Cite as follows: Bernd Schoeller, Tobias Widmer and Bertrand Meyer, *"Making specifications complete through models"*, to appear in *"Architecting Systems with Trustworthy Components"*, eds Ralf Reussner, Judith Stafford and Clemens Szyperski, Springer-Verlag Lecture Notes in Computer Science, 2006.

Making specifications complete through models

Bernd Schoeller, Tobias Widmer, and Bertrand Meyer

[1] Eidgenössische Technische Hochschule Zürich, Switzerland
[2] IBM Research, Zürich, Switzerland
[3] Eidgenössische Technische Hochschule Zürich, Switzerland

Abstract. Good components need precise contracts. In the practice of Design by Contract™, applications and libraries typically express, in their postconditions and class invariants, only a subset of the relevant properties. We present:

- An approach to making these contract elements complete without extending the assertion language, by relying on "model classes" directly deduced from mathematical concepts.
- An actual "Mathematical Model Library" (MML) built for that purpose
- A method for using MML to express complete contracts through abstraction functions, and an associated theory of specification soundness.
- As a direct application of these ideas, a new version of a widely used data structure and algorithms library equipped with complete contracts through MML.

All the software is available for download. The approach retains the pragmatism of the Design by Contract method, suitable for ordinary applications and understandable to ordinary programmers, while potentially achieving the benefits of much heavier formal specifications.
The article concludes with a discussion of applications to testing and program proving, and of remaining issues.

1 Introduction

Professional-quality components should be accompanied by precise specifications, or "contracts", of their functionality. Contracts as written today are often incomplete; we will discuss how to make them complete through the use of a model library.
The rest of section 1 discusses contracts and the problem of how to make them complete. Section 2 outlines the key element of our solution: the notion of model. Section 3 describes our application of this concept: the Mathematical Model Library (MML) which we have developed for this work. Section 4 explains how then to use MML to turn incomplete contracts into complete ones. Section 5 describes how we applied this approach to provide a completely contracted version of a widely used data structure and fundamental algorithms library. Section 6 presents a comparison with earlier uses of models for specification. Section 7 is a conclusion and presentation of future work.

1.1 Contracts

Before they will accept a large-scale switch to Component-Based Development, organizations with a significant stake in the correct functioning of their software need some guarantee that the components they include in their applications will themselves perform correctly. The first step is to know what exactly each of these components is supposed to do.
The Design by Contract™ techniques of Eiffel address this issue: every component is characterized by contract elements specifying its abstract relationship to other software elements. An individual operation (feature) has a *precondition*, stating what initial conditions it expects from its callers, and a *postcondition* stating what it provides in return; a group of operations (class) has an *invariant*, stating consistency conditions which each of these operations must preserve and each initialization mechanism (creation procedure) must ensure initially.
Design by Contract provides a number of advantages [18]: a methodological basis for analysis, design and implementation of correct software; automatic documentation, such as the class abstracters present in Eiffel environments extract from the class texts themselves; help for project management; a disciplined approach to inheritance, polymorphism and dynamic binding; and support for testing and debugging, including [6] component tests automatically generated and run from the contracts. An important characteristic of these techniques as available in Eiffel is that they are not for academic research but for practical use by developers, and indeed libraries such as EiffelBase [17, 15, 8] covering fundamental data structures and algorithms are extensively equipped with contracts. This distinguishes the context of the present study from extensions to Java or other languages (such as JML [14] or iContract [12]), which require the use of tools, libraries and language extensions different from what most.
This pragmatic focus also explains why Design by Contract distinguishes itself from more heavy-duty "formal methods" in its attitude to specification *completeness*: you can benefit from the various advantages of contracts mentioned above even if your contracts express only part of the relevant specification properties. More precisely, in the practice of Design by Contract as illustrated by the Eiffel libraries:

- Preconditions tend to be complete. Specifying "**require** *cond*" enables the routine to assume that condition *cond* will hold on entry, and not to provide any guarantee if it doesn't. Clearly, this is safe only if the routine specifies such conditions exhaustively.
- Postconditions and class invariants, however, are often underspecified; the next section will give typical examples. Unlike with preconditions, there is no obviously disastrous consequence; operations simply advertise less than they guarantee or (in the invariant case) maintain. The same holds for other uses of contracts: loop invariants and loop variants.

Why are such specification elements incomplete? There are three common justifications:

- Economy of effort (or, less politely, laziness): expressing complete specifications would require more effort than is deemed beneficial.

– Limitations of the specification language: in the absence of higher-level mechanisms, such as first-order predicate calculus (*"for all"*, and *"there exists"* quantifiers), some specifications appear impossible to express; an example would be *"All the elements of this list are positive"*.
– The difficulty of expressing postconditions that depend on a previous state of the computation

This discussion will show that there is no theoretical impossibility, and will propose an approach that makes it possible to express complete specifications and apply them to practical libraries such as EiffelBase [8].

1.2 Incomplete contracts

A typical feature exhibiting incomplete postconditions is *put* from class *STACK* of EiffelBase describing the abstract notion of stack, and its descendants providing various implementations of stacks. It implements the "push" operation on stacks (the name *put* is a result of the strict consistency policy of Eiffel libraries [15, 17]). In its "flat" form taking inheritance of assertions into account, it reads

```
put (v: like item)
        -- Push 'v' onto top.
    require
        not_full: not full
    ... Implementation, or "deferred" mark ...
    ensure
        item_on_top: item = v
        count_increased: count = old count + 1
    end
```

The query *item* yields the top of the stack, and the query *count* its number of items; *full* tells whether a stack's representation is full (never true for an unbounded stack).
The precondition is complete: if the stack is not full, you may always push an element onto it. The postcondition, however, is not: it only talks about the number of items and the top item after the operation, but doesn't say what happens to the items already present. As a result:
– It leaves some questions unanswered, for example, what will get printed by

```
create stack.make_empty
stack.put (1)
stack.put (2)
stack.remove
print (stack.item)
```

whereas the corresponding abstract data type specification [18] is sufficient to compute the corresponding mathematical expression: *item (remove (put (put (new, 1), 2)))*.

– It leaves the possibility of manifestly erroneous or hostile implementations, for example one that would push v but change some of the previously present items.

The specification of *STACK*, like most specifications in existing libraries, tells the truth, and tells only the truth; but it does not tell the whole truth.

For most practical applications of Design by Contract, these limitations have so far been considered acceptable. But it is desirable to go further, in particular to achieve the prospect of actual *proofs* of class correctness. Proving a class correct means proving that its implementation satisfies its contracts; this will require the specifications to be complete.

1.3 Approaches to completing the contracts

To address the issue of incomplete specifications, and obtain contracts that tell the whole truth, we may envision several possibilities.

A first solution is to extend the assertion language. In Eiffel and most other formalisms that have applied similar ideas, assertions are essentially Boolean expressions, with two important additions:

– The **old** notation, as used in the last postcondition clause (labeled count_increased:), making it possible to refer to the value of an expression as captured on routine entry (a "previous state of the computation" as mentioned in the earlier terminology).
– The **only** clause of ECMA Eiffel [21] (similar to the "modifies" clause of some other formalisms), stating that the modifying effect of a feature is limited to a specific set of queries; a clause **only** *a, b, ...* is equivalent to a set of clauses of the form q = **old** q for all the queries q not listed in *a, b, ...*.

This conspicuously does not include first-order predicate calculus mechanisms.

It is conceivable to extend the assertion language to include first-order constructs; the Object Constraint Language [27] for UML has some built-in quantifiers for that purpose. We do not adopt this approach for several reasons. One is that first-order calculus is often insufficient anyway; it doesn't help us much to express (in a graph class) an assertion such as "the graph has no cycles". Another more practical reason is that it is important in the spirit of Design by Contract to retain the close connection between the assertion language and the rest of the language, part of the general **seamlessness** of the method. In particular, for applications to testing and debugging — which will remain essential until proofs become widely practical — it is important to continue ensuring that assertions can be evaluated at reasonable cost during execution. This rules out properties of the form "For all objects, ..." or "For all objects of type *T*, ...". Properties of the form "For all objects in data structure *D*, ...", on the other hand, easy to handle through Eiffel's **agent** mechanism [7, 22]. For example, we state that "all values in the list of integers *il* are positive" through the simple Boolean expression

*il.for_all (***agent** *is_positive)*

using a simple query *is_positive*. In the absence of an agent mechanism, it would be still possible, although far more tedious and less elegant, to write a special function for any such case, here *all_positive* applying to a list of integers.

A second solution is to rely on extra features that express all the properties of interest. *all_positive* is a simple example, but we may extend it to more specific features; for example a class such as *STACK* may have a query *body* yielding the stack consisting of all the items except the top one (the same that would result from a "pop" command). We can then add to put a postcondition

body === **old** *Current*

where === is *object equality*. This technique works and has the advantage that it is not subject to the limitations of first-order predicate calculus; in our graph example we may write a query *acyclic* — a routine in the corresponding class — that ascertains the absence of cycles. The disadvantage, however, is to pollute classes with numerous extra features useful for specification only. In addition, we must be particularly careful to ensure that such features can produce no state change. The solution retained below is in part inspired by this approach but puts the specification features in separate classes with impeccable mathematical credentials.

A third solution would be to refer explicitly, in contracts, to internal (non-exported) elements of the objects' state. This is partially what a query such as *body* does, in a more abstract way. But the need for complete specification is not a reason to break the fundamental rules of information hiding and data abstraction.

For the record, we may mention here a fourth solution, as used in some specifications of the ELKS library standard [26], based on [23] and relying on recursive specifications. In the absence of a precise semantic theory it is not clear to us that the specifications are mathematically well-founded.

2 Using Models

The approach we have adopted for specifying libraries retains some of the elements of the second and third solutions above, but through a more abstract technique for describing the state.

2.1 The notion of model

The basic idea is to consider that a software object — an instance of any particular class — is a certain computer representation of a certain mathematical entity, simple or complex, called a *model* for the object, and to define the semantics of the applicable operations through their effect on the model.

As model for a stack, for example we may choose a *sequence*, with the convention that the last element of the sequence corresponds to the top

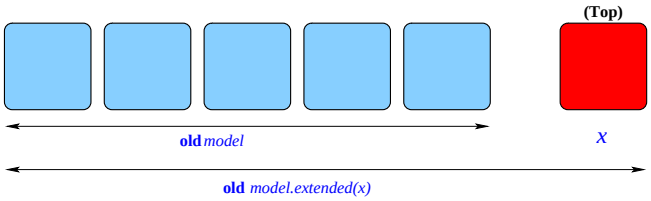

Fig. 1. Sequence model of a stack

of the stack (although the reverse convention would work too). Figure 1 illustrates this.
Then the effect of *put* can be specified through the model: *put* simply adds the new element at the end of the sequence. We will express this below and see that the existing postconditions (*count* increased by one, *item* denoting the new element) become immediate consequences of this property.

2.2 A model library

The model for a software object, as noted, is a mathematical object, such as a set, a sequence, a number, or some combination of any such elementary objects. But we still want to preserve the seamlessness of the approach; this would not be the case if we expressed contracts in a separate mathematical notation, for example a mathematical specification language.
It turns out that a language such as Eiffel is perfectly appropriate to express such concepts. For example we can write a class *MML_SEQUENCE [G]* that directly models the mathematical notion of sequence, and use it in lieu of the mathematical equivalents, as long as we observe a golden rule:

Model Library Principle

Model classes may not have commands.

A *command* (as opposed to *query*), also called a procedure, is a feature that modifies the object state; this is also excluded for purposes of specification. ("Creation procedures" will, however, be permitted, as they are necessary to obtain the mathematical objects in the first place.) For example the *MML_SEQUENCE* class may not have a procedure *extend*, which would modify a sequence by adding an element at the end; but it has a query *extended* such that *s.extended (v)* denotes another sequence with the elements of *s* complemented by an extra one, *v*, at the end.

In Eiffel a query may be implemented as either a *function* ("method" in some programming languages' terminology) or an *attribute* ("field", "data member", "instance variable"). The "Principle of Uniform Access" implies that the difference is not visible from the outside. As detailed below, the basic model classes will be deferred, meaning that they stay away from any choice of implementation; this is how client classes will see them. Implementation classes are also provided (section 3.7) for testing purposes; these classes, representing mathematical objects, may implement some queries as attributes. In other words the corresponding objects have a state, but this causes no conceptual problem since the Model Library principle guarantees that the state is immutable.

Our library of such model classes is called the *Mathematical Model Library* (MML). It is important to note that MML is couched in the programming language — Eiffel — for purposes of expressiveness, convenience and seamlessness only; underneath the syntax, it is a direct expression of well-known and unimpeachable mathematical concepts as could be expressed in a mathematical textbook or in a formal specification language such as Z [30] or B [1].

Instead of relying on explicit knowledge of the state, the contracts will rely on abstract properties of the associated model. We add to every relevant class a query

model: SOME_MML_TYPE

and then rely on *model* to express complete contracts for the class and its features. Taking advantage of the "selective export" facility of Eiffel [16, 22], we declare *model* in a clause labeled

feature {*SPECIFICATION*}

which implies that, depending on the view they choose, client programmers will, through the documentation tools in the environment, either see it or not. For simple-minded uses, it is preferable to ignore it; as soon as one is interested in advanced specification, tests or proofs, it is preferable to retain it.

The model describes a kind of abstract implementation of the concept underlying a class. As an implementation, however, it is purely mathematical and does not interfere with the rest of the class. In particular, the approach described here has no effect whatsoever on performance in normal operational circumstances, where contract monitoring is usually disabled. If contract monitoring is on (for debugging or testing), options should be available to include or exclude the extra "model contracts".

2.3 Model example

Let us now express how to use the notion of model on our earlier example of an unbounded stack. We use an *MML_SEQUENCE* as model for a stack. To this effect we add to class *STACK* a query:

```
feature {SPECIFICATION} -- Model

    model: MML_SEQUENCE [G]
        ensure
            not_void: Result /= Void
```

Based on this model, we can complete the contract of *put* by adding the model-based properties:

```
    put (v: like item)
            -- Push 'v' onto top.
        require
            not_full: not full
        do
            ... Implementation ...
        ensure
            model_extended: model === old model.extended (v)
            item_on_top: item = v
            count_increased: count = old count + 1
        end
```

The assertion *model* === **old** *model.extended (v)* states that the model after the feature invocation is the same as the model before the feature invocation except that *v* has been added to the end of the sequence. In a formalism such as Z it would be expressed as the following before-after predicate (where :: is the operator for appending a value to a sequence):

$$model' = model :: v$$

We now have a completely contracted version of *put*: the postcondition specifies the full effect, without revealing details about the implementation [32].

2.4 Theories and models

As detailed in section 6 (devoted to the comparison with earlier work), models have already been used in several approaches to program specification and verification, notably Larch and JML.

In general, these approaches treat models as additions to the *specification framework*, each created in response to a particular specification need. The model classes themselves have to be contracted in the existing specification language (without model features); the meaning of the model is based solely on its own contracts. We again get into the problems of underspecification, this time within the model library.

MML does not integrate the models into the specification framework, but into the *specification language*. We postulate that the models correctly reflect their theoretical counterparts which, as a consequence, define their semantics. To reason about assertions using models, we translate them into the underlying theory. The contracts of model classes directly reflect

axioms and theorems of the associated theory, assumed to have been proved (often long ago, and documented in mathematical textbooks), so we can just take them for granted.
As our basic theory, we choose typed set theory. It is a well-defined formalism, easy to understand for the average software developer; the typed nature of modern programming languages such as Eiffel makes types a familiar concept. A formalization intended for software modelling purposes can be found in the language of the B method [1] (whose program refinement aspects, studied in [28] in relation to contracts, are not relevant for this work). The availability of theorem provers for this theory [25, 31] is an added advantage.

3 Model Library Design

The model library is designed around a set of deferred classes describing the interfaces of the modeling abstractions.

3.1 Model classes

Eiffel's inheritance and genericity mechanisms enable us to model the definition of powersets, relations, functions, sequences, bag and graphs in terms of sets and pairs. The result reflects the definitions of [1]. Inheritance in particular provides a direct way to represent the subtype relation.
Figure 2 on the next page is a BON diagram of the inheritance structure of the principal deferred classes. The top type is *MML_ANY*, with two direct heirs *MML_SET* and *MML_PAIR*. (All the class names have the *MML_* suffix, omitted in the figure except for *MML_ANY*.)

