Strengthening Eiffel Contracts using Models

Bernd Schoeller
Swiss Federal Institute of Technology, Chair of Software Engineering
CH-8092 Zurich – Switzerland
bernd.schoeller@inf.ethz.ch

Abstract

Creating proper contracts as interface specifications for software components is a key quality for the usability of the component in various contexts. A major goal of software engineering is to extend the expressiveness of these contracts and to enable component developers to use contracts for their own benefit with as little overhead as possible. One source for this overhead might be changes in language or paradigm.

A powerful possibility to express complex contracts for components is the use of behavioral models. This paper explores how these model specifications can be introduced into the Eiffel language, exploiting only standard language features and mechanisms. It also examines how these model-based contracts can be used to derive proof obligations as a starting point for formal verification.

1 Introduction

Specification of software components is an important but difficult task. It is important as it enables the client of a software component to understand the requirements and benefits he will get by using the component. Also, specifications make it possible for the component's creator to hide the implementation from his clients. This decoupling of the client's and the creator's view on a component creates the freedom needed for reuse and evolution of the component.

But specification is difficult as it is very hard not to introduce errors into the specifications. Components can be under-specified, leaving space for misleading assumptions on the client or author side of a component. Or the component is over-specified, removing the freedom for reuse and evolution or creating a component that is impossible to implement correctly.

The *frame problem* [1] is the best known specification problem. It describes the whole range of problems connected to describing *what has not changed* during the execution of an operation.

But the frame problem is not the only problem when dealing with specifications. Others include the possibility to introduce errors into the specification, to create a "too implementation oriented" specification, to introduce contradictions or too strong or too weak specifications.

It cannot be expected that all problems connected to correct specifications will be solved in the near future. Still, a lot can be gained by creating even suboptimal specifications. Specifications can already be used for model checking, formal verification of implementations, debugging as well as for for documentation purposes.

In the context of software components, the documentation purpose of contracts is significant. Software components are independent items of deployment that enable developers reuse of code in changing contexts. Usually, the author is not the same person as the user of a component. Therefore the documentation is major communication instrument between the author and the client. High-quality documentation can remove ambiguity and avoid misunderstandings.

Software engineering should help developers in creating good specifications for their components. A key to the success of this endeavor is to provide software developers with methods and tools that enable developers to express their specifications in an easy and straight-forward way. Optimally, this is done before or at the time the code is created in the same programming language as the code itself.

1.1 Model-based Reasoning

One possibility for expressing the specifications is to relate the operations that the user of the component can invoke to transformations of models for the component. Models are mathematical constructs that can capture the structural semantics of the data stored in the component. Examples for models are sequences, sets, trees, graphs, integers — but also other properties like value-types or patterns.

The idea of using models to reason about program execution dates back to the very early days of program verification. Already 1972, Hoare [2] related program verification to abstract data types and models.

This approach to model-based reasoning on programs has been rediscovered in the recent years with the integration of specification languages into modern program development. The possibility of using static analysis tools on the basis of these specifications makes them very attractive. Examples of systems that have incorporated the idea of model based specifications into their systems are the Java Modeling Language [3] or the Larch/C++ tool [4]. The idea of applying models in the Design by Contract method have been developed by Meyer in [5].

1.2 Design by Contract

Design by Contract[6] is a methodology that has been introduced to software development by the Eiffel language [7]. It enables users to use assertion mechanisms to create specifications (i.e. contracts) for object-oriented software components[1].

Eiffel uses standard boolean expressions of the language to express these contracts. In contrast, many other specification languages use first-order predicate logic or similar description techniques. Using direct constructs of the language is very important as the software developer has to be motivated to use contracts while developing his application. This is one aspect of the seamless development method as advertised by the Eiffel language (see [8, p. 931]).

[1]There is no commonly accepted definition of the term "component". Many definitions exist. Here we consider components software elements that are meant to be used by other software elements and not directly in interaction with the environment of the software system.

Poetzsch-Heffter criticizes operational interface specifications as used in Eiffel (among other things) to be “suitable for verification”[9]. He comes to this conclusion from side-effect or termination problems that are introduced by boolean expressions and the difficulty of evaluating such contracts. Though this might be true for arbitrary boolean expressions, specific kinds of expression[2] can be fed into a model-checker or verification tools. There is no conceptual gap between the specification language and the implementation language. The benefit is that seamless development process, advertised by Eiffel, is retained.

