2, \ldots \in M_C$ and a query $q \in Q_C$ returning objects of some class T, the qualified calls $o_1.m_1; o_1.m_2; \cdots$ and $o_2.m_1; o_2.m_2; \cdots$ (with identical actual arguments where appropriate) drive o_1 and o_2 in states such that if q is reference-bound then $o_1.q \equiv_T o_2.q$, and if q is value-bound then $o_1.q \triangleq_T o_2.q$. Intuitively, two objects are equivalent with respect to $\asymp_C$ if a client cannot distinguish them by any sequence of calls to public features. Abstract equality defines an *abstract object space*: the quotient set $A_C = C/\asymp_C$ of C by $\asymp_C$. As a consequence, two objects are equivalent w.r.t. $\asymp_C$ iff they have the same *abstract (object) state*. Any concrete set that is isomorphic to A_C is called a *model* of C.

[1] Constructors need no special treatment and can be modeled as queries returning new objects.

Example 2. Consider a class implementing a *queue*. If the *remove* operation were not part of the interface, no element in the queue would be accessible to clients but the one that was enqueued first; the model of such a class would be $\mathbb{N} \times G$: a set of pairs recording the number of elements and the head element of generic type G. Including *remove* in the interface, as it usually is the case for queues, allows clients to access all the elements in the order of insertion. Hence, two queues with full interfaces are indistinguishable iff they have all the same elements in the same order, which makes G^* (sequences of elements) a model for queues.

Model Classes. The model of a class C is expressed as a tuple $D_C = D_C^1 \times D_C^2 \times \ldots \times D_C^n$ of *model classes*.Model classes are immutable classes designed for specification purposes; essentially, they are wrappers of rigorously defined mathematical entities: elementary sorts such as Booleans, integers, and object references, as well as more complex structures such as sets, bags, relations, maps, and sequences. The Mathematical Model Library (MML) [12] provides a variety of such model classes, equipped with features that correspond to common operations on the mathematical structures they represent, including first-order quantification. For example, class MML_SET models sets of elements of homogeneous type; it includes features such as membership test and quantification.

Model Queries. Every class C provides a collection of public *model queries* $S_C = s_C^1, s_C^2, \ldots, s_C^n$, one for each component model class in D_C. Each model query s_C^i returns an instance of the corresponding model class D_C^i that represents the current value of the i-th component of the model. Clauses in the class invariant can constrain the values of the model queries to match precisely the abstract object space. Consider, for example, the model of LINKED_LIST (Figure 2): model query *index*: INTEGER returning the cursor position should be constrained by an invariant clause $0 \leq \mathit{index} \leq \mathit{sequence.count} + 1$. A meta-annotation **note** model: $s_C^1, s_C^2, \ldots$ lists all model queries of the class.

It is likely that some model queries (such as *index* in the example above) are already used in the implementation before models are added explicitly; additional model queries (such as *sequence*) return the remaining components of the model for specification purposes. Our approach prefers to implement such additional model queries as functions rather than attributes. This choice facilitates a purely descriptive usage of references to model queries in specifications. In other words, instead of augmenting routine bodies with bookkeeping instructions that update model attributes, routine postconditions are extended with clauses that describe the new value returned by model queries in terms of the old one. This has the advantage of enforcing a cleaner division between implementation and specification, while better modularizing the latter at routine level (properties of model attributes are typically gathered in the class invariant).

Model-Based Contracts. Let C be a class equipped with model queries and let its interface I_C be partitioned into queries Q_C and commands M_C. Q_C now includes the model queries $S_C \subseteq Q_C$ together with other queries $R_C = Q_C \setminus S_C$. The rest of the section contains guidelines to writing model-based contracts for commands in M_C and queries in R_C.

The *precondition* of a feature is a constraint on the abstract states of its value-bound arguments and, possibly, on the actual references to its reference-bound arguments.

```
note model: bag
class COLLECTION [G]
  bag: MML_BAG [G]

  is_empty: BOOLEAN
    ensure Result = bag.is_empty end

  wipe_out
    ensure bag.is_empty end

  put (v: G)
    ensure bag = old bag.extended (v) end
end
```

```
note model: sequence
class DISPENSER [G]
inherit COLLECTION [G]

  sequence: MML_SEQUENCE [G]

  invariant
    bag.domain = sequence.range
    bag.domain.for_all ( agent (x: G): BOOLEAN
                  bag [x] = sequence.occurrences (x) )
end
```

Fig. 4. Snippets of classes COLLECTION (left) and DISPENSER (right) with model-based contracts.

The target object, in particular, can be considered an implicit value-bound argument. For example, the precondition *map.domain* [*k*] of feature *put* in class TABLE (Figure 2), refers to the abstract state of the target object, given by the model query *map*, and to its actual reference-bound argument *k*.

Postconditions should refer to abstract states only through model queries. This emphasizes the components of the abstract state that a feature modifies or relies upon, which in turn facilitates understanding and reasoning on the semantics of a feature.

The *postcondition of a command* defines a relation between the prestate and the poststate of its arguments and the target object. More precisely, the postcondition mentions only abstract values of its value-bound arguments and possibly the actual references to its reference-bound arguments; the target object is considered value-bound both in the prestate and in the poststate.

It is common that a command only affects a few components of the abstract state and leaves all the others unchanged. Accordingly, the *closed world assumption* is convenient: the value of any model query $s \in S_C$ that is not mentioned in the postcondition is assumed not to be modified by the command, as if s = **old** s were a clause of the postcondition. When the closed world assumption is wrong, explicit clauses in the postcondition should establish the correct semantics.

The *postcondition of a query* defines the result as a function of its arguments and the target object (with the usual discipline of mentioning only abstract values of value-bound arguments and target object and possibly actual references to reference-bound arguments). Value-bound queries define the abstract state of the result, whereas reference-bound queries describe an actual reference to it. For example, compare the postcondition of the reference-bound query *item* from class LINKED_LIST (Figure 3) with the postcondition of the value-bound query *duplicate* in the same class.

A clear-cut separation between queries and commands assumes *abstract purity* for all queries: executing a query leaves the abstract state of all its arguments and of the target object unchanged.

Inheritance and Model-Based Contracts. A class C' that inherits from a parent class C may or may not re-use C's model queries to represent its own abstract state. For every model query $s_C \in S_C$ of the parent class that is not among the child's model

queries $S_{C'}$, C' should provide a *linking invariant*: a formula that defines the value returned by s_C in terms of the values returned by the model queries $S_{C'}$ of the inheriting class. This guarantees that the new model is indeed a specialization of the previous model, in accordance with the notion of sub-typing inheritance.

A properly defined linking invariant ensures that every inherited feature has a definite semantics in terms of the new model. However, the new semantics may be weaker; that is, incompleteness is introduced (see Section 3.2).

Example 3. Consider class COLLECTION in Figure 4, a generic container of elements whose model is a bag. Class DISPENSER inherits from COLLECTION and specializes it by introducing a notion of insertion order; correspondingly, its model is a sequence. The linking invariant of DISPENSER defines the value of the inherited model query *bag* in terms of the new model query *sequence* and ensures that the semantics of features *is_empty* and *wipe_out* is unambiguously defined also in DISPENSER. At the same time, the model-based contract of command *put* in COLLECTION and the linking invariant are insufficient to characterize the effects of *put* in DISPENSER, as the position within the sequence where the new element is inserted is irrelevant for the bag.

3.2 Completeness of Contracts

The notion of *completeness* for the specification of a class gives an indication of how accurate the contracts are with respect to the model of that class. An incomplete contract does not fully capture the effects of a feature, suggesting that the contract may be more detailed or, less commonly, that the model of the class — and hence its interface — is not abstract enough. A dual notion of soundness is definable along the same lines; for brevity, this section only presents the more interesting notion of completeness.

Completeness of a Model-Based Contract. The specification of a feature f of class C denotes two predicates $\mathbf{pre}_f$ and $\mathbf{post}_f$. $\mathbf{pre}_f$ represents the set of objects of class C that satisfy the precondition[2]. If f is a command, $\mathbf{post}_f$ has signature $C \times C$ and denotes the pairs of target objects before and after executing the command. If f is a query with return type T, $\mathbf{post}_f$ has signature $C \times T$; it denotes the pairs of target and returned objects for value-bound queries; and the pairs of target object and returned reference for reference-bound queries. In both cases $\mathbf{post}_f$ does not refer to the target object after executing the query because all queries are assumed to be abstractly pure.

- The *postcondition of a command* m is *complete* iff: for every $o, o'_1, o'_2 : C$ such that $\mathbf{pre}_m(o)$, $\mathbf{post}_m(o, o'_1)$, and $\mathbf{post}_m(o, o'_2)$ it is $o'_1 \asymp_C o'_2$.
- The *postcondition of a value-bound query* q is *complete* iff: for every $o : C$ and $t_1, t_2 : T$ such that $\mathbf{pre}_q(o)$, $\mathbf{post}_q(o, t_1)$, and $\mathbf{post}_q(o, t_2)$ it is $t_1 \asymp_T t_2$.
- The *postcondition of a reference-bound query* q is *complete* iff: for every $o : C$ and $t_1, t_2 : T$ such that $\mathbf{pre}_q(o)$, $\mathbf{post}_q(o, t_1)$, and $\mathbf{post}_q(o, t_2)$ it is $t_1 \equiv_T t_2$.

[2] For simplicity, the following definitions do not mention feature arguments; introducing them is, however, straightforward.

A postcondition is complete if all the pairs of objects that satisfy it are equivalent (according to the right model of equivalence). This means that the complete postcondition of a command defines its effect as a mathematical *function* (as apposed to a relation) from A_C to A_C. Similarly, the complete postcondition of a query defines the result as a *function* from A_C to A_T if the query is value-bound and to the set of references if the query is reference-bound.

Example 4. The contracts of features *is_empty*, *wipe_out*, and *put* in class COLLECTION (Figure 4) are complete; the postcondition of *put*, in particular, is complete as it defines the new value of *bag* uniquely. In the child class DISPENSER, however, the inherited postcondition of *put* becomes incomplete: the linking invariant does not uniquely define *sequence* from *bag*, hence inequivalent sequences (for example, one with *v* inserted at the beginning and another one with *v* at the end) satisfy the postcondition.

Completeness in Practice. As the previous example suggests, reasoning informally — but precisely — about completeness of model-based contracts is often straightforward and intuitive, especially if the guidelines of Section 3.1 have been followed. Completeness captures the uniqueness of the (abstract) state described by a postcondition, hence postconditions in the form **Result** = *exp* and similar, where *exp* is a side-effect free expression, are painless to check for completeness.

Example 5. Consider the following example, from class ARRAY whose model is a map.

```
1  fill (v: G ; l, u: INTEGER) -- Put 'v' at all positions in ['l', 'u'].
2    require map.domain [l] and map.domain [u]
3    ensure map.domain = old map.domain
4           ( map | {MML_INT_SET} [[l, u]] ).is_constant (v)
5           ( map | (map.domain − {MML_INT_SET} [[l, u]]) ) =
6                   old ( map | (map.domain − {MML_INT_SET} [[l, u]]) )
7    end
```

The following reasoning shows that the postcondition is complete: a map is uniquely defined by its domain and by a value for every key in the domain. The first clause of the postcondition (line 3) defines the domain completely. Then, let k be any key in the domain. If $k \in [l, u]$ then the second clause (line 4) defines *map* (*k*)= *v*; otherwise $k \notin [l, u]$, and the third clause (lines 5–6) postulates *map*(*k*) unchanged.

