DISS. ETH NO. 17610

Making classes provable through contracts, models and frames

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
(ETH Zürich)

for the degree of
Doctor of Sciences

presented by
Bernd Schoeller
Diplom-Informatiker, TU Berlin

born
April 5th, 1974

citizen of
Federal Republic of Germany

accepted on the recommendation of
Prof. Dr. Bertrand Meyer, examiner
Prof. Dr. Martin Odersky, co-examiner
Prof. Dr. Jonathan S. Ostroff, co-examiner

2007

ABSTRACT

Software correctness is a relation between code and a specification of the expected behavior of the software component. Without proper specifications, correct software cannot be defined.

The *Design by Contract* methodology is a way to tightly integrate specifications into software development. It has proved to be a light-weight and at the same time powerful description technique that is accepted by software developers. In its more than 20 years of existence, it has demonstrated many uses: documentation, understanding object-oriented inheritance, runtime assertion checking, or fully automated testing.

This thesis approaches the formal verification of contracted code. It conducts an analysis of Eiffel and how contracts are expressed in the language as it is now. It formalizes the programming language providing an operational semantics and a formal list of correctness conditions in terms of this operational semantics.

It introduces the concept of *axiomatic classes* and provides a full library of axiomatic classes, called the *mathematical model library* to overcome problems of contracts on *unbounded data structures*.

This thesis argues that modular verification is essential for the reuse of trusted object-oriented code. Modular verification introduces problems with hidden interference of components, known as the *frame problem*. This thesis introduces the concept of *dynamic frame contracts* and shows how such contracts can overcome the frame problem, at the same time retaining full information hiding and being faithful to the inheritance relation.

The thesis includes an experimental implementation of a fully automated verifier called *Ballet*. This verifier transforms Eiffel into proof obligations that are handed over to a fully automated theorem prover.

ZUSAMMENFASSUNG

Korrekte Software beschreibt eine Relation zwischen Programmtext und einer Spezifikation des zu erwartenden Verhaltens einer Software-Komponente. Ohne eine geeignete Spezifikation ist der Begriff *korrekter Software* bedeutungslos.

Design by Contract ist eine Methode die Spezifikationen in den Softwareentwicklungsprozess einbindet. Sie hat sich bewährt als eine einfache und gleichermaßen mächtige Technologie und wird von Entwicklern angenommen. In den 20 Jahren seit ihrer Entstehung haben sich viele Anwendungsfelder für Design by Contract ergeben: Dokumentation, Verständnis der objektorientierten Vererbung, Überprüfung zur Laufzeit, oder vollautomatisches Testen.

Diese Dissertation beschäftigt sich mit der formalen Verifikation von *contracted Code*. Sie analysiert die Programmiersprache Eiffel und wie in dieser Programmiersprache Verträge (Contracts) benutzt werden können. Eine operationelle Semantik formalisiert die Programmiersprache. Die Beweisverpflichtungen zur Überprüfung der Korrektheit werde definiert.

Sie entwickelt den Begriff der *axiomatischen Klasse* und entwickelt eine Bibliothek solcher Klassen, die *Mathematical Model Library*, um Probleme mit Verträgen über unbeschränkten Datenstrukturen zu beheben.

Diese Dissertation argumentiert, dass modulares Beweisen essentiell für die Wiederverwendungen von vertrauenswürdigen, objektorientierten Komponenten ist. Eine modulare Beweisführung scheitert an versteckten Interferenzen verschiedener Komponenten. Dieses Problem ist unter dem Namen *Frame Problem* bekannt. Diese Dissertation erweitert Design by Contracts um *Dynamic Frame Contracts*, um diese Probleme zu beheben. Dynamic Frame Contracts bewahren die Kapselung von Komponenten und sind verträglich mit der Vererbungsbeziehung der objektorientierten Entwicklung.

Die Dissertation enthält die experimentelle Implementierung eines vollautomatischen Beweiswerkzeugs mit dem Namen *Ballet*. Dieses Werkzeug wandelt Eiffel Programmtext in Beweisverpflichtungen für einen automatischen Beweiser um.

Acknowledgements

First and foremost, I would like to thank my supervisor *Prof. Bertrand Meyer*. Working as part of his research group and his constant personal input and feedback formed the basis for my understanding of software engineering, language semantics and verification of object-oriented programs.

I would like to thank *Prof. Jonathan Ostroff*. The possibility to stay with his research group in Toronto had a great influence on my understanding of models and frames. I also want to include the other members of his research group, *Faraz Torshizi*, *Hai Feng Huang*, *David Makalsky* and *Chen-Wei Wang*.

My gratitudes goes many people at the "Chair of Software Engineering" at ETH Zürich. This includes *Prof. Peter Müller*, who was always available for questions on Spec# and JML, and the other members: *Manuel Oriol*, *Till Bay*, *Ilinca Ciupa*, *Adam Darvas*, *Werner Dietl*, *Hermann Lehner*, *Andreas Leitner*, *Yann Müller*, *Martin Nordio*, *Michela Pedroni*, *Marco Piccioni*, *Joseph Ruskiewicz* and *Jason Wei*, and the former members *Vijay d'Silva*, *Arnaud Bailly*, *Karine Arnout*, *Susanne Cech Previtali* and *Piotr Nienaltowski*. Each one of them, through smaller or more extensive discussions, contributed to the overall result of this thesis.

If would like to thank *Prof. Martin Odersky*, for volunteering on short notice to become second co-examiner, for giving me the opportunity to present my work at his research group at EPFL and for the feedback given.

Finally, I would like to thank all members of my family (the number increased significantly during my studies) and my girlfriend, *Jana Steinbrück*, for providing all the needed personal support required to endure the ups and downs of my scientific work.

CONTENTS

List of Definitions

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Current software development is characterized by a high increase in complexity. Each generation of products and systems exceeds the previous one by an order of magnitude in terms of line of code or number of people involved. To keep up with this development, programmers are forced to a high level of reuse and adaptation of existing systems. Software is constantly modified, few systems are written from scratch. Legacy software is ubiquitous.

At the same time, the demands for quality software are rising: The integration of more and more systems into the Internet increases the need for security against malicious attackers. Transportation systems with lives at stake have a high demand for software reliability and robustness. Financial systems demand correctness. Constant changes require good maintainability.

1.1 Towards trusted components

Software reuse and software quality complement each other. Without a high level of reuse, the extra effort that needs to be invested into software quality does not pay off. Without a high level of software quality, reuse is difficult; the more a piece of software is reused, the more its deficiencies come to the fore.

This relation builds a vicious circle: many reusable components are of bad quality, and developers suffer from these deficiencies. They do not trust software quality and prefer to re-implement instead of reusing (even high quality) components. This has the consequence that the level of reuse remains low and an investment into the quality of the software

components does not pay off.

To break this circle, two problems must be solved: First, software quality must be improved by investing into reusable software. Software engineering can help in this progress by developing new methods and tools for software development that detect quality flaws and enforce quality standards.

Second, we have to find better ways to build trust into software. We have to provide developers with evidence that the software is of a certain level of quality. This evidence can be: access to the source, clear specifications and documentations, certification by independent organizations or reproducible test. Also, an already high-level of reuse creates trust, resulting in a self-accelerating process.

Software quality has many dimensions. The focus of this work is *functional correctness*. Intuitively, we call software correct, if it does what we expect it to do. Formally, a software is correct if and only if, when executed on a well-defined machine, will transform the state of the machine in way that is defined by its *functional specification*.

Proving that a software component is *correct with respect to a given functional specification* means to establish a relation between specification and implementation. It does not show that the software is indeed valid for the given problem, but it allows a critical reduction of complexity, as the specification is normally much simpler and more revisable than the implementation. This creates trust into the implementation.

1.2 The grand challenge of the verifying compiler

In 2005, Hoare and Misra proposed a vision of a grand challenge:

> *The ideal of correct software has long been the goal of research in Computer Science. We now have a good theoretical understanding of how to describe what programs do, how they do it, and why they work. This understanding has already been applied to the design, development and manual verification of simple programs of moderate size that are used in critical applications. Automatic verification could greatly extend the benefits of this technology.*
>
> [...] *the time is ripe to embark on an international Grand Challenge project to construct a program verifier that would use logical proof to give an automatic check of the correctness of programs submitted to it.* (Hoare and Misra, [35])

The grand challenge of developing a verifying compiler contains many small steps that have to be taken one at a time. Programming languages have to be developed on the basis of formal theory, with a sound understanding of what the expected outcome of an execution is. Provers have to become powerful enough to take off the burden from the developer to do the proof himself. Functional specifications have to be written, they are an inherent requirement to formal verification. Developers have to be taught how to use the new programming and specification languages, and the supporting tools for verification.

1.3 Summary and outlook

This thesis contributes to the "Grand Challenge" project as proposed by Hoare and Misra. It presents a full approach to the verification of object-oriented programs, written in a real-world programming language.

Starting with the analysis of the language and its properties, it develops an full operational semantics covering nearly all features of the language. A framework of model classes is developed to make it possible for the specification language to talk about complex data structures. The programming language is extended by frame specifications, required for modular software verification and bottom-up development. Finally, all theories are integrated into a tool, "Ballet", which illustrates the automatic verification of code and integration of formal methods into software development.

CHAPTER 2

ESSENTIAL RESULTS

This chapter guides the reader through the problem domain and essential results of the presented thesis. The short description illustrates problems and corresponding solutions with examples. Later chapters contain the formalization and implementation of the solution strategies.

2.1 Overview

The overall goal of this thesis is to advance theories and technologies for integrating formal methods and verification for the Eiffel programming language, the Design by Contract methodology and the object-oriented paradigm as a whole.

The work addresses four issues: a formal semantics for Eiffel, models, the frame problem and the implementation of an automatic verifier.

Formal semantics for Eiffel: With ISO/ECMA Eiffel [56], the Eiffel programming language has received an extensive description of its syntax and semantics. It uses free-form text and informal diagrams. This thesis contains an operational semantics for the Eiffel programming language. Such a formal, rigorous semantics is a prerequisite for any application of formal methods to Eiffel. It is defined in chapter 5.

The definition of the programming language semantics uses a three-step approach to define code, state and execution. It does not operate on the language directly, but instead uses a intermediate form as programming model which can also be used as target for other object-oriented programming languages. The operational semantics creates insights not only into Eiffel, but into universal mechanisms of object-oriented software execution like subtyping, genericity or dynamic binding.

Models: Contracts for dynamic and unbounded classes: When compared to runtime checking or testing, formal verification requires a higher level of consistency and completeness of contracts. This thesis identifies a concept of *abstract object state*, captured by *state models* as a adequate approach to the creation of complete and consistent contracts. It develops a methodology of creating contracts based on models. It describes a lightweight integration of models in form of a model library. The model library is implemented and its interaction with specification, verification, and runtime assertion checking is explored. The concept of models, together with the design and implementation of the model library appears in chapter 6.

Modularity, composability and the frame problem: Bottom-up development requires the composability of software components. But software components cannot be composed to solve new, more complex problem if they interfere with each other. This is known as the *frame problem* [46, 62]. Specifying and proving non-interference of software components is critical for modular proofs of reusable components.

Specification of non-interference poses new challenges that cannot be solved with existing mechanisms available in Design by Contract. Mechanisms in other specification languages [41, 4] have always sacrificed flexibility or information hiding to make assumptions of non-interference possible.

This thesis introduces the concept of *Dynamic Frame Contracts*. With dynamic frame contracts, it is possible to solve the full frame specification problem without cutting back on flexibility or information hiding. Dynamic frame contracts are described and formalized in chapter 7.

Fully automated tools for formal verification: Formal verification needs tools, because human reasoning is too costly and error prone. The *Ballet verifier* is a tool for the functional verification of object-oriented programs that was developed as part of this thesis. It is tightly integrated into the EiffelStudio IDE and implements the weakest-precondition verification. It is based on Boogie, a weakest-precondition proof obligation generator for an intermediate language BoogiePL [19], and Simplify [20], a fully automatic prover for first-order predicate logic. The design and implementation of the tool are described in chapter 8.

As a preparation to these four basic contributions, the thesis conducts an analyzes of current language constructs for Design by Contract in Eiffel and their possibilities and limitations with regards to formal reasoning and specification (chapter 3).

All together, this thesis gives a complete guide to techniques, methodologies and technical tools for the modular verification of functional cor-

rectness for object-oriented code.

The following sections describe the issues and the results of each of the four main contributions by giving illustrations and intuitive examples, highlighting some of the achievements.

2.2 Language semantics

A programming language semantics describes the *execution* of a program on an idealized computer-like machine. The machine has a state, the execution of the program changes the state into new states, thus defines a sequence of states (also called a *trace*).

There are three well-established approaches to describe the semantics of a programming language: operational, denotational and axiomatic [63].

We choose to use a structural operational model [73] describing the programming language and its execution. This contrasts the existing specification mechanisms of Design by Contract. Design by Contract uses pre-, postconditions and invariants, concepts related to Hoare-style reasoning, which would imply the use of an axiomatic semantics.

There are two advantages of defining the semantics of an object-oriented language supporting Design by Contract using an operational description.

First, using a different approach has proved to give good insights into "dark corners" of the language.

Second, an operational semantics is normally more precise than an axiomatic semantics. The axiomatic semantics only defines how properties of the state are maintained or transformed during execution.

An operational semantics has to define the resulting state of a program execution precisely, starting from an arbitrary (within the constraints of a state invariant) given state. It is of course possible to define a complete description of a programming language with an axiomatic semantics, but we claim that it is easier to forget important details when compared to an operational semantics.

The operational semantics defined in this thesis is made out of three distinct parts: the *static model*, the *state model* and the *execution model*.

The *static model* defines code as a mathematical object, formulating data types for classes, features, instructions, expressions and contracts. The static model is not a direct mapping of the language into mathematics. Instead, it describes the static part of a program on a high level of abstraction, removing concepts like identifiers, generic derivations, signature redefinitions or code inheritance (but not subtyping).

An important achievement of the static model is the separation of essential object-oriented constructs from "syntactic sugar": language constructs that are normally analyzed by a compiler and could be removed from the code by unfolding or other semantically neutral refactoring steps.

The *state model* defines the state of the execution of an Eiffel program. We use abstract data types to define this state. We have identified three major components of the state of an Eiffel program: heap, environment and global state.

The heap is the part of the memory where objects are created. We regard the heap as unbounded, ignoring the need for a garbage collector. This is an adequate abstraction as the garbage collector must not interfere with the correct behavior of the program and we ignore possible "out of memory" environment errors when verifying Eiffel programs.

The environment is an abstraction of the stack. Because of the way that we formulate the call rules in the operational semantics, there is no need for an explicit stack-like data structure. Instead, the environment captures all the properties that are local to a routine, like arguments, local variables, **`Current`** or **`Result`**.

The global state is needed only to capture the evaluations of *once* routines that are global for the whole application.

Once we have a precise definitions for static Eiffel code as well as the dynamic state of the machine, the *executional model* defines the effect of Eiffel code execution on the state. This is the core of the operational semantics.

We do not only consider the execution of instructions and the evaluation of expressions, but also the execution of expressions with their potential side-effects. We define precisely what we mean by side-effect free (pure) expressions and we show why we still need to consider changes to the state of these side-effect free expressions.

Taking all the three models together, we are able to prove properties of the Eiffel language, as well as properties of code written in Eiffel. This gives us the foundations for proving properties of Eiffel.

The definition of the operational semantics yields insights the domain of object-oriented programming. We consider the definition of semantics for Eiffel as a universal contribution to the domains of object-oriented programming.

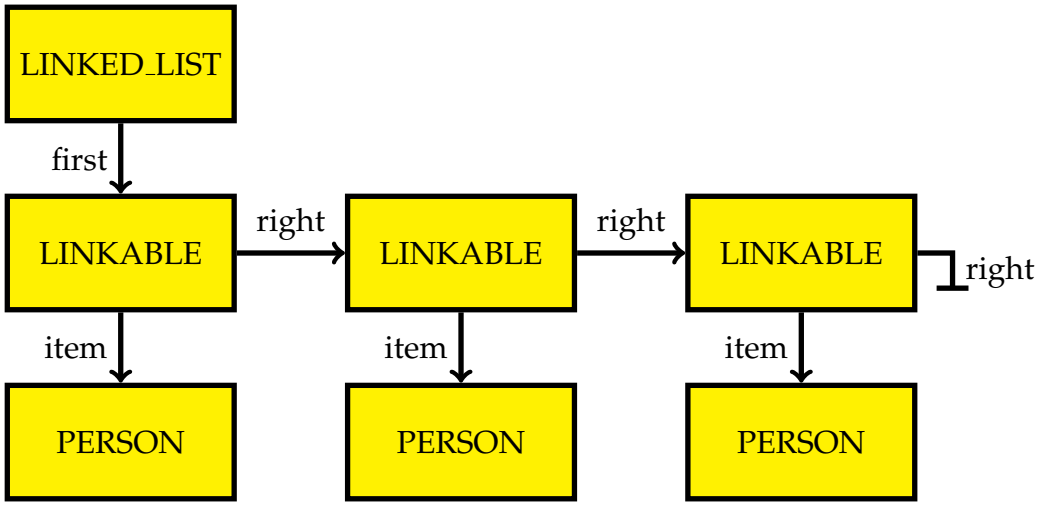

Figure 2.1: Object structure of a linked list.

2.3 Models and model contracts

To model the semantics of object-oriented program execution requires a precise notion of *state of an object*. Specifications express how commands change queries with respect to the object's state: preconditions describe requirements on the object's state for commands or queries, postconditions relate pre- and poststates for commands and describe abstraction functions for queries. Class invariants constrain the object's state.

Surprisingly, there are very few precise definitions of what the object state actually is. As a result of this thesis, we have developed a clear, precise definition of object state.

Looking at the fields of an object to define its state is not sufficient: the state of a linked list, as shown in figure 2.1, is not only defined by the first pointer, but also by the fields of the cells (of type `LINKABLE`) that it uses. Other fields of the object might not contribute to the state. For example fields that are used for caching or internal book-keeping do not contribute to the state of an object.

An alternative to define the object's state would be by defining the state of an object using all fields of all objects that are reachable by the object through the heap reference structure. But, while the cells of the linked list contribute to the state of the list, the fields of the persons contained in the list should not be part of the state.

Furthermore, using fields to define object state is difficult when com-

bined with inheritance and subtyping. For example, it should be possible to talk about the state of a deferred class to understand the semantics of its contracts.

There are cases where subtypes add extensions to the state. These extensions are not visible to parent classes. For example the state of a `PERSON` object might be defined by the name of the person, while the state of a `STUDENT` object extends this state by a student identification number. With the goal of modular reasoning and incremental verification in mind, it is necessary to reason about feature invocations on the basis of the static type of the target defined in the code and not on the basis of the actual dynamic type of the object during execution.

The key to address all these issues is the introduction of *models* into the Eiffel language and the Design by Contract methodology. The model of an object is a description of the object's state in form of a mathematical value.

By mathematical value, we mean a value from some underlying mathematical domain. In our case, we use set theory as the mathematical domain. The model might be for example a set, a relation, a function or a sequence.

The model is not part of the system state, defined by the heap, the stack and global variables. A model is a state abstraction defined through an *abstraction function* on the system state. This abstraction function is also called *model query*.

Assuming that all values of our typed first-order set theory are elements of the mathematical domain MD. Then an abstraction function is a function that takes the full $State$ of the execution and a single $Object$ and computes the corresponding element of MD.

$$mq : State \times Object \rightarrow MD$$

As already mentioned, the abstraction function is also called *model query*. This is because of two reasons:

- The abstraction function is always relative to an object. It is thus equivalent to an object-oriented query.
- To integrate models with the existing Design by Contract mechanisms, the *model queries* are defined as actual queries in the programming language, thus they are features defined in classes.

A single object can have multiple model queries. There might be different queries defined in parent classes that abstract the state in terms of the

parents abstraction. Also, we might want to split up different parts of the object state by providing specialized model queries.

A concrete example of this is the type `LIST[ANY]` in the EiffelBase library. The type is deferred and defines the concept of linear data structure with an integrated cursor. The state of a linked list has two dimensions: the sequence of elements stored in the linked list and a cursor position (mathematically, a sequence is a function from the a finite prefix of natural numbers to objects identities). The cursor position is a linear value that can be easily captured by an integer. The model query for the list class is:

$$\mathsf{model} : (State \times List) \rightarrow (seq(any) \times Integer)$$

To make access easier, we can give names to the different components. For example by calling $model_{seq}$ the sequence part of the model, and $model_{index}$ the cursor position. They will then become the selectors for the components of the resulting cross-product.

$$\forall S, l : model(S, l) = (model_{seq}(S, l), model_{index}(S, l))$$

With models, we can be much more expressive when it come to specifying behavior. For example, without models it is very difficult to describe the precise effect of an extend operation for a lists. The specification is restricted to a fixed set of object equivalences and to the visible state, something that is not sufficient for data structures with an unbounded size and information hiding, like linked lists. This results in heavy underspecifications.

The goal is to describe that the resulting list is equivalent to the old list, with a single element appended. With models, the specification can be done directly on the model abstraction, using operators defined for types in the model. For example, assuming that $++$ defines concatenation of two sequences, the effect of the concatenation is defined as (using **old** to reference the pre-state of the operation):

$$\begin{aligned} model_{seq}(State, Current) =& model_{seq}(\textbf{old}\ State, Current) ++ \\ & model_{seq}(\textbf{old}\ State, other) \\ model_{index}(State, Current) =& model_{index}(\textbf{old}\ State, Current) \end{aligned}$$

This is a full, precise description of the append operation of lists that does not give away details of the implementation: we still can implement the list as a linked list, a doubly linked list or an automatically resizing array. Also, by going over all components of the model, details that have been

forgotten are easier to identify: in the case of a linked list, it is easy to forget to specify what happens to the cursor position for operations that change the list.

The expressiveness of such an approach depends on the expressiveness of the mathematical theory that is used to capture the models. The types and functions have to be expressive enough to model classes implementing a wide range of functionalities.

To prevent reinventing the wheel, the mathematical theory used in the thesis is derived from a mathematical foundation that was already used in an other specification language, B [1]. B has been used successfully in large commercial applications from embedded devices to full security systems, giving confidence that it is suited to express complex specifications in many problem domains.

The B language defines a number of operations on sets and relations that makes it easy to use and very powerful at the same time. Operations like projection, closure or restrictions on the domain or range of a relation create small predicates for very complex properties. The B language is specially good at avoiding quantifiers and set comprehension. For example, the set of all persons contained in a linked list l that has the structure shown in figure 2.1 can be expressed as:

$$item[right * [\{first(l)\}]]$$

(with "first", "right" and "last" being relations between objects, [] is the relational image operator in B, $*$ the reflexive transitive closure).

The mathematical notation defined in this thesis restricts the B language to typed first-order predicate logic and set theory on finite sets. There are different reasons for this choice:

- It is important for the overall approach to keep the specifications executable. Proof technology is still developing and it is obvious that it will not be feasible, or even possible to have every line of code verified fully automatically by pressing a button. Executable specifications can be used for testing, system robustness or bi-simulation.

- All data structures in a system will always be finite. There are only very few cases where it makes sense to give an infinite model to a finite amount of data.

- The Eiffel language includes a very powerful type system. The type system can help to prevent numerous mistakes at early stages of the development. With typed set theory, it is possible create a mapping

between B and Eiffel types. Compared to the original B language this is not a major change: all existing B systems implement their own type checker on top of the B language to support the user, with a similar effect.

The following section describes the integration of models and model-based specifications into the programming language.

2.4 A library of models

Extending Eiffel by a modelling language using first-order predicate logic and typed set theory has to be as non-intrusive as possible. Existing Eiffel developers should not have to learn a full, new specification language, mathematical symbols scare non-mathematicians. Mapping mathematical symbols to the standard ASCII representation of program text can make the code difficult to read. Also, extensions to the programming language always require that all tools dealing with the language are updated and extended to cope with the new syntax. This, again, easily opens a whole myriad of problems with existing Eiffel code.

To solve these problems, many extensions to the specification language offered the new syntax as specially marked comments. Examples are JML[10] or Javadoc[81]. But this produces a gap between the language including the comments and the language without the comments. Regular tools cannot check the comments for type consistency, syntax highlighting will not work on the commented sections. An important decision that was made early in the development of Eiffel, is to let the developers use the same language for specification and implementation.

A better way of extending a programming language non-intrusively is to describe the language extension as a library. Libraries are a natural way to extend expressiveness, every developer is familiar with using libraries.

The ability to define models and the underlying mathematical notation in form of a library was made possible by the concept of **axiomatic classes** into the programming language. Axiomatic classes map concepts that are defined outside of the programming language into types and interfaces of the programming language.

To define the semantics of an axiomatic class we will use neither the implementation nor even on an abstract interface specification, but simply rely on some existing mathematical theory, for example set theory. The axiomatic class simply restates some concept from that theory, such as total functions, in the syntax of the programming language.

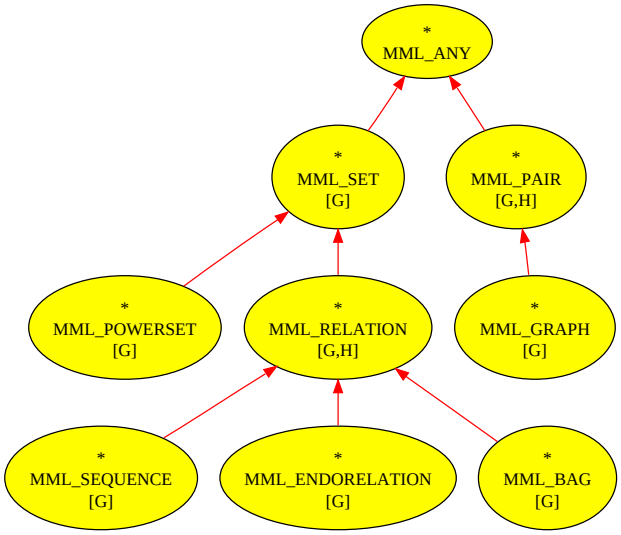

Figure 2.2: Class hierarchy of MML

The notion of axiomatic class is not really new. In the core Eiffel library (ELKS, the Eiffel Library Kernel Standard), the contracts for such classes as INTEGER and BOOLEAN describe only some partial properties of the corresponding abstractions; this implicitly refers the reader to a pre-existing understanding of the concepts, which must come from mathematics. In some cases the reference is explicit; for example the "note" paragraph that introduces class NUMERIC reads:

> *Objects to which numerical operations are applicable. Note: The model is that of a commutative ring.*

It is important to stress, because the semantics is defined by the theory, that contracts of an axiomatic class are irrelevant for the semantic understanding of the class, though they still may be helpful for documentation purposes.

The library of models developed as part of this thesis is called *MML*, the *mathematical model library*. It consists of the interfaces of the nine classes shown in figure 2.2. The class `MML_ANY` is equivalent to the mathematical domain MD, which was suggested in the last section as the range of the abstraction function.

An achievement of this thesis is that we were able to reuse the type system available in Eiffel as the type system of the typed set theory. If an expression given in terms of the MML library type checks, it is also well

```
extend (v: G) is
   -- Ensure that structure includes 'v'.
  require
    extendible: extendible
  deferred
  ensure
    item_inserted: is_inserted (v)
  end
```

Listing 2.1: Original contract of *extend* from class *COLLECTION*

typed in the underlying mathematical theory. This is made possible by the choice of classes and their subtype relationships, many covariant redefinitions on the result types, and the extensive use of generic parameters and feature renaming.

Using MML, it is easy to strengthen the contracts of existing libraries in Eiffel. For example, listing 2.1 shows the original contract of the *extend* feature of the class *COLLECTION* in EiffelBase. It is easy to see that such a contract is inadequate as a full specification for extending a collection. The feature *is_inserted* is a special helper feature that is always comparing the argument with the last object inserted into the collection. It is useless for the interface of a collection, other than to contract *extend*. Also, this feature tells which element was added last to a collection, something that should not matter for the underlying abstract data type.

It is very difficult to add any other contract to the *extend* feature. The reason is that it is on a very high level of abstract: more or less all other data structure classes are subtypes of *COLLECTION*. It is not possible to say anything about the number of elements contained in *COLLECTION* after *extend*. The count is increased for bags and lists, but not always for sets or tables.

Listing 2.2 now shows the contract of the extend feature using models. The model definition is also included, although the model query is actually defined in *CONTAINER*, a parent class of *COLLECTION*.

The model of a *CONTAINER* is defined as a mathematical bag. This means that very early in the definition of the hierarchy a strong statement about the future behavior of the type and its subtypes was made. An instance of *CONTAINER* or any of its subtypes should behave like a bag.

When comparing the precondition of the two versions of *extend* from listings 2.1 and 2.2, the benefits of this strong semantic commitment is clear. Without models, the precondition of *extend* is just *extendable*,

```
model: MML_BAG [G] is
   -- Model of a general container
  deferred
  ensure
    result_not_void: Result /= Void
  end

extend (v: G) is
   -- Ensure that structure includes 'v' iff 'v' can be
      added.
  require
    can_add_element: can_extend (v)
  deferred
  ensure
    model_updated: model |=| old model.extended (v)
  end

can_extend (v: G) : BOOLEAN is
   -- Can 'v' be added?
  deferred
  ensure
    extendable_relation: Result implies extendable
  end
```

Listing 2.2: Contract of *extend* from class *COLLECTION* using models

with is a property talking in general about the possibility of modifying a collection by extension. But the rigorous use of models have forced us to add a precondition to *extend* that needs to consider the argument.

All subtypes of *COLLECTION* have to conform to this type. This is also true for the *SET* type. The model for *SET* is a mathematical sets. But because of the subtyping rules and object-oriented polymorphism, it is also necessary to regard the *SET* as a *CONTAINER*. The implementer of the set needs to make a choice on how to *project* the set model into the bag model of the parent class. Projection means defining a total function from the model of the child to the model of the parent.

As part of the library design, the choice was made to regard the set as a bag directly, but to constrain the values that can be added to it. To do this, *extend* needs a feature that can test, also in the parent class, if the value can be added to the *COLLECTION*. The definition of *extend* has to take this into account.

```
can_extend (v: G) : BOOLEAN is
   -- Can 'an_item' be added?
   -- True only if it is not already in the set.
  do
   Result := extendible and not has(v)
  ensure then
   element_not_in_set: Result implies (not has (v))
  end
```

Listing 2.3: *can_extend* as redefined in *SET* using models

```
can_extend (v: G) : BOOLEAN is
   -- Can 'an_item' be added?
   -- Equivalent to 'extendable' for bags.
  do
   Result := extendable
  ensure then
   always_true: Result = extendable
  end
```

Listing 2.4: *can_extend* as redefined in *BAG* using models

Although the model of the collection is a bag, the collection does not need to behave like a bag by allowing arbitrary addition of elements. This is expressed by the *can_extend* precondition. The subclasses *SET* and *BAG* define themselves as sets and bags by adding a constraint to the *can_extend* predicate as shown in listings 2.3 and 2.4.

The application of models and model-based contracts to existing Eiffel code has led to the discovery of many weaknesses and errors in the design of the corresponding libraries. An extensive study using models was done on EiffelBase [51], a library used extensively within the Eiffel community. The analysis has shown weaknesses and errors, specially in the inheritance hierarchy and the conformance relation, but sometimes even on the level of a single class. A redesign for EiffelBase was suggested that cleans up these problems an promises to be more consistent than the current implementation.

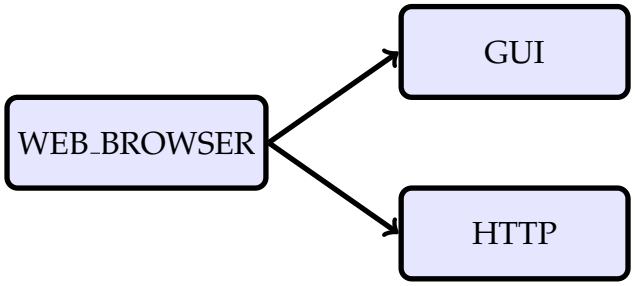

Figure 2.3: Composing components in a client/supplier relationship

2.5 Composability and the frame problem

Development in Eiffel is based on bottom-up development: small software components, completely described by their contracts, can be put together to solve a more complex problem. The term *component* is intentionally not defined here, as we are just trying to motivate an intuitive understanding. We will later use more precise terminology, like *object*, *module* or *class*. Figure 2.3 illustrates such a relation: the component `WEB_BROWSER` combines the two components `GUI` and `HTTP` to solve the problem of a graphical web-browser.

This relationship is called the *client/supplier* relation. The component `WEB_BROWSER` is a client of the components `HTTP` and `GUI`. Note that there is no direct relationship between the two suppliers.

It should be possible to develop the two components `GUI` and `HTTP` independently from each other, and then later join them to form the larger application `WEB_BROWSER`. This is called *bottom-up development*. When combining bottom-up development and information hiding, the only information available to the `WEB_BROWSER` is the information provided through the functional specification of the components, their contracts. So, any implementation of `GUI` and `HTTP` that fulfills the specification of their interfaces can be used, and at any time it is possible to exchange the correct implementation of `GUI` or `HTTP` by another one without breaking the overall application.

Can we always compose the `GUI` and `HTTP` components to form a bigger component? The answer is no, as illustrated by figure 2.4. The problem is that the implementation of both components make use of a fourth component `NETWORK`. It is not unusual for the `GUI` to use a network component. For example in Unix, graphical user interfaces communicate with the screen using the network stack.

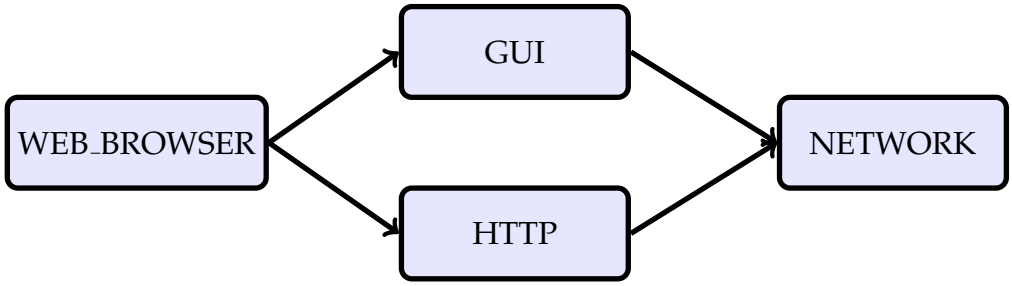

Figure 2.4: Interference of two components through their implementation.

Now, imagine that network is handled through a device that can only be opened once. Both components will try to allocated the network resource, which breaks the application.

This illustrates the frame problem in software development. Neither the `GUI` nor the `HTTP` component can state that it uses the `NETWORK` component; this would reveal details of their implementation, violating the rules of information hiding.

Also, it is not possible for one component to declare that it cannot be used together with the other. There are two reasons: first, each component should not know about the implementation of the other component, as already stated above. Second, in an open-world development, the number of potential components is infinite. It is impossible to enumerate all components that interfere.

The solution is to devise is a mechanism that allows `GUI` and `HTTP` to describe that they might use some shared resource, but on a level that is sufficiently abstract to maintain information hiding. `WEB_BROWSER` then takes these descriptions to detect possible problems of interference. These abstract descriptions are called *frame specifications*.

2.6 The frame problem in Eiffel

In Eiffel, components become objects (instances of classes), specifications become contracts. In an attempt to verify Eiffel code, the frame problem arises already in the most primitive examples. In the following, we illustrate the frame problem using only boxed integers and a simple copy operation between them. Boxed integers are reference objects whose only purpose it is to store an integer value. This problem is already sufficiently complex that it is indeed not possible to specify and verify it using existing mechanisms provided by the Eiffel language.

```
class INT_STORE

feature -- Access
  item: INTEGER
    -- Value stored

feature -- Change
  set_item (a_value: INTEGER) is
      -- Set 'item' to 'a_value'.
    ensure
      item_set: item = a_value
end
```

Listing 2.5: Interface of a boxed integer

```
  copy (source: INT_STORE; target: INT_STORE) is
      -- Copy 'item' of 'source' to 'target', making both
         values the same.
    require
      source_not_void: source /= Void
      target_not_void: target /= Void
    do
      target.set_item (source.item)
    ensure
      value_copied: source.item = target.item
    end
```

Listing 2.6: Copy operation of a boxed integer

The interface of the boxed integer is captured in listing 2.5. We intentionally restrict ourselves to the interface: it is the only information the client knows and we want to maintain information hiding.

The copy operation is shown in listing 2.6. The goal of the copy operation is to make sure that both values are equivalent after the copy operation.

Most of the time, the implementation of `copy` is correct. But it is not possible to verify that `copy` is correct. If it would be possible, then the underlying theory has to be unsound, as there are ways to correctly (with respect to its contract) implement `INT_STORE` that breaks the implementation of copy.

One possible implementation would be to use a object that does not

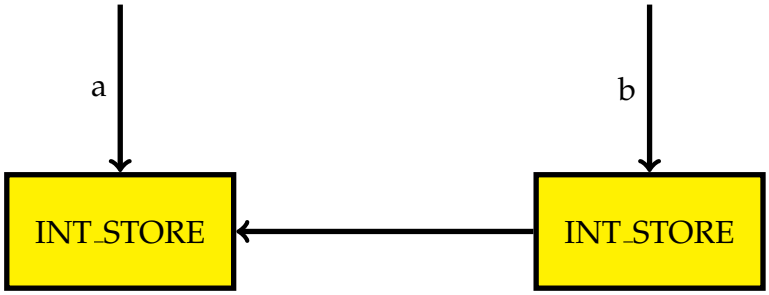

Figure 2.5: Proxy object structure

store its plain value, but instead as a difference to another object. This is illustrated in the object structure of figure 2.5. The object b has a reference to object a. Instead of storing the value directly, it might store its value as the difference to the value stored in a (see the implemenation of `REL_INT_STORE` in listing 2.7).

If we set a to 3 and then b to 5, then the value stored directly in b is just 2, though `item` would correctly return 5. If we later change a to 7, then the value of b is changed implicitly to 9. If we now try to copy the value of b to a, we get a postcondition violation: the value 9 will be stored in a, but at the same time, the value of b is changed to 11. The two values read from `a.item` and `b.item` are not the same.

2.7 Dynamic frame contracts

There are numerous specification mechanisms for frames. Explicit *modify clauses* enumerate all the attributes effected by an operation. This is a violation of information hiding, as it reveals details of the implementation [62]. Also, it does not work well together with inheritance, where the decision of the precise implementation of an interface is done in a subclass.

Data groups [44] are an attempt to remedy the problems of information hiding. They allow the abstract specification of attributes on the level of the interface. Later, a number of concrete variables are associated with the data groups in the hidden part of the class or its subclasses. Still, the state is required to be implemented as attributes of the object. The implementation of a proxy data structure as shown above remains a problem.

Dynamic frames are a novel and seminal approach developed by Ioannis Kassios [36, 37]. Prior approaches have all tried to make frames static, the attributes captured in the frames are fixed by the specification. Dy-

namic frames regards frame specifications as part of the runtime state of the program. Frames are sets of memory locations, captured themselves in variables of the program state. Kassios' work is based on an artificial, simple language. Inheritance as well as implementation of interfaces was modeled through the use of abstract refinement.

Dynamic Frame Contracts as introduced and defined in this thesis take over the essential ideas of Kassios' dynamic frames, but integrate them into a real-world programming language and open world reasoning. We do this by combining dynamic frames with the Design by Contract methodology.

The idea behind dynamic frame contracts is as follows: resources of Eiffel program are captured by an opaque data type called `FRAME`. Think of frames as being sets of objects with their corresponding fields.

As an extension to the Design by Contract mechanism, every feature can have **`use`** and **`modify`** clauses. The **`modify`** clause is an expression of type `FRAME` yielding the frame of resources that might be modified by the execution of the routine. The **`use`** frame an expression of type `FRAME` yielding the set of resources that is read during the execution of the feature, thus may influence its behavior and its return value. Both clauses are evaluated in the pre-state of a routine execution.

The idea to regard frame specifications as executable code that needs to be evaluated at run-time is the fundamentally new idea behind dynamic frames contracts. They fit amazingly well with the Design by Contract methodology as prominent in Eiffel, and the use of regular expressions of the programming language as contractual specifications.

The benefit of frame specifications is gained by the **dynamic frame rule** that is defined later. Informally it says that if the frame of a query q and command c are disjoint, then the result value of q will not change by the execution of c.

In practice, dynamic frame contracts use opaque queries, called frame queries, that return specific sets of resources. To show the non-interference of two software components, it is necessary to maintain the disjointness of the corresponding frame queries.

To illustrate the benefits of the approach, dynamic frame contracts are added to the `INT_STORE` example. An opaque query `representation` is introduced into the interface of the integer store that yields the representation frame of the boxed integer, meaning all resources that are used to store the integer value. The `item` and `set_item` feature are extended by **`use`** and **`modify`** frames, as shown in listing 2.8.

The frame rule states that a command does not influence a query if their frames do not overlap. This requirement is added as a precondition

of the `copy` feature, enforcing it to be fulfilled by the client of the copy operation. The extended contract to `copy` is shown in listing 2.9. It is now possible to verify the functional correctness of `copy`.

To summarize, dynamic frame contracts are an extremely powerful, although sometimes verbose specification mechanism that solves the problems of framing in object-oriented programs. The underlying proof theory needs to know about sets and very few set operations. In contrast to existing technology suggested for framing like ownership structures that impose a tree structure over the heap, dynamic frame contracts seem to be better suited to specify problem with possible alias situations like proxies, singletons or publish/subscribe frameworks. Dynamic frame contracts integrate well with Eiffel, supporting the Design by Contracts mechanisms and other parts of the methodology like uniform access and command-query separation.

2.8 Ballet verifier

Ballet is a verification tool for Eiffel implemented as part of this thesis, based on the results described above. It uses the Boogie tool chain developed by Microsoft Research for the automatic verification of Eiffel classes[4].

During the VSTTE conference 2005, Rustan Leino gave a demo of the Boogie verifier for Spec#. In the context of this talk he invited other research groups to consider the intermediate language of the verification process, called BoogiePL [19], as their target language for verification.

BoogiePL is an abstract language used by Boogie between Spec# and the proof obligations that are handed off to the theorem prover. It provides its users with an uniform interface to first-order predicate logic and with a complete framework for weakest precondition reasoning. At the same time, it is sufficiently abstract not to impose a specific understanding of object-orientation, or to even enforce the object-oriented paradigm at all.

The overall structure of the Ballet tool chain is shown in figure 8.2. Ballet is integrated into the Eiffel IDE and operates on a specific intermediate representation of the Eiffel compiler, called the *Eiffel Byte Code*. The Eiffel Byte Code is a late intermediate representation that is used directly for all the other Eiffel IDE code generation facilities like C code generation, .NET code generation or the melting ice interpreter.

From Eiffel Byte Code, Ballet generates BoogiePL programs. BoogiePL code encodes feature as procedures assumptions and proof obligations for the verification by `assume` and *assert* instructions in these procedures. Thus BoogiePL is not meant to be executed as it includes non-constructive

descriptions and non-determinism.

A background theory in form of function declarations and axioms needs to be added to BoogiePL. For Ballet, the background theory defines such concepts as the global memory structure and the basic set theory that is required for the verification of models and frame.

The Boogie tool will then create proof obligations in first-order predicate logic from these instructions by doing weakest precondition reasoning over the procedures. These proof obligations are then handed of to a fully automatic theorem prover. Currently, two different provers are supported: Simplify, a classic prover for first-order predicate logic, and Z3, an in-house development by Microsoft Research to overcome some of the limitations of Simplify.

Should any of the provers find violation of any of its proof obligations, an error message is propagated back through the tool chain, pointing the developer to the possible Eiffel instruction or assertion that causes the violation.

```
class REL_INT_STORE inherit INT_STORE

create
  make

feature -- Initialization

  make (o: INT_STORE) is
     -- Initialize 'other'
    require
      not_void: o /= Void
    do
      other := o
    end

feature -- Access

  item: INTEGER is
    -- Value stored
    do
      Result := intern + other.item
    end

feature -- Change

  set_item (a_value: INTEGER) is
     -- Set 'item' to 'a_value'.
    do
      inter := a_value -- other.item
    end

feature{NONE} -- Implementation

  intern: INTEGER
  other: INT_STORE

end
```

Listing 2.7: Implementation of *INT_STORE* that stores its value relative of another *INT_STORE*

```
class INT_STORE

feature -- Access
  item: INTEGER
     -- Value stored
    use
      representation

feature -- Change
  set_item (a_value: INTEGER) is
     -- Set 'item' to 'a_value'.
    modify
      representation
    ensure
      item_set: item = a_value

feature -- Frame queries
  representation: FRAME
    ensure
      not_void: Result /= Void
end
```

Listing 2.8: Interface of a boxed integer with frames

```
  copy (source: INT_STORE; target: INT_STORE) is
     - Copy 'item' of 'source' to 'target', making both
        values the same.
    require
      source_not_void: source /= Void
      target_not_void: target /= Void
      representation_disjoint:
   source.representation.is_disjoint(target.representation)
    do
      target.set_item (source.item)
    ensure
      value_copied: source.item = target.item
    end
```

Listing 2.9: Copy operation of a boxed integer with frames

CHAPTER 3

SPECIFICATIONS IN EIFFEL

Eiffel is one of the few programming languages that combines specification and efficient implementation into a single language. Meyer motivates the integration of specification constructs into the actual programming language by emphasizing the importance of *seamless development*:

> *The object-oriented approach is ambitious: it encompasses the entire software life-cycle. When examining object-oriented solutions, you should check that the method and language, as well as the supporting tools, apply to analysis and design as well as implementation and maintenance. The language, in particular, should be a vehicle for thought which will help you through all stages of your work.* (Meyer [52, p. 22])

Excluding any other alternative approach to specification is further stressed by the "single model principle", as described by Paige and Ostroff:

> *Independently generated multiple models of a system cause more problems than they solve in developing software. It is claimed that multiple models are useful because they allow developers to work independently on different parts of a software system, and thereafter their individual work can be integrated. We have already remarked on problems with this approach, particularly with consistency: checking that one independently created model does not contradict a second independently created model is a very complicated problem, even for small systems. The problem is avoided by obeying the single model principle.* (Paige and Ostroff, [69])

The *single model principle* (sometimes also called the *single document principle* or *single product principle*) is one of the key properties of the Eiffel language and methodology. The single model is the Eiffel language source

text. Different *views* [69] can be provided, but they do not carry extra information. The use of a single language enforces the consistency of the model.

As a consequence, specifications are limited to the expressiveness of Eiffel. Every property of the specification needs to have a representation in the programming language.

There are numerous specification problems that are not easy to express in current Eiffel. For example, Eiffel does not offer specifications for real-time properties or specifications on information flow, as needed by security concerns. Specifications available in Eiffel are *functional specifications*, specifying relations between pre- and post-states, and invariants on the state.

In this chapter, we analyze the existing mechanisms available in Eiffel. This includes the type system, contracts and assertions. We look at their expressiveness and their theoretical foundations. The focus of the analysis is formal reasoning, that is the ability to describe properties of executions of the code and to prove these properties.

3.1 Type safety

Eiffel is centered around the concept of **classes** and **features**. Classes are *abstract data types* (ADTs) [45]. The functions of the ADT become features (methods, attributes) of the class. Furthermore, a class may or may not provide an implementation for the features to be executed on a machine. If a full implementation is provided for every feature, a class is called *effective*, otherwise it is called *deferred*.

Based on the idea of classes as types, Eiffel introduces a **type system**. All types define a partial ordering $\sqsubseteq$ with the type `ANY` as the top and the type `NONE` as bottom element. One type A is said to conform to another type B if $A \sqsubseteq B$ holds.

Every *entity* (attribute, variable, argument, return value) is declared to be of a specific type: that is a class with the generic parameters bound to other types. The type system defines *type safety*: any value attached to that entity at run-time is an instance of the corresponding type or any of its subtypes.

A *static type checker* is used at compile time to verify that the property of type safety is maintained for any execution of the program. This is called *static type safety*, in contrast to a *dynamic type checker*, a system for monitoring type safety at run-time and throwing exceptions should a type violation occur.

The checker verifies static type safety modularly, by only considering the class text of some type `T` and the class text of classes inherited (directly or indirectly) by `T`, to certify that the implementation of `T` is indeed type safe.

A modular static type checker for current Eiffel cannot be sound. This is caused by two language constructs in Eiffel that make it impossible to certify type safety in a modular way: covariant redefinitions and the use of generic parameters to type arguments.

Meyer [52, pp. 633] suggests the use of non-modular, global analysis to ensure static type safety. This endeavor seems to be very difficult: Keller [38] tries to implement a sound global analysis using several approaches, even by extending the Eiffel language, but fails to provide a working solution.

The GOBO tool chain for Eiffel [5] offers a tool called `gelint` implementing a global analysis algorithm. While the tools indeed manages to identify type violations not found by the modular type checker of the compiler, the tool is neither sound nor complete.

There is another problem, demonstrated by `gelint`: even if a global analysis is able to detect an error, it rarely gives a meaningful and understandable error message. A problem found through global analysis implies that the error is caused by the interaction of different modules in the system and not by a single module alone. So, both modules can be considered type safe by themselves, and only joining them to form a system creates the error.

3.2 Contracts

The Design by Contract [49] methodology of Eiffel allows the developer to integrate specifications, called **contracts**, into the code of Eiffel programs. It also defines requirements (related to these contracts) on executing features of the class. If it can be shown that any execution will fulfill these requirements, the class text is called **correct** with respect to its contract. This property is also called **class consistency** [56, sec. 8.9.16].

- Every class text can be accompanied by a **class invariant**. The class invariant is a set of boolean expressions. The evaluation of all these expressions has to yield **`true`** at the beginning and the end of the execution of any feature of that class.
- Every feature can be accompanied by a set of boolean expressions called **pre-** and **postconditions**. Any execution of the feature started

in a state where all evaluations of preconditions yield **true**, will terminate and yield a state where all evaluations of the postconditions yield **true**. Postconditions have the ability to evaluate certain subexpressions already in the pre-state, using the keyword **old**. Postcondition form a relation between the pre- and post-state.

- For **qualified feature calls** of the form `t.f(a)`, the value of `t` must not be equivalent to **Void** and — should `f` have preconditions — the evaluation of these preconditions on target `t` with arguments `a` yields **true**.

- **check instructions** containing boolean expressions can be added to the code at the instruction level. The evaluation of these boolean expressions must yield **true** whenever the check instruction is encountered at runtime.

- For the specification of loops, Eiffel introduces **loop invariants** and **loop variants**. Loop invariants are boolean expressions. Loop invariants must evaluate to **true** at the start of the loop and at the end of each loop iteration. Loop variants are integer expressions. They must evaluate to a positive value after each iteration and that this value is smaller than the value of the previous iteration.

As seen by these definitions, Eiffel uses regular language expressions (with the exception of the keyword **old**) to express specifications. This approach has been adopted by most other languages featuring Design by Contract ([4], [41]). Here are some of its advantages:

- There is no "conceptual break" between the specification and the implementation language. Developers do not have to switch back and forth between two notations.

- Important properties are often abstracted into boolean queries. These queries can be used in contracts, removing the need to duplicate the effort in two language.

- Expressions are executable. This makes it trivial to monitor correct behavior of the program at run-time.

At the same time, here are also problems introduced by using expressions for contracts. In the next sections, we give an overview about these problems and suggested solutions.

```
class A feature

  x: INTEGER

  f1 is
    do
      x := 1
    ensure
      post1: f2
      post2: x = 2
    end

  f2: BOOLEAN is
    do
      x := 2
      Result := True
    end

end
```

Listing 3.1: Side-effects in contracts

3.3 Side-effects of contracts

The use of regular programming language expressions for specifications raises the question of potential side-effects in these queries. If the evaluation of the queries has an effect on the state of the system, then we might be facing a "Heisenberg"-like problem in which the instrument of our measurement (expressions used in contracts) influences the subject of the measurement (the implementation).

We might get different behavior based on whether we evaluate the contracts or not. Lets consider the Eiffel code of listing 3.1, demonstrating this effect.

This extract should not be correct: the post-condition `post2` of `f1` is not satisfied by the execution of the body of the routine. Run-time monitoring will not detect this flaw, as the evaluation of `post1`, which happens before `post2`, will make `post2` yield **`true`**.

Fortunately, we are not in physics. The problem does matter for the monitoring of contracts at run-time as described in section 3.2. But for formal reasoning, we can exclude the evaluation of the expressions from the execution of the program. The only point that matters for the evaluation

```
f(x,y:INTEGER)
  require
    pre: x / y >= 1
```

Listing 3.2: Side-effects in contracts

of the contracts is the return value. Instead of really executing the code, it is possible to say "what would be the return value of the expression if we would execute the contract".

In listing 3.1, we would consider **True** to be the return value of `f2`, but we ignore the update of `x := 2` in `f2`. Instead, the can assume that `post2` is evaluated independely of `post1`. Weakest precondition reasoning reveals that the code is indeed incorrect.

3.4 Partial functions in contracts

More delicate than side-effects in contracts is the possibility for the contract to use partial functions. Partial functions are feature invocations that carry a precondition on the arguments, or the use of an entity as target of the call that might have the value **Void**.

Listing 3.2 illustrates this. The value of the precondition `pre` for the feature `f` is not defined if `y` is equal to 0. There are two ways to interpret such a contract:

- Every precondition must be total: A *division by zero* contract violation is thus the fault of the supplier as the supplier did not provide a total function as contract.
- A precondition may be partial: It is the obligation of the client to analyze the contract and detect the *implicit precondition* `y /= 0`.

ISO/ECMA [56, sec. 8.26.10] defines the second interpretation to be the one used in the Eiffel programming language. The caller of a feature does not only need to satisfy the preconditions, but also has to be sure that all implicit preconditions of the features used in the contract are satisfied.

3.5 Pure object-orientation

The ISO/ECMA standard defines only few elements that constitute an expression. This is done to minimize the complexity of the language and

prominent also in other "pure" object-oriented languages such as Smalltalk [31] or Ruby [78].

- **Feature invocation** is used to call queries on the current or other objects. Feature invocations are not total, as they require the target object to be different from **`Void`**. Any operator others than = and /= (which is equivalent to **`not`** (E1 = E2)) requires that the first argument to the operator is not **`Void`** and is mapped to a feature invocation of the corresponding **`prefix`** or **`infix`** feature. Also, a feature invocation has to respect the precondition of the feature called.
- It is possible to create new objects using a **creation expression** or **agent expressions**. Some classes like *`STRING`*, *`TUPLE`*, *`INTEGER`* or *`BOOLEAN`* offer object creation through *manifest expressions*.
- **Object equality** of reference types using the = and /= operator. These are the only operators not defined in the context of a class are thus not features. The semantics of the = is different for *reference* and *expanded* types: for reference types, the reference are compared. On expanded types, a feature called `is_equal` is called.
- Expressions used in features can reference **arguments** or declared **local variables**.
- **`Void`** is a constant reference, referencing the "non-object".

This enumeration is not complete (see the syntax rules in the ISO/ECMA standard [56, sec. 8.28.1]), but includes the most important constructs in Eiffel expressions. The mathematics framework provided by the language standard only introduces equational reasoning.

The reduction to very few constructs in the assertion language has advantages and disadvantages for formal verification of software. The main advantage is that the semantic description of the language remains small. The disadvantage is the need to define even the most simple language constructs in terms of features and contracts. We approach this problem through the use of *axiomatic classes* as introduced in the next section.

3.6 Axiomatic classes

The ISO/ECMA Eiffel standard defines a set of basic types [56, sec. 8.30]. Basic types are boolean, natural and integer numbers, characters, float point values, pointers and strings.

The specification of these classes is regarded as outside of the scope of the language standard. Instead, these classes are defined in ELKS [65], the Eiffel Language Kernel Standard. ELKS is an Eiffel interface description making extensive use of free form comments. The contracts provided are strong under-specifications. The FreeELKS project [6] provides further refinements of this specification and a partial implementation. The final implementation is provided using *built-in* mechanisms in Eiffel compilers and code generation technology.

For the verification of Eiffel classes, this means that there is no formal description of the Eiffel basic types. Reasoning about feature invocations on these classes cannot be done. As most of these basic types (specially `BOOLEAN` and `INTEGER`) are expanded, it is not even possible to bootstrap their semantics using equational reasoning of ADT specifications.

A solution to this problem is to single out a certain set of classes as *axiomatic classes*.

Definition 3.1: Axiomatic class

A class is "axiomatic" if its interface describes a concept defined by a mathematical theory rather than by its contracts.

Using this approach, we can relate the class `BOOLEAN` to a boolean algebra. Naturals and integers are introduced, for example, using the Peano axioms.

Axiomatic classes are not only helpful to introduce basic types into the language. They also make it possible to declare more complex data types using standard mathematics. They are an important vehicle for the definition of model classes in chapter 6.

3.7 Language semantics

In section 3.2 we have defined the correctness of a given class text. We have related the correctness of a class text to values computed by the evaluation of expressions at certain points of the execution.

To prove the correctness of an Eiffel class text, we have to understand what it means to execute Eiffel code on a von-Neumann style machine. Only then are we able to prove that the evaluation of a expression (given as a program text) in a certain state will always yield a certain value. We have to understand the given program text as a mathematical object.

The theory that relates program text and mathematics is called a *formal language semantics*. Eiffel does not have a formal language semantic for the

full language. Some small subsets have been formalized ([70], [64], [54], [55]). The ISO/ECMA Eiffel standard itself is not a formal language semantic, as it uses free language text which is insufficient for mathematical reasoning.

Without the availability of a sound formal language semantic for Eiffel, there cannot be formal verification of Eiffel programs. The definition of a formal semantic for Eiffel is thus the first step and defined in chapter 5.

3.8 Contract-based reasoning

Eiffel regards classes as software modules ("*Classes should be the only modules.*", [52, p. 24]) and promotes information hiding between them:

> *The designer of every module must select a subset of the module's properties as the official information about the module, to be made available to authors of client modules. Application of this rule assumes that every module is known to the rest of the world* [...] *through some official description, or* ***public*** *properties.* [...] *The public properties of a module are also known as the* ***interface*** *of the module.* (Meyer [52, p. 51])

Authors of client modules are restricted in their reasoning to the information that is available in the interface specification of the supplier.

This has major impact on reasoning about the correctness of Eiffel code: until now, we had assumed that we could show correctness by evaluating the implementation of the expression using some formal language semantic. But the implementation is not part of the interface. We cannot reason about code that is not available to us.

Instead, we have to rely on the contracts of the features that are used in the expression. Furthermore, we have to rely on the correctness of the hidden implementation of the features with respect to these contracts. Reasoning about a given class text by only using the interface specifications of its suppliers is called *modular reasoning*.

Definition 3.2: Modular verification

A proof, that a property P of a class C holds, is **modular**, if it only relies on the properties of its own source code, its ancestors, and the interface of its direct and indirect suppliers is taken into account.

We cannot restrict ourselves to just the direct suppliers of C: the contracts of the direct suppliers of C are themselves written in terms of feature calls on other suppliers. We can only stop when we encounter contracts written entirely using calls on axiomatic classes or reference equality.

Considering listing 3.3 containing the class `A` and the contracts of classes `B` and `C`. To prove the correctness of `prove_me`, we have to take the contract of `B` into account. To reason about the contract of `value` in `B`, we have again to look up the contract of `C`.

Unfolding can stop there, as the specification of the feature call `>` is defined in the class `INTEGER`. `INTEGER` is an axiomatic class, modelling integer values. The specification is given through a mathematical theory and not though the feature contracts.

Iteratively looking up contracts of used features, replacing the arguments, is called *unfolding contracts*. The result of unfolding contracts is a set of boolean expressions.

We unfold the contracts until sufficient knowledge to prove correctness is accumulated, or until only reference equalities or feature invocations of axiomatic classes are left. Proving `prove_me` by unfolding the postcondition in listing 3.3 looks as follows:

$$\texttt{Result} := \texttt{x.value}\ \textbf{ensure}\ \texttt{Result} > 10$$

is reduced by weakest-precondition reasoning to

$$\texttt{x.value} > 10$$

adding the postcondition of `value` of class `B`

$$(\texttt{x.value} = \texttt{x.y.value}) \Rightarrow \texttt{x.value} > 10$$

adding add the postcondition of `value` of class `C`

$$(\texttt{x.value} = \texttt{x.y.value} \wedge \texttt{x.y.value} > 10) \Rightarrow \texttt{x.value} > 10$$

which can be simplified to

$$\texttt{true}$$

Unfolding illustrates a simple reasoning mechanism for contracts. We can model unfolding in mathematics through functions, similar to the lambda calculus.

```
class A feature

  x: B

  prove_me: INTEGER is
    do
      Result := x.value
    ensure
      Result > 10
    end
end

class B feature

  y: C

  value: INTEGER
    ensure
      Result = y.value
end

class C feature

  value: INTEGER
    ensure
      Result > 10

end
```

Listing 3.3: Contract-based reasoning

3.9 Recursive contracts

The contract of a feature `f` is called *directly non-recursive* if `f` is never encountered while unfolding the contract of `f`. The contract of a feature `f` is called *non-recursive* if all features encountered during unfolding of `f` are directly non-recursive. Otherwise, the contract of feature `f` is *recursive*.

Assuming that the program text of all features is finite, it is straight forward to prove that unfolding will always terminate: assuming that the program text contains n features and the longest contract of a feature contains m feature invocations, then m^n is an upper bound the number of unfolding operations (the unfolding operations form a tree with the max-

```
is_equal (other: IMMUTABLE_LIST[G]): BOOLEAN
   -- Does 'Current' contain the same items as 'other',
   -- in the same order?
 require
   other_exists: other /= Void
 ensure
   same_contents:
     Result = (is_empty and other.is_empty) or else
            ((not is_empty and (not other.is_empty))
             and then
             (head = other.head and
              tail.is_equal (other.tail)))
```

Listing 3.4: Recursive contract for `is_equal` in `IMMUTABLE_LIST[G]` as described in [58].

imum depth n and the maximum number of children per node of m).

In regular Eiffel, only the basic types are defined as axiomatic classes. Most basic types are fixed-sized data structures: integer, boolean, float or pointer have a predefined memory footprint. The only basic type without a fixed-size is string.

If we ignore the possibility to encode arbitrary data into strings, contract-based reasoning about non-recursive contracts can only express properties on a bounded part of the memory. Non-recursive contracts are thus inappropriate for the description of unbounded data structures like lists or stacks.

To solve this problem, Mitchell and McKim [58] advertise the use of recursive contracts. They contract the *is_equal* feature of a class *IMMUTABLE_LIST[G]* using the contract shown in listing 3.4.

The use of recursive contracts as a specification makes it possible to contract unbounded data structures using bounded contracts. But there are a two problems introduced by this technique:

- Reasoning about recursive contracts might require fixpoint reasoning, as the number of unfold operations can be defined by values only known at run-time. The fixpoint computation might require the specification of an invariant, but Eiffel does not provide a language construct for this.

- Unfolding the recursive contract might never terminate. If the contract does not terminate, there is no clear understanding of the se-

mantics with respect to correctness. Proof of termination might require the specification of a variant.

Because of these reasons, recursive contracts seem to make it more difficult to reason about Eiffel programs. Recursive contracts are difficult to read and understand.

To avoid the need for recursive contracts for unbounded data structures, we suggest the definition of axiomatic classes modelling unbounded data and to use these classes to contract unbounded data structures. The definition of such classes is described in chapter 6.

3.10 Agent-based contracts

Another technique to describe properties of unbounded data structures is the use the *agents*. Agents are feature invocations encapsulated into objects, similar to mathematical closures.

With agents, it is possible to express mathematical quantifiers like $\exists$ or $\forall$ over finite data structures by the definition of features `there_exists` or `for_all` that take an agent as argument and evaluate the agent on all elements of the data structure.

This approach creates problems. First, agents cannot be equipped with contracts. It is thus impossible apply modular reasoning to code using agents. Instead, we have to track the agent through the code from its creation up to its call.

Second, there are no features using agents in the basic types of Eiffel. Thus, the semantics of features using agents are defined again through the contract of the feature.

3.11 Specification features

The last possibility to specify properties of unbounded data structures in Eiffel is the use of *specification features*. Specification features are features only defined to test an invariant property of the data structure. They can be used to simulate a for-all quantification, using the argument as the quantified variable. Listing 3.5 shows the definition of a feature that certifies that a list of integers is sorted.

A specification feature reasons on the pre-state of the computation, describing its assertions within an **`old`** clause. Thus, it is not possible for the feature to establish the postcondition by its own implementation. Instead,

```
sort_test_feature (index: INTEGER)
   -- Test feature that the integer list is sorted.
 require
   valid_index: index > 1 and index <= count
 ensure
   is_sorted: old (i_th (index-1) < i_th(index))
```

Listing 3.5: Specification feature for a sorted list of integers

the postcondition becomes an invariant. But in contrast to an invariant, the test feature can work with arguments that have unknown values.

The drawback of this approach is that specification features pollute the interface of a class with useless definitions. Though they make it possible to provide all-quantified invariants, they seem to be a very unorthodox way of providing specifications for classes. Until now, we have never seen the use of specification predicates in contracts.

CHAPTER 4

MODULAR REASONING AND VERIFICATION

The need for modular verification is not obvious. Instead we might argue that proving properties about programs is such a difficult task that we should first try to go for the "easier road" by proving properties in a non-modular way.

In this chapter, we define what we mean by modular verification. Also, we motivate why reasoning Eiffel needs to be modular. We argue that a non-modular approach to verification violates the foundations of object-oriented development and is thus not acceptable.

4.1 Modular boundary

Defining modular verification means defining a precise *modular boundary*: which information is taken into account to prove that a class is correct. If this information changes, the proof may be invalidated. Any information that is not taken into account may change arbitrarily without violating the correctness proof of the program.

The goal is to minimize the set of information that is inside this boundary and thus to make the proofs as robust as possible to changes in the system.

Assume that we want to prove the correctness of an implementation of some class C. Obviously, the source text of C is required for the proof. Other information might be about:

- **direct parents**, class that C directly inherits from.

- **indirect parents**, the closure of the parent relationship, thus all ancestors of `C`.
- **direct suppliers**, classes that are used by `C`.
- **indirect suppliers**, the closure of the supplier relationship.
- **other classes** that are used in the system.

Orthognal to the source to the information is the question of information hiding. The proof might either rely on the full class text, or restrict itself to the information provided by the interface.

4.2 Modularity in inheritance

The interface specification of parent classes is needed for reasoning about the correctness of a class: the implementation of a feature may call inherited features using "unqualified calls" [56, sec. 8.23.4], the semantics of these feature calls is carried in the feature contracts. Also, the inheritance relation demands that redefined features must satisfy the inherited contracts, and is only allowed to weaken the precondition and to strengthen the postcondition [52, p. 578].

But relying only on the interface specification is not sufficient. Reasoning about the child requires information not provided by the interface.

1. Because of *uniform access* [52, p. 57], we do not know through the interface specification if a given argument-less query is a function or an attribute. But the assignment operator `:=` is only allowed for attributes. Thus, it is necessary to know whether a given argument-less query is an attribute or a function.
2. Unqualified feature calls do not have to respect export restrictions. Thus, it is possible to call features that are exported to `NONE`. These features are not part of the interface specification.
3. Features do not have to preserve the class invariant when subject to an unqualified call. The implementation of a feature is undefined when called without a valid class invariant.
4. Feature redefinitions have to fulfill all inherited contracts. This includes postconditions that are using private features, and are thus not part of the interface specification.

5. Last but not least, inheritance and partial redefinition of feature implementations can expose the *fragile base class problem* as defined by Mikhajlov and Sekerinski [57]. The solution to this problem is to reprove all inherited features (proving the *flattened text* of a class). This requires full knowledge of the feature implementation.

For the verification of Eiffel code, it is impossible to form a modular boundary on the basis of the inheritance relation. Instead, we will assume that we always have full knowledge of the text of all direct and indirect parent classes. Using these classes, we are able to create the flattened class text.

4.3 Modularity of suppliers

To reason about the effect of feature invocations, we have to include the interface specification of all classes defining the static types of the entities we are using in our (flattened) class text. The legality and effect of a feature invocation on the entity can then be understood by reasoning on the type, and the pre- and postconditions carried by the interface specification.

It is possible to call an inherited feature on the entity. Consequently, the interface specification of the supplier has to be the "fattened interface specification", that is the interface specification of all parent classes.

The interface specifications of child classes are not needed, although at runtime the actual object stored in the entity may be of a subtype defined by a child class. The reason lies in the rules for inheritance: the precondition of a parent class implies the precondition of the child class, and the postcondition of the child class implies the postcondition of the parent class. We can reason on the basis of static type of an entity and on the basis of the dynamic type of the attached object. Doing this, we loose information, but it reduces our modular boundary and very often the exact dynamic type is not known anyway.

As explained in section 3.8, to reason only on the basis of contract, we have to unfold these contracts. Thus, the contract of features used in contracts needs to be included within out modular boundary. The modular boundary needs to include the interface specification of all classes that take part in this unfolding.

4.4 The imperative of modular verification

Verification of Eiffel must be modular. Otherwise, we sacrifice important advantages of the object-oriented paradigm and we violate the Eif-

fel development process and methodology. The next sections summarizes the advantages of modular verification over an "a posteriori" verification strategy using global analysis.

4.4.1 Non-modular verification is not easier

As personal experience with teaching loop invariants suggests, it is much more difficult to come up with the right functional specifications than to do the actual proof. Even during non-modular verification, we will invent these theorems and carry them around in our proofs. But, in contrast to modular verification using full specifications, these theorems do not become part of the specification. So they have to be invented again every time a feature is used.

Modular verification has to put all properties of a feature into the contract of the feature. This simplifies reusing proofs and limits the domain of discourse for a proof. The set of proof obligations and hypothesis remains manageable.

4.4.2 Cluster development model

Eiffel suggests the cluster development model [52, pp. 926]. The idea is to develop different parts of the software in parallel, each containing, among other things, an own validation and verification phase.

Non-modular verification would instead require that the validation and verification phase come after all required modules have been implemented, putting the verification phase outside of the cluster model and instead at the end of the development cycle.

4.4.3 Software reuse

The cluster development model promotes software reuse. After successful implementation and verification, it should be possible to factor out parts of the software to make them available to other systems.

If the proof of correctness is done using non-modular reasoning, it is not possible to extract, from the proof of the whole system, proofs of individual parts. The part that is factored out for reuse has to be delivered without a certificate of correctness.

With modular verification, the correctness proof of a class becomes a certificate of correctness of the class. This certificate can be delivered together with the class. The class becomes a "trusted component", a software component with the certified quality of correctness.

4.4.4 Modular continuity

The main motivation for object-orientation is the introduction of modularity into the development process. Understanding that software maintenance takes up a large part of the software development cycle, changes to the software should stay confined to the corresponding modules, as expressed by Modular Continuity:

> *A method satisfies Modular Continuity if, in the software architectures that it yields, a small change in a problem specification will trigger a change of just one module, or a small number of modules.*
> (Meyer [52, p. 44])

Non-modular verification violates Modular Continuity: specification boundaries form the natural fence to confine change to a module or a small number of modules. If the proofs do not rely on these specification boundaries, the proof needs to be redone completely after the change.

4.4.5 Complete functional specifications

Modular verification forces us to write full functional specifications for all suppliers. Without modular specifications, there is no check that the contracts of suppliers are sound or complete.

The problem of partial specifications is intricate: the fact that a specification is partial will not be revealed during the correctness proof. But when the correctness proof is used as a certificate for the system quality, then deploying the system in an environment will reveal these deficiencies. This is late in the development process and the costs of fixing the deficiencies are very high.

The development process should discover holes in the specifications as early as possible. Modular verification enforces this.

4.4.6 Subtyping and polymorphism

It is possible to subtype a class to extend it (see *Open Closed Principle*, [52, p. 57]). Through polymorphic attachment, the static type of an entity and the dynamic type of its value may diverge. Eiffel provides a set of subtyping rules [52, pp. 835] relating the contract of the parent class and the child class. The purpose of these rules: any implementation that is correct for the child's contract is also correct for the parent's contract.

If we can prove the correctness of a class using only the contract of the static type of an entity (and not its implementation) we know that class is

correct irrespectively of the dynamic type of the entity. This is a tremendous reduction of proof complexity, as we do not have to take dynamic binding into account.

4.5 Summary

We regard the ability to reason modularly about Eiffel code as a key quality for the success of an Eiffel verification technique. Modular reasoning here means the reasoning on the basis of:

- Text of the class.
- Text of all direct or indirect parent classes.
- Interface specification of all direct and indirect suppliers.

The success of object-orientation relies heavily on modularization, bottom-up development and information hiding. Going for a non-modular verification technique would result in a mode of reasoning violating object-orientation.

We have also seen that it is not possible to apply modular reasoning for the inheritance relation. Instead, child classes always need to have all details of the inherited code to certify correctness. Information hiding is not possible through the inheritance relation. Privacy relations such as `private` or `protected` as available in C++, C# or Java just obscure this insight.

CHAPTER 5

LANGUAGE SEMANTICS

There is little point in verifying Eiffel without a well-defined semantics for the language. The two successive Eiffel standards, ETL2 [50] and ECMA-367 [56], provide extensive descriptions of the Eiffel language, but the semantics is still provided in an informal, natural language form that is not suited for formal reasoning.

The purpose of this chapter is to define a significant subset of Eiffel through the introduction of a natural, big-step operational semantic description[63].

An early version of this semantics was published as Eiffel0 [76]. The goal of Eiffel0 was mainly to explain the semantics of dynamic frame contracts (see chapter 7). The semantics presented in this chapter significantly extends Eiffel0, including weak purity, inheritance/subtyping, genericity, once and creation procedures.

The language described tries to be as close as possible to the official ECMA standard, but explicitly excludes some new language features:

- **Attached types** [56, sec. 8.11.19] an **object tests** [56, sec. 8.24.1]: The problem with these constructs is that they are neither widely used nor very well understood. It can be expected that these constructs will change significantly in future revisions of the Eiffel language.
- **Exception handling**: Eiffel uses exception handling to introduce robustness, to describe behavior in the case of incorrect software. Correct code should not throw exceptions. Removing exception handling from the operational semantics dramatically simplifies reasoning.
- **Agents**: reasoning with agents in Eiffel is problematic, as they do not carry contracts.

The semantics description of the Eiffel programming language proceeds in three separate steps:

1. In the **static model**, we define sets and functions that describe the Eiffel code as a high-level description. The functions declared are constant throughout the execution of a program, as they describe the analyzed program code, not the dynamic behavior at run-time. We assume that all compilation checks have been performed on the Eiffel source code and we do indeed have legal Eiffel code.

2. The **state model** describes the state of a machine executing Eiffel code. The state is defined as an abstract data type.

3. The **Execution model** describes the effect of executing Eiffel code as defined in the static model on the state as defined in the state model by a von-Neumann type of machine.

Each model is based on the previous model, thus forms an extension. The given structure makes it possible to replace any of the later models while keeping the previous one.

The static model is not a direct model of the language or its abstract syntax tree. It simplifies the language. The simplifications are not achieved by removing language features, but instead by removing redundant information and flattening (resolving moving the inherited code from the parent to the child). These transformations are described in section 5.3.4.

5.1 Related Work

A first description of the semantics of Eiffel was published in the book "Eiffel, the language" [48], superseded in 1992 by a second revision [50]. A third revision is currently in preparation.

The main definition of the Eiffel language is the ECMA/ISO Eiffel standard[56]. This standard is the official document for the definition of the Eiffel language. It incorporates a number of extensions to the language as it currently implemented in Eiffel compilers, for example the work on attached types or type safety. Some of these extensions might never get implemented as the Eiffel ECMA committee is constantly reviewing these extensions and is experimenting with alternative solutions.

Formal definitions of the Eiffel programming language are contained in the work by Paige and Ostroff[70] as well as in the research on SCOOP

by Nienaltowski[64]. Another semantics for a core subset of Eiffel was done by Thomas Bietenhader [8].

A first version of the theory presented here was published under the name of Eiffel0 [76].

The heap model presented here is based on the work by Müller[61] for Java.

5.2 Mathematical notation

The mathematical notation used in this thesis is derived from two different sources:

- First-order logic and Zermelo-Fraenkel set theory as introduced in the B book by Abrial [1].
- The notation for defining abstract data types introduced by Müller in his thesis [60].

Appendix A gives a summary of uncommon symbols.

Except for the heap model, all definitions and axioms are created specifically for the Eiffel language, but try to follow text book examples for the definition on an operational sematics. The heap model is based on a given semantics for Java [60].

We always first express definitions and axioms mathematically, and then explain the connected semantics in free form text, referencing the labels of the formulas. This makes it possible to use the definition as reference, separating formalization from informal text.

Basic sets used for reasoning are called *carrier sets* (sometimes called *sorts*). All carrier sets are not empty and disjoint from each other.

Regular sets start with capital letters, functions and relations — if not using specific infix operators — have names starting with lower-case letters. If a name is made out of multiple words, "CamelCase" is used.

Arbitrary function symbols combining prefix, postfix and infix notation are defined using the underscore _ to declare open argument positions. Binary functions using symbols use infix notation.

Proofs are a sequence of $H_1, H_2, \cdots \vdash G$ theorems, where $H_1, H_2, \ldots$ are a list of hypothesis, and G is a given goal of the proof. Proofs are shown using backwards reasoning, simplifying the theorem to simpler theorems using implication rules. These proof steps are introduced by a justficiation and the $\Rightarrow$ symbol.

5.3 Static model

The static model is a high-level view on the program text after compilation. It defines Eiffel program text at a very late part of the compilation phase. All names have been resolved and dynamic binding tables have been computed.

5.3.1 Classes and Types

$$Class \text{ is a finite carrier set} \tag{5.1}$$
$$Type \text{ is a carrier set} \tag{5.2}$$
$$\mathsf{constraints} : Class \to Type* \tag{5.3}$$
$$\sqsubseteq : Type \leftrightarrow Type \tag{5.4}$$
$$\mathsf{classOf} : Type \to Class \tag{5.5}$$
$$\mathsf{parameters} : Type \to Type* \tag{5.6}$$
$$(\forall i : v_i \sqsubseteq \mathsf{constraints}(C)_i) \Rightarrow C[\overrightarrow{v}] \in Type \tag{5.7}$$
$$A \sqsubseteq A \tag{5.8}$$
$$(A \sqsubseteq B) \wedge (B \sqsubseteq C) \Rightarrow (A \sqsubseteq C) \tag{5.9}$$
$$(A \sqsubseteq B) \wedge (B \sqsubseteq A) \Rightarrow (A = B) \tag{5.10}$$
$$FrozenType \subset Type \tag{5.11}$$
$$\forall T \in Type, F \in FrozenType : T \sqsubseteq F \Rightarrow T = F \tag{5.12}$$
$$EType \subseteq FrozenType \tag{5.13}$$
$$RType = Type \setminus EType \tag{5.14}$$
$$\texttt{ANY} \in RType \tag{5.15}$$
$$\forall T \in Type : T \sqsubseteq \texttt{ANY} \tag{5.16}$$
$$\texttt{NONE} \in RType \tag{5.17}$$
$$\forall T \in RType : \texttt{NONE} \sqsubseteq T \tag{5.18}$$
$$\texttt{ANY} \neq \texttt{NONE} \tag{5.19}$$

The Eiffel system is defined by a set of classes (5.1). Each class can be uniquely identified by a name, which we write using Eiffel-style capital letters: *INTEGER*, *BOOLEAN*, etc. The set of all classes is finite, as the program text is finite.

Each class defines a sequence (vector) of generic parameters and constraints $\overrightarrow{p}$ for these parameters (5.3). Constraints are types (5.2). If no explicit constraint is given, then we assume that *ANY* (5.15) is the implicit constraint. The sequence may be empty. A type is created from a class by

instantiating the generic parameters with types that conform to the constraints.

If $\overrightarrow{p}$ is the empty sequence, then the type can be written as C instead of $C[]$. In contrast to the set of classes, the set of types is infinite.

Every type can be queried for its constructing class (5.5) and its type parameters (5.6).

As the set of types is potentially infinite, it is impossible to compute the set of all dynamic types at compilation time. This differs from the definition of *dynamic type set* in ETL2, which implies that the dynamic type set of an entity can be computed at compilation time:

> *The set of possible dynamic types for an entity or expression* `x` *is called the* ***dynamic type set*** *of* `x`*.* [...] *It is possible to determine the dynamic type set of* `x` *through analysis of the classes in the system to which x belongs, by considering all the attachment and reattachment instructions involving* `x` *or its entities.* (Meyer, ETL2 [50, p. 323])

The relation $\sqsubseteq$ (5.4) is the *subtype* relation. It is a partial ordering of types, and is thus reflexive (5.8), transitive (5.9) and antisymetric (5.10). $A \sqsubseteq B$ reads as "type `A` conforms to type `B`".

Some types are frozen (5.11): they do not have subtypes (5.12). A given type can be either an expanded type (5.13) or a reference type (5.14). Expanded types are always frozen.

The special type `ANY` is a reference type (5.15). It is the supertype of all types (5.16). The type `NONE` is a reference type (5.17) and the common subtype of all reference types (5.18). `ANY` and `NONE` are different types (5.19).

5.3.2 Features

$$\mathit{Feature} \text{ is a carrier set} \tag{5.20}$$

$$\mathsf{defIn} : \mathit{Feature} \to \mathit{Type} \tag{5.21}$$

$$\mathsf{features} : \mathit{Type} \to \mathbb{P}(\mathit{Feature}) \tag{5.22}$$

$$\forall T \in \mathit{Type} : \mathsf{features}(T) = \mathsf{defIn}^{-1}[T] \tag{5.23}$$

$$\mathsf{version} : \mathit{Type} \to (\mathit{Feature} \nrightarrow \mathit{Feature}) \tag{5.24}$$

$$\forall T \in \mathit{Type} : \mathrm{ran}(\mathsf{version}(T)) \subseteq \mathsf{features}(T) \tag{5.25}$$

$$\forall B \in \mathit{Type} \setminus \{\texttt{NONE}\}, A \in \mathit{Type} : B \sqsubseteq A \Rightarrow \mathsf{features}(A) \subseteq \mathrm{dom}(\mathsf{version}(B)) \tag{5.26}$$

$$\mathsf{features}(\texttt{NONE}) = \varnothing \tag{5.27}$$

$$\mathsf{args} : \mathit{Feature} \to \mathit{Type}* \tag{5.28}$$

$$\mathsf{ret} : \mathit{Feature} \nrightarrow \mathit{Type} \tag{5.29}$$

$$\forall m \in \mathrm{ran}(\mathsf{version}), f \in \mathit{Feature}, T \in \mathit{Type} : \mathsf{ret}(f) = T \Rightarrow \mathsf{ret}(m(f)) \sqsubseteq T \tag{5.30}$$

$$\forall m \in \mathrm{ran}(\mathsf{version}), f \in \mathit{Feature}, n \in \mathrm{dom}(\mathsf{args}(f)), T \in \mathit{Type} : \mathsf{args}(f)(n) = T \Rightarrow \mathsf{args}(m(f))(n) \sqsubseteq T \tag{5.31}$$

$$\mathit{Query} = \mathrm{dom}(\mathsf{ret}) \tag{5.32}$$

$$\mathit{Command} = \mathit{Feature} \setminus \mathit{Query} \tag{5.33}$$

Eiffel code defines a set of features (5.20). Though features are defined in classes, we regard features as introduced by types: features are always relative to certain type (5.21), they never appear in more than one type. A type can be queried for its features (5.22).

Features are linked together through the subtyping relation: for every feature f of A and any subtype B of A, there exists exactly one feature g in B that is the *descendant version* of f of A. This is expressed by the version function (5.24) and axioms (5.25) and (5.26). The only exception to this rule is the type `NONE`, that does not define any features (5.27), although it is a subtype of all reference types. Excluding `NONE` from this rule creates the problem of *Void calls* (a.k.a. null-pointer exceptions), causing version to be undefined if we want to look up the descendant version of some feature in **`Void`**.

Each features defines a sequence of argument types (5.28), and possibly a return type (5.29). The types for arguments and the return type of descendant versions must conform to the type in the original version (5.31) (5.30). This means that arguments and return values can be covariantly redefined in subtypes. As described in section 3.1, this may break the type system.

Queries are all features defining a return type (5.32); the other features are called *Commands* (5.33).

5.3.3 Program text

$$\mathit{Expr} \text{ is a carrier set} \tag{5.34}$$

$$\mathit{Instr} \text{ is a carrier set} \tag{5.35}$$

$$\mathit{Local} \text{ is a carrier set} \tag{5.36}$$

$$\texttt{Current} \in \mathit{Local} \tag{5.37}$$

$$\texttt{Result} \in \mathit{Local} \tag{5.38}$$

$$\mathit{ManifestConstant} \text{ is a carrier set} \tag{5.39}$$

$$\mathsf{typeOf} \in \mathit{Expr} \rightarrow \mathit{Type} \tag{5.40}$$

Actual implementations in Eiffel are supplied by program text. We differentiate between **instructions** (5.35) and **expressions** (5.34). Expressions are used to compute values and to express contracts. Instructions describe state change and flow of control.

The grammar for expressions is extended by a construct not available in Eiffel: the possibility to assign to local variables in expressions. The target local used for this assignment always has to be a fresh variable that is not used somewhere except in the constructs that have explicitly rewritten. The introduction of this limited form of assignment into expressions makes it possible to simplify many of the other language constructs, like **`old`** expressions or argument passing. The simplifications are discussed in section 5.3.4.

The following grammars for recursive trees defines these two data structures describing program text:

$$\begin{aligned}
\mathit{Expr} := {} & \mathit{Local} \\
& \mid (\mathit{Local} \ := \ \mathit{Expr}); \mathit{Expr} \\
& \mid \texttt{create}\ \mathit{Feature}[(\overrightarrow{\mathit{Local}})] \\
& \mid \mathit{ManifestConstant} \\
& \mid \mathit{Local} = \mathit{Local} \\
& \mid [\mathit{Local}.]\mathit{Feature}[(\mathit{Local}, \ldots)]
\end{aligned} \tag{5.41}$$

$$\begin{aligned}
\mathit{Instr} := {} & \epsilon \\
& \mid [\mathit{Local}.]\mathit{Feature}[(\mathit{Local}, \ldots)] \\
& \mid \mathit{Local} \ := \ \mathit{Expr} \\
& \mid \mathit{Feature} \ := \ \mathit{Local} \\
& \mid \mathit{Instr}\ [;]\ \mathit{Instr} \\
& \mid \textbf{from until}\ \mathit{Local} \\
& \quad \textbf{invariant}\ \mathit{Expr} \\
& \quad \textbf{variant}\ \mathit{Expr} \\
& \quad \textbf{loop}\ \mathit{Instr} \\
& \ \textbf{end} \\
& \mid \textbf{if}\ \mathit{Local}\ \textbf{then}\ \mathit{Instr}\ \textbf{else}\ \mathit{Instr}\ \textbf{end} \\
& \mid \textbf{check}\ \mathit{Expr}\ \textbf{end} \\
& \mid \texttt{attribute}
\end{aligned} \tag{5.42}$$

Depending on the context, program text may reference a number of *local entities* (5.36). Local entities are arguments of surrounding feature, declared local variables, **Current** (5.37) and **Result** (5.38). In every context (contract, routine body), local entities have a well-defined static type.

In section 3.5, we have described that Eiffel expressions are based on a few number of constructs. We can see that only 6 production rules for expressions and 9 rules (including the special `attribute` rule) are needed to define the full grammar of executable code in Eiffel. It is sufficient that the only instruction referencing expressions is the assignment to local variables.

ManifestConstant(5.39) are special creation instructions creating constant values. Examples are integer values like `0`, `1`, `-40` or `4711` to create instances of *INTEGER*, **True** and **False** to create instances of *BOOLEAN* or **"Hello"** to create the corresponding instance of *STRING*.

We are referencing features directly, assuming that resolving the names has already been done statically by the compiler. All features that appear in code are features of the of the static type of the target of the call. The static type of any expression is available through the *typeOf* function (5.40).

Instructions form the body of feature implementations. They are structured code. The special instruction `attribute` is used to mark that a feature is implemented as an attribute. It may only appear at the top of an instruction tree.

5.3.4 Language simpliciations

The given grammar assumes that we can do a number of simplifications at compile-time without changing the semantics of the code:

1. Instructions in the **from** clause are moved in front of the loop:
   from *I1* **until** *E1* **invariant** *E2* **loop** *I2* **end**
   is rewritten to
   I1 **from until** *E1* **invariant** *E2* **loop** *I2* **end**

2. Empty **else** clauses are added to all **if** instructions:
   if *E* **then** *I* **end**
   is rewritten to
   if *E* **then** *I* **else end**

3. **elseif** clauses are removed and replaced by corresponding inner **if** clauses:
   if *E1* **then** *I1* **elseif** *E2* **then** *I2* **else** *I3* **end**

is rewritten to
`if E1 then I1 else if E2 then I2 else I3 end end`

4. **debug** instructions are removed.

5. Multi branch instructions (**inspect**) are replaced by semantically equivalent **if** instructions.

6. Object creation always happens in expressions: `create x.make`
is rewritten to
`x := create make`
As all features, in our case `make` are directly linked to a specific type, adding information on the target type, like in `create {T}.make` is not necessary.

7. Calls to **old** have been replaced by assigning to fresh local variables in the precondition and then to reference these local variables in the postcondition instead of the **old** expression:
`require P do C ensure Q(old E)end`
is rewritten to
`require X := E ; P do C ensure Q(X)end`
where `X` is a fresh local variable.

8. Arguments to features or creators are always local variables. Arguments that are not local variables are replaced by an assignment to a fresh local variable and then the use of this local variable as argument:
`t.f(a)`
is rewritten to
`X := a ; t.f(X)`
where `X` is a fresh local variable.

9. The same rule holds for the target of a call:
`t.f(a)`
is rewritten to
`X := t ; X.f(a)`
where `X` is a fresh local variable.

10. Also, we remove expressions from the condition expressions in **if** clauses:
`if E then A else B end`
is rewritten to

```
X := E ; if X then A else B end
```
where *X* is a fresh local variable.

11. In the case of loops, we remove expressions from termination conditions:
```
from until T invariant I variant V loop B end
```
is rewritten to
```
TL := T
from until TL invariant I variant V loop
B ; TL := T end
```
where *TL* is a fresh local variables.

12. All calls of `Precursor` have been flattened out by the inherited implementation. Flattening of `Precursor` can always be done as recursion is not possible using such calls: The inheritance tree is always finite and acyclic. Calls to `Precursor` functions are flattened by creating a private function in the current class and duplicating the implementation of the inherited function. `Precursor` can never access an attribute, as redefining attributes is not allowed.

All simplifications are possible by an analysis of the given program text, using simple refactoring rules or other semantically neutral transformations.

5.3.5 Contracts

$$\mathsf{pre} : \mathit{Feature} \rightarrow \mathit{Expr} \tag{5.43}$$

$$\mathsf{post} : \mathit{Feature} \rightarrow \mathit{Expr} \tag{5.44}$$

$$\mathsf{inv} : \mathit{Type} \rightarrow \mathit{Expr} \tag{5.45}$$

All features carry contracts. Each feature has a *precondition* (5.43) and a *postcondition* (5.44). Each type has a *class invariant* (5.45). Contracts are boolean expressions.

We can assume that every feature has a pre- and a postcondition, and that every type has an invariant: if no contract is expressed in the Eiffel code or inherited, then the expression is the *ManifestConstant* **`True`**.

Contracts include all inherited contracts, properly combined following the Eiffel inheritance rules. The precise rules governing inheritance require an understanding of state and contract evaluation. They are defined in section 5.8.

5.3.6 Implementation

$$\mathsf{body} : \mathit{Feature} \nrightarrow \mathit{Instr} \tag{5.46}$$
$$\mathit{EffType} \subseteq \mathit{Type} \tag{5.47}$$
$$\mathit{DefType} = \mathit{Type} \setminus \mathit{EffType} \tag{5.48}$$
$$\forall T \in \mathit{EffType}, F \in \mathit{features}(T) : F \in \mathrm{dom}(\mathsf{body}) \tag{5.49}$$
$$\mathit{Attribute} = \mathsf{body}^{-1}[\texttt{attribute}] \tag{5.50}$$
$$\mathit{Attribute} \subseteq \mathit{Query} \tag{5.51}$$
$$\mathit{Routine} = \mathrm{dom}(\mathsf{body}) - \mathit{Attribute} \tag{5.52}$$
$$\mathit{Function} = (\mathrm{dom}(\mathsf{body}) \cap \mathit{Query}) - \mathit{Attribute} \tag{5.53}$$
$$\mathit{OnceFunction} \subseteq \mathit{Function} \tag{5.54}$$
$$\mathit{Creator} \subseteq (\mathit{Command} \cap \mathsf{defIn}^{-1}[\mathit{EffType}]) \tag{5.55}$$

Features may carry implementations, defined by the body function (5.46). Some types are *effective* (5.47), all others are *deferred* (5.48). An effective type defines an implementation for every feature (5.49). For other types, the body function may be partial.

Feature implemented using the `attribute` instruction are contained in the set $\mathit{Attribute}$(5.50). Attributes are always queries (5.51). All other feature with implementations are called *routines* (5.52). Implemented queries that are not attributes are called *functions* (5.53).

The implementation of a function might be declared as **`once`** (5.54). Such an implementation is only executed once on the first call. Future calls to the feature yield the same return value as the first call, without the execution of the feature.

A subset of the features for a given effective type are used to initialize instances of this type. These features are called *creation procedures* (5.55). Creation procedures are always commands.

5.4 State model

$$\mathit{State} \text{ is a carrier set} \tag{5.56}$$
$$\mathit{Heap} \text{ is a carrier set} \tag{5.57}$$
$$\mathit{Env} \text{ is a carrier set} \tag{5.58}$$
$$\mathit{Global} \text{ is a carrier set} \tag{5.59}$$
$$\mathsf{env} : \mathit{State} \to \mathit{Env} \tag{5.60}$$

$$\mathsf{heap} : State \to Heap \tag{5.61}$$
$$\mathsf{global} : State \to Global \tag{5.62}$$

In an operational semantic, we define the execution of a program as a state transformation, thus always considering a given piece of code and state, and describing the resulting state. In this section, we define the data structure that is sufficiently precise to describe the state of an Eiffel program at a discrete point in time.

The state (5.56) has three components: the heap (5.57), the environment (5.58) and the global state (5.59). The heap describes objects and their associated attributes. The environment describes the state of local variables and parameters, so information that is local to a feature execution. Finally, the global state is used to store the return values of **once** features, making the available for future look up.

To access the components of a state, we use total functions (5.60), (5.61) and (5.62). As applications of these functions happen so often, indexes are used: if s is a state, then s_H is its heap, s_E is its environment and s_G is its global state. Also, we can construct a state with a given environment, heap and global state by using a tuple notation, i.e. (h, e, g).

Each component has got an access function written $c(l)$ and an update function written $c\langle l := v\rangle$. Because it is defined for heaps, environments and global states, both functions are overloaded.

5.4.1 Heap

$$Obj \text{ is a carrier set} \tag{5.63}$$
$$Loc \text{ is a carrier set} \tag{5.64}$$

$$\_.\_ : Obj \times Attribute \to Loc \tag{5.65}$$
$$\_(\_) : Heap \times Loc \to Obj \tag{5.66}$$
$$\_\langle\_ := \_\rangle : Heap \times Loc \times Obj \to Heap \tag{5.67}$$
$$\mathsf{new} : Heap \times EffType \to Obj \tag{5.68}$$
$$\_\langle\_\rangle : Heap \times EffType \to Heap \tag{5.69}$$
$$\mathsf{alloc} : Obj \times Heap \to \mathrm{bool} \tag{5.70}$$
$$\mathsf{typeof} : Obj \to EffType \tag{5.71}$$

$$\forall H \in Heap, l \in Loc, L, K, X, Y \in Obj, f, g \in Attribute, T \in Type :$$

$$L \neq K \vee f \neq g \Rightarrow H\langle L.f := X\rangle(K.g) = H(K.g) \tag{5.72}$$

$$H\langle L.f := X\rangle(L.f) = X \tag{5.73}$$

$$\mathsf{alloc}(L, H) \Rightarrow H\langle T\rangle(L.f) = H(L.f) \tag{5.74}$$

$$H\langle T\rangle(\mathsf{new}(H).f) = \texttt{Void} \tag{5.75}$$

$$\mathsf{alloc}(X, H\langle l := Y\rangle) \Leftrightarrow \mathsf{alloc}(X, H) \tag{5.76}$$

$$\mathsf{alloc}(X, H\langle T\rangle) \Leftrightarrow \mathsf{alloc}(X, H) \vee X = \mathsf{new}(H, T) \tag{5.77}$$

$$\mathsf{alloc}(H(l), H) \tag{5.78}$$

$$\neg\mathsf{alloc}(\mathsf{new}(H, T), H) \tag{5.79}$$

$$\mathsf{typeof}(\mathsf{new}(H, T)) = T \tag{5.80}$$

The heap model is taken from the heap model introduced by Müller [61]. The name of some sets are changed and the concept of object initialization introduced.

It differentiates between *objects* (5.63) and *locations* (5.64). Objects are references to instances of types. They are made up of a number of locations. Locations are created in the heap and carry information in form of object references. The heap keeps track of the values stored in locations and the set of objects currently allocated.

All functions (5.65)–(5.71) are total. Although not every attribute is available in every object, we regard the values of these attributes as arbitrary. Objects on the heap are always instances of effective types. The semantics of *Heap* is defined by the axioms (5.72)–(5.80).

5.4.2 Environment

$$\begin{aligned} \_(\_) &: Env \times Local \rightarrow Obj \\ \_\langle \_ := \_\rangle &: Env \times Local \times Obj \rightarrow Env \end{aligned}$$

$$\forall E \in Env, L, K \in Local, x \in Obj :$$

$$E\langle L := x\rangle(L) = x \tag{5.81}$$

$$L \neq K \Rightarrow E\langle L := x\rangle(K) = E(K) \tag{5.82}$$

$$\mathsf{alloc}(E(L), H) \tag{5.83}$$

The environment maps locals to corresponding values. It is thus a simple associative array. Again, functions (5.81) and (5.81) are total. Although

not every local is available in the context of every program text, we regard the values of these attributes as arbitrary. The semantics of Env is defined by the axioms (5.81)–(5.82). Axiom (5.83) states that objects stored in the environment are always allocated.

Sometimes an environment only matters with respect to the values it defines for specific variables. The notation $\langle a := x, b := y, \dots \rangle$ stands for $E\langle a := x\rangle\langle b := y\rangle \dots$ where E is an arbitrary environment; the only properties that we may use in connection with this notation are those of the listed variables.

5.4.3 Global State

$$\mathsf{empty} \;:\; Global \tag{5.84}$$
$$\_(\_) \;:\; Global \times OnceFunction \to Obj \tag{5.85}$$
$$\_\langle \_ := \_ \rangle \;:\; Global \times OnceFunction \times Obj \to Global \tag{5.86}$$
$$\mathsf{stored} \;:\; Global \times OnceFunction \to \mathrm{bool} \tag{5.87}$$

$$\forall G \in Global, F, H \in OnceFunction, x \in Obj :$$
$$G\langle F := x\rangle(F) = x \tag{5.88}$$
$$F \neq H \Rightarrow G\langle F := x\rangle(H) = E(H) \tag{5.89}$$
$$\mathsf{stored}(G, \mathsf{empty}) = \mathrm{FALSE} \tag{5.90}$$
$$\mathsf{stored}(G\langle F := x\rangle, F) = \mathrm{TRUE} \tag{5.91}$$
$$F \neq H \Rightarrow \mathsf{stored}(G\langle F := x\rangle, H) = \mathsf{stored}(G, H) \tag{5.92}$$
$$\mathsf{alloc}(E(L), H) \tag{5.93}$$

The global state keeps track of the return values created by the execution of **once** routines. At the beginning of the execution, the global state is empty.

The work only considers a very limited form of **once** evaluations, *once per type*. This means that every type has its own set of once features, independent of parent types or other generic derivations. This is a more restricted form when compared to the definitions [56, sec. 8.23.21] in ISO / ECMA, which consider the explicit dynamic binding version of a feature. Thus, multiple types may share a single once variable. It is possible to implement the full semantics of once as in ISO/ECMA by using equivalence classes of features that share a once variable.

Once per object is not considered here, as it can be implemented using an explicit caching attribute. Once per thread requires the concept of

multi-threading, and this thesis only considers non-concurrent executions of Eiffel.

Again, we use total functions (5.84)–(5.87) as the signature of the data type. Looking up a feature in the empty global state yields an arbitrary value. The semantics of $Global$ is defined by the axioms (5.88)–(5.92). Axiom (5.93) states that objects stored in the global state are always allocated.

5.4.4 Monomorphic state relation

$$\_ \sqsubseteq \_ : State \times State \tag{5.94}$$

$$\begin{aligned} s \sqsubseteq s' \Leftrightarrow & \exists \tau : Obj \rightarrowtail\!\!\!\!\!\rightarrow Obj | \\ & \forall o : Obj | \mathsf{typeof}(o) = \mathsf{typeof}(\tau(o)) \\ & \forall o : Obj, a : Attribute | \mathsf{alloc}(o, s_H) \Rightarrow \tau(s_H(o.a)) = s'_H(\tau(o).a) \\ & \forall l : Local | \tau(s_E(l)) = s'_E(l) \\ & \forall f : OnceFunction | \tau(s_G(f)) = s'_G(f) \\ & \forall f : OnceFunction | \mathsf{stored}(f, s) = \mathsf{stored}(f, s') \end{aligned} \tag{5.95}$$

A state is called *monomorphic* (in terms of category theory [24]) with relation to another state, if it is equivalent, except for object identities and new objects. Such a relation is needed when we want to show that two states are equivalent, but a different order in which objects are created might lead to different object identities or new allocated, but not reachable object.

The monomorphism relation between states is transitive and reflexive. It is not an isomorphism, as the second state can have more allocated objects than the first one.

Equivalent changes to monomorphic states preserve the monomorphic property. If $s \sqsubseteq s'$, then the following theorems hold:

$$(s_H\langle l := o\rangle, s_E, s_G) \sqsubseteq (s'_H\langle l := o\rangle, s'_E, s'_G) \tag{5.96}$$

$$(s_H\langle T\rangle, s_E, s_G) \sqsubseteq (s'_H\langle T\rangle, s'_E, s'_G) \tag{5.97}$$

$$(s_H, s_E\langle l := o\rangle, s_G) \sqsubseteq (s'_H, s'_E\langle l := o\rangle, s'_G) \tag{5.98}$$

$$(s_H, s_E, s_G\langle l := o\rangle) \sqsubseteq (s'_H, s'_E, s'_G\langle l := o\rangle) \tag{5.99}$$

$$(s_H, s_E, s_H) \sqsubseteq (s_H\langle T\rangle, s_E, s_G) \tag{5.100}$$

Proof outline: Assuming that f is the (existing) monomorphism relation between s and s'. The equations (5.98) and (5.99) do not modify the heap,

we can define f', the new mapping to be equivalent to f. Equations (5.97) and (5.96) changes the heap, but because of axiom (5.74), all existing objects retain their attribute values. The difference between f' and f is the $\mathsf{new}(s_H)$ object. The initialization rule (5.75) causes all attributes to be mapped to **`Void`**, making both objects equivalent for all functions.

5.5 Execution model

$$\langle \_, \_ \rangle \rightsquigarrow \_ \subseteq \mathit{Instr} \times \mathit{State} \times \mathit{State} \tag{5.101}$$

$$\mathbf{eval}(\_, \_) : \mathit{Expr} \times \mathit{State} \rightarrow \mathit{Obj} \tag{5.102}$$

$$[\![\_, \_]\!] \rightsquigarrow \_ \subseteq \mathit{Expr} \times \mathit{State} \times \mathit{State} \tag{5.103}$$

The execution model describes the effect of executing program text on a given state. We assume that the execution of Eiffel program text is always deterministic: when started in the same state, the same program text always yields the same resulting state.

The model is defined through two relations and a function. Relation (5.101) defines how an instruction transforms a given state. Relation (5.103) defines how an expression transforms a given state. Function (5.102) defines the resulting value of an expression evaluation. The two relations and the function are mutually recursive.

The relation (5.101) is called the *state transformation relation*. The expression $\langle I, s \rangle \rightsquigarrow s'$ reads as "when started in a state s, the execution of the instructions I results in the state s'".

To describe expression evaluation, we must consider two effects: a return value is computed (5.102) and the query might change the state as a side-effect (5.103). This is even true for code respecting command/query separation: the evaluation of the expression can allocate new objects and can assign values to these objects.

For the side-effect, the relation has a similar syntax as state transformation relation has for instructions. The expression $[\![E, s]\!] \rightsquigarrow s'$ informally reads as "when evaluating E in a state s, the state will be changed to state s'". The actual return value is computed by the $\mathbf{eval}(E, s)$ function. It reads as "evaluating expression E in state s yields $\mathbf{eval}(E, s)$".

In the definition of the execution model, we will use a number of free, all-quantified variables. They are defined as follows: $e, e^1, e^2, \cdots \in \mathit{Expr}$, $a \in \mathit{Attribute}$, $f \in \mathit{Function}$, $c \in \mathit{Creator}$, $s, s', s'', s^0, s^1, \cdots \in \mathit{State}$, $a^0, a^1, \cdots \in \mathit{Obj}$, $arg^1, arg^2, \ldots, l, l', l'', l^0, l^1, \cdots \in \mathit{Local}$, $mc \in \mathit{ManifestConstant}$.

The following sections contain the formal definition of the operational semantics.

5.5.1 Evaluating expressions

$$\mathbf{eval}(l, s) = s_E(l) \tag{5.104}$$

$$\mathbf{eval}(l = l', s) = (s_E(l) = s_E(l')) \tag{5.105}$$

$$\frac{\langle l := e, s\rangle \rightsquigarrow s'}{\mathbf{eval}(l := e; e', s) = \mathbf{eval}(e', s')} \tag{5.106}$$

$$\mathbf{eval}(\texttt{create}\, c(l^1, \ldots, l^n), s) = \mathsf{new}(s) \tag{5.107}$$

$$\mathbf{eval}(mc, s) = \mathsf{new}(s) \tag{5.108}$$

$$\mathbf{eval}(l.a, s) = s_H(s_E(l).a) \tag{5.109}$$

$$\frac{\left[\begin{array}{c} f \notin \mathit{OnceFunction} \vee \neg\mathsf{stored}(s_G, f) \\ B = \mathsf{body}(\mathsf{version}(\mathsf{typeof}(s_E(l)))(f)) \\ E = \langle \texttt{Current} := s_E(l), arg^1 := s_E(l^1), \ldots, arg^n := s_E(l^n)\rangle \\ \langle B, (s_H, E, s_G)\rangle \rightsquigarrow s' \end{array}\right]}{\mathbf{eval}(l.f(l^1, \ldots, l^n), s) = s'_E(\texttt{Result})} \tag{5.110}$$

$$\frac{\left[\, f \in \mathit{OnceFunction} \wedge \mathsf{stored}(s_G, f)\right]}{\mathbf{eval}(l.f(l^1, \ldots, l^n), s) = s_G(f)} \tag{5.111}$$

Reading a local entity causes a look up in the environment (5.104). Reference equality means looking up both local variables of the equation in the environment and then comparing the objects (5.105). The assignment to a fresh, local variable in expressions is evaluated by computing the effect of the assignment on the environment, and the to continue the evaluation of the remaining expression (5.106).

Creating a new object is done by mere allocation of the object on the heap. There is not need to execute the creation feature, as this will have no effect on the value of the expression (5.107), though it might have an effect on its side-effect, as we will see below. Manifest constants are also created in a similar way (5.108).

Reading an attribute requires evaluating the target and look up the attribute on the target object on the heap (5.109). Reading a local attribute `e` is done using **`Current`**`.a`.

Calling functions (5.110) will first look up the actual body of the target feature using the rules for dynamic binding: using the dynamic type of the target object and the static feature to be called, we compute the corresponding dynamic feature to be called using the version relation. We then look up the body implementing the dynamic feature by using the body function.

A new environment E is created for the execution of the body that binds the target to `Current` and the arguments to the corresponding values. The body is then executed in a state composing of the original heap and global state, and the temporary environment. The result value is then gained by looking up the local **`Result`** in the environment of the post-state.

If the feature called is a once function that has already been executed, then the pre-computed value is just returned (5.111).

5.5.2 *Side effects in expressions*

$$[\![l, s]\!] \rightsquigarrow s \tag{5.112}$$

$$[\![l = l', s]\!] \rightsquigarrow s \tag{5.113}$$

$$[\![l.a, s]\!] \rightsquigarrow s \tag{5.114}$$

$$\frac{\begin{bmatrix} \langle l := e, s\rangle \rightsquigarrow s' \\ [\![e', s']\!] \rightsquigarrow s'' \end{bmatrix}}{[\![l := e; e', s]\!] \rightsquigarrow s''} \tag{5.115}$$

$$\frac{\begin{bmatrix} f \notin \mathit{OnceFunction} \\ B = \mathsf{body}(\mathsf{version}(\mathsf{typeof}(s_E(l)))(f)) \\ E = \langle \texttt{Current} := s_E(l), arg^1 := s_E(l^1), \ldots, arg^n := s_E(l^n)\rangle \\ \langle B, (s_H, E, s_G)\rangle \rightsquigarrow s' \end{bmatrix}}{[\![l.f(l^1, \ldots, l^n), s]\!] \rightsquigarrow (s'_H, s_E, s'_G)} \tag{5.116}$$

$$\frac{\left[\begin{array}{c} f \in \mathit{OnceFunction} \\ \neg\mathsf{stored}(s_G, f) \\ B = \mathsf{body}(\mathsf{version}(\mathsf{typeof}(s_E(l)))(f)) \\ E = \langle \texttt{Current} := s_E(l), arg^1 := s_E(l^1), \ldots, arg^n := s_E(l^n)\rangle \\ \langle B, (s_H, E, s_G)\rangle \rightsquigarrow s' \end{array}\right]}{[\![l.f(l^1, \ldots, l^n), s]\!] \rightsquigarrow (s'_H, s_E, s'_G\langle f := s'_E(\texttt{Result})\rangle)} \tag{5.117}$$

$$\frac{\left[\begin{array}{c} f \in \mathit{OnceFunction} \\ \mathsf{stored}(s_G, f) \end{array}\right]}{[\![l.f(l^1, \ldots, l^n), s]\!] \rightsquigarrow (s_H, s_E, s_G)} \tag{5.118}$$

$$\frac{\left[\begin{array}{c} E = \langle \texttt{Current} := \mathsf{new}(s), arg^1 := s_E(l^1), \ldots, arg^n := s_E(l^n)\rangle \\ \langle \mathsf{body}(c), (s_H\langle \mathsf{defIn}(c)\rangle, E, s_G)\rangle \rightsquigarrow s' \end{array}\right]}{[\![\texttt{create}\, c(l^1, \ldots, l^n), s]\!] \rightsquigarrow (s'_H, s_E, s'_G)} \tag{5.119}$$

$$[\![mc, s]\!] \rightsquigarrow s \tag{5.120}$$

To support weak purity (see section 5.7), the operational semantics has to support side-effects in the evaluation of expressions. The $[\![\_, \_]\!] \rightsquigarrow \_$ relation

Reading local entities causes no side-effects (5.112). Neither does comparing two locals (5.113) nor reading an attribute (5.114).

Assigning to a local variable requires the evaluation of the expression, together with the side-effect of doing the actual assignments (defined by the rules for instruction evaluation) (5.115).

The side-effect of the function call (5.116) is defined in a similar way to the computation of the result value. The resulting state is taken from the heap and global state of the post-state of the execution, while the environment remains unchanged. I a similar way, the execution of a creation routine (5.119) is defined by executing the body (there is no need for dynamic binding here, as the target type is fixed) of the creation routine on the new object.

The first execution of a once routine stores the computed value in the global state (5.117) after execution. If there is already a value available, then the this feature invocation does not have a side-effect, as just the precomputed value is returned (5.118).

The precise definition of *ManifestConstant* is defined by the semantics of the underlying type of that manifest constant. We consider manifest constants to be of types `BOOLEAN`, `INTEGER` or `STRING`. All of these type are axiomatic classes as introduced in section 3.6. The creation of a new instance of an axiomatic class causes no side-effects on our state.

5.5.3 *Executing instructions*

$$\langle \epsilon, s \rangle \rightsquigarrow s \tag{5.121}$$

$$\frac{\left[\begin{array}{c} \langle S^1, s \rangle \rightsquigarrow s' \\ \langle S^2, s' \rangle \rightsquigarrow s'' \end{array}\right]}{\langle S^1 S^2, s \rangle \rightsquigarrow s''} \tag{5.122}$$

$$\frac{[\![E, s]\!] \rightsquigarrow s'}{\langle l := E, s \rangle \rightsquigarrow (s'_H, s'_E \langle l := \mathbf{eval}(E, s) \rangle, s'_G)} \tag{5.123}$$

$$\langle a := l, s \rangle \rightsquigarrow (s_H \langle s_E(\texttt{Current}).a := s_E(l) \rangle, s_E, s_G) \tag{5.124}$$

$$\frac{\left[\begin{array}{c} s_E(C) = \text{TRUE} \\ \langle S1, s \rangle \rightsquigarrow s' \end{array}\right]}{\langle \texttt{if}\, C\, \texttt{then}\, S1\, \texttt{else}\, S2\, \texttt{end}, s \rangle \rightsquigarrow s'} \tag{5.125}$$

$$\frac{\left[\begin{array}{c} s_E(C) = \text{FALSE} \\ \langle S2, s \rangle \rightsquigarrow s' \end{array}\right]}{\langle \texttt{if}\, C\, \texttt{then}\, S1\, \texttt{else}\, S2\, \texttt{end}, s \rangle \rightsquigarrow s''} \tag{5.126}$$

$$\frac{\left[\begin{array}{c} s_E(C) = \text{FALSE} \\ \langle S, s \rangle \rightsquigarrow s' \\ \langle\, \texttt{from}\, \texttt{until}\, C\, \texttt{invariant}\, I\, \texttt{variant}\, V\, \texttt{until}\, S\, \texttt{end}, s' \rangle \rightsquigarrow s'' \end{array}\right]}{\langle\, \texttt{from}\, \texttt{until}\, C\, \texttt{invariant}\, I\, \texttt{variant}\, V\, \texttt{until}\, S\, \texttt{end}, s \rangle \rightsquigarrow s''} \tag{5.127}$$

$$\frac{\left[\, s_E(C) = \text{TRUE} \right]}{\langle\, \texttt{from}\, \texttt{until}\, C\, \texttt{invariant}\, I\, \texttt{variant}\, V\, \texttt{until}\, S\, \texttt{end}, s \rangle \rightsquigarrow s} \tag{5.128}$$

$$\frac{\left[\begin{array}{c} B = \mathsf{body}(\mathsf{version}(\mathsf{typeof}(s_E(l)))(f)) \\ \langle B, (s_H, \langle \texttt{Current} := s_E(l), arg^1 := s_E(a^1), \ldots, arg^n := s_E(a^n) \rangle, s_G) \rangle \rightsquigarrow s' \end{array}\right]}{\langle l.f(a^1, \ldots, a^n), s^0 \rangle \rightsquigarrow (s'_H, s_E, s'_G)} \tag{5.129}$$

$$\langle \texttt{check}\, l\, \texttt{end}, s\rangle \rightsquigarrow s \tag{5.130}$$

The empty operation ϵ (skip) does not change the state (5.121). Sequential composition uses an intermediate state s' to evaluate two commands in a row(5.122).

Local assignments consider the side-effect to evaluate the expression, then update the local variable in the environment (5.123). Assignments to attribute also evaluate the expression, considering its side-effect, and then assign the result to the target field of the current object (5.124).

We have two axioms for the case-distinction:: The first one (5.125) can be applied if the condition evaluates to **`true`**, executing the **`then`** branch in this case. The second one (5.126) can be applied if the condition evaluates to **`false`** and executes the **`else`** branch.

Similarly, loops also have two rules: One for the execution of the loop body by unfolding the loop once (5.127), and one for the termination of the loop (5.128).

Calling commands (5.129) is similar to calling functions (5.110), only that the result value is ignored. Check instruction are basically a skip (5.130).

5.6 Monotonic properties of the state

This section lists a number of theorem about the state and its execution.

5.6.1 Object allocation

Once an object has been allocated, it never becomes unallocated by the execution of an instruction (5.131) or an expression (5.132).

$$\begin{aligned} &\forall I \in \mathit{Instr}, s, s' \in \mathit{State}, o \in \mathit{Obj} : e \\ &\langle I, s\rangle \rightsquigarrow s' \wedge \mathsf{alloc}(s_H, o) \Rightarrow \mathsf{alloc}(s'_H, o) \end{aligned} \tag{5.131}$$

$$\begin{aligned} &\forall E \in \mathit{Expr}, s, s' \in \mathit{State}, o \in \mathit{Obj} : \\ &[\![E, s]\!] \rightsquigarrow s' \wedge \mathsf{alloc}(s_H, o) \Rightarrow \mathsf{alloc}(s'_H, o) \end{aligned} \tag{5.132}$$

Proof outline: All semantic rules compute the post-state heap by modifying the pre-state heap, using the operations defined in the abstract data type. The abstract data type definition for heaps does not contain a function that deallocates an object. We ignore memory size and garbage collection.

5.6.2 Once setting

If a once function has been evaluated, the execution of Eiffel will never overwrite the created value or unset it ((5.133) and (5.134)).

$$\begin{aligned} &\forall I \in Instr, s, s' \in State, f \in OnceFunction : \\ &(\langle I, s\rangle \rightsquigarrow s' \wedge \mathsf{stored}(s_G, o)) \Rightarrow (\mathsf{stored}(s'_G, o) \wedge s_G(f) = s'_G(f)) \end{aligned} \tag{5.133}$$

$$\begin{aligned} &\forall E \in Expr, s, s' \in State, f \in OnceFunction : \\ &(\llbracket E, s\rrbracket \rightsquigarrow s' \wedge \mathsf{stored}(s_G, o)) \Rightarrow (\mathsf{stored}(s'_G, o) \wedge s_G(f) = s'_G(f)) \end{aligned} \tag{5.134}$$

Justification: All semantic rules compute post-state global state from pre-state global state, using the operations defined in the abstract data type. The abstract data type definition for global state does not contain a function that "unsets" a once value.

5.7 Weak purity

While queries may cause side-effects, Eiffel demands that these side-effects are not visible: they are not allowed to create an effect onto the execution of the program other than their return value. This property is called *Command/Query Separation* in OOSC2. It is summarized by the statement that "asking a question should not change the answer" [52, pp. 748].

The chapter on side-effects in OOSC2 differentiates between *concrete side-effects* and *abstract side-effects*. Concrete side-effects are created by assignments to attributes or calls of commands [52, p. 757]. Abstract side-effects are "concrete side effects that can change the value of a non-secret query".

There are a number of obstacles that make it difficult to transform these definitions into theorems of the operational semantics:

- The definition of concrete side-effects only talks informally about memory allocation. More specifically, it ignores the problem of side-effects created by the memory allocation itself, and by calling commands on these newly created objects.
- The definition of abstract side-effects is difficult to formalize, as it talks about all future evaluations of arbitrary queries and their results.
- There are a number of assumptions about references in Eiffel and the memory model: The only operation available on references is

comparison. There is no way to find out which object was created first, or how many objects were created during the execution of a routine. These assumptions are never stated explicitly, but instead are implied by the full language definition. The assumptions are required, otherwise pure object creation or function calls could lead to observable side-effects.

A number of theorems describe our requirements that queries be side-effect free. The non-observable object creation is captured theorem (5.107) and (5.119): the evaluation of a query, its side-effect or the effect of an instruction is effected by the creation of an object.

Furthermore, there is the problem of weak purity [16]: even with full command-query separation, query can have side-effects. A query is allowed to create new objects, and arbitrarily change the state of these objects. These are state changes, although they are not visible from the caller. To allow such changes, we define *purity* as a property of the evaluation of expressions and creation routines as relative to the objects that existed before the execution.

Definition 5.1: Pure query

A query $q \in Query$ is called *pure* if fields belonging to objects that were allocated before the evaluation of the query have the same value after the evaluation of the query:

$$\begin{gathered} \forall s, s' : State, o \in Obj, a \in Attribute | \\ [\![q, s]\!] \rightsquigarrow s' \wedge \mathsf{alloc}(o, s_H) \\ \Rightarrow s_H(o.a) = s'_H(o.a) \end{gathered} \tag{5.135}$$

Definition 5.2: Pure creation routine

A creator $c \in Creator$ is called *pure* if fields belonging to objects that were allocated before the execution of the creator, with the exception of **`Current`**, have the same value after the evaluation of the query:

$$\begin{gathered} \forall s, s' : State, o \in Obj, a \in Attribute | \\ [\![q, s]\!] \rightsquigarrow s' \wedge \mathsf{alloc}(o, s_H) \wedge o \neq s_E(\texttt{Current}) \\ \Rightarrow s_H(o.a) = s'_H(o.a) \end{gathered} \tag{5.136}$$

With regards to correctness, all queries and creators are required to be pure. Any expression evaluation containing pure queries will also be pure:

$$\begin{gathered}\forall E \in Expr, s, s' : State, o \in Obj, a \in Attribute| \\ [\![E, s]\!] \rightsquigarrow s' \wedge o \in \mathsf{alloc}(o, s_H) \\ \Rightarrow s_H(o.a) = s'_(o.a)\end{gathered} \tag{5.137}$$

Proof outline: The only subexpression in the syntax of expressions that modifies the heap is the invocation of a query and the create expression. Both subexpressions have to be pure, thus the whole expression is pure.

To simplify matters, we will also assume that the evaluation of once queries for the first time, or for in future invocations, does not change the result value of expressions. This simplification is acceptable, understanding that **once** is primarily used to model singletons.

$$\forall E \in Expr, s : State|ExprEs = ExprE(s_H, s_E, s'_G) \tag{5.138}$$

5.8 Correctness

To be called correct, an implementation needs to satisfy a number of properties. When verifying code, these properties become proof obligation. It, using mathematical reasoning, we can show that a piece of code satisfies all proof obligations, then we call the code to be *functionally correct*.

Every proof obligation is associated with a component, a class. If all proof obligations associated with a class text have been proved, the class is correct.

Postconditions: Every postcondition has to evaluate to true. We can assume the precondition and invariant.

$$\begin{gathered}\forall f \in Feature \\ \mathbf{eval}(\mathsf{pre}(f), s) = \mathrm{TRUE} \quad \wedge \\ \mathbf{eval}(\mathsf{inv}(\mathsf{defIn}(f)), s) = \mathrm{TRUE} \quad \wedge \\ \langle \mathsf{body}(f), s \rangle \rightsquigarrow s' \quad \Rightarrow \\ \mathbf{eval}(\mathsf{post}(f), s') = \mathrm{TRUE}\end{gathered} \tag{5.139}$$

Invariants: The class invariant has to be re-established at the end of a feature execution.

$$\begin{gathered}\forall f \in \mathit{Feature} \\ \mathbf{eval}(\mathsf{pre}(f), s) = \mathrm{TRUE} \quad \wedge \\ \mathbf{eval}(\mathsf{inv}(\mathsf{defIn}(f)), s) = \mathrm{TRUE} \quad \wedge \\ \langle \mathsf{body}(f), s \rangle \rightsquigarrow s' \quad \Rightarrow \\ \mathbf{eval}(\mathsf{inv}(\mathsf{defIn}(f)), s') = \mathrm{TRUE}\end{gathered} \tag{5.140}$$

Other than these two proof obligations, there are also a number of proof obligations that need to be satisfied for some syntactical constructs.

Let s be a state under which a given instruction I or expression E is evaluated. s must be a legal state of the execution of the routine, assuming that the precondition and class invariant held at the start of the routine.

Void calls: Targets of calls may never be void.

$$E = l.f(l^1, \ldots, l^n) \quad \Rightarrow \quad s_E(l) \neq \texttt{Void} \tag{5.141}$$

$$I = l.f(l^1, \ldots, l^n) \quad \Rightarrow \quad s_E(l) \neq \texttt{Void} \tag{5.142}$$

Precondition holds: The precondition of the feature called has to be established.

$$\begin{gathered}E = l.f(l^1, \ldots, l^n) \quad \Rightarrow \\ \mathbf{eval}(\mathsf{pre}(f), (s_H, \langle \texttt{Current} := s_E(l), arg^1 := s_E(l^1), \ldots, arg^n := s_E(l^n) \rangle, \\ s_G)) = \mathrm{TRUE}\end{gathered} \tag{5.143}$$

$$\begin{gathered}I = l.f(l^1, \ldots, l^n) \quad \Rightarrow \\ \mathbf{eval}(\mathsf{pre}(f), (s_H, \langle \texttt{Current} := s_E(l), arg^1 := s_E(l^1), \ldots, arg^n := s_E(l^n) \rangle, \\ s_G)) = \mathrm{TRUE}\end{gathered} \tag{5.144}$$

Check instructions: Check instructions must evaluate to **`true`**.

$$I = \textbf{check}\ \mathrm{C}\ \textbf{end} \quad \Rightarrow \quad \mathbf{eval}(C, s) = \mathrm{TRUE} \tag{5.145}$$

Loop variants and invariants: The loop invariant must be established at the beginning of the loop and at the end of the loop. The variant must be positive and decreasing.

$$
\begin{aligned}
&I = \textbf{from until } \mathrm{C} \textbf{ invariant } \mathrm{I} \textbf{ variant } \mathrm{V} \textbf{ loop } \mathrm{B} \textbf{ end} \\
&\langle \textbf{from until } \mathrm{C} \textbf{ invariant } \mathrm{I} \textbf{ variant } \mathrm{V} \textbf{ loop } \mathrm{B} \textbf{ end}, s\rangle \rightsquigarrow s' \quad \Rightarrow \\
&\qquad \mathbf{eval}(I, s) = \mathrm{TRUE} \qquad (5.146) \\
&\qquad \mathbf{eval}(V, s) \geq 0 \qquad (5.147) \\
&\qquad \mathbf{eval}(I, s') = \mathrm{TRUE} \qquad (5.148) \\
&\qquad \mathbf{eval}(V, s') \geq 0 \qquad (5.149) \\
&\qquad \mathbf{eval}(V, s) > \mathbf{eval}(V, s') \qquad (5.150)
\end{aligned}
$$

Loop variant and invariant checking: The loop variant and invariant needs to be established after each iteration of the loop.
Purity: All queries and creators have to be pure, see (5.135) and (5.136).

CHAPTER 6

MODEL-BASED CONTRACTS

Understanding the concept of "object state is necessary to specify object-oriented components using Design by Contract. Preconditions describe requirements on the state of the target or arguments when a feature may be called. Postconditions relate the pre-state to the post-state of the execution. Invariants describe global constraints of the state of the object.

Only considering the fields of one object to define its state is not sufficient: many objects rely on numerous "supporting objects" that participate in defining the object's state. For example, a list object (instance of `LIST[G]`) relies on a number of cell objects (instances of `LINKABLE[G]`); a database row might load contents from the database *on demand* from the external source.

We cannot, at the other extreme, consider all objects reachable from the object of interest: a person does not change its state when the company for which the person is working hires another person, although the person might have a reference to the employer and the employer object has a list of employees. Considering everything reachable to be part of the state makes reasoning non-modular.

Furthermore, we might not know the dynamic type of the object: reasoning happens on the basis of the static type of the entity. If the static type is based on a deferred class, we do not have sufficient knowledge about the fields of that class.

The solution used in this work and explained in this chapter is to use *models* to describe the state of objects.

A model is a high-level abstraction of the state of an object. The model itself is not an object structure, but a mathematical value. It is immutable in the same way that the number 5 or the set $\{1, 2, 3\}$ are immutable: evaluating $1 + 5$ yields the value 6, but does not change the number 5. Adding

a value 7 to the set $\{1, 2, 3\}$ will give the set $\{1, 2, 3, 7\}$, but does not change the original set.

At any time, objects can be queried for their model. Contractual obligations are expressed by stating properties of these models. The query that yields the state is called the *model query*. The model query is an abstraction from the concrete state of the heap to a mathematical value representing the state of a single object.

6.1 Related work

An early article that suggests the use of models to capture state abstractions was published by Hoare in 1972[34].

ESC/Java [28] has a special construct `dttfsa` (Damn the Torpedoes, Full Speed Ahead) that translates expressions directly into proof-obligations. Using such constructs, it is possible to create libraries whose semantics is defined in terms of the underlying theorem prover.

The Caduceus [27] tool for the verification of C programs follows a similar approach, allowing define properties directly in terms of the underlying Coq verification system.

Both approaches depend on the underlying theorem prover. Models as presented in this thesis are more abstract: our goal is not only the verification of software with a specific verification tools or environment, but the extension of full Design by Contract as a whole.

Darvas and Müller [17] suggest the introduction of a general `mapped_to` clause to the contract language of JML and Jive that can be used to relate classes and methods to types and functions in an underlying theorem prover. Here, it is possible to target different provers by explicitly stating which prover is targeted in the code.

Meyer describes the approach in OOSC2 [52, pp. 400-402]. He suggests to define a strong relation between the programming language and mathematics by the development of a library called IFL [53].

Mitchell and McKim [58] suggested to introduce an immutable list data structure into Eiffel, but used extensive recursive contracts.

Earlier results of this work have been published by in [84] and [77]. Applications of the library of models have been conducted by Zietzling in his semester thesis[85].

Further references are contained in the comparison in section 6.6.

6.2 Mathematical language

We may express properties of the state of an object through the object's model. We have to introduce a mathematical theory describing models to relate and compare them. Together with the theory we have to define a language to express the properties.

The theory should be powerful enough to express the properties we need. At the same time, it should be simple enough to allow fully automated proofs of programs.

The B Method [1] is a formal method used for top-down software development. It was defined in 1996 by J.-R. Abrial. The goal was to develop a methodology for program development that makes it feasible to prove the functional correctness of large software systems, without suffering from problems of combinatorial explosion.

Together with the B methodology comes a mathematical theory and notation used to express invariants, specifications and events. This mathematical theory is based on first-order predicate logic and Zermelo-Fraenkel set theory (with the axiom of choice)[29]. In fact, it is a limited form of ZFC, as it drops a number of axioms that are not required [1, p. 56], including the *foundation axiom*, which makes ZFC difficult to use for many non-mathematicians. Instead, B requires of a type-checker to exclude non-meaningful axioms.

The notation is mostly a subset of the Z specification language [79][80]. It uses standard operators for predicates and sets like $\Rightarrow$, $\wedge$, $\in$, $\subseteq$ or $\cap$. Some of the operators are less commonly used, for example the description of a partial injection using $\rightarrowtail$, or domain subtraction of a relation using $\lhd$. A table of operators is included in appendix A.

As the basis for our mathematical modelling language, we have chosen the mathematical language introduced by the B method.

6.3 Finite and typed models

Although we consider the notation and selection of operators of B to form the basis of our mathematical theory of models, the mathematical foundation is changed. The theory of models is *typed set theory on finite sets*. The change from ZFC to typed set theory is a natural one: the Eiffel programming language itself is a strongly typed language.

First, we will never encounter objects whose static type is unknown. Typed set theory is easy to check. More importantly, its axioms are easy to

understand for a developer who is already used to work with a strongly typed programming language.

Second, we consider states of objects to be always finite values. At a first glance, this seems obvious as the state of the system itself is always finite, bound by the size of the heap. The following example illustrates why it would still be nice to have a theory supporting infinite models:

The class *RANDOM* implements a random number generator. It inherits from *COUNTABLE_SEQUENCE* [*INTEGER*]. The mathematical model of a random number generator is thus a sequence of values, as the model is inherited, too. Assuming that we are not working with a fixed seed and create finite circle of pseudo-random numbers, this sequence is potentially infinite.

6.4 MML: a mathematical model library

The *Mathematical Model Library* is an implementation of set theory in terms of a programming language. We reuse the capabilities of the programming language to express mathematical expressions.

The library is available for current versions of Eiffel from the URL `http://se.inf.ethz.ch/people/schoeller/mml.html` under the permissive *Eiffel Forum License, version 2* (an OSI-approved license similar to the MIT or BSD license).

We have already mentioned that mathematical values are considered immutable. All mathematical operations yield new values, but do not change the existing ones. In contrast, objects in Eiffel are mutable: the state of the object can change by changing the values of object fields. When implementing mathematical values in terms of Eiffel classes, the objects have to be immutable: they do not change their value after creation. Assuming command/query separation, this results in the *model library principle*:

Definition 6.1: Model Library Principle

Model classes may not have commands. Queries in a model class may only rely on queries of the class itself and public queries of other model classes. Model objects are never compared by reference.

The model library principle is a refinement of the definition in [53]. The first restriction makes it impossible to change fields of the object, as only commands can change fields. The second restriction enforces that queries in model classes cannot take some mutable part of the state outside of the

model into account when computing a result value. The third restriction ensures that the object identity has no effect on comparisons. Together, they enforce that the class is *immutable* and behaves as a *mathematical value*.

There are two advantages of describing models using known concepts of the programming language instead of a language extension or using some kind of specific modelling language:

- Developers do not need to learn new notations. All reference documentation of the language remains the same.
- The tools available for the language automatically extend onto the new notation.

6.5 Library design

A first version of the library was developed by Tobias Widmer as part of his master thesis[84]. The goal of the second library design was to reuse as much as possible the possibilities of the Eiffel type-checker to implement the type-checker for typed set theory.

This resulted in a cleaned up hierarchy, shown in the BON [83][82] diagram in figure 6.1. The diagram does not show the details of the inheritance relation between classes, which are as follows:

- `MML_RELATION[G, H]` is a subtype of `MML_SET[MML_PAIR[G, H]]`, defining a relation to be a set of pairs.
- `MML_ENDORELATION[G]` is a subtype of `MML_RELATION[G, G]`, to offer specific functions that only make sense on relations where the domain and range type are equivalent (for example to compute closures).
- `MML_GRAPH[G]` is a subtype of `MML_PAIR[MML_SET[G], MML_ENDORELATION[G]]`, defining the set of vertices and edges of the graph.
- `MML_BAG[G]` is a subtype of `MML_RELATION[G, INTEGER]`. The bag is a function that relates a given element to the number of occurrences that the element has in the bag.
- `MML_POWERSET[G]` is a subtype of `MML_SET[MML_SET[G]]`. Powersets defined operations like generalized union and intersection.

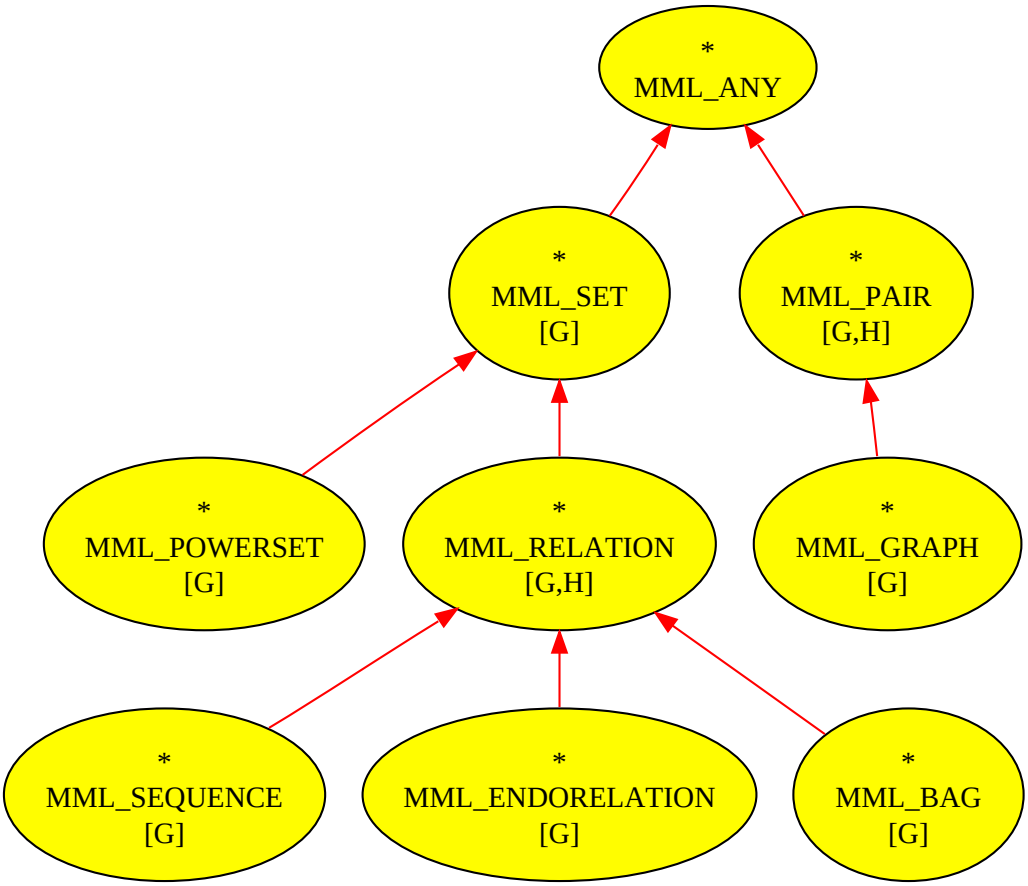

Figure 6.1: Class hierarchy of MML

- *MML_SEQUENCE[G]* is a subtype of *MML_RELATION[INTEGER,G]*. A sequence is a function of the prefix of natural numbers to values stored in the sequence.

The subtype relation has been chosen carefully to be aligned with the definitions of the different types given in the B book. [1] This makes it use each class polymorphically, the same way that we are used from mathematics.

Although the graph class *MML_GRAPH* is included in the library, it has not been fully explored and implemented, and is thus excluded from some of the following discussions.

The classes also try to retain type information by covariant redefinition of return types whenever possible. For example, in *MML_SET[G]* the signature of the feature *extended* is

```
extended (v:G): MML_SET[G]
```

In class *MML_RELATION[G]*, the feature is redefined from the inherited signature

```
extended (v:MML_PAIR[G,H]): MML_SET[MML_PAIR[G,H]]
```

which would follow the inheritance clause, to the signature

```
extended (v:MML_PAIR[G,H]): MML_RELATION[G]
```

as every the resulting value of adding a pair to a relation is still a relation.

In class *MML_SEQUENCE[G]*, a similar redefinition is not allowed, as adding an arbitrary pair to a sequence might break the property of sequences that the domain is a finite prefix of the natural numbers. *extended* in *MML_SEQUENCE[G]* has the same signature as in *MML_RELATION[G,H]*.

6.6 Comparison of model libraries

The Java Modelling Language [10][43] offers a library of models as available as part of the of the `org.jmlspec.models` package [42].

Because of the complexity of the Java object model, the JML model library contains a total of 465 classes. Many of these classes express relations between value and reference types.

The JML models are not axiomatic, their semantics is still defined by their contracts with all the related problems as shown in section 3.9. The JML model library is not derived from a coherent mathematical modelling language.

In contrast, MML models are axiomatic. their semantic definition is defined purely by the underlying theory, using the vehicle of a library to offer a shallow embedding of the underlying theory.

The ESpec workbench[67] for Eiffel defines a model library called ML. The ML library was developed in parallel with MML, addressing similar goals.

The ESpec verifier transforms Eiffel programs into code for the "Perfect Developer" verification system. ML offers a shallow embedding of the data types available in Perfect Developer in form of a library. Similar to MML, ML classes are thus implicitly defined by the semantics of the underlying data types in Perfect Developer.

The data types from Perfect Developer do not form a consistent set theoretical framework. Instead, they are a collection of unrelated types for bags, sets and sequences. This makes ML-based difficult to port to other theorem provers.

All three libraries, MML, ML and jmlspec, offer an implementation that can be monitored at runtime.

6.7 Applying models

Feature `put` from class `STACK[G]` of EiffelBase describing the abstract notion of stack is a typical feature exhibiting incomplete postconditions. It implements the "push" operation on stacks (the name `put` is a result of the strict consistency policy of Eiffel libraries [47, 51]). The "flat" form taking inheritance of assertions into account is shown in listing 6.1.

The query `item` yields the top of the stack, and the query `count` its number of items. Remove is the implementation of "pop", removing the top element of the stack.

The precondition is complete: if the stack is not empty, you may always remove an element from it or ask for the top element. The postconditions, however, are not: they only refer to the number of items and the top item after the operation, but do not say what happens to the items already present. As a result:

- It leaves some questions unanswered. For example, what will get printed by the code of listing 6.2, whereas the corresponding abstract data type specification [52] is sufficient to compute the corresponding mathematical expression: *item (remove (put (put (new, 47), 11)))*.

- 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.

```
item: G is
   -- Top element
  require
    not_empty: not is_empty
  do
    ...
  end

put (v: G) is
   -- Push 'v' onto top.
  do
    ...
  ensure
    item_on_top: item = v
    count_increased: count = old count + 1
  end

remove is
   -- Remove the top of the stack.
  require
    not_empty: not is_empty
  do
    ...
  ensure
    count_decreased: count = old count + 1
  end
```

Listing 6.1: Contracts of *STACK* without models

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

With models from MML, we now have the possibility to give a full (in terms of the data abstraction) contract of the behavior of a stack. First, a model query needs to make the state of a stack explicit. The mathematical abstraction of a stack is a sequence of numbers, with the new stack elements added to the end (they could also be added to the front, which would be a different state abstraction). The model query is defined with the following signature:

```
feature -- Model queries
  model: MML_SEQUENCE [G]
```

```
create stack.make_empty
stack.put (47)
stack.put (11)
stack.remove
print (stack.item)
```

Listing 6.2: Incomplete contracts exposed in *STACK*

The contract of *put*, *remove* and *item* is then extended by references to the behavior on the model, as shown in listing 6.3. Using the mathematical definitions of the operations (see appendix B), we can rewrite the contracts into their corresponding mathematical forms. This is shown in listing 6.4.

With the added model contracts, it is possible to understand the example given in listing 6.2. The proof sketch, using Hoare style intermediate state annotations, is shown in listing 6.5. It shows that the print command will output the number 47.

This small example illustrates how models can be used to improve the expressiveness of contracts for unbounded data types.

6.8 Models and inheritance

Models work well together with the concept of inheritance. This is done by projecting the model of the child onto the model of the parent. A binding invariant expresses this projection, making it possible to understand the child model in terms of the parent model. Because of renaming, we are able to give the inherited model an expressive name.

For example, we might regard the width and the height as the model of a abstract geometrical *FIGURE*. It is defined as a pair of two *REAL* values:

```
deferred class FIGURE

feature -- Model queries

  model: MML_PAIR [REAL,REAL]
    -- Width and height of the figure

feature -- Tranformations

  scale (factor: REAL)
    require
      factor_positive: factor > 0
```

```
    ensure
      width_adjusted: width = old width * factor
      height_adjusted: height = old height * factor

end
```

A circle is a geometrical figure, but the model of the circle is its diameter. We define the class `CIRCLE` using the following inheritance relation, model query and binding invariant:

```
class CIRCLE

inherit
  FIGURE
    rename
      model as model_of_figure
    end

feature -- Model queries

  model: REAL
    -- Diameter of the circle

invariant
  model_binding_to_figure: model_of_figure.first = model
      and model_of_figure.second = model
end
```

The binding invariant makes it possible to understand the contractual obligations inherited from parent feature in terms of the child model. The `scale` command will have to adjust the diameter of the circle, following the inherited obligations of its postconditions and the class invariant.

6.9 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 (see figure 6.2).

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:

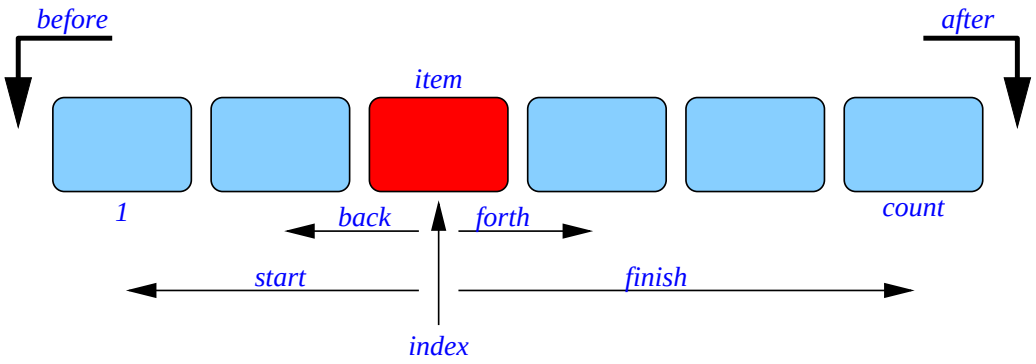

Figure 6.2: *LINKED_LIST* with active cursor

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

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

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

 model_sequence: MML_SEQUENCE [G] is
  do
    ...
  end
```

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

```
 model: MML_PAIR [MML_SEQUENCE [G],INTEGER] is
    -- Model of the list
  do
    create {MML_DEFAULT_PAIR [MML_SEQUENCE [G], INTEGER}
       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 func-

tion 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".

6.10 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 6.2: Soundness of a Model

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

1. The classic preconditions are equivalent to the model preconditions.
2. The model postconditions imply the classic postconditions.
3. The model invariant implies the 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. At the same time, the model precondition will give the full information about when the feature can be called.

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:

```
item_on_top: item = v
count_increased: count = old count + 1
model_changed: model.equals (old model.extended (v))
```

From the postconditions of *count* (not shown above) and *item*, we know that

```
count = model.cardinality
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.equals (old model.extended (v)) and
item = model.last implies
item = v

model.equals (old model.extended (v)) and
count = model.cardinality and
old count = old model.cardinality implies
count = old count + 1
```

Using the mathematical translations (see appendix B), we can create the proof obligations:

$$\begin{aligned}
\text{model}' = (\text{model} \cup (\text{card}\,\text{model} + 1, v) \land \\
\text{item}' = \text{model}'(\text{card}\,\mathit{model}') \Rightarrow \text{item}' = v \\
\text{model}' = (\text{model} \cup (\text{card}\,\text{model} + 1, v) \land \\
\text{count} = \text{card}\,\text{model} \land \\
\text{count}' = \text{card}\,\text{model}' \Rightarrow \text{count}' = \text{count}
\end{aligned}$$

Both properties can be verified using the underlying set theory. 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 extensively by Meyer [53].

6.11 Default implementation

The MML library contains a default implementation for models based on arrays as the main underlying data structures. The models are implemented in the following way:

- ***MML_PAIR*** is implemented as a class with two attributes, each containing the corresponding value. The implementing class is called *MML_DEFAULT_PAIR*.
- ***MML_SET*** is implemented in two ways. First, as an array containing the values stored in the set, called *MML_DEFAULT_SET*. Second, the class *MML_RANGE_SET* models the set of the numbers contained in the interval $a \ldots b$ by just storing the lower and upper bound of the interval. This implementation is used when we access the domain of a sequence.
- ***MML_RELATION*** is implemented in the class *MML_DEFAULT_RELATION* mostly by inherting the implementation of *MML_DEFAULT_SET* and relying on instances of *MML_DEFAULT_PAIR*, thus using an array of references to pairs.
- ***MML_BAG*** is implemented in *MML_DEFAULT_BAG* by inheritance from the implementation relation. A bag is thus an array of elements plus a number denoting the occurrences of the element in the bag.
- ***MML_ENDORELATION*** is implemented in *MML_DEFAULT_ENDORELATION* by inheriting the implementation from *MML_DEFAULT_RELATION*.
- ***MML_POWERSET*** is implemented in *MML_DEFAULT_POWERSET* by inheriting the implementation from *MML_DEFAULT_SET*.
- ***MML_SEQUENCE*** does not reuse the implementation of parents. Instead, it uses *ARRAY* directly to implement the sequence. This results in a large re-implementation of all features, projecting the *ARRAY* to the inherited queries, adapting the structures if necessary.

Although the implementation tries to be efficient, experiments show that the run-time monitoring of models has a major impact on the executing speed. Because models are immutable, most computations on models require heavy-weight copy operations, with an algorithmic complexity of $O(n)$. The computation of the intersection of two sets is in $O(n * m)$.

On a current machine, we have created lists using the following code:

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

Table 6.1: Time measurements of the test case with 1,000 elements.

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

Table 6.2: Time measurements of the test case with 100,000 elements.

```
create linked_list.make

from
  counter := 1
until
  counter > number_of_elements
loop
  linked_list.put_front (counter)
  counter := counter + 1
end
```

Tables 6.1 and 6.2 shows the results of messuring the performance impact of model-based contracts in EiffelBase. The table shows the performance of the code with and without assertion checking enabled for inserting 1,000 and 100,000 elements into the linked list.

There remains a great potential for optimizations, as all model structures are immutable and lazy evaluation and on-the-fly changes of the internal representation are possible. Hashing might also speed up search operations, but require that the values stored in the data structures are indeed *HASHABLE*.

The is also the possibility to perform runtime evaluation of model-based contracts only up to a certain size of the data structure. This would require the adoption of a three-valued boolean logic with the ability to express that the result was not known because the values were too large.

6.12 Implementing the abstraction function

The relationship between a concrete software object and its MML model is its "abstraction function" (a notion introduced in [34] in the form of the "representation function", its inverse, actually multi-valued).

If we want to enable runtime monitoring using the default implementation suggested above, we also have to provide an implementation of the abstraction function. For example, an abstraction function for the `ARRAYED_STACK[G]` class is implemented in listing 6.6.

The example illustrates that the implementation of models can (and very often have to) access parts the state that are not directly accessible using the public interface of the class. Models have to capture the full state of an object, and not only what is currently available through the public interface.

Model queries always return an attached (non-void) result in the sense of ISO/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.

6.13 Applying models to EiffelBase

The main target of our study the EiffeBase library[51][47][25]. EiffelBase is the ideal target of study:

- EiffelBase is an old and mature library.
- EiffelBase has been used extensively in commercial and non-commercial applications. Few other Eiffel library are applied on such a wide-spread domain of problems.
- The library has been carefully crafted, making heavy use of classifications and separating concerns and later merging these concepts into implementations, using multiple inheritance.
- The inheritance hierarchy is deep, with many intermediate classes only introducing one or two new concepts.
- Data structure libraries are typically good at exposing the problems adhered by the introduction of models.

Our work with EiffelBase can be split into two parts. In the first part of our research, we tried to add model-based contracts to EiffelBase. The goal of this endeavor was to see what the underlying abstraction of the EiffeBase classes are and to discover potential modelling defects in EiffelBase.

In the second part, we redesigned parts of the EiffelBase library with the knowledge about its deficits gained in the first part. We concentrate below on the hierarchy needed for `LINKED_LIST`, but as this class is very low in the inheritance hierarchy, this means that we have to redesign a large part of the library.

6.13.1 *Analyzing existing EiffeBase*

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. A full specification for each class was done by Widmer[84].

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 dis-

tinguished. The linearity is not necessary for an implementation of logic quantifiers.

A full list of problems discovered can be found in [84].

An interesting problem was discovered when trying to analyze the *put* operation through the inheritance tree, and its relation to the *extend* command and *extendible* query, describing the extension of the container by an element. We think that this kind of error is typical for the problems that can be discovered using models, so it is explained here in detail.

The class *COLLECTION* introduces the operations *put* and *extend* as the same feature:

```
extend (v: G)
   -- Ensure that structure includes 'v'.
  require
    extendible: extendible
  deferred
  ensure
    item_inserted: is_inserted (v)
  end
```

Feature *is_inserted* is an alias for *has*. The predicate *extendible* is introduced as an opaque precondition, modelling the fact that data structure can be full and reject the inclusion of a new object:

```
is_inserted (v: G): BOOLEAN is
   -- Has 'v' been inserted by the most recent insertion?
   -- (By default, the value returned is equivalent to
      calling
   -- 'has (v)'. However, descendants might be able to
      provide more
   -- efficient implementations.)
  do
    Result := has (v)
  end

extendible: BOOLEAN is
   -- May new items be added?
  deferred
  end
```

The aliases *put* and *extend* are split up into two distinct features in class *BAG*. The post-condition of *extend* is extended by the assertion that the number of occurrences of the argument *v* will be increased by *extend*, while *put* does not ensure this property:

```
extend (v: G) is
   -- Add a new occurrence of 'v'.
  deferred
  ensure then
    one_more_occurrence:
     occurrences (v) = old (occurrences (v)) + 1
  end
```

At this point, it is not clear what *put* on a bag will be. From the current contractual obligations, *put* might add v to the data structure if v is already contained. It has to make sure that v is contained (following the opaque definition of *is_inserted*). This suggests the definition of *put* to be similar to a set-like insert, that only adds v if it is not already contained. But because of the contract of *extend*, *SET* cannot inherit from *BAG*.

The main conceptual mistake of this model is introduced in the class *CHAIN*, an indirect subclass of *BAG* through the classes *ACTIVE* and *CURSOR_STRUCTURE*. *CHAIN* describes itself in the header comment as a "possibly circular sequences of items, without commitment to a particular representation". It has a very complex inheritance clause, inheriting twice from *SEQUENCE*.

CHAIN redefines *put* as an alias for the *replace* operation. This overwrites the "current item" (a notion introduced in class *ACTIVE*) of the structure with a new value:

```
put (v: like item) is
   -- Replace current item by 'v'.
   -- (Synonym for 'replace')
  do
    replace (v)
  ensure then
     same_count: count = old count
  end
```

Flattening all the inherited contracts for *put* reveals the problem. Although *put* does not add new elements to the data structure, it requires the data structure to be extendible:

```
put (v: like item)
   -- Replace current item by 'v'.
   -- (Synonym for replace)
  require -- from COLLECTION
    extendible: extendible
  do
    replace (v)
  ensure -- from COLLECTION
```

```
    item_inserted: is_inserted (v)
  ensure then
    same_count: count = old count
  end
```

This inherited contract makes is necessary to weaken the precondition of `put` in `FIXED_LIST`, an implementation of a list with a maximum number of elements:

```
put (v: like first) is
    -- Replace current item by 'v'.
    -- (Synonym for 'replace')
  require else
    True
  do
    replace (v)
  end
```

Other examples of features that are suffering from the imprecise use of `extendible` are features like `fill` defined in class `COLLECTION` that "fills with as many items of *other* as possible". Again, as `FIXED_LIST` is not extendible, this feature is useless here. Using export restrictions, `fill` is exported to `NONE` for `FIXED_LIST`, creating a potential CAT-call when fixed lists are regarded as any parent data structure.

The next section describes efforts to re-design EiffelBase, using the insights gained.

6.13.2 Designing a new EiffeBase

The goal of the re-design was to see the changes that are necessary to EiffelBase to add model contracts (and dynamic frame contracts, as discussed in chapter 7). The new library only implements everything needed for `LINKED_LIST`. This is sufficient as linked list uses many of the concepts introduced by abstract parent classes, covering nearly all of EiffelBase. A class diagram showing the different classes covered is shown in figure 6.3.

The new library is called CCEiffelBase, with CC standing for *complex contracts*. To prevent collision with existing data structures, all classes of the CCEiffelBase library have `CC_` prepended in front.

We introduce the features `extend`, `put` (and later `force_end`) as variations of the same operations on the model. The difference between the features is the level of "defensiveness":

- `extend` is defined as non-defensive of putting elements into the container. It adds an element to the collection, and will always fail if this

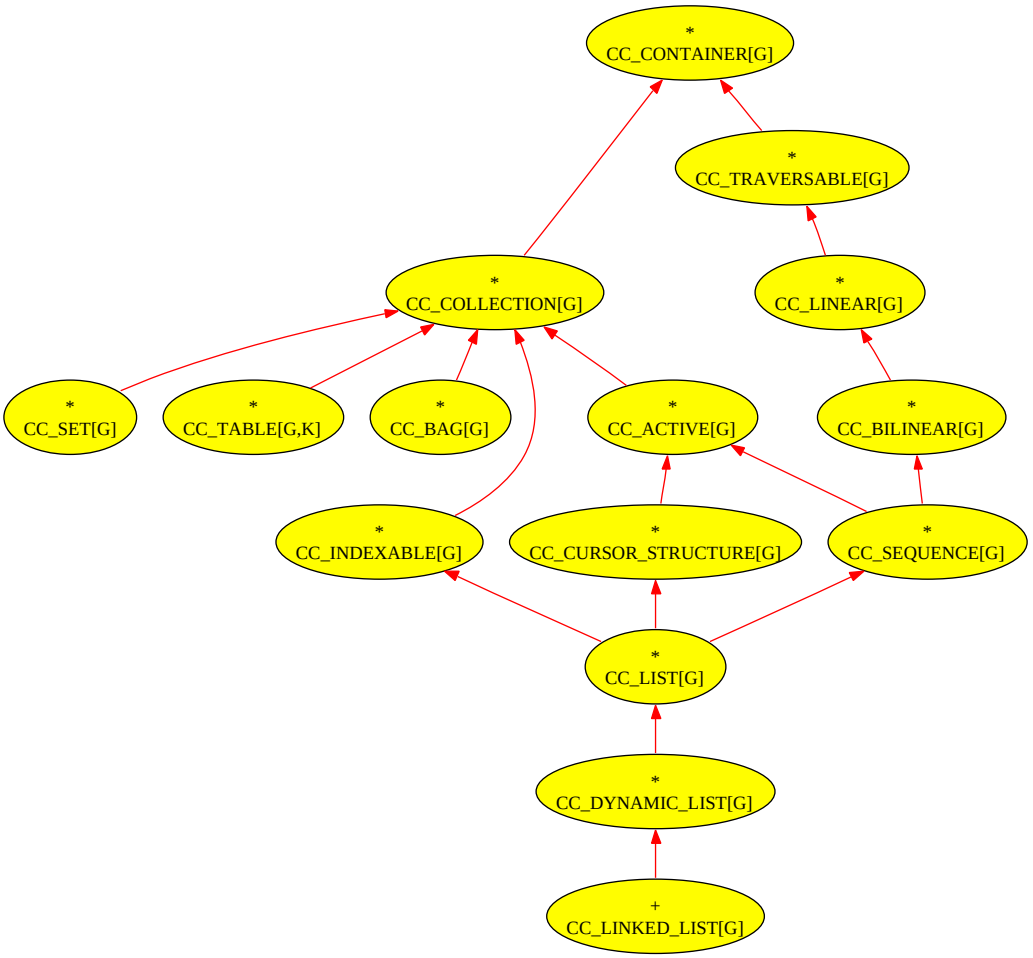

Figure 6.3: Class hierarchy of CCEiffelBase

is not possible (for example if the element is already contained in a set). It has the precondition *can_extend(v)*.

- *put* is defined as the medium-defensive version. It represents the underlying mathematical operation. For example, it will just not add the element to a set to prevent duplication or overwrite entries in a table. It has the precondition *can_put(v)*.

- *force_end* is the most defensive version. It will does not have any precondition, and will do its best to add the element. For example, it will resize arrays. It is introduced in *CC_SEQUENCE*.

The predicates *can_extend(v)* and *can_put(v)* replace the *extendible* query. The problem with *extendible* was that could not take the element to be added into account, making it unsuited for sets or tables.

The model of *CC_CONTAINER* is a bag of elements:

```
model: MML_BAG[G]
   -- Model of a general container
  ensure
    not_void: Result /= Void
```

This model is not changed in *CC_COLLECTION*. The contracts of *put* and *extend* using models are as follows:

```
put (v: G)
   -- Ensure that structure includes 'v'.
  require
    can_add_element: can_put (v)
  ensure
    model_updated: model |=| old model.extended (v)
```

```
extend (v: G)
   -- Ensure that structure includes 'v' iff 'v' can be
      added.
  require
    can_add_element: can_extend (v)
  ensure
    model_updated: model |=| old model.extended (v)
```

The model is changed at a number of points in the inheritance relation. This is always done by renaming the old definition to a name *model_X*, where *X* is the name of the class where the model was introduced, and the defining a new model. The two models are then related using the class invariant.

- *CC_SET*[*G*] defines the model to be *MML_SET*[*G*]:
  model.*equals*(*model_container*.*domain*)
- *CC_TABLE*[*G*,*K*] defines the model to be *MML_RELATION*[*K*,*G*]:
  model_container.*equals*(*model*.*range_bag*)
- *CC_INDEXABLE*[*G*] defines the model to be *MML_RELATION*[*INTEGER*,*G*]:
  model_container.*equals*(*model*.*range_bag*)
- *CC_TRAVERSABLE*[*G*] defines the model to be *MML_PAIR*[*MML_SEQUENCE*[*G*],*INTEGER*]:
  model_container.*equals*(*model*.*first*.*range_bag*)
- *CC_LIST*[*G*] has to merge multiple models. Its model is *MML_PAIR*[*MML_SEQUENCE*[*G*],*INTEGER*], but it redefines the model from *CC_INDEXABLE* to be equivalent to the sequence:
  model_indexable.*equals*(*model*.*first*)
  and it merges the *model_container* from all inherited data structures.

Following the inheritance relation and the model contracts, the *put* feature defined in *CC_COLLECTION* has the following contracts in *CC_LIST*:

```
put (v: G)
   -- Ensure that structure includes 'v'.
   -- (from CC_COLLECTION)
  require -- from CC_COLLECTION
   can_add_element: can_put (v)
  ensure -- from CC_COLLECTION
   model_updated: model_container |=| old model_container
      .extended (v)
   model_correspondends: model_container.contains (v)
   frame_confined: confined representation
  ensure then -- from CC_SEQUENCE
   new_count: count = old count + 1
```

The contractual obligations of *put* make it impossible to implement replace: the list, if seen as a bag, will not remove any element, but instead add one occurrence of v.

The solution to this problem is to use *rename* instead of *redefine*: the *put* feature in *CC_LIST* is a different feature from the *put* feature introduced in *CC_COLLECTION*. The inheritance clause renames the *put* as inherited from *CC_CURSOR_STRUCTURE* to *put_end*, to describe that v is added to the

end of the list. The new put feature (inherited from *CC_INDEXABLE*) has the following contract:

```
put (v: G) is
    -- Replace current item by 'v'.
  require
    not_off: not off
    model_precondition: model.first.domain.has(index)
  ensure
    item_replaced: item = v
    model_update: model.first.equals (old model.
      domain_restricted_by (index).extended_by_pair (index,
         v))
    model_cursor_not_moved: model.second = old model.
       second
```

There are numerous other details that were changed from the original EiffelBase by the introduction of models:

- Because of the constraints of the model library, all containers are finite. This made it possible to remove the *FINITE*, *UNBOUNDED* and *BOX*.

- *CC_TABLE* does not inherit from *CC_BAG*. The idea in the original EiffelBase library was that a table is a bag with the elements accessible by keys. Now the abstraction of the bag has already been moved to *CC_COLLECTION*. The bag is now a "real bag".

- In the same spirit, *CC_ACTIVE* is also not a heir of bag anymore. *CC_ACTIVE* introduced the concept of *item*, a current element contained in the structure that can be replaced or removed. There are bags that are not active, and active objects that are not bags.

- In EiffelBase *BILINEAR* inherits twice from *LINEAR*. This double inheritance relation does not make any sense, as all of the features introduced by the second inheritance relation are hidden through the *select* statement anyway.

- The *fill* feature has been discarded for the reasons mentioned in the last section.

- *Strict command-query* separation was introduced. This was not the case for a number of queries in the EiffelBase implementation, as they moved the cursor.

- *cursor* was renamed to *cursor_position*, as the object really only stores the cursor position (the index value), but cannot be used to directly access the elements of the container.

A final observation: an open question that was raised numerous times during discussions about active data structures was whether two lists that contained the same elements but had their cursor at different positions were considered as *equivalent* in the terms of *is_equal*. CCEiffelBase answers this question by introducing following postcondition for *is_equal* in *CC_LIST*:

```
feature -- Comparison
  is_equal (other: like Current)
    ensure
      Result = (model.first |=| other.model.first)
```

This expresses that two lists are equivalent if the underlying sequences are the same. The cursor position does not matter for comparing the value of two lists.

6.14 Summary

This chapter has motivated and introduced the concept of models for writing contracts in Design by Contract. Models are representations of mathematical values in the domain of a programming language.

We have introduced models by defining a model library. Each class and feature in the model library was related to some mathematical set or function. We have demonstrated how the proper use of genericity and subtyping can translate the types of the underlying theory into the type system of the programming language.

With the model library, it is possible to define the "state of an object" precisely by providing an abstraction function — here always called *model* — that yields a value of the mathematical domain. We have called this abstraction function the *model query*. Using the model query, it is possible to improve the contract for commands and queries, without the need to fall back to recursive contracts.

Models give a better understanding of the inheritance relation, introducing the concept of model redefinition (which manifests in the renaming of the inherited model and the introduction of a new model) and the definition of glueing invariants between inherited and actual models.

Models are an important addition to the Design by Contract language, and our experiments illustrate that the force the developer to have a better

understanding of his abstractions and object state. At the same time, they are necessary to make contracts strong enough for formal verification.

```
item: G is
   -- Top element
  require
    not_empty: not is_empty
    model_not_empty: not model.is_empty
  do
    ...
  ensure
    model_definition: Result = model.last
  end

put (v: G) is
   -- Push 'v' onto top.
  do
    ...
  ensure
    item_on_top: item = v
    count_increased: count = old count + 1
    model_changed: model.equals (old model.extended (v))
  end

remove is
   -- Remove the top of the stack.
  require
    not_empty: not is_empty
    model_not_empty: not model.is_empty
  do
    ...
  ensure
    count_decreased: count = old count + 1
    model_changed: model.equals (old model.front)
  end
```

Listing 6.3: Contracts of *STACK* using models

```
item: G is
    -- Top element
  require
    not_empty: not is_empty
    model_not_empty: model ≠ ∅
  do
    ...
  ensure
    model_definition: Result = model(card(model))
  end

put (v: G) is
    -- Push 'v' onto top.
  do
    ...
  ensure
    item_on_top: item = v
    count_increased: count = old count + 1
    model_changed: model' = model ∪ (card(model), v)
  end

remove is
    -- Remove the top of the stack.
  require
    not_empty: not is_empty
    model_not_empty: model ≠ ∅
  do
    ...
  ensure
    count_decreased: count = old count + 1
    model_changed: model' = {card(M)} ⩤ model
  end
```

Listing 6.4: Mathematical translations of model contracts in *STACK*

```
create stack.make_empty
{stack.model = ∅}
stack.put (47)
{stack.model = {(1, 47)}}
stack.put (11)
{stack.model = {(1, 47), (2, 11)}}
stack.remove
{stack.model = {(1, 47)}}
print (stack.item)
```

Listing 6.5: Proof *STACK*

```
feature{SPECIFICATION} -- Model queries

  model: MML_SEQUENCE [G] is
      -- Model of the stack
    local
      i: INTEGER
    do
      from
        i := a.lower
      until
        i > a.upper
      loop
        Result := Result.extended (a.i_th (i))
        i := i + 1
      end
    end
```

Listing 6.6: Implementation of the abstraction function for class *ARRAYED_STACK*.

CHAPTER 7

DYNAMIC FRAME CONTRACTS

In their seminal paper "Some Philosophical Problems from the Standpoint of Artificial Intelligence" from 1969 [46], McCarthy and Hayes gave an extensive study of problems and possible solutions that the artificial intelligence community faced at that time. This paper identified and coined the term "frame problem" as one of the open problem of formal specifications for real world phenomena. The frame problem was informally described as follows:

> [...] *in proving that one person could get into conversation with another, we were obliged to add the hypothesis that if a person has a telephone he still has it after looking up a number in the telephone book. If we had a number of actions to be performed in sequence we would have quite a number of conditions to write down that certain actions do not change the values of certain fluents. In fact with* n *actions and* m *fluents we might have to write down* mn *such conditions.* (McCarthy and Hayes [46, p. 30])

Applying more current software engineering terminology, we can translate situation as *state*, fluent as *state function* and action as a *state transformation function*, a fluent where the range is another state. Propositional fluents are *state properties*.

McCarthy's and Hayes's description of the frame problem is optimistic; they assume that there is indeed a well-known number n or actions and another well-known number m of fluents, such that it is possible to specify the full effect of an action on all fluents. Whether or not this information is known when writing down formalizations, and what to do if the information is not known, is the question covered by this chapter.

7.1 Closed world reasoning

One approach to transform the unbounded, unknown set of fluents into a bounded, well-known one is *closed world* reasoning [74]. In artificial intelligence, closing down the world means giving a *default* for all properties that have not been mentioned so far or cannot be deduced by the given knowledge base. For example, if the knowledge base contains "Fred has a sister Ginny.", and we ask if "Fred has a brother George?", we would deduce "No", as we would have mentioned the brothers of Fred if he had one in the underlying knowledge base.

Open-world reasoning does not allow such assumptions. We have never implied that George is a brother of Fred, nor have we implied the opposite. We can regard open world reasoning as closed world reasoning with a default value of "unknown" in ternary logic.

Formally, closed world reasoning means the extension of the knowledge base by a closed world assumption CWA. If a property holds on the knowledge base KB without the closed world assumption, is must also hold if we enrich the knowledge base with the closed world assumption:

$$\frac{KB \vdash P}{KB, CWA \vdash P}$$

The introduction of the *CWA* needs to be sound: it must not introduce contradictions into the knowledge base.

A typical example of closed world reasoning is implemented in Prolog's *negation as failure* policy [14], as demonstrated by the script of listing 7.1. Although we never add the knowledge that Fred is not a sibling of George, Prolog implies this information, as it cannot prove the opposite.

```
sibling(fred,ginny).
?- sibling(fred,george).
No
```

Listing 7.1: Prolog example of closed world reasoning

Applying closed world reasoning to the frame problem means to assume that all fluents that are not mentioned in the specification of an action will have a default new value, for example the same value as they had in the situation in which the action was applied. Thus, all fluents not mentioned in the specification are invariant under the action.

7.2 Frame problem in software development

The frame problem arises in software engineering as well as in artificial intelligence. It arises when we combine formal reasoning, information hiding and bottom-up software development.

Bottom-up software development means we want to be able to develop different components to fulfill certain tasks of the software. Later in the process, we combine these components into larger components to fulfill more complex task. The existing components become subcomponents of the new components. We repeat this process until the final component fulfills the overall requirements for the system under development. The top component is then the actual application.

The advantage of bottom-up development is the ability to reuse existing software in different contexts. It is possible to develop libraries of components independently, without specific knowledge of the context they will be used in later. Components from different sources can be combined without the developers of the original components knowing about this combination. Bottom-up development is a widespread approach to software development.

Information hiding [72] separates *interface* and *implementation*. The interface describes *what* a software component does, while the implementation has all the details of *how* the software component achieves its functionality. When using an existing component in a new component, only the (hopefully small) interface description needs to be consulted. This results in a separation of concerns and a reduction of complexity.

For formal verification, information hiding turns into *modular reasoning* [61]. Modular reasoning means that it should be possible to verify the correctness of a component only knowing the interfaces of the components it uses.

For the frame problem, we can see that it is not sufficient for the interface to talk about the pure functionality of a component. It is also necessary for the interface to carry information on how it interacts with other components when combined into an application. Trying to implement the example from McCarthy mentioned earlier, we get the code shown in listing 7.2.

Following McCarthy's reasoning, we should mention as part of the specification of the `lookup_number` in `TELEPHONE_BOOK` that looking up a number will not change the ownership of telephones, affecting the value returned by `telephone` or class `PERSON`.

This is contradicting with the rules of bottom-up development if `PERSON` and `TELEPHONE_BOOK` are developed independently. Only later, both classes

```
check
  person.telephone /= Void
end

-- We know that the person has a telephone at this point
-- during the execution.

telephone_book.lookup_number ("Smith")

-- Do we still know that we have a telephone?
-- (otherwise we get a Void call on the next line)

person.telephone.dial_number (telephone_book.last_number)
```

Listing 7.2: McCarthy telephone problem as code

are joined together into a single application. The interface specification of *PERSON* cannot talk about telephone books, nor can the interface specification of *TELEPHONE_BOOK* talk about persons and their telephones.

A solution to this dilemma is to adopt a meaningful default, that is to work under a closed world assumption as introduced by Reiter [74]. We develop specifications for software components to describe what is affected by a certain operation. Then, we assume that everything not mentioned by the specification remains unchanged.

The major engineering problem that needs to be solved is the combination of these specifications with the information hiding principles. Describing explicitly all the changes of a component would reveal *how* a component is implemented. This is a violation of information hiding, where the specification should only mention *what* a component does.

The next section contains the application of framing to the object-oriented model as introduced in chapter 5. We then give a short overview of existing approaches to the frame problem in existing verification system. We introduce the idea of dynamic frames as a new approach and discuss the advantages and disadvantages of the approach when compared with previous work. We integrate dynamic frames into Design by Contract by proposing "Dynamic Frame Contracts" and formalize these contracts. Finally, we apply dynamic frame contracts to real world examples and identify common contract patterns.

7.3 Related work

For program verification, significant work has been done to specify and enforce encapsulation of data. The most universal approach might be the definition of *separation logic* [75][71][66].

Many papers approach solving frame problems by preventing unwanted *leaking* of references within the program execution, guaranteeing the encapsulation of the data by preventing access [2][9][59][21]. The universe type systems is used in Spec#[4] to maintain class invariants that rely on foreign objects. All of these approaches is in common that they violate information hiding by lacking an approach to specify read effects. Also, they constrain the object structure.

Static *modify clauses* (in contrast to the modify clauses based on dynamic frames as presented in this thesis) are annotations added to the contract language to restrict the set of object fields [33][13][10].

Static modify clauses introduce problems with information hiding and subtyping. Data groups are a specification technique to counter these problems by introducing abstract variable groups and only later assign specific attributes to these groups[44].

Few papers are not only concerned with possible changes of the state, but also to specify which functions are effected by change. Greenhouse and Boyland coined the term read-effect[32] for this phenomenon.

VDM introduces the specification of read and write effects into its refinement calculus[7]. Reading and writing to variables is specified explicitly. A global frame can be used to enforce the independance of different refinements. As a refinements calculus with top-down developement and without dynamic storage allocations, the actual problems encountered are very different from the work presented here.

Early work on framing in Eiffel was done by Mitchell and McKim [58]. This work included only adding standard descriptions of the form x = **old** x to post-conditions, an approach which is not sufficient to overcome the problems of framing.

Dynamic frames as applied in this thesis have been invented by Kassios and extensively covered as part of his thesis[37] as well as a conference paper[36]. A summary of of Kassios' work is contained in section 7.5.

A comparison between Kassios' *Dynamic Frames* and *Dynamic Frame Contracts* as presented in this thesis is given in section 7.7.

7.4 Frames in the object-oriented programs

Section 5.4 introduced a model of the state of an object-oriented program. The state is define by three components: environment, heap and global state.

The frame problem is encountered in program execution whenever we call a feature. Only at this point during the execution do we use information hiding: we potentially cross the boundary between two classes, with the client class only aware of the supplier's interface.

Every feature execution has its own environment (see equation (5.116); and (5.129)). The current environment cannot be changed by the execution of another feature. All values stored in the environment are never changed by the call, we can assume them to be constant.

Values set in the global state will never change, and we can assume that a routine has no way to tell whether a **once** feature has already been evaluated or not (this is an explicit simplicification, as there are indeed ways to detect the execution of a once routine, for example by side-effects).

The part of the state that is interesting for frame specifications is the heap. This is where unwanted side-effects can take place.

The heap is contains two parts: a mapping from locations to objects (see equation (5.66)) and which objects are allocated (see equation (5.70)). Section 5.4.4 shows that object allocation alone does not produce visible side-effects. Consequently, the frame problem can disregard allocation and work with a similified heap from locations to objects.

For a concrete heap H, we will call this mapping σ. The mapping can be constructed by closing the first argument in function $\_(\_)$ (5.66). The type of σ is $Loc \to Obj$.

The heap itself is unbounded; there can be an arbitrary number of locations and objects. Frame specifications for object-oriented programs have to identify a bounded set of locations whose values are read or modified during the execution of a routine. We call this set *the frame of the routine*. All other locations, outside of the frame, have to be constant. Given a transformation of σ into a state σ' and a frame f, a legal state transformation would be the transformation that keeps all locations outside of the frame constant:

$$f \lhd \sigma = f \lhd \sigma'$$

This equation of framing is too strong, as it ignores the allocation of new objects and the change of these new objects. Knowing that state abstractions only use allocated objects, we can restrict our definition of framing

to all locations of allocated objects.

The following equations for framing introduced by Kassios [36]:

$$\Xi f \triangleq f \lhd \sigma = f \lhd \sigma' \tag{7.1}$$

$$\Delta f \triangleq \Xi(Used \setminus f) \tag{7.2}$$

This definition of Δf describes that all locations outside of the frame f *that are used by allocated objects in the pre-state of the computation* have their values unchanged by the computation.

Definition 7.1: Framing

Giving a state transformation T that transforms the locations of the heap H from σ to σ', a given set of locations f **frames** the transformation T if and only if Δf holds for T.

This definition allows the state transformation to change any of the locations named in the frame, and also all locations of object unallocated in the pre-state of the computation.

If the specifications of a feature in object-oriented programs gives sufficient information about the frame needed by the feature to fulfill its contract, it is easily possible to show the non-interference of the sequential execution of two computations, if their frames do not overlap.

7.4.1 Read vs write effects

When reasoning about frames, we want to know which state predicate P is affected by which state transformation T. The state predicate P might be defined independently of the state transformation T, for example P might be a query of one library, while T is a command defined in another library. As noted, this makes it impossible to include a specification of the effect of T on P in the specification of P or T.

We solve this problem by splitting the specification into two parts. We know that the effect of T on P is conveyed by the state. T's changes to the state are called its *write effect*. P's use of the state is called its *read effect* [32].

Definition 7.2: Write effect

The write effect of a state transformation T on the state s is the set of locations in the heap s_H whose values are changed between the pre- and the post-state of the state transformation T.

Definition 7.3: Read effect

The read effect of a state predicate P on the state s is the set of locations in the heap s_H that are read to compute the resulting value of P.

Knowing that the heap is the only means of communication between P and T, we can combine read and write effect to compute the actual effect of T on P. We assume that the environment, global state and object creation plays no role in read and write effects of T.

7.5 Dynamic Frames

In his thesis [37], Ioannis Kassios has developed an object-oriented programming and verification model. To solve the problem of framing in his programming model, Kassios introduced the idea of "dynamic frames". This section summarizes the idea of dynamic frames.

Kassios's programming model defines modules. Subtyping and the interface/implementation relation are handled through mathematical refinement of the modules.

Modules define specification variables, program variables and operations. Refinement of two machines means that the axioms of the refining module imply the axioms of the refined module.

Along with program variables, which are part of the state, the model introduces specification variables, defined in terms of other variables. Operations are axioms defining state transitions, expressing a relation between a pre-state and a post-state. The value of a variable x in a post-state is marked by a tick: x'.

Listing 7.3 shows the definition of a polar point data type that implements scaling. Listing 7.4 shows a possible implementation of the point data type using Cartesian coordinates.

```
module Point
  spec var a ∈ ℝ
  spec var r ∈ ℝ

  scale (f ∈ ℝ) ensures r' = r*f
end module
```

Listing 7.3: Definition of a polar point

```
module Point
  prog var x ∈ ℝ
  prog var y ∈ ℝ
  spec var a = ...
  spec var r = √(x² + y²)

  scale (f ∈ ℝ) ensures x' = x*f ∧ y' = y*f
end module
```

Listing 7.4: Implementation of a point using Cartesian coordinates

One of the proof obligations of the refinement would be to show that the new definition of `scale` implies the old definition.

$$
\begin{gathered}
(x' = xf \land y' = yf \land r = \sqrt{x^2 + y^2} \land r' = \sqrt{x'^2 + y'^2} \Rightarrow r' = rf) \Leftrightarrow \\
(\sqrt{x'^2 + y'^2} = \sqrt{x^2 + y^2} f) \Leftrightarrow \\
(\sqrt{(xf)^2 + (yf)^2} = \sqrt{x^2 + y^2} f) \Leftrightarrow \\
(\sqrt{(x^2 + y^2) f^2} = \sqrt{x^2 + y^2} f) \Leftrightarrow \\
(\sqrt{x^2 + y^2} f = \sqrt{x^2 + y^2} f)
\end{gathered}
$$

7.5.1 Object-orientation

Object-orientation is introduced by a heap structure similar to the one introduced in section 5.4. Program variables are linked to certain address locations.

Attributes are introduced making program variables shortcuts to looking up values in the heap. Program variable x becomes an attribute by looking up the value at $addr_x$ in the heap.

$$x = \sigma(addr_x)$$

There is a special variable called *self*. Evaluating an expression on another object means to replace *self* by the actual reference to the object:

$$p.E = E(p/self)$$

7.5.2 Frame variables

A *dynamic frame* is a specification variable whose value is a set of locations. As shown in listing 7.4, the frame is defined by an abstraction function on the program state.

This, as introduced by Kassios, is different from previous approaches: the frame is an abstraction of the full state. It has the full power of a state-dependent expression. Frames can change any time the state changes.

Kassios introduces framing as part of the specification of a variable. Frames are the memory locations that contribute to this variable. Let f be a frame and v be a variable, we express that f frames v by writing:

$$f \,\mathbf{frames}\, v \quad \triangleq \quad f \subseteq \mathit{Used} \wedge \forall \sigma'.\Xi f \Rightarrow v' = v \tag{7.3}$$

This is the original definition from Kassios [36]. The following expanded version of (7.3) might be more clear:

$$\frac{f \subseteq \mathit{Used} \quad \forall \sigma, \sigma'.(f \lhd \sigma = f \lhd \sigma') \Rightarrow v = v'}{f \,\mathbf{frames}\, v} \tag{7.4}$$

With a frame specification, we give a global invariant specifying the relation between a variable and its frame: if no location included in the frame f in the pre-state of any state transformation is changed by that state transformation, then the value of the variable v will not change.

If v is a program variable, then the frame specification is trivially fulfilled if the address of the program variable is always included in the frame. If v is a specification variable, the all the addresses of program variables that are used to compute the value of v has be in the frame. Otherwise, the frame specification is not fulfilled by the implementation.

The power of dynamic frame specifications is the possibility of underspecifying frame variables. This is because the following theorem holds for domain restrictions:

$$g \subseteq f \wedge (f \lhd \sigma = f \lhd \sigma') \Rightarrow (g \lhd \sigma = g \lhd \sigma') \tag{7.5}$$

Proof:

$$(g \subseteq f), (f \lhd \sigma = f \lhd \sigma') \vdash (g \lhd \sigma = g \lhd \sigma')$$
$$\Rrightarrow \text{Definition of } \lhd$$
$$(g \subseteq f), (\forall o : o \in f \Rightarrow \sigma(o) = \sigma'(o)) \vdash (\forall o : o \in g \Rightarrow \sigma(o) = \sigma'(o))$$
$$\Rrightarrow \text{Definition of } \subseteq$$
$$(\forall o : o \in g \Rightarrow o \in f), (\forall o : o \in f \Rightarrow \sigma(o) = \sigma'(o)) \vdash (\forall o : o \in g \Rightarrow \sigma(o) = \sigma'(o))$$
$$\Rrightarrow \text{Simplification}$$
$$(\forall o : (o \in g \Rightarrow o \in f), (o \in f \Rightarrow \sigma(o) = \sigma'(o))) \vdash (\forall o : o \in g \Rightarrow \sigma(o) = \sigma'(o))$$
$$\Rrightarrow \text{Transitivity of } \Rightarrow$$
$$(\forall o : o \in g \Rightarrow \sigma(o) = \sigma'(o)) \vdash (\forall o : o \in g \Rightarrow \sigma(o) = \sigma'(o))$$
$$\square$$

Theorem (7.5) specifies that a given frame specification may describe a superset of the actual frame specification that is given in later refinements of the programs. As refinement is used for inheritance and information hiding, the frame in the implementation has to be a subset of the frame in the specifications, and redefinitions of frames in subtypes are only allowed to specify smaller frames.

Two variables are independent if their dynamic frames are disjoint. This makes is possible to maintain the knowledge about one abstraction over some state transformation if the state transformation is constrained by the Δ predicate (7.2). This is expressed by the following theorem:

$$g \,\mathbf{frames}\, y \wedge f \cap g = \varnothing \wedge \Delta f \Rightarrow y = y' \tag{7.6}$$

Proof:

$$(g \,\mathbf{frames}\, y), (f \cap g = \varnothing), \Delta f \vdash y = y'$$
$$\Rrightarrow \text{Definition (7.3)}$$
$$(g \subseteq \mathit{Used}), \forall o, o'.(g \lhd o = g \lhd o' \Rightarrow y = y'), (f \cap g = \varnothing), \Delta f \vdash y = y'$$
$$\Rrightarrow \text{Definition (7.2)}$$
$$(g \subseteq \mathit{Used}), \forall o, o'.(g \lhd o = g \lhd o' \Rightarrow y = y'), (f \cap g = \varnothing), \Xi(\mathit{Used} \setminus f) \vdash y = y'$$
$$\Rrightarrow \text{Definition (7.1)}$$
$$(g \subseteq \mathit{Used}), \forall o, o'.(g \lhd o = g \lhd o' \Rightarrow y = y'), (f \cap g = \varnothing),$$
$$(\mathit{Used} \setminus f) \lhd \sigma = (\mathit{Used} \setminus f) \lhd \sigma' \vdash y = y'$$
$$\Rrightarrow \text{Instantiation of the } \forall \text{ with } \sigma, \sigma'$$

$$(g \subseteq \mathit{Used}), (g \lhd \sigma = g \lhd \sigma' \Rightarrow y = y'), (f \cap g = \varnothing),$$
$$(\mathit{Used} \setminus f) \lhd \sigma = (\mathit{Used} \setminus f) \lhd \sigma' \vdash y = y'$$
$$\Rrightarrow \text{Simplification (set-theory)}$$
$$(g \lhd \sigma = g \lhd \sigma' \Rightarrow y = y'), (\mathit{Used} \setminus f) \lhd \sigma = (\mathit{Used} \setminus f) \lhd \sigma' \wedge$$
$$g \subseteq (\mathit{Used} \setminus f) \vdash y = y'$$
$$\Rrightarrow \text{Theorem (7.5)}$$
$$y = y' \vdash y = y'$$
$$\square$$

Theorem (7.6) is the key theorem to dynamic frame reasoning: if we are able to ensure that two frames are disjoint, and we use one to frame an arbitrary specification variable and restrict the modification of the state to the second frame, we know that the modification will not change the value of the specification variable.

As we have seen, the disjointness property of frames is the property that needs to be maintained while reasoning about program executions.

7.5.3 Modular reasoning

Maintaining the disjointness of the frames requires understanding of how frames change by state updates. Again, we face the dilemma of modular reasoning: how can one module state that is does not change the frame in another module if the two modules are developed independently?

The answer is *self-framing frames*. As frames are defined by specification variables, frames themselves can carry frames. Most important, they can frame themselves:

$$g \textbf{ frames } g$$

This means that the frame (the set of addresses) will only change if some the content of any of the addresses it contains changes. But if the change is limited to a frame f disjoint from g, then g will not change.

$$g \textbf{ frames } g \wedge (f \cap g = \varnothing) \wedge \Delta f \Rightarrow g = g'$$

The property g **frames** g is added to the specification of the module defining g and the property Δf is added to the specification of the module defining f. The client that is integrating the two modules then can maintain the property $(f \cap g = \varnothing)$ over state transitions restricted by Δf.

For example, knowing in the module defining f that f will not change makes it easy to maintain disjointness

$$g \textbf{ frames } g \wedge (f \cap g = \varnothing) \wedge \Delta f \wedge f = f' \Rightarrow (f' \cap g' = \varnothing)$$

which follows trivially by the fact that neither f nor g is changed by the state transformation.

7.5.4 Object creation

The property $f = f'$ is very restrictive. We might want a state transition to change the frame. For example, adding a new element to a linked list requires enlarging the frame of the list representation by the new cell that builds up the list.

If we allow f to change arbitrarily, then we cannot maintain the disjointness from g. But we know that g does not contain any unallocated objects (7.3). Enlarging f by newly allocated objects will maintain the disjointness. Enlarging the frame by new objects is defined by the Λ predicate:

$$\Lambda f \triangleq f' \subseteq f \cup \mathsf{Unused} \tag{7.7}$$

Λf is a predicate on a state transformation, relating pre- and post-state. It can only be used to specify operations. Specifying an operation with Λf makes it easy to maintain the disjointness of two frames:

$$g \textbf{ frames } g \wedge (f \cap g = \varnothing) \wedge \Delta f \wedge \Lambda f \Rightarrow (f' \cap g' = \varnothing)$$

7.5.5 Summary

Dynamic frames as introduced by Kassios [36, 37] defines frames as sets of resources that can cause interference, in the case of object-orientation memory locations on the heap.

They are dynamic as they are state abstractions, called specification variables: whenever the state changes, the value of the frame can change as well.

Framing invariants limit changes of variables: if a variable is framed by a frame f, the framing invariant tells us that the value of the variable will only change if a location in f changes its value. By showing that a state transformation will not change any of the values in f, the value returned by the query framed by f will not change.

Showing that a state transformation does not change f is done by defining a second frame g disjoint from f. By restricting the change of the state transformations to the locations in g, we know that all location in f keep

their value and the return value of the query is invariant over the state transformation.

Frames are variables. It is possible to frame frames by other frames. It is possible to have frames frame themselves. Self-framing frames help us to develop modular specifications for software components.

Reasoning about dynamic frames requires maintaining disjointness properties of frames. As long as frames are disjoint, software components that are limited in their specification by these frames will not interfere with each other.

Reasoning about the disjointness of frames comes very naturally. Disjoint frames partition the heap. In that respect, reasoning about heap partitions is similar to reasoning in separation logic [75], but avoids the definition of a full new logic. These partitions are used by components that make up the system. If two components always work on separate partitions to fulfill their contracts, then the two partitions trivially do not interfere with each other.

Frame variables can carry their own specifications. The stronger the specification of the frame variable, the more details are known to the client and the easier it is for the client to show non-interference. As an obligation to the supplier, strong specifications for frame variables restrict possible implementations or subtypes, limiting the set of objects that can be used.

In contrast to static ownership, frames do not put any restrictions on passing around references in the system. An "owner as modifier" policy is not required. Frames work well with the inheritance relations and with global data.

7.6 Frames as contracts

Kassios's theory is developed on the basis of a non-existing programming language. The language only contains elementary constructs needed illustrate and reason about pointer structure and framing.

Industrial strength programming languages integrate many concepts into a single, consistent whole. The integration of frame specifications into Eiffel needs to be consistent with all existing language constructs and the overall design philosophy. For Eiffel, this means:

- The language extension has to be backward compatible with existing Eiffel code. Existing Eiffel has to compile without changes.

- The semantics of existing code needs to be sound without the new

annotations. Code without annotations must not imply or infer a specification.

- The number of new keywords introduced into the language should be minimal. Keywords should be align with existing ones, using easy and understandable regular English words. At the same time, it should be uncommon that the keywords are used as regular identifiers for features or class names.
- It must be possible to annotate frame specifications with tags to describe their semantics.
- Annotated code must be readable even for a person not familiar with the Eiffel language.
- It has to be easy to for tools and readers to differentiate between interface and implementation. It must be possible to extract the interface from code.
- Eiffel uses expressions to express contracts, including calls to other features. It should also be possible to use expressions in frame specifications.
- Numerous runtime notions are not available in the Eiffel language, except through means like reflection or introspection. It is not possible to access encapsulated objects and it is not possible to access the set of all allocated objects. Frame specifications must not talk explicitly about such objects.

We introduce a new class called `FRAME` into the language that captures sets of resources. Similar to the model classes introduced in chapter 6, instances of `FRAME` are immutable and have a value semantics.

`FRAME`s only offer a limited number of operations: existing sets can be checked for inclusion ($x \in F$), subset relations ($F \subseteq G$) and disjointness of sets ($F \cap G = \varnothing$). Frames can be constructed by creating the empty frame ($\varnothing$), by including resources into existing frames ($F \cup x$) and by joining frames existing frames ($F \cup G$).

It is not necessary to create intersections of frames, except for the purpose of checking disjointness, or to reduce the size of a frame by any other means. All quantifications are also not needed.

Kassios' specification variables are generalized using expressions: expressions are universal state abstraction functions. The resulting type of specification variables that capture frames has to be `FRAME`.

Using expressions to describe frames is consistent with the general use of expressions in other parts of the contract language. This introduces new needs for specifications. Kassios' simplified language was clearly split into two parts: observations of the state are handled using (specification or programming) variables. State transformations are done by before/after predicates. Only observations, that means variables, have to be framed.

When using expressions as frame specifications, commands can contribute to the result of an expression: the expression can call some function, what itself makes use of some command while computing its **Result** value. It has to be possible to frame command executions.

To express frames, we introduce the new keyword **use** into the language. The syntax of a **use** clause is similar to the syntax of **require** or **ensure** clauses such that it annotates a given feature with a number of expressions.

```
feature
  some_query (arg: ARG_TYPE): RESULT_TYPE
    use
      some_tag: some_expression -- Type: FRAME
```

These expressions describe the resources that a given feature uses to accomplish its task: computing a result value or updating a number of (potentially different) resources. If all resources described by the expressions remain unchanged by a state transformation, then the resulting effect of the feature invocation is also unchanged.

f **frames** g becomes `g use f` in Eiffel code. The frame becomes part of the specification of a feature, limiting its implementation similar to strong postconditions.

Complementary to **use** clauses, we introduce **modify** clauses. Modify captures the intend of the Δ function defined by Kassios (7.2): it limits possible changes. As we demand command/query separation, **modify** clauses only make sense as part of the contract for commands. Annotating a feature with **modify** `f` is semantically equivalent with writing Δf, thus $\Xi(Used \setminus f)$.

```
feature
  some_command (arg: ARG_TYPE)
    use
      some_use_tag: some_expression -- Type: FRAME
    modify
      some_tag: some_expression -- Type: FRAME
```

Adding this feature as an extra clause make **modify** annotations symmetric to **use** annotations. An alternative would be to add modify expres-

sions to **ensure** clauses, similar to **old** expressions, losing symmetry and making it more difficult to reason about modify clauses in the context of inheritance.

7.7 Extending dynamic frames

Adopting Kassios' dynamic frames to a real-world programming language required a number of changes. The following major differences exist between Kassios' dynamic frames and Dynamic Frame Contracts:

- Kassios's frame specifications need access to the set of all allocated objects. Dynamic frame contracts do not require this.
- As an executable programming language, Eiffel needs to clearly separate the implementation from the specification. Kassios's refinement calculus does not provide this, treating inheritance and implementation using refinement.
- Queries are allowed to use commands for their implementation. Kassios's specification variables do not allow to execute a sequence of operations on the state to compute result values.
- Kassios introduces a specific symbol (δ) to frame read effects on variables, while adding write effects to the postcondition of an operation. The actual write effects have to be extracted from the postcondition. Dynamic frame contracts establish a symmetry between read and write effects, making them more accessible.
- Dynamic frame contracts are executable. In section 7.15, we give instructions on how to monitor frames at runtime.
- As frames in Dynamic Frame Contracts are regular queries in the language, they can contain arguments. Kassios's specification variables are all argument-less variables.

Dynamic frame contracts try to make frame specifications easy to identify and read. They try to maintain the verbosity and unambiguity of the Eiffel programming language.

7.8 Granularity of Frames

Until now we have talked about frames on an abstract level as sets "resources". Kassios defines frames to contain set of memory locations, meaning fields of objects.

Frame specifications are used to exclude non-interference of components that are developed independently. These components make use of subcomponents, by instantiating the subcomponents classes. In the general case, the two components will not work on different fields of shared instances, but use different instances.

This observation allows frames introduced by the `FRAME` class to be more course grained than frames introduced by Kassios. Instances of `FRAME` capture *sets of objects*, and regard all the fields of the objects contained in the set as the resources captured by the frame.

Frames as sets of objects are easier to understand and specify. There is no easy way of describing a field in Eiffel. Writing down the field will result in an evaluation of the field. Strings are cumbersome and are not easy to check for consistency by the parser. Agents are a possible way to specify fields, but are heavy weight and the notation of equivalence of two agents is not well defined. In general, uniform access makes it difficult to identify fields and inheritance may change a function to a field.

Both approaches, sets of fields and sets of objects, limit frame reasoning to the domain of objects within the object-oriented paradigm. Frames could also capture other kinds of resources, like files on the file system or states of hardware components. This is beyond the scope of this thesis.

7.9 Confining change

The Λ relation (7.7) used in the postcondition of a command describes that the change to a frame is limited. The frame in the post-state can only contain objects that were either in the frame before, or that have been created during the execution of the command.

It is not possible to express this property with existing language constructs, as there is (outside of introspection) no way to tell which object were created during the execution of a routine.

To solve this problem, we introduce the **`confined`** keyword, that is similar to the Λ relation as introduced by Kassios. The boolean predicate **`confined`** `f` can only be used in postconditions (similar to old) and expresses that the new frame does not contain any objects in the post-state

```
do
  l := Allocated - f -- l is a fresh local id
  -- Routine implementation
ensure
  confine_f: f ∩ l = ∅
end
```

Listing 7.5: Sketch of flattened **confined** *f* postcondition

that were not already in the frame in the pre-state, except for object that were created during the execution of the routine.

7.10 Formalization

use and **modify** clauses are introduced into the static model of the programming language similar to pre- and postconditions. We define two new functions that yield the expressions for the use and modify frame:

$$\mathsf{use} : \mathit{Feature} \to \mathit{Expr} \tag{7.8}$$

$$\mathsf{modify} : \mathit{Feature} \to \mathit{Expr} \tag{7.9}$$

The result type of the expressions always has to be *FRAME*.

The **confined** keyword is not present in the static model. It is instead replaced by storing the used objects minus the value of the frame in the precondition at the start of the execution, and then demanding that the value of the frame in the postcondition is a subset to the stored set. The outline of such a transformation is sketched in listing 7.5. This makes is necessary — in this specific case — to allow access to the set of all allocated objects. The set $\mathit{Allocated}$ is the set of all allocated objects. It is defined as:

$$\mathit{Allocated} : \mathit{State} \to \mathbb{P}\ \mathit{Obj} \tag{7.10}$$

$$\mathit{Allocated}(s) = \{o \in \mathit{Obj} | \mathsf{alloc}(o, s_H)\} \tag{7.11}$$

7.10.1 Use sets

The use set of an execution is the set of all objects o that have fields read during the execution of a routine or the evaluation of an instruction. Reading a field from an object means using the $\_(\_)$ function defined in (5.66).

We define two functions to compute the use set of a given expression and instruction:

$$\mathbf{UseSetE}(\_,\_) : \mathit{Expr} \times \mathit{State} \to \mathbb{P}\ \mathit{Obj} \tag{7.12}$$

$$\mathbf{UseSetI}(\_,\_) : \mathit{Instr} \times \mathit{State} \to \mathbb{P}\ \mathit{Obj} \tag{7.13}$$

Similar to the definitions given in the operational semantics, we define the two functions inductively over all possible evaluation trees:

$$\mathbf{UseSetE}(l, s) = \varnothing \tag{7.14}$$

$$\mathbf{UseSetE}(l = l', s) = \varnothing \tag{7.15}$$

$$\frac{\langle l := e, s\rangle \rightsquigarrow s'}{\mathbf{UseSetE}(l := e; e', s) = \mathbf{UseSetI}(l := e, s) \cup \mathbf{UseSetE}(e', s')} \tag{7.16}$$

$$\frac{\left[\begin{array}{c} B = \mathsf{body}(c) \\ E = \langle \texttt{Current} := \mathsf{new}(s), arg^1 := s_E(l^1), \ldots, arg^n := s_E(l^n)\rangle \end{array}\right]}{\mathbf{UseSetE}(\texttt{create}\, c(l^1, \ldots, l^n), s) = \mathbf{UseSetI}(B, (s_H, E, s_G))} \tag{7.17}$$

$$\mathbf{UseSetE}(mc, s) = \varnothing \tag{7.18}$$

$$\mathbf{UseSetE}(l.a, s) = \{s_E(l)\} \tag{7.19}$$

$$\frac{\left[\begin{array}{c} f \notin \mathit{OnceFunction} \vee \neg\mathsf{stored}(s_G, f) \\ B = \mathsf{body}(\mathsf{version}(\mathsf{typeof}(s_E(l)))(f)) \\ E = \langle \texttt{Current} := s_E(l), arg^1 := s_E(l^1), \ldots, arg^n := s_E(l^n)\rangle \end{array}\right]}{\mathbf{UseSetE}(l.f(l^1, \ldots, l^n), s) = \mathbf{UseSetI}(B, (s_H, E, s_G))} \tag{7.20}$$

$$\frac{\left[\, f \in \mathit{OnceFunction} \wedge \mathsf{stored}(s_G, f)\right]}{\mathbf{UseSetE}(l.f(l^1, \ldots, l^n), s) = \varnothing} \tag{7.21}$$

$$\mathbf{UseSetI}(\epsilon, s) = \varnothing \tag{7.22}$$

$$\frac{\big[\,\langle S^1, s\rangle \rightsquigarrow s'\big]}{\mathbf{UseSetI}(S^1 S^2, s) = \mathbf{UseSetI}(S^1, s) \cup \mathbf{UseSetI}(S^1, s')} \tag{7.23}$$

$$\mathbf{UseSetI}(l := E, s) = \mathbf{UseSetE}(E, s) \tag{7.24}$$

$$\langle a := l, s\rangle \rightsquigarrow = \varnothing \tag{7.25}$$

$$\frac{\big[\,s_E(C) = \text{TRUE}\big]}{\mathbf{UseSetI}(\texttt{if}\, C\, \texttt{then}\, S1\, \texttt{else}\, S2\, \texttt{end}, s) = \mathbf{UseSetI}(S1, s)} \tag{7.26}$$

$$\frac{\big[\,s_E(C) = \text{FALSE}\big]}{\mathbf{UseSetI}(\texttt{if}\, C\, \texttt{then}\, S1\, \texttt{else}\, S2\, \texttt{end}, s) = \mathbf{UseSetI}(S2, s)} \tag{7.27}$$

$$\frac{\left[\begin{array}{c} s_E(C) = \text{FALSE} \\ \langle S, s\rangle \rightsquigarrow s' \\ R = \mathbf{UseSetI}(\,\texttt{from}\,\texttt{until}\, C\, \texttt{invariant}\, I\, \texttt{variant}\, V\, \texttt{until}\, S\, \texttt{end}, s')\end{array}\right]}{\left[\begin{array}{c}\mathbf{UseSetI}(\,\texttt{from}\,\texttt{until}\, C\, \texttt{invariