How to Contracting Eiffel Classes with Models

A Methodological Tutorial

Bernd Schoeller <bernd.schoeller@inf.ethz.ch>
ETH Zurich, Switzerland

Overview

The *Mathematical Model Library* (MML)[1] is a library of immutable data structures for the Eiffel programming language. The data structures are derived from first-order predicate logic and set theory on finite sets. The goal of the library is to enable developers to improve the way classes are contracted using the *Design by Contract* methodology.

Together with MML comes an own methodology of how to apply model contracts to classes. This document describes this methodology and how to apply it to actual classes.

Runtime Checks

MML offers default implementations for the different models. The default implementations were developed with efficiency in mind. Still, models tend to become much slower than the actual implementations and also when compared to classical contracts. The two major reasons are:

1. Models represent immutable classes. As a result, few changes happen *in place*. Instead, almost every time a data structure is changed, it needs to be copied (at least as long as the Eiffel environment does not provide a reference counting mechanism).
2. Models always capture the full state of the object. Compared to the regular queries, the model query computation and the actual operations work on data structures that are normally at least an order of magnitude more complex than the ones of regular contracts.

A run-time environment should thus be able to detect MML contracts and specifically disable those. As an alternative, there exists the idea of developing a version of MML based on three-valued-logic that would make it possible to replace regular MML by an adjustable MML version that does not do any or only feasible checks.

Contracting Outline

There are different steps involved to add *model-based contracts* to a deferred class. The next chapters gives a detailed description of each on of these steps. The basic steps are for any class:

1. Define a suitable model that reflects the state of instances of the class on the level of your abstracts.
2. Restrict the valid values of the model by using an appropriate invariant.
3. Contract the features by describing the effect of commands relative to the model and projections of the model to query results.

Should you define a full *implementation* of the class, then the following extra work needs to be done:

1 http://se.inf.ethz.ch/people/schoeller/mml.html

4. Implement an abstraction function.
5. Contract the implementation with models (check, loop invariants etc.).

If your class inherits from another class that provides model contracts, then add the following step. The step is only necessary if the model of the new class is different from the model of the old class:

6. Rename the old model features and link the old model with the new model through the use of a gluing invariant.

The result will be a class that is *fully contracted using models*. The purpose of the next sections is to explain each step in detail and to give example on how to apply MML to real-world classes.

Step 1: Defining a Suitable Model

The very first step in the development of a model contract is the definition of a model. The model is defined by a query called `model` that returns the model. For example if you decide that the model of a counter is an integer value, you define the model through the following code:

```
model: INTEGER
      -- Abstraction Function
   deferred
   end
```

As you can see, the feature is abstract. We do not provide an implementation of an *abstraction function* as there currently is no concrete state to abstract from. This is fine for deferred classes. With non-deferred classes, we have to prove an implementation. This is described in the part *Abstraction Function Implementation* below.

In most cases, a simple INTEGER as model will not be sufficient. The state of the object is rather captured by a more complex data structure. The model feature is defined as a query that returns a model defined in terms of the data types in the MML library. For example a database: a database is formed by a set of data entries, each one relating keys to a values.

```
model: MML_SET [MML_RELATION [STRING,STRING]]
      -- Abstraction Function
   deferred
   end
```

The model can be composed of any data type. But you have to remember that there are three different types of classes: *expanded* classes, *reference* classes and the *MML* classes. Creating models from expanded classes and MML classes compares by value as expected. But with models defined in terms of reference classes, the `is_equal` function is used for comparison. Normally, `is_equal` compares references.

For example, if we model a collection of people, we might define the following model:

```
model: MML_SET [PERSON]
      -- Abstraction Function
   deferred
   end
```

If we define the model that way, we have to be aware that the model will not change if we change the first name of a person. This is an expected and necessary behavior of the methodology, otherwise model based contracts would over-specify the behavior of data structures. The property the adding a

person to a database does not change any of the data of persons is a property of the frame and not handled through MML, but through frame properties like modification clauses or dynamic frame sets.

The model might be constructed of different parts. It might be helpful to split up the model into sub-models, joining them into a global model structure. For example, the `LIST[G]` class in EiffelBase is has an integrated cursor. The model is constructed of the sequence of elements and the cursor position. The resulting model features would look like the following:

```
feature -- Model

   model: MML_PAIR [MML_SEQUENCE[G],INTEGER]
         -- Abstraction Function
      do
         Result := create {MML_DEFAULT_PAIR}.make
            (model_sequence,model_index)
      end

   model_sequence: MML_SEQUENCE[G]
         -- Abstraction of the list elements
      deferred
      end

   model_index: INTEGER
         -- Abstraction of the cursor position
      deferred
      end

invariant
   model_sequence_definition: model_sequence |=| model.frist
   model_index_definition: model_index |=| model.second
```

The given invariant links the different model abstractions. They form a gluing invariant between the models. The invariant is the best location to place these properties. The mail query `model` must still form a superset of the other models.

Step 2: Model Invariants

The invariants can also be used to restrict the legal values of the models. Not every value stored in the model might make sense. For example, if a list will never contain Void elements, the invariant can specify it with the following expression:

```
model_does_not_contain_void: not model.range.contains(Void)
```

The purpose if the model invariant is to describe precisely which values of the models are legal and which ones can be rejected. As a consequence, all other public invariants of the class should be implied by the model invariant. All other invariants then are concerned with the internal representation of the model, thus they are not part of the public contract.

Step 3: Contract using Model Contracts

Terminology: Preconditions and postconditions of features can rely on properties of the model. We

will call such contracts *model-based contracts*. Then there are contracts that do not use any references to the current state, but only put constrains on arguments. These will be called *argument contracts*. All other contracts will be call *classical contracts*.

The Eiffel methodology promotes the application of the command query separation principle. As a consequence of this principle, all queries must be side-effect free. The model of the data structure does not change. Thus, the postcondition of queries should always contain:

```
model_unchanged: model |=| old model
```

Other than that, there is no constrain on the *model-based* and *argument contracts*. The contract must be strong enough for clients to be usable, without revealing details of the implementation.

The situation is different with *classical contracts*. These contracts can be helpful as a fall back when the execution of *model-based contracts* is not feasible. Also, they can be easier to read that the model-based contracts. Third, the classical contracts can be inherited from earlier steps of the software development life cycle or from legacy code.

Still, classical contracts should not contain any contractual information that is not also expressed in the model-based contract. So, the other contracts have to be stronger than the classical contracts.

[model-based precondition] ^ [argument precondition] ⇒ [classical precondition]
[model-based postcondition] ^ [argument postcondition] ⇒ [classical postcondition]

Step 4: Abstraction Function Implementation

At some point in the inheritance hierarchy an internal representation of the data has be chosen. That is also the point were the abstraction function needs to be defined that translates the internal representation into the model.

The abstraction function normally uses one of the implementation classes of MML. The default implementations are all-purpose representations, but sometimes alternative implementations might be more optimal (for example using MML_RANGE_SET to model a set of numbers in a counter, if it is known that this range is between a lower and an upper bound).

For the example of the `ARRAYED_LIST [G]`, inheriting from the `LIST [G]` as defined above, the abstraction function might look somewhat like the following:

```
feature -- Model

   model_sequence: MML_SEQUENCE[G]
         -- Abstraction of the list elements
      local
         i: INTEGER
      do
         from
            create {MML_DEFAULT_SEQUENCE [G]}Result.make_empty
            i := 0
         until i > count loop
            Result := Result.extended(item_at(i))
            i := i + 1
         end
      end

   model_index: INTEGER
         -- Abstraction of the cursor position
      do
         Result := index
      end
```

We do not have to redefine the actual model query as it was split up into the two partial parts and the joining of the two parts was already defined in the parent.

If the classes that are used for the data representation already have model queries, then those queries should be used to compute the full model. As an example, consider a birthday book that relates names to birthdays. It is implementation by two equal-sized arrays, one containing the names (called `name_array`), the other the birthdays (called `date_array`). The arrays provide model queries in the form of sequences. The abstraction of the birthday book is a relation between names and birthdays. The abstraction function is implemented in the following way:

```
model: MML_RELATION [STRING,DATE]
      -- Abstraction Function
   do
      Result := composed (name_array.model.inversed,
                          date_array.model)
   end
```

(`composed` is functional composition of two relations)

Step 5: Contract the Implementation

Contracting the implementation with model contracts is done in the same ways as regular contracts. As the final goal of model-based specifications should be the formal verification of the classes using proof tools, the assertions that are added to the code have to be much stronger. Loops must carry reasonable loop invariants and variants.

Step 6: Rename and linking the old Model Features

In the case of inheritance, the `model` query inherited from parent classes might not return the correct model of the child. This is the case when the model of the child is more specific than that of the parent (data refinement).

Still, because of polymorphism, the child has to provide a model that is advertised by the parent. Otherwise the inherited model contracts would become meaningless. As a result, the model query must be renamed and redefined.

Renaming makes space for the new model query. Still the renamed name should point out the perspective that is taken on the state of the child. The model query should be named `model_for_x`, where `x` is the name of the parent.

Next, the model query has to provide an implementation that projects the model of the child onto the model of the parent. This relation should also be captured in its contract in form of an invariant, the *gluing invariant* of the data refinement.

In the following example we have a collection that is modeled as a bag and a list that is modeled as a sequence. When the list inherits from the bag, its model needs to be projected into the bag.

```
class LIST [G]

inherit
   COLLECTION [G]
      rename
         model as model_for_collection
      end

feature -- Model

   model: MML_SEQUENCE [G] is
         -- Abstraction Function
      deferred
      end

   model_for_collection: MML_BAG [G] is
         -- Projection of the list into a bag
      do
         Result := model.range_bag
      end

invariant
   list_collection_glue: model_for_collection |=| model.range_bag
end
```