How useful is completeness in practice? As a norm, completeness is a valuable yardstick to evaluate whether the contracts are sufficiently detailed. This is not enough to guarantee that the contracts are correct — and meet the original requirements — but the yardstick is serviceable methodologically to focus on what a routine really achieves and how that is related to the abstract model. As a result, inconsistencies in specifications are less likely to occur, and the impossibility of systematically writing complete contracts is a strong indication that the model is incorrect, or the implementation is faulty. Either way, a warning is available before attempting a correctness proof.

While complete postconditions should be the norm, there are recurring cases where incomplete postconditions are unavoidable or even preferable. Two major sources of benign incompleteness are:

```
1 note mapped_to: "Sequence G"
2 class MML_SEQUENCE [G]
3   extended (x: G): MML_SEQUENCE[G]
4     -- Current sequence extended with 'x' at the end
5     note mapped_to: "Sequence.extended(Current, x)"
6     do ... end
7 ...
8 end
```

```
9  type Sequence T = [int] T ;
10 function Sequence.extended ⟨T⟩ (Sequence T, T)
11     returns (Sequence T);
12 axiom (∀ ⟨T⟩ s: Sequence T, x:T •
13   Sequence.extended(s, x) =s[Sequence.count(s)+1 := x]) ;
14 axiom (∀ ⟨T⟩ s: Sequence T, x: T •
15   Sequence.count(Sequence.extended(s, x)) =
16             Sequence.count(s)+1);
17 ...
```

Fig. 5. Snippets from class MML_SEQUENCE (left) and the corresponding Boogie theory (right).

- inherently *nondeterministic or stochastic* specifications and
- usage of *inheritance* to factor out common parts of (complete) specifications.

As an example of the latter consider class DISPENSER in Figure 4, a common parent of STACK and QUEUE. Based on the interface, its model has to be isomorphic to a sequence, but the postcondition of feature *put* cannot define the exact position of the new element in that sequence: a choice compatible with the semantics of STACK will be incompatible with QUEUE and vice versa.

In such cases, reasoning about completeness is still likely to improve the understanding of the classes and to question constructively the choices made for interfaces and inheritance hierarchies.

3.3 Verification: Proofs and Runtime Checking

This subsection outlines the main ideas behind using model-based contracts for verification with formal correctness proofs and with runtime checking for automated testing. Its goal is not to detail any particular proof or testing technique, but rather to sketch how to express the semantics of model-based contracts within standard verification frameworks.

Proofs. The *axiomatic* treatment of model classes [12] is quite natural: the semantics of a model class is defined directly in terms of a theory expressed in the underlying proof language, rather than with "special" contracts. The mapping often has the advantage of reusing theories that are optimized for effective usage with the proof engine of choice. In addition, the immutability (and value semantics) of model classes makes them very similar to mathematical structures and facilitates a straightforward translation into mathematical theories.

We are currently developing an accurate mapping of model classes and model-based contracts into Boogie [2]. First, the mapping introduces axiomatic definitions of MML model classes as Boogie theories; annotations in the form **note** mapped_to connect MML classes to the corresponding Boogie types (see Figure 5 for an example). Then, each model query in a class with model-based contracts maps to a Boogie function that references a representation of the heap. For example, the model query *sequence* in LINKED_LIST becomes **function** LinkedList.*sequence*(HeapType, *ref*) **returns** (Sequence *ref*). Axioms that connect the value returned by the function to the attributes of the

translated class are generated from the postconditions of model queries. The issue of providing such postconditions (*abstraction functions*) is outside the scope of current paper as here we are only concerned with interface specifications. Finally, model-based contracts are translated into Boogie formulas according to the mapped_to annotations in model classes.

Runtime Checking and Testing. Most model classes represent *finite* mathematical objects, such as sets of finite cardinality, sequences of finite length, and so on. All these classes can have an implementation of their operations which is executable in finite time; this supports the runtime checking of assertions that reference these model classes.

Testing techniques can leverage runtime checkable contracts to fully automate the testing process: generate objects by randomly calling constructors and commands; check the precondition of a routine on the generated objects to filter out valid inputs; execute the routine body on a valid input and check the validity of the postcondition on the result; any postcondition violation on a valid input is a fault in the routine.

This approach to contract-based testing has proved very effective at uncovering plenty of bugs in production code [10], hence it is an excellent "lightweight" precursor to correctness proofs. Contract-based testing, however, is only as good as the contracts are; the weak postconditions of traditional Design by Contract, in particular, leave many real faults undetected. Runtime checkable model-based contracts can help in this respect and boost the effectiveness of contract-based testing by providing more expressive specifications. Section 4 describes some testing experiments that support this claim.

4 Model-Based Contracts at Work

This section describes experiments in developing model-based contracts for real object-oriented software written in Eiffel. The experiments target two non-trivial case studies based on data-structure libraries (described in Section 4.1) with the goal of demonstrating that deploying model-based contracts is feasible, practical, and useful. Section 4.2 discusses the successes and limitations highlighted by the experiments.

4.1 Case Studies

The first case study targeted EiffelBase [4], a library of general-purpose data structures widely used in Eiffel programs; EiffelBase is representative of mature Eiffel code exploiting extensively traditional Design by Contract. We selected 7 classes from EiffelBase, for a total of 304 features (254 of them are public) over more that 5700 lines of code. The 7 classes include 3 widely used container data structures (ARRAY, ARRAYED_LIST, and LINKED_LIST) and 4 auxiliary classes. Our experiments systematically introduced models and conservatively augmented the contracts of all public features in these 7 classes with model-based specifications.

The second case study developed EiffelBase2, a new general-purpose data structure library. The design of EiffelBase2 is similar to that of its precursor EiffelBase;

EiffelBase2, however, has been developed from the start with expressive model-based specifications and with the ultimate goal of proving its full functional correctness — backward compatibility is not one of its primary aims. This implies that EiffelBase2 rediscusses and solves any deficiency and inconsistency in the design of EiffelBase that impedes achieving full functional correctness or hinders the full-fledged application of formal techniques. EiffelBase2 provides containers such as arrays, lists, sets, tables, stacks, queues, and binary trees; iterators to traverse these containers; and comparator objects to parametrize containers with respect to arbitrary equivalence and order relations on their elements. The current version of EiffelBase2 includes 46 classes with 460 features (403 of them are public) totaling about 5800 lines of code; these figures make EiffelBase2 a library of substantial size with realistic functionalities. The latest version of EiffelBase2 is available at `http://eiffelbase2.origo.ethz.ch`.

4.2 Results and Discussion

How Many Model Classes? Model-based contracts for EiffelBase used model classes for Booleans, integers, references, (finite) sets, relations, and sequences. EiffelBase2 additionally required (finite) maps, bags, and infinite maps and relations for special purposes (such as modeling comparator objects). This suggests that a moderate number of well-understood mathematical models suffices to specify a general-purpose library of data structures.

Determining to what extent this is generalizable to software other than libraries of general-purpose data structures is an open question which belongs to future work. Some problem domains may indeed require domain-specific model classes (e.g., real-valued functions, stochastic variables, finite-state machines), and application software that interacts with a complex environment may be less prone to accurate documentation with models. However, even if writing model-based contracts for such systems proved exceedingly complex, some formal model is required if the goal is formal verification. In this sense, focusing model-based contracts on library software is likely to have a great payoff through extensive reuse: the many clients of the reusable components can rely on expressive contracts not only as detailed documentation but also to express their own contracts and interfaces by combining a limited set of well-understood, highly dependable components.

Another interesting remark is that the correspondence between the limited number of model classes needed in our experiments and the classes using these model classes is far from trivial: reusable data structures are often more complex than the mathematical structures they implement. Consider, for example, class SET: EiffelBase2 sets are parameterized with respect to an equivalence relation, hence the model of SET is a pair of a mathematical set and a relation; correspondingly, the postcondition of feature *has* relies on the model by defining **Result** = **not** (*set* $*$ *relation.image_of* (*v*)).*is_empty*. Another significant example is BINARY_TREE: instead of introducing a new model class for trees or graphs, BINARY_TREE concisely represents a tree as a map of paths to values, where paths are encoded as sequences of Booleans.

How Many Complete Contracts? Reasoning informally, but rigorously, about the completeness of postconditions — along the lines of Section 3.2 — proved to be

straightforward in our experiments. Only 18 (7%) out of 254 public features in EiffelBase with model-based contracts and 17 (4%) out of 403 public features in EiffelBase2 have incomplete postconditions. All of them are examples of "intrinsic" incompleteness mentioned at the end of Section 3.2; EiffelBase2, in particular, was designed trying to minimize the number of features with intrinsically incomplete postconditions.

These results indicate that model-based contracts make it feasible to write systematically complete contracts; in most cases this was even relatively straightforward to achieve. Unsurprisingly, using model-based contracts dramatically increases the completeness of contracts in comparison with standard Design by Contract. For example, 42 (66%) out of 64 public features of class LIST in the original version of EiffelBase (without model-based contracts) have incomplete postconditions, including 20 features (31%) without any postcondition.

Contract-Based Testing with Model-Based Contracts. The standard EiffelBase library has been in use for many years and has been extensively tested, both manually and automatically. Are the expressive contracts based on models enough to boost automated testing finding new, subtle bugs? While preliminary, our experiments seem to answer in the affirmative. Applying the AutoTest testing framework [10] on EiffelBase with model-based contracts for 30 minutes discovered 3 faults; none of them would have been detectable with standard contracts. Running these tests did not require any modification to AutoTest or model classes, because the latter include an executable implementation.

The 3 faults reveal subtle mistakes that have gone undetected so far. For example, consider an implementation of routine *merge_right* in LINKED_LIST (not shown for brevity); the routine merges a linked list *other* into the current list at the cursor position by modifying references in the chain of elements. The routine deals in a special way with the case when the cursor in the current list is *before* the first element; in this case the *first_element* reference is attached directly to the first element of the other list. This is not sufficient, as the routine should also link the end of the other list to the front of the current one, otherwise all elements in the current list become inaccessible. The original contract does not detect this fault; in particular the postcondition clause *count* = **old** *count* + **old** *other.count* is satisfied as the attribute *count* is updated correctly, but its value does not reflect the actual content of the new list. On the contrary, the clause *sequence* = **old** (*sequence.front* (*index*)+ *other.sequence* + *sequence.tail* (*index* + 1)) of the complete model-based postcondition specifies the desired configuration of the list after executing the command, which leads to easily detecting the error.

5 Related Work

Hoare pioneered the usage of mathematical models to define and prove correctness of data type implementations [7]. This idea spawned much related work; the following paragraphs shortly summarize a few significant representatives, with particular focus on the approaches that are closest to the one in the present paper.

Algebraic Notations. Algebraic notations formalize classes in terms of (uninterpreted) functions and axioms that describe the mutual relationship among the functions.

The most influential work in algebraic specifications is arguably Guttag and Horning's [6] and Gougen et al.'s [5], which gave a foundation to much derivative work. The former also introduced a notion of *completeness*. Algebraic notations emphasize the calculational aspect of a specification. This makes them very effective notations to formalize and verify data types at a high level of abstraction, but does not integrate as well with real programming languages to document implementations in the form of pre and postconditions.

Descriptive Notations. Descriptive notations formalize classes in terms of simpler types — ultimately grounded in simple mathematical models such as sets and relations — and operations defined as input/output relations. Descriptive notations can be used in isolation to build language-independent models, or to give a formal semantics to concrete implementations. Languages and methods such as B [1] pursue the former approach; other specification languages and tools such as Jahob [15] are examples of the latter approach. Descriptive notations are apt to develop correct-by-construction designs and to accurately document implementations, often with the goal of verifying functional correctness; using them in contracts, however, introduces a new notation on top of the programming language, which requires additional effort and expertise from the programmer. This weakness is shared by algebraic notations alike.

Design by Contract Approaches. Design by Contract [9] introduces formal specifications in programs using the same notation for implementation and annotations, in an attempt to make writing the contracts as congenial as possible to programmers. The Eiffel programming language epitomizes the Design by Contract methodology, together with many similar solutions for other languages such as Spec$^\#$ [2] for C$^\#$. As we discussed also in the rest of the paper, using a subset of the programming language in annotations often does not provide enough expressive power to formalize (easily) "complete" functional correctness.

Model-Based Annotation Languages. The Java Modeling Language (JML) [8] is likely the approach that shares the most similarities with ours: JML annotations are based on a subset of the Java programming language and the JML framework provides a library of model classes mapping mathematical concepts. While sharing a common outlook, the approaches in JML and in the present paper differ in several details. At the technical level, JML prefers model variables while our method leverages model queries; each approach has its merits, but model queries have some advantages (discussed in Section 3.1). A notational difference is that JML extends Java's expressions with notations for logic operators, while our method reuses Eiffel notation such as agents to express quantifications and other aspects. In terms of scope, our approach strives to be more methodological and systematic, with the primary target of fully contracting complete libraries of data structures, while keeping the additional effort required to the programmer to a minimum. The present paper extends in scope the previous work of ours on model-based contracts [13, 12], and systematically applies the results to the re-design and re-implementation of a rich library of data structures. The experience gained in this practical application also prompted us to refine and rethink aspects of the previous approach, as we discussed at length in the rest of the paper.

6 Conclusions and Future Work

The present work introduces a method to write strong interface specifications for reusable object-oriented components. The method is soundly based on the concept of model and features a notion of specification completeness which is formal, yet easy to reason about. The application of the method to the development of a library of general-purpose data structures demonstrates its practicality and its many uses in analysis, design, and verification.

Future work includes short- and long-term goals. Among the former, we plan to apply model-based contracts to more real-life examples, including application software from diverse domains. A user study will try to confirm the preliminary evidence that model-based contracts are easy to write, understand, and reason about informally. Longer term work envisions integrating model-based contracts within a comprehensive verification environment.

Acknowledgements to Marco Piccioni, Stephan van Staden, and Scott West for comments on a draft of this paper.

References

1. J.-R. Abrial. *The B-book*. Cambridge University Press, 1996.
2. M. Barnett, R. DeLine, M. Fähndrich, B. Jacobs, K. R. M. Leino, W. Schulte, and H. Venter. The Spec$^{\#}$ programming system: Challenges and directions. In *VSTTE 2005*, volume 4171 of *LNCS*, pages 144–152. Springer, 2008.
3. P. Chalin. Are practitioners writing contracts? In *Rigorous Development of Complex Fault-Tolerant Systems*, pages 100–113, 2006.
4. http://freeelks.svn.sourceforge.net.
5. J. A. Gougen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In *Current Trends in Programming Methodology*, pages 80–149. Prentice Hall, 1978.
6. J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Inf.*, 10:27–52, 1978.
7. C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.
8. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, 55(1-3):185–208, 2005.
9. B. Meyer. *Object-oriented software construction*. Prentice Hall, 2nd edition, 1997.
10. B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf. Programs that test themselves. *Computer*, 42(9):46–55, 2009.
11. N. Polikarpova, C. A. Furia, and B. Meyer. Specifying reusable components. Extended version available at http://arxiv.org/abs/1003.5777.
12. B. Schoeller. *Making classes provable through contracts, models and frames*. PhD thesis, ETH Zurich, 2007.
13. B. Schoeller, T. Widmer, and B. Meyer. Making specifications complete through models. In *Architecting Systems with Trustworthy Components*, pages 48–70, 2004.
14. http://vsr.sourceforge.net/.
15. K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In *PLDI 2008*, pages 349–361. ACM, 2008.