3.2 Specifics of model objects

Mathematical objects are different from software objects:

- Mathematical objects are normally immutable: the operation $5 + 1$ does not change 5, but instead describes another number. Similarly, we cannot "add" an element to a set; rather, we describe new sets by union or intersection of existing sets like in $\{a, b, c\} \cup \{d\} = \{a, b, c, d\}$.
- They have no notion of identity distinct from their value.

Software objects do not have these properties: they have a mutable state, and an identity independent from their value. MML classes, although expressed in Eiffel, represent mathematical objects and hence must satisfy immutability and not rely on object identity.

3.3 Immutability

Enforcing immutability means that an instance of an MML class, once created, will never change its state. All features of the class other than creation procedures are *pure* (side-effect-free) queries.

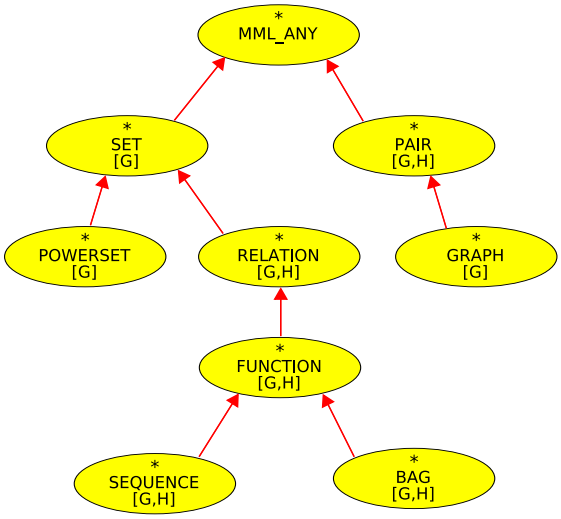

Fig. 2. A BON diagram of MML

3.4 Comparing mathematical objects

Not relying on object identity means that comparison operations will never apply to references, but to objects.
Object equality in Eiffel has a predefined version, *default_is_equal*, which compares objects field-by-field, and a redefinable version, *is_equal*, whose semantics also governs the = equality operator. Neither is adequate, however, for defining the equality of model objects, because two Eiffel objects cannot be equal unless they have the same type; in mathematics this is too strong a requirement, even with the type approach we are following. For example an object of type *MML_RELATION [X, Y]* can never be equal, in the Eiffel sense, to an object of type *MML_SET [MML_PAIR [X, Y]]*, whereas mathematically they may represent the same concept (a relation is a set of pairs).
For that reason, *MML_ANY* introduces a special query *equals* to represent mathematical object equality. Its descendants redefine it to describe their specific notions of equality. Every comparison of MML objects should use equals, not *is_equal*. To guarantee this and avoid mistakes, *MML_ANY* and descendants do not export *is_equal*.
Here is the specification of equal in *MML_ANY*:

```
equals alias "===" (other: MML_ANY): BOOLEAN
        -- Is other mathematically equivalent to current object?
    require
        other_not_void: other /= Void
    ensure
```

symmetric: **Result implies** *(other.equals (***Current***))*
yes_if_equal_as_objects: *is_equal (other)* **implies Result**

The first postcondition clause expresses symmetry, the second that object equality implies mathematical equality (although, as noted, not necessarily the other way around).
The precondition refers to **Void** values, which will not arise with mathematical objects. This clause will go away thanks to the ECMA Eiffel standard [21] which deals with this issue statically; all MML types will be attached [20] and hence statically guaranteed non-void.
The **alias** clause makes it possible to use $a === b$ as shorthand for *a.equals (b)*.

3.5 Class overview

Here are some of the features of the MML classes in the top part of the hierarchy as shown in figure 2.
MML_SET is the basic class for the definition of sets as models. It implements most basic operators. Examples of available predicates on sets are *is_member* ($x \in A$), *is_subset* ($A \subseteq B$), *is_proper_subset* ($A \subset B$) or *is_disjoint* ($A \cap B = \emptyset$). Other operators include *united* ($A \cup B$), *intersected* ($A \cap B$), *subtracted* ($A - B$), *cartesian_product* ($A \times B$) and so on. The class also provides a non-deterministic choice operator called *any_item*.
MML_PAIR represents tuples of cardinality two. All other types can be described in terms of *MML_PAIR* and *MML_SET*.
MML_RELATION describes relations viewed as sets of pairs. Thanks to inheritance we adapt set operations into operations on relations.
The class then adds another substantial set of relation-specific features: queries such as *is_reflexive* and *is_transitive*, transformations such as *image* and *inversed*.
Relational composition causes the only problem with using the language's type mechanisms to model set-theoretical type rules. The notion of relation involves two generic parameters, representing the types of the source and target sets. But the expression *r1.composed (r2)* requires a third generic type, the target set of *r2*. This cannot be modelled directly since only classes, not features, may have generic parameters.
Our solution is to take *ANY* as the type of the second argument. This has sufficed for the examples we have encountered so far, but we may have in the future to add a third generic parameter to the class just for the sake of the composition operator.
MML_FUNCTION describes possibly partial functions, viewed as a special case of relations. It defines such concepts as partiality and surjectivity.
MML_SEQUENCE, MML_BAG, MML_GRAPH provide the library with a richer set of modeling concepts. Sequences in particular provide part or all of the model for many concepts, including lists, strings, files and others for which the ordering of data is important.