In the following sections, I will explore how the Eiffel contract language can be used in conjunction with models to increase the expressiveness of these contracts. Then I describe a rigid set of rules to define and use these models and I will propose approaches to verify contracts based on models.

1.3 Immutable Objects as Models

Immutable objects are objects that never change once they are created. The identity of an immutable object — in contrast to a normal object — is defined by its state and not by some arbitrary pointer value. That means that two objects that have the same values are considered equal. Eiffel, like most other programming languages, does not directly support the construction of immutable objects[3]. It is possible to define constraints on the source of a class that will always result in the class to produce immutable objects.

Immutable objects have the benefit to represent the original idea of mathematical constructs much better than normal objects do. Another advantage — which is not directly connected to this paper — is that fields of immutable objects can be “in-lined” into programs, using copy operations instead of reference sharing and replacing objects “in place” instead of allocating new memory for the results of expressions. Immutable classes (classes that define immutable objects) are very similar to basic types in languages like Java or C#, or expanded types in Eiffel.

Immutable Eiffel objects offer the possibility to use the standard Eiffel mechanisms for model-based assertion checking without extending the language. They can benefit from all the advantages that normal Eiffel assertions have (e.g. runtime evaluation, documentation).

2 Using Immutable Objects for Specifications

The JML[11] specification language offers a large set of standard models to be applied in specifications. Figure 1 describes a sequence model in the context of Eiffel. The excerpt of the class SEQUENCE_MODEL shows the contract of two operations, concatenation of two sequences as well as a query for the first n elements of a sequence. As it is defining an immutable class, the operations never change the object itself, but produce new objects based on the current object and the arguments. The example also demonstrates how immutable objects can have much more expressive contracts.

[2]“Specific kinds of expressions” means expressions which are syntactically or semantically constrained to enable static analysis.

[3]A language that supports immutable objects is the Sather language [10]. In Sather, immutable objects replace the expanded construct available in Eiffel.

Especially the possibility to use recursive contract definitions (as demonstrated in the first_n feature) increases the strength of these contracts significantly.

```
class interface
   SEQUENCE_MODEL[G] -- (excerpt)

feature -- Extension
   concat (val: like Current): like Current
         -- Append val to the current sequence
      ensure
         count_added: Result.count = Current.count + val.count
         front_is_current: Result.first_n (Current.count).equals (Current)
         rest_is_val: Result.last_n (val.count).equals (val)

feature -- Deletion
   first_n (n: INTEGER): like Current
         -- First n elements of the sequence
      require
         n_not_too_small: n >= 0
         n_not_too_large: n <= count
      ensure
         empty_case: (n = 0) implies Result.is_empty
         non_empty_case: (n > 0) implies Result.front.equals (first_n (n - 1))

end -- class SEQUENCE_MODEL
```

Figure 1: Excerpt from a sequence model class

2.1 Model Definition

To start working with models, we define a special query in the class, that should contain model-based contracts that constructs the model from the current state of the instances. This query should be side-effect free, except for the creation of the model object.

As for every piece of unverified code, the creation of the model can be error prone. Fortunately, this does not introduce errors into the specification and is part of the class implementation. And although it could affect the execution of test-cases, errors should get detected quite easily using the model's own contracts.

For example, a stack class is defined in terms of a sequence model. This would result in a feature model: SEQUENCE_MODEL [G]. The implementation would need to access hidden features of the class to construct the stack, as the construction of the sequence is not possible from the four standard features push, pop, top and is_empty without producing side-effects.

The implementation of such a feature for class STACK in EiffelBase, based on the class SEQUENCE_MODEL mentioned above, is shown in figure 2. The feature linear_representation gives a linear representation of an object that can be accessed sequentially. It is used to get the internal representation of the stack and to produce the corresponding sequence.

```
feature{MODEL} -- Model definitions
   model: SEQUENCE_MODEL [G] is
         -- The model of the current stack
      local
         linear: LINEAR [G]
      do
         create Result.empty_sequence
         linear := linear_representation
         from linear.start until linear.exhausted loop
            Result := Result.append (linear.item)
            linear.forth
         end
      ensure
         model_not_void: Result /= Void
      end
```

Figure 2: Definition of a model for the class STACK in EiffelBase

On creation, the model is equivalent to a model that represents the structure of the created object. With every change to the structure, the model is changed as well. This is expressed by postconditions relating the new value of the model to its old value.

Most contracts can be expressed in terms of the model. The boolean expression language of Eiffel is now usable as a functional programming language allowing recursion and case differentiations (using the **implies** construct as shown in figure 1).

Eiffel enables the user to restrict features access to specific classes and its subclasses by adding the class name in curly braces behind the feature clause. The definition of the model can be exported to a special MODEL class, as shown in figure 2. This export restriction enables the user to optionally show or hide all the model contracts by using the class MODEL as a perspective class when generating the short or the interface view [4] from the code. This will cause not only the model definition to become hidden, but also all contracts that are using the model to express assertions. This is a result of the requirement that public contracts must not use any non-exported features. All contracts that are not relying on models (in the future called non-model contracts) remain in the interface description.

```
feature {MODEL} -- Model definitions
    seq_model: SEQUENCE_MODEL[G] is ...
    set_model: SET_MODEL[G] is ...

feature -- Modification
    extend (v: G)
            -- Extend the list by one element v.
        require
            not_in_model: not set_model.has (v)
        ensure
            model_extended: sequence_model = old sequence_model.append (v)
            added_to_model: set_model = old set_model.add(v)
invariant
    model_size: set_model.count = seq_model.count
    set_correspondance: seq_model.for_all (agent set_model.has (?))
    seq_correspondance: set_model.for_all (agent seq_model.has (?))
```

Figure 3: Multiple model specification

2.2 A Library of Models

It is possible to capture different conceptual dimensions for the same software component by using multiple models at once. To express that a list does not contain any elements twice, one model connects the list to a sequence and a second model connects the list to a set. How to define such contracts is demonstrated in figure 3: Multiple models — as shown — can be connected with each other using invariants. These invariants directly show the relation between the different models. In this case, the size of the sequence must be the same as the size of the set and every element in the sequence has to be an element of the set. The two models are connected with a "gluing invariant" (as available in the B language[12]).

On the basis of this observation, we can create a large number of possible models and provide them as a library to the user. The exact model can then be derived from combining different models. This drastically improves the coverage for potential datastructures by our models. It can be expected that the number of models for a proper standard model library should include the same number of models as there are currently for the JML[11] jmlspec library (around 70). The work on creating this library has begun and will be provided to the Eiffel developer community on the basis of the Eiffel Forum (Open-Source) license.

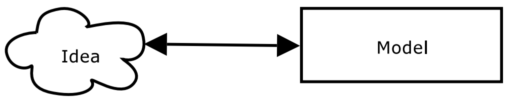
[4]The short and interface views are special interface descriptions that can be automatically created from Eiffel classes. These view only show the contracts of the features implemented or defined in a class (short view) or the whole interface of a class, including the features inherited from parent classes (interface view).

3 Formal Verification with Models

What exactly can be gained by using models for the verifiability of classes? Models can — on the basis of their contracts — be transformed into a representation that is understandable by model proof environments[5].

In the next subsections, I will show a number of different proof goals for the model-based contracts. The proof goals go into very different directions, either connecting the model with the implementation, with other contracts or even with mathematical theories.

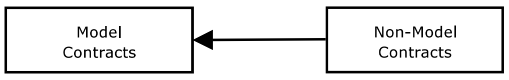
3.1 Semantic Verification of Models



Mathematics has a long tradition for standard definitions of constructs. Peano's Axioms defines the idea of natural numbers; set theory offers a number of axioms for the definitions of the operations on sets. An equivalence proof will create the trust that a given model really represents the mathematical construct that it was meant to be modelling.

Since the models are provided as a library, they can be reused in many different contexts. Thus, the benefit of doing these proofs scales and justifies the increased work while providing the models. I am planning to do these proofs for the Eiffel model library.

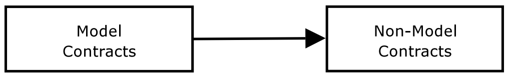
3.2 Correctness of Non-Model Contracts



Models enable the user to verify the correctness of the non-model contracts towards the model. This can be done by transforming the model as well as the non-model contracts into mathematical formulas and proving that the model implies the non-model contracts. This requires a mathematical theory also for the non-model contracts.

This strategy can also be used in a multi-model context. Here, it can be shown that the different models are really compatible with each other when connected by an invariant (as shown in figure 3).

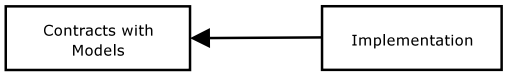
3.3 Completeness of Non-Model Contracts



[5]Some transformations have been done manually into the Atelier B proof environment for the B language. It can be expected that this process can be automated in the future.

By reversing the above strategy, the completeness of the non-model contracts can be shown. This is done by proving that the non-model contracts imply the model contracts.

3.4 Correctness of the Implementation



Last, but not least, models can be helpful when trying to prove the code. There are two reasons why a model-based contract should be easier to prove than the non-model contract.

First, the model-based contract is more complete concerning the behavior of the object. This is especially beneficial when trying to prove that one component behaves correctly when using another component known only by its contracts.

Second, as the model can be reused in many places, a large theory can be created for these models in the form of contracts and invariants. This theory is provided with the standard library. Such a theory improves the provability of code while using automated or manual provers.

4 Summary and Conclusion

Behavioral specifications with models as introduced by JML or the Larch tool offer a powerful mechanism to specify contracts. This paper shows how model-based specifications can be fully implemented using standard Eiffel and the Design by Contract method.

It has been shown how immutable objects can be used to capture the behavior in specifications. Model-based specifications integrate seamlessly into current Eiffel implementations without the introduction of new compiler features or the extension of the Eiffel syntax. Models can be extracted at any time from classes using special model creation features.

Normal export restrictions can be used to show or hide the model-based contracts from the short and interface views of classes. Models can be left in the code and used only by specific analysis tools, while they are transparent to developers.

Different structural aspects for the same class can be captured by different models. The combinations of different models empower the user to capture even complex types with the help of models. Gluing invariants connect these models.

Different verification techniques offer the possibility to reason about model-based contracts. These proofs can be analysis oriented, by showing that the contract really identifies the mathematic idea. They can also be design oriented by relating models to non-model contracts. Finally they can be implementation oriented, trying to prove the code correctness.

5 Acknowledgements

I would like to thank my supervisor, Prof. Bertrand Meyer, for developing the initial idea of the application of models in contracts and ideas about their proofs. I would also thank Karine Arnout, Susanne Cech, Till Bay and Markus Keller for reviewing this paper and giving me valuable hints and comments.

References

[1] J. McCarthy and P. J. Hayes, "Some philosophical problems from the standpoint of artificial intelligence," in *Machine Intelligence 4* (B. Meltzer and D. Michie, eds.), pp. 463–502, Edinburgh University Press, 1969. reprinted in McC90.

[2] C. Hoare, "Proof of correctness of data representations," *Acta Informatica*, vol. 1, no. 4, pp. 271–281, 1972.

[3] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll, "An overview of JML tools and applications," Tech. Rep. R0309, NIII, 2003.

[4] G. T. Leavens, "Larch/C++, an interface specification language for C++," tech. rep., Iowa State University, Ames, Iowa 50011 USA, August 1997.

[5] B. Meyer, "Towards practical proofs of class correctness," in *ZB 2003* (D. B. et al., ed.), LNCS 2651, pp. pp. 359–387, Springer-Verlag Berlin, 2003.

[6] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, pp. 40–51, October 1992.

[7] B. Meyer, *Eiffel: the language*. Prentice Hall, New York, NY, 1992.

[8] B. Meyer, *Object-Oriented Software Construction*. Prentice Hall, 2 ed., 1997.

[9] A. Poetzsch-Heffter, "Specification and verification of object-oriented programs." Habilitationsschrift, Technische Univeristät München, January 1997.

[10] S. Omohundro and C.-C. Lim, "The sather language and libraries," Tech. Rep. TR-92-017, International Computer Science Institute, Berkeley, Ca., 1991.

[11] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: A behavioral interface specification language for Java," Tech. Rep. 98-06t, Department of Computer Science, Iowa State University, June 1998.

[12] J.-R. Abrial, *The B-Book – assigning programs to meanings*. Cambridge University Press, 1996.