Chair of
Software Engineering

Complete Contracts for EiffelBase

Marco Zietzling

02-906-998
Semester Thesis
Summer 2007

Chair of Software Engineering
Department of Computer Science
ETH Zürich

Bernd Schoeller
Prof. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

Abstract

Formal verification of programs is a very important topic. But on the other hand it is rather difficult to be able to verify a program if only the source code is given. Using Eiffel, not only the code itself can be used but also contracts. Each feature has preconditions and postconditions which can be used to enhance and simplify the verification process. But unfortunately the contracts currently used in Eiffel are under-specifications in the sense that there are always some properties when invoking Eiffel features that are not expressed through its contracts. To be able to write complete contracts, Dynamic Frame Contracts are added for each feature and Models are used to enhance existing contracts.

In this semester thesis, we try to redevelop a major part of the Eiffel-Base Library to support Dynamic Frames and Models. A focus is on the LINKED_LIST class and all its supporting classes to allow full functional specifications and therefore formal verification.

Acknowledgments

I would like to thank my supervisor Bernd Schoeller for his competent support and helpful advices. He always had time for answering my questions and giving hints concerning implementation and general library design. I also would like to thank the other members of the *ballet* team (Raphael Mack and Jason (Yi) Wei).

Additionally I would like to thank my girlfriend, my family and my friends for the support during my studies and this thesis.

Contents

Chapter 1

Introduction

1.1 Motivation

The contracts currently used in Eiffel (cf. [Mey92b, Mey92a, Mey97]) are strong under-specifications. There are always some properties when invoking Eiffel features that are not expressed through the contracts. This is specially true when it comes to express what does not change through feature invocation. This makes the formal verification of Eiffel impossible, as in the case of verification we have to assume the worst possible implementation that still satisfies the contract.

To overcome these lacks, two approaches will be used to add full functional specifications to Eiffel: Frames and Models.

- Dynamic Frame Contracts (DFC) (cf. [SO]) enable us to talk about infinite sets of objects. They are contractual expressions describing the read effect and the write effect of a feature. With DFC we can solve the *frame problem* (cf. Section 2.1) and exclude unwanted side effects.
- Models are mathematical structures which describe the abstract state of an object. These mathematical structures are built out of sets, relations, functions and sequences. The currently available library called MML (cf. [Wid04], downloadable from [MML07]) provides this functionality. This model library is based on a typed set theory of finite sets.

This semester thesis is an effort to redevelop a major part of the EiffelBase Library to support Dynamic Frame Contracts and Models. The goal is to

redesign and implement the LINKED_LIST class in the EiffelBase Library and all supporting classes to allow full functional specifications and therefore formal verification. The redesign of the EiffelBase Library structure should lead to a cleaner and smoother Base Library.

1.2 Related Work

Since the *frame problem* (cf. Section 2.1) is a fairly old problem, a lot of ideas and solutions exist. Earlier solutions [Mül01, LN02, LM04] impose several restrictions to the programmer. The newest solution by Kassios [Kas06] proposes a formal theory without programming restrictions. Based on this solution, Schoeller et al. added Dynamic Frames to the Eiffel language (cf. [SO, Sch06]).

The idea of using models for program specification is relatively old (cf. [Hoa72]). JML [LBR98] provides models for the Java language. Spec# [BLS04] provides predefined sets and sequences for the C# language. For Eiffel, the MML [Wid04, Sch03, SWM06] has been developed.

This semester thesis is the first approach to combine these two techniques (Dynamic Frames and Models) in one single design to allow full specifications and therefore formal verification.

1.3 Overview

In Chapter 2 we present a short overview over the concept of Frames. In Chapter 3 a short introduction into the concept of Models is given. In Chapter 4 the redesigned and reimplemented Library is presented. In particular the new Library using DFCs and Models is introduced (Section 4.1) and differences to the original EiffelBase Library are emphasized (Section 4.1.1). Finally we present our results, conclusions and ideas for future work in Chapter 5.

Chapter 2

Frames

In this Chapter the *frame problem* is introduced and a short overview over the topic of Frames and in particular Dynamic Frames is given. For more information see also [Kas06, SO, Sch06].