3.6 Quantifiers

To model quantifiers, we use Eiffel's *agent* mechanism. Agents are objects encapsulating features, and hence functionality. These objects are immutable, so the introduction of agents does not affect the "pure" (side-effect-free) requirement on the model library.
The agents we use for our model-based specifications represent predicates. For example $\forall x \in S.P(x)$ will appear as *S.for_all(***agent** *P(?))* where the question mark represents the bound variable — "open argument"' in Eiffel terminology; an agent expression like **agent** *P(?)* where all arguments are open can be abbreviated into just **agent** *P*.
For modularity and ease of use, all the basic quantifier mechanisms based on this technique are grouped into a specific class (a "facet" abstraction [32]) called *MML_QUANTIFIABLE*, with the following two features.

```
feature -- Quantifiers

    there_exists (predicate: FUNCTION [ANY, TUPLE [G], BOOLEAN]):
                  BOOLEAN is
        -- Does current contain an element which satisfies
        -- predicate ?
    require
        predicate_not_void: predicate /= Void
    deferred
    ensure
        definition: Result =
            (not for_all (agent negated (?, predicate)))
    end

    for_all (predicate: FUNCTION [ANY, TUPLE [G], BOOLEAN]):
             BOOLEAN is
        -- Does current contain only elements which satisfy
        -- predicate ?
    require
        predicate_not_void: predicate /= Void
    deferred
    ensure
        definition: Result =
            (not there_exists (agent negated (?, predicate)))
    end
```

The contracts capture the relations of $\forall$ and $\exists$. *negated* is a feature from the class *MML_FUNCTIONALS* offering generic functionals such as negation and composition on predicates defined by agents. We may note in passing that this class and *MML_QUANTIFIABLE* achieve — thanks in particular to agents — the side goal of providing, within the Eiffel framework, a substantial subset of the mechanisms of functional languages such as Haskell.

3.7 Implementing the model classes

MML classes as seen so far are all deferred (abstract). A deferred class may have no direct instances; correspondingly, it need not provide any implementation for its features. Non-deferred (concrete) classes, directly describing software objects, are called *effective* [18].
If we are interested in completely contracted classes for proving purposes, deferred classes are clearly sufficient. There is no need for direct instances of model objects, for implementation of model features, or more generally for execution.
If, on the other hand if we are also interested in equipping classes with complete contracts for the purpose of *testing* them more effectively, we will need implementations — effective versions of the original classes.
As a result of these observations, MML includes a set of reference implementations, one provided (as an effective descendant) for each of the directly usable deferred classes.
All implementations assume that the sets are finite and small enough to be represented through *ARRAY* or *LINKED_LIST* data structures. This is sufficient for the problems we have tackled so far.
Most of the work for the default implementation is done in the two classes *MML_SET* and *MML_PAIR*. *MML_SET* uses the *ARRAYED_SET* data structure of EiffelBase. *MML_PAIR* just defines two variables *one* and *two* to represent the values of a pair.
Because typed set theory allows describing all other structures (bags, sequences etc.) in terms of these two, their implementation builds on implementations of sets and pairs.

4 Using models to achieve complete contracts

The model library as sketched in the previous section enables us to reach our original goal of equipping realistic, practical classes with complete contracts. We now explore this process and its application to some important classes of the EiffelBase library.

4.1 Devising a model

The first step in equipping a class with model-based complete contracts is to choose a model that will adequately capture the state of its instances; in the *STACK* example the choice was sequences.
As with the basic object-oriented design issue of of finding the right inheritance or client relation, there is no general, infallible process. [32] gives some hints.
For example, a mathematical relation is probably the right model for classes describing hash tables or other dictionary-like structures. As another hint, the EiffelBase placement of the random number generator class as as a descendant of *COUNTABLE_SEQUENCE* suggests sequences as the model for this notion.

4.2 The abstraction function

We may call the relationship between a concrete software object and its MML model its "abstraction function" (a notion introduced in [11] in the form of the "representation function", its inverse, actually multi-valued). For the *ARRAYED_STACK* class we use the following model:

```
feature{SPECIFICATION} -- Model

    model: MML_SEQUENCE [G] is
            -- Model of the stack
        local
            l: LINEAR[G]
        do
            create {MML_DEFAULT_SEQUENCE [G]}Result.make_empty
            l := linear_representation
            from
                l.start
            until
                l.off
            loop
                Result := Result.prepended (l.item)
                l.forth
            end
        end
```

Model queries always return an attached (non-void) result in the sense of ECMA Eiffel. They have no feature-specific contracts (preconditions or postconditions), but may have associated constraints as part of the class invariant. Any implementation of the abstraction function (potentially useful, as noted, for applications to testing) may only rely on the invariant.

4.3 Composite models

In many of the more advanced examples it is not realistic to capture the complete state of a data structure through an atomic model built directly from one of the classes of MML, such as a single sequence in the examples above. As an example, consider the EiffelBase class *LINKED_LIST*, describing a sequence of values equipped with a *cursor* to facilitate traversal and manipulation (figure 3).
To describe the full state, we may use a tuple of a sequence s and a cursor position n, yielding an abstraction function of type:

$$model : LINKED\_LIST[G] \Rightarrow SEQUENCE[G] \times \mathbb{N}$$

To build this abstraction function into the class we first define an abstraction for each component of the model:

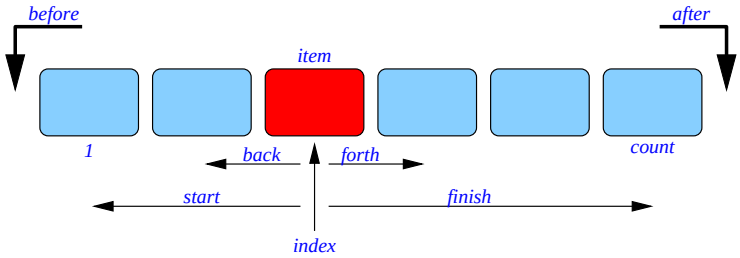

Fig. 3. *LINKED_LIST* with active cursor

```
feature{SPECIFICATION} -- Model

    model_index: INTEGER is
            -- Model of the cursor position
        do
            Result := index
        end

    model_sequence: MML_SEQUENCE [G] is
            -- Model of the list when regarded as a sequence
        do
            ...
        end
```

Then we create a common model by pairing the two components:

```
    model: MML_PAIR [SEQUENCE [G],INTEGER] is
            -- Model of the list
        do
            create {MML_DEFAULT_PAIR}Result.
                make(model_sequence,model_index)
        end
```

Our experience shows that this is a convenient practice. In particular we have retained the technique, illustrated in all the above examples, of always using a single *model* query expressing the entire abstraction function and yielding a single object; if the model conceptually involves several components — in the last example, a sequence and an integer — we turn them into a single one by taking advantage of the MML classes for pairs and sets. This rule yields a consistent style and enables us to refer for any class to "the model" and "the abstraction function".

4.4 Classic and model contracts

Most Eiffel classes, especially in libraries, are equipped with some contracts expressing important elements of their intended semantics. We will call them *classic contracts* in contrast to contracts relying on the model library, called *model contracts*.
Classic contracts are usually easy to understand for programmers, even those who may be put off by more formal approaches. But, as noted, they are often incomplete, especially postconditions and invariants. With the help of model contracts we should be able to check that they are at least *sound*, according to the following definition:

Definition: Soundness of a Model

A classic contract for a model-equipped class is sound if:

1. Every classic precondition implies the corresponding model precondition.
2. Every model postcondition implies the corresponding classic postcondition.
3. Every model invariant implies the corresponding classic invariant.

In the informal terms used at the beginning of this discussion: model contracts give us "all the truth"; classic contracts, the only ones that less advanced or less interested programmers will see, are sound if what they tell, while perhaps not the full truth, is still "the truth".
To this effect, condition 1 guarantees that every call that appears correct to a client programmer working on the sole knowledge of the classic contracts will indeed satisfy all the required conditions — even if it might satisfy more than strictly needed.
Condition 2 guarantees that every call will, on return, deliver every condition promised to clients - even if it might deliver more than classically advertised.
Condition 3 guarantees that the consistency constraints expected of instances of a class actually hold.
On the basis of this definition, let us examine the soundness of the *STACK* specification extract. The interesting part is the postcondition, consisting of three clauses, two classic and one model-related:

ensure
model_is_extended: *model* === **old** *model.extended (v)*
item_pushed: *item* = *v*
count_increased: *count* = **old** *count* + *1*

From the invariant, we know that

invariant
count_defined_through_model: *count* = *model.count*
item_defined_through_model: *item* = *model.last*

By combining the assertions of the postcondition and the invariant, we can derive the following two proof obligations to verify the soundness of the classical contracts:

(model === old model.extended (v)) **and**
(item = model.last)
  implies
(v = item)

(model === old model.extended (v)) **and**
(old count = old model.count) **and**
(count = model.count)
  implies
(count = old count + 1)

Both properties can be easily verified using a theory for sequences. The notion of soundness is particularly interesting in combination with inheritance. It is possible to prove soundness at an abstract level, in a deferred class such as *STACK*, without having to redo the proof in effective descendants such as *ARRAYED_STACK*. This point was discussed in [19].

5 Specification of a Full Library

As a testbed for the approach described here, and a major application of interest in its own stake, we considered EiffelBase [8], a reusable, open-source library of data structures provided with the Eiffel environment. Making heavy use of multiple-inheritance and genericity,the classes of EiffelBase include not only implementations of the data structures but also offer a rich set of deferred classes that capture useful concepts such as abstract containers, common traversal strategies, and mathematical structures such as "ring" and total order. The full design of the library is discussed in [17].

5.1 Overall structure

We produced a fully contracted version of the structural classes of EiffelBase; a significant endeavor since that part of the library includes 36 classes totalling 1853 exported (public) features.

The process of completing the specifications brought to light numerous inconsistencies in the library. Using model specifications, we were able to come up with a cleaned up hierarchy for EiffelBase. Figure 4 on the following page presents a BON diagram of this hierarchy. A full specification for each class appears in [32].

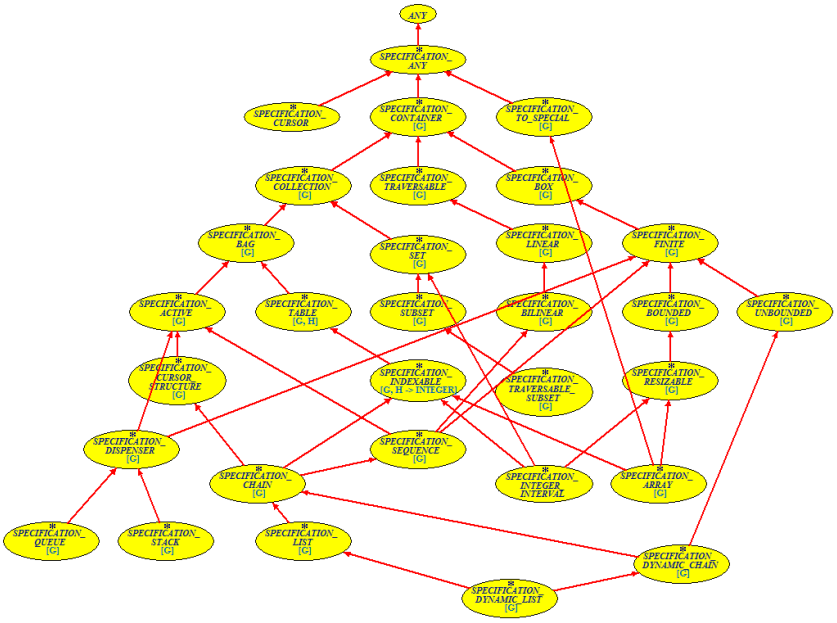

Fig. 4. The inheritance hierarchy of EiffelBase

5.2 Models of DYNAMIC_LIST

As an illustration of the work involved in this reengineering of EiffelBase for complete contracts, we consider a typical class. *DYNAMIC_LIST* is the parent for the implementation of lists through arrays (*ARRAYED_LIST*) and linked structures *LINKED_LIST*. Dynamic lists, like EiffelBase lists in general (see figure 4) are "active": they contain a movable cursor with a current cursor position.

The classes of our reengineered library bear the names of the corresponding EiffelBase classes prefixed by *SPECIFICATION_*, for example *SPECIFICATION_DYNAMIC_LIST*.

Four different models are available to describe the state of the dynamic list. They are inherited from the parent classes and describe different possible views of lists:

```
feature{SPECIFICATION} -- Model
    model_bag: MML_BAG [G]
            -- Bag model for the list
            -- (from SPECIFICATION_BAG)
    model_indexable: MML_RELATION [INTEGER,G]
            -- Table model for the list
            -- (from SPECIFICATION_TABLE)
    model_cursor: INTEGER
```

```
        -- Cursor model for the list
        -- (form SPECIFICATION_TRAVERSABLE)
    model_sequence: MML_SEQUENCE [G]
        -- Sequence model for the list
        -- (from SPECIFICATION_TRAVERSABLE)
```

All four relations are connected by the invariant of *SPECIFICATION_DYNAMIC_LIST*. For example the value of the cursor is limited by the size of the sequence:

```
model_cursor >= model_sequence.lower_bound − 1
model_cursor <= model_sequence.upper_bound + 1
```

The domain of the bag has to be the range of the sequence:

```
model_bag.domain === model_sequence.range
```

This shows how the class invariant can be used as a so-called *gluing invariant* between the different mathematical abstractions of the list.

5.3 Problems discovered

Most problems we found in EiffelBase were caused by heavy underspecifications, contradictions in contracts and flaws in the taxonomy. Here are some examples:

- The equality relation of "active" (cursor-based) data structures might involve not only elements of the structure, but also a cursor position and other internal data. All active data structures were missing a clear specification of whether they should be regarded equivalent if they have the same data but different cursor positions.
- The class *TRAVERSABLE_SUBSET* does not inherit from class *TRAVERSABLE*, even though it implements all features offered by *TRAVERSABLE*. This design decision prohibits polymorphic use.
- The features *prune* and *prune_all* in class *SEQUENCE* move the cursor to off, even if the element to be pruned is not present in the sequence.
- The feature *wipe_out* in class *ARRAY* is marked as obsolete. Obsolete feature clauses are not the proper way to declare a feature as inapplicable.
- The class *BILINEAR* inherits twice from *LINEAR* to implement bilinearity. This makes specification difficult, as it is not always clear which iteration features are derived for which inheritance relation.
- Internal cursors and functionals such as *for_all*, *there_exists* and *do_all* do not represent the same concept and should be distinguished. The linearity is not necessary for an implementation of logic quantifiers.

A full list of problems discovered can be found in [32].

6 Related Work

Models have been used before for software specification. Early work by Hoare [11] suggested the use of models. The Larch language and toolset [10] relies on models for program verification. In contrast to our approach, Larch introduces a special language for the specifications of models. This creates a conceptual separation between the model-based specifications and the programming language. Special projects provide embedding mechanisms of Larch models into such languages as Smalltalk [4] and C++ [13].

JML [14, 3] applies models to the domain of modular specifications of Java programs. JML includes an extensive model library for the specification of object-oriented programs, offering more than a hundred Java classes describing very diverse specification mechanisms. The core of the library comprises structural classes such as *JMLSequence* and *JMLValueSet*. The technique presented in this paper is strongly related to the *model variables* of JML [5]. The major difference is that JML model variables introduce the notion of state into the contractual specification. We view models as abstraction functions, without model variables. In addition, as explained earlier, we treat models as an extension to the contractual language and not as part of the surrounding framework.

Müller, Poetzsch-Heffter and Leavens [24] extend the use of model variables and procedures to the field of *frame properties*. We have not explicitly addressed this important issue here. Our working hypothesis is that to the extent that the model expresses all the properties of interest any effect the software's operations may have on properties not covered by the model is irrelevant. (Eiffel can, as noted, express frame properties through the newly introduced **only** postcondition clause, but the precise relation between models and the **only** clause still needs to be explored).

The Z specification language [30, 29] and the B method [1] have both been used to apply set theory to specify software in conjunction with before-after predicates. Our work is intended to provide Eiffel contracts with the same expressive power.

ASMs [9] and AsmL [2] use the concept of models and introduce model variables with a related notion of model programs. This yields *executable* specifications since one may treat the model as an abstract program that operates on the abstract state denoted by the model variables. By this, AsmL can provide executable specifications. The verification process consists of showing that the implementation is a behavioral subtype of the specification.

Mitchell and McKim [23] introduced models in the context of Eiffel and Design by Contract.

7 Conclusion

The framework described here appears to allow the development of libraries with complete contracts, not too difficult to write yet still understandable by any programmer who cares to learn a few basic concepts. We are continuing to apply this process to the EiffelBase library, which

lies at the core of many applications and hence plays a major practical role. Research work that will immediately benefit from this effort includes:

- Our ongoing effort to produce proofs that the classes indeed satisfy their contracts.
- Complementary work on entirely automatic ("push-button") tests of components based on their contracts [6], evidently made all the more interesting if the contracts are more extensive.

So far we have mostly applied our model-based techniques to libraries such as EiffelBase describing fundamental computer science concepts. Although we believe they can also be fruitfully applied to more application-oriented classes, or to graphical libraries such as EiffelVision, this remains to be demonstrated and is one of the next challenges.

The effort of producing complete contracts for EiffelBase has already born fruit: while the library has been carefully designed and is reused in many commercial and non-commercial applications, the process has uncovered a number of technical and conceptual flaws. These will be reported and fixed in the "classic" EiffelBase, although we definitely hope that — in line with the applied nature of this work and its intention, thanks to Eiffel's built-in contracts, to serve the direct needs of operational developments — the version with complete contracts will become the reference.

The mere process of writing the complete contracts and the resulting improvements to classic EiffelBase has already shown that more complete specifications improve the Design by Contract process and lead to clearer abstractions.

References

1. Jean-Raymond Abrial. *The B-Book – assigning programs to meanings*. Cambridge University Press, 1996.
2. Mike Barnett and Wolfram Schulte. The ABCs of specifications: AsmL, behavior, and components. *Informatica*, 25(4):517–526, November 2001.
3. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. Technical Report R0309, NIII, 2003.
4. Yoonsik Cheon and Gary T. Leavens. The larch/smalltalk interface specification language. In *ACM Transactions on Software Engineering and Methodology*, number 3, pages 221–253. ACM Press, July 1994.
5. Yoonsik Cheon, Gary T. Leavons, Murali Sitaraman, and Stephen Edwards. Model variables: Cleanly supporting abstraction in design by contract. Technical Report 03-10, Iowa State University, April 2003.
6. Ilinca Ciupa and Andreas Leitner. Automatic testing based on design by contract. In *Proceedings of Net.ObjectDays 2005*. tranSIT Thüringer Anwendungszentrum für Software-, Informations- und Kommunikationstechnologien GmbH, 2005. (to be published).

7. Paul Dubois, Mark Howard, Bertrand Meyer, Michael Schweitzer, and Emmanuel Stapf. From calls to agents. *Journal of Object-Oriented Programming*, 12(6), 1999.
8. Eiffel Software. *EiffelBase*, August 2005. `http://archive.eiffel.com/products/base/`.
9. Yuri Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.
10. John V. Guttag, James J. Jorning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specifications*. Springer-Verlag, New York, N. Y., 1993.
11. C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.
12. R. Kramer. iContract - the Java(tm) Design by Contract(tm) tool. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 295, Washington, DC, USA, 1998. IEEE Computer Society.
13. Gary T. Leavens. Larch/C++, an interface specification language for C++. Technical report, Iowa State University, Ames, Iowa 50011 USA, August 1997.
14. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06t, Department of Computer Science, Iowa State University, June 1998.
15. Bertrand Meyer. Tools for the new culture: Lessons from the design of the eiffel libraries. *Communications of the ACM*, 33(9):40–60, September 1990.
16. Bertrand Meyer. *Eiffel: the language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.
17. Bertrand Meyer. *Reusable software: the Base object-oriented component libraries*. Prentice-Hall, 1994.
18. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2 edition, 1997.
19. Bertrand Meyer. A framework for proving contract-equipped classes. In Egon Börger, Angelo Gargantini, and Elvinia Riccobene, editors, *Abstract State Machines 2003, Advances in Theory and Practice, 10th International Workshop, Taormina (Italy), March 3-7, 2003*, pages 108–125. Springer-Verlag, 2003.
20. Bertrand Meyer. Attached types and their application to three open problems of object-oriented programming. In Andrew Black, editor, *ECOOP 2005 (Proceedings of European Conference on Object-Oriented Programming, Edinburgh, 25-29 July 2005)*, number 3586 in LNCS, pages 1–32. Springer Verlag, 2005.
21. Bertrand Meyer, editor. *Eiffel Analysis, Design and Programming Language*. ECMA International, June 2005. As approved as International Standard 367.
22. Bertrand Meyer. Eiffel: The language. Third Edition, ongoing work as published at `http://se.ethz.ch/~meyer/ongoing/etl/`, August 2005.
23. Richard Mitchell and Jim McKim. *Design by Contract, by example*. Addison-Wesley, 2002.

24. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15:117–154, 2003.
25. Tobias Nipkow, Laurence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Springer, 2004.
26. Nonprofit International Consortium for Eiffel (NICE). *The Eiffel Library Standard*, June 1995. TR-EI-48/KL.
27. Object Management Group. *UML 2.0 OCL Specification*, November 2003. adopted specification, ptc/13-10-14.
28. Bernd Schoeller. Strengthening eiffel contracts using models. In Hung Dang Van and Zhiming Liu, editors, *Proceeding of the Workshop on Formal Aspects of Component Software FACS'03*, number 284 in UNU/IIST Report, pages 143–158, September 2003.
29. J. M. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, 1989. January.
30. J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, second edition edition, 1992.
31. Steria, Aix-en-Provence, France. *Atelier B Interactive Prover User Manual*.
32. Tobias Widmer. Reusable mathematical models. Master's thesis, ETH Zürich, July 2004.