2.1 Frame problem

The *frame problem*, the problem of how to formalize framing requirements, is a central issue in formal logic, first studied in the context of artificial intelligence in the 1960s. It was initially formulated as the problem of expressing a dynamical domain in logic without explicitly specifying which conditions are *not* affected by an action. Artificial intelligence tries to model the logical thinking of a human being. For the brain it is fairly simple to connect many things logically. But for a computer or a robot, an infinite number of things has to be specified. The *frame problem* is that specifying only which conditions are changed by an action do not allow, in logic, to conclude that all other conditions are not changed. To solve this problem one could write so called frame axioms which are logic formulas describing explicitly the non-effects of each action. But since most actions do not affect most properties this solution is not suitable for large systems.

The *frame problem* has then been taken up by philosophers. In philosophy, it is about rationality in general, whether it is possible to limit the scope of the reasoning required to derive the consequences on an action. In computer science the problem gained importance concerning program verification. Program verification tools are but logic tools which analyze a pro-

gram. Knowing what the program should do, for example from a contract, and then decide if the program really does things right.

A significant aspect of the behavior of any operation is what parts of the world it leaves unchanged. Usually, a specification for an operation is split into a functional requirement, which describes what changes the specified operation brings about and a framing requirement, which describes the frame of the operation, the part of the world which the operation is allowed to change. Anything outside the frame is left unchanged. In a modular setting, we don't know all the variables of the program at the time we specify a computation. There is no proper way to write a contract specifying that only some variables are changed but any other variable is left unchanged. One solution would be that such a statement is translated into a relational specification at the client side when all involved variables are known. But in the presence of pointers, this translation might be unsound. Independence of variables is important for guaranteeing absence of abstract aliasing. But the combination of two important features of object oriented programming, namely encapsulation and the support for pointers, makes the *frame problem* really hard to solve because the combination introduces the possibility of aliasing.

2.2 Dynamic Frames

As stated above, the *frame problem* in general is to specify which variables are affected by a computation and which are not. Existing solutions concerning the *frame problem* guarantee the absence of abstract aliasing but for the price of more or less strong restrictions for the programmer. [Kas06] introduced a formal theory that supports specification variables and pointers without programming restrictions. The specification language is strong enough to express the desired property that at the present state, the values of two variables are independent.

For Kassios, a frame is nothing more than a specification variable whose value is a set of memory locations. The new contribution of Kassios is that the frame itself is part of the state. The frame is like a function depending on the state. It is not a constant but it is dynamic which means the set of memory locations can change. For example suppose we have a frame which frames a whole list of elements. If we insert a new element, it is automatically framed too and its memory address is added to the frame variable. The virtual address of the frame or the specification variable respectively can be itself

part of the frame. Which in fact means that frames could frame themselves.

Kassios introduced three new definitions or framing specifications which are defined below. The following notation is used: f defines a frame, v defines a specification variable, v' defines the final value of a specification value, x and y define program variables, σ defines a state, Used defines the set of used memory locations, Unused defines the set of unused memory locations. For more detailed explanations please cf. Section 2.2 in [Kas06].

The first specification (Equation 2.1) is *preservation* which is satisfied by every computation that does not touch frame f.

$$\Xi f := \sigma' \triangleright f = \sigma \triangleright f \tag{2.1}$$

The second specification (Equation 2.2) is *modification* which is satisfied by every computation that only touches region f or at most allocates new memory. That implies all allocated locations other than f are not touched.

$$\Delta f := \Xi(\text{Used} \setminus f) \tag{2.2}$$

The third specification (Equation 2.3) is the *swinging pivot requirement* which does not allow the frame f to increase in any other way than allocation of new memory.

$$\Lambda f := f' \subseteq f \cup \text{Unused} \tag{2.3}$$

Equation 2.4 states that a variable v depends only on locations in f. Leaving these locations untouched preserves the value of v.

$$f \textbf{ frames } v := \forall \sigma' \cdot \Xi f \Rightarrow v' = v \tag{2.4}$$

Independence of two variables can be expressed as disjointness of dynamic frames. Disjointness of frames is an important property. It is desired to establish and preserve disjointness. To preserve disjointness, dynamic frames usually frame themselves.

$$f \textbf{ frames } x \ \wedge \ g \textbf{ frames } y \ \wedge \ \mathit{disjoint}[f; g] \ \wedge \ \Delta f \Rightarrow y' = y \tag{2.5}$$

$$\mathit{disjoint}[f; g] \ \wedge \ \Delta f \ \wedge \ \Lambda f \Rightarrow \mathit{disjoint}[f'; g'] \tag{2.6}$$

Listing 2.1: put feature of the CC_COLLECTION class.

```
feature -- Element change
  put (v: G) is
      -- Ensure that structure includes 'v'.
    require
      can_add_element: can_put (v)
    use
      use_own_representation: representation
    modify
      modify_own_representation: representation
    deferred
    ensure
      model_updated: model |=| old model.extended (v)
      model_corresponds: model.contains (v)
      confined representation
    end
```

2.3 Dynamic Frames in Eiffel

Every feature in every class in the new base library uses new contract clauses. They are called *use* and *modify* and are used just like normal contracts. An example is given in Listing 2.1.

The *use* clause specifies which frames are used (read) by calling this feature. In contrast, the *modify* clause specifies which frames are possibly modified by calling this feature. If a feature is a query, only a *use* clause is necessary. If a feature is a command, both *use* and *modify* clauses are needed. Additionally, a command should provide in its *ensure* clause the statement '**confined** *frame_name*'. This statement ensures the third of Kassios framing specifications (Equation 2.3).

Chapter 3

Models

Reusable components should have precise specifications of their functionality. The creation of such precise specifications is a difficult but important task because precise specifications enable the client to understand the requirements for the usage of these components. An additional benefit for the component's creator is implementation hiding. Such a decoupling of different views on a component (author versus client) is needed for reuse and evolution of these components. [Sch03] gives an overview over the problem of creating precise specifications.

The creation of specifications in general is error-prone and often components are under-specified or over-specified. If they are under-specified, they leave too much space for misleading assumptions. If they are over-specified, they possibly violate information hiding, making reuse and evolution nearly impossible. As already described in Section 2.1, the *frame problem* is the best known specification problem because it is nearly impossible to clearly describe what has not changed during execution of a feature.

Models are mathematical constructs that describe the abstract behavior of a data type without dealing with the actual implementation. Examples are sequences, sets, bags or relations. The idea was to enable precise specifications through the usage of models. Even Hoare in 1972 (cf. [Hoa72]) related program verification to abstract data types and models.

3.1 Mathematical Model Library (MML)

The mathematical model library (MML) resulted from the Master Thesis of Tobias Widmer (cf. [Wid04]) and an improved version can be downloaded from [MML07]. This library introduced a way to make specifications complete by using pure mathematical models that describe the abstract behavior of data types.

Models are represented by immutable objects. Immutable objects are objects that never change once they are created. Therefore the identity of such a model objects is defined through its state. That means that two objects that have the same state (and therefore have the same values) are considered equal. Operations on immutable objects do not change the object itself, but produce new objects based on the current object and the arguments.

The MML is used in this semester thesis to create models in classes and to be able to write model contracts for features of the new base library.

3.2 Models in Eiffel

Every class in the new base library contains a special query that constructs the model from the current state of the instance. This query should be side-effect free and should only create the model object. This query is named *model*.

An example of such a *model* query is given in Listing 3.1. Here a bag is chosen as a model for a general container to contain its elements. The bag is created using a default implementation from the MML. Then all elements of the container are added to the bag object using an intermediate linear representation.

Such a *model* query is then used in contracts of features of this class. An example is given in Listing 3.2 in the *ensure* clause.

Model queries themselves can be built from several sub-models. An example is given in Listing 3.3. Here the *model* is built from a *model_sequence* which represents the sequence itself with all elements and a *model_cursor* which represents the current cursor position of the internal cursor. The final *model* is a pair of *model_sequence* and *model_cursor*. Two traversable objects are therefore equal if and only if they contain the same elements in the same order and the internal cursor is at the same location.

Listing 3.1: model feature of the CC_CONTAINER class.

```
feature -- Model
  model: MML_BAG [G] is
      -- Model of a general container
    use
      use_own_representation: representation
    local
      linear: CC_LINEAR [G]
    do
      create {MML_DEFAULT_BAG [G]} Result.make_empty
      linear := linear_representation

      from
        linear.start
      until
        linear.off
      loop
        Result := Result.extended (linear.item)
        linear.forth
      end
    ensure
      result_not_void: Result /= Void
    end
```

Listing 3.2: put feature of the CC_COLLECTION class.

```
feature -- Element change
  put (v: G) is
      -- Ensure that structure includes 'v'.
    require
      can_add_element: can_put (v)
    use
      use_own_representation: representation
    modify
      modify_own_representation: representation
    deferred
    ensure
      model_updated: model |=| old model.extended (v)
      model_corresponds: model.contains (v)
      confined representation
    end
```

Listing 3.3: Model and sub-models from the CC_TRAVERSABLE class.

```
feature -- Model
  model_sequence: MML_SEQUENCE [G] is
      -- Model of a general traversable structure
    use
      use_own_representation: representation
    local
      linear: CC_LINEAR [G]
    do
      create {MML_DEFAULT_SEQUENCE [G]} Result.
         make_empty
      linear := linear_representation

      from
        linear.start
      until
        linear.off
      loop
        Result := Result.extended (linear.item)
        linear.forth
      end
    ensure
      result_not_void: Result /= Void
    end

  model_cursor: INTEGER is
      -- Model of a cursor for traversal
    use
      use_own_representation: representation
    deferred
    end

  model: MML_PAIR [MML_SEQUENCE [G], INTEGER] is
      -- Model of the traversable structure
    use
      use_own_representation: representation
    do
      create {MML_DEFAULT_PAIR [MML_SEQUENCE [G],
        INTEGER]} Result.make_from (model_sequence,
         model_cursor)
    ensure
      result_not_void: Result /= Void
    end
```

Chapter 4

New library with Complete contracts

One part of this semester thesis was the redesign of the EiffelBase Library (cf. [Eif05]). The goal was to get a cleaner and smoother base library. The new base library should use Dynamic Frame Contracts and Models as discussed in Section 2.3 and Section 3.2. The redesign was focused on LINKED_LIST class and all supporting classes.

The current EiffelBase Library is over engineered and sometimes messy. Some features have names which do not comply with the own Eiffel style guidelines (cf. Chapter 26 in [Mey97]) or which do not reflect the true meaning of this feature or even of a class. An example for this is cursor which in Eiffel is rather a cursor position rather than a real cursor.

4.1 EiffelBase with Complete Contracts

As stated above the new base library was focused on the LINKED_LIST class and all supporting classes. In total there are about 20 classes in the library. Every class has a 'CC_' prefix in front of the name to be able to use this library in parallel to the original EiffelBase Library.

4.1.1 Differences to EiffelBase

Actually there are a lot of differences to the original EiffelBase Library. We try to give a short overview over these changes and explain the basic concepts and ideas that led to these differences.

Finite containers In contrast to the EiffelBase Library the new library is always finite. To be more precise, only finite containers are available.

TABLE not a heir of BAG The TABLE class is no longer a heir of the BAG class. Originally in the EiffelBase Library the idea was that a table is a bag of elements which can be directly accessed through an index. But for us the mathematical concept behind a bag and a table are too different to be directly related.

ACTIVE not a heir of BAG The ACTIVE class is no longer a heir of the BAG class. An active object is defined as a data structure, which at every stage has a possibly undefined current item. Here again the mathematical concept behind an active and a bag are too different to be directly related.

INDEXABLE not a heir of TABLE The INDEXABLE class is no longer a heir of the TABLE class.

Feature renaming Some features in the original EiffelBase Library have names which do not reflect the actual meaning or which are not compliant with the Eiffel style guidelines (cf. Chapter 26 in [Mey97]).

- Feature cursor in CURSOR_STRUCTURE class has been renamed to cursor_position because it actually describes a position rather than a cursor object as known for example from [Bez07].
- Features isfirst and islast in the LIST class have been renamed to is_first and is_last to comply with the Eiffel style guidelines.
- Features replace and remove from ACTIVE have been renamed to replace_item_with and remove_item in CC_ACTIVE to make clear that these features operate on the current item.

BILINEAR class In the EiffelBase Library the BILINEAR class inherits two times from the LINEAR class. One time to actually inherit features which are then redefined to fit the bilinear case. The other time to be able to use features of the original linear class inside the implementation. In the new library, only one inheritance relation is used. The necessary implementation is done manually.

extend vs. put vs. force We tried to really define and implement differences between these features which are doing essentially the same but having different meanings.

- extend is defined as the non-defensive version of putting a new element into a container. This implies the strongest preconditions.
- put is defined as the medium-defensive version. This is more or less the mathematical point of view. For example if an element is already included in a set, the operation is just ignored.
- force is defined as the defensive version. This implies the weakest preconditions. This feature does everything to ensure a successful operation. The force feature would for example double an array if there is no more free space in it.

To be able to realize these different behaviors, new flags have been introduced. In particular can_extend(x) and can_put(x) are added to the preconditions of extend and put features.

Fill feature discarded The fill feature has been discarded in the COLLECTION class because this would have introduced some nasty model contracts including under-specifications.

Command-query separation A strict command-query separation has been introduced. This was not the case in the original EiffelBase Library. Some features, mainly in the LINEAR class, were queries but in fact they move the internal cursor which obviously should not be the case because queries are defined as side-effect free. The solution used in the new library was to save the internal cursor, do the operation and then move the cursor back to the old position. This was also necessary for the new Dynamic Frame Contracts (cf. Section 2.3) because a query has only a *use* clause, a command has a *use* and a *modify* clause.

CURSOR renamed to CURSOR_POSITION Because the cursor in the EiffelBase Library is in fact a cursor position, the corresponding class in the new library has been named CURSOR_POSITION. The cursor feature has also been renamed to cursor_position.

Discarded classes Some classes which are supporting classes of the LINKED_LIST class in the EiffelBase Library have been discarded in the new library.

- The classes BOX, FINITE and UNBOUNDED have been discarded because, as stated above, only finite containers are available.

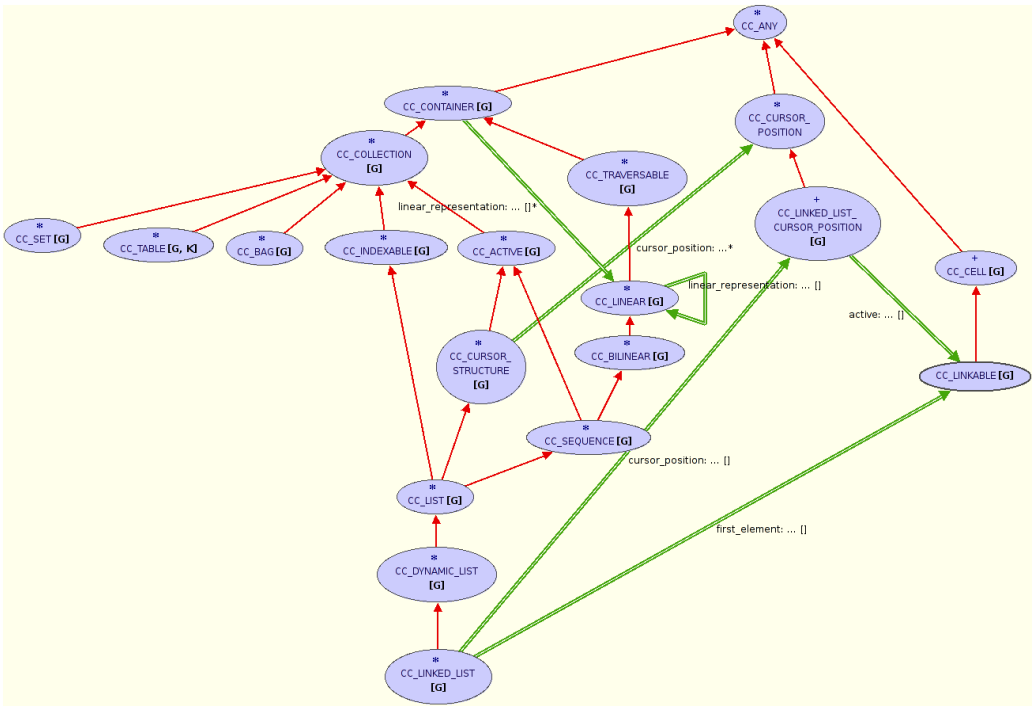

Figure 4.1: Overview over all classes in the new EiffelBase Library with complete contracts.

- The CHAIN class has been discarded and all features have been directly included in the LIST class.
- The DYNAMIC_CHAIN class has been discarded and all features have been directly included in the DYNAMIC_LIST class.

Figure 4.1 shows the final 'CC_EiffelBase' Library as a BON diagram. The only effective class is CC_LINKED_LIST. This class provides the same functionality as the original LINKED_LIST class from the EiffelBase Library.

Chapter 5

Results and Conclusions

5.1 Usage of Frames and Models in Eiffel

A part of this semester thesis was to experiment with the usage of Dynamic Frame Contracts and models in practice. An easy usage is of course necessary because developers should be encouraged to use these new mechanism. The usage would definitely not be easy if developers have to learn a new language or write too much just to give a complete specification.

The usage of frames and Dynamic Frame Contracts in particular is rather simple. As stated before in Section 2.3, if a feature is a query, only a *use* clause is necessary. If a feature is a command, both *use* and *modify* clauses are needed. Additionally, a command should provide in its *ensure* clause the statement '**confined** *frame_name*'. This is straight forward and not too much overhead to write and a part of the additional text could be done automatically using an improved feature creation wizard.

When it comes to models, the developer himself has to think which model could be used to abstractly represent the class he is about to write. Additionally, for each feature model contracts have to be added and class invariants concerning its model have to be written. This is not straight forward and needs more thinking. But this has also positive aspects because the developer has to clearly specify what its class is about to do by defining a model for it.

Table 5.1: Time measurements of the test case with 1'000 elements.

	LINKED_LIST		CC_LINKED_LIST	
Contracts	with	without	with	without
Time	0m0.005s	0m0.003s	0m13.539s	0m0.003s

Table 5.2: Time measurements of the test case with 100'000 elements.

	LINKED_LIST		CC_LINKED_LIST	
Contracts	with	without	with	without
Time	0m0.603s	0m0.045s	162m11.846s (canceled)	0m0.046s

5.2 Time measurements

Two simple test cases have been created. The first test case creates an empty list and inserts 1'000 elements, the second test case does the same but inserts 100'000 elements. This is done for the original LINKED_LIST class from the EiffelBase Library and for the new CC_LINKED_LIST class of the new library. The execution of these test runs has been measured using the unix 'time' command. The test runs were done in *finalized* mode of the Eiffel project. One time with all contracts enabled and one time without contracts at all. For testing, the following computer was used: Dell Inspiron 6400 notebook with Intel Core2Duo processor 1.8 GHz and 2 GB RAM on Ubuntu Linux 6.06 with a 2.6.15 kernel. In Table 5.1 and Table 5.2 the timing results for each test case are shown. Remark: the test case with 100'000 elements in the CC_LINKED_LIST class with all contracts enabled has been canceled after nearly three hours of computation (cf. Table 5.2).

As can be seen by the results of Table 5.1 and Table 5.2, the test runs without contracts using LINKED_LIST and CC_LINKED_LIST take nearly the same time which was expected. In both test cases, the test runs using CC_LINKED_LIST with contracts take definitively more time because of the model contracts. Models, as stated already in Section 3.1, are immutable objects and therefore an operation on an immutable object does not change the object itself, but produce a new object based on the current object and the arguments. This leads to a massive computational overhead. In the first test case with 1'000 elements, the execution time using CC_LINKED_LIST is nearly 4'500 times larger if model contracts are enabled then if they are disabled. This also explains the cancellation of the second test case with 100'000 elements using CC_LINKED_LIST with contracts enabled.

5.3 Future Work

A major goal for the future is certainly to add more effective classes to the new EiffelBase Library with Complete Contracts to be able to use the library.

Other suggestions for future investigation are listed below in random order:

- Better integration into the Eiffel Studio IDE so that Dynamic Frame Contracts could be generated automatically.
- Integrate a possibility to deactivate model contracts during runtime. Like in the *finalizing* dialog where contracts in general can be enabled or disabled.

Bibliography

[Bez07] Eric Bezault. Gobo eiffel project, April 2007. http://www.gobosoft.com/.

[BLS04] Mike Barnett, K.R.M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. *CASSIS*, LNCS 3362, 2004.

[Eif05] Eiffel Software. *EiffelBase Library*, August 2005. http://archive.eiffel.com/products/base/.

[Hoa72] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[Kas06] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. *Formal Methods*, 2006.

[LBR98] 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.

[LM04] K.R.M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *Proceedings of ECOOP'04: European Conference on Object-Oriented Programming*, volume 3068, pages 491–516, 2004.

[LN02] K.R.M. Leino and G. Nelson. Data abstraction and information hiding. In *TOPLAS*, volume 24, 2002.

[Mey92a] Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, October 1992.

[Mey92b] Bertrand Meyer. *Eiffel: the language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2 edition, 1997.

[MML07] *Mathematical Model Library (MML)*, January 2007. http://se.inf.ethz.ch/people/schoeller/mml.html.

[Mül01] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, Fern-Universität Hagen, 2001.

[Sch03] 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.

[Sch06] Bernd Schoeller. Eiffel0: An object-oriented language with dynamic frame contracts. Technical Report 542, Chair of Software Engineering, Department of Computer Science, ETH Zurich, December 2006.

[SO] Bernd Schoeller and Jonathan Ostroff. Dynamic frame contracts. to be published.

[SWM06] Bernd Schoeller, Tobias Widmer, and Bertrand Meyer. Making specifications complete through models. *Architecting Systems with Trustworthy Components*, 3938, 2006.

[Wid04] Tobias Widmer. Reusable mathematical models. Master's thesis, ETH Zurich, July 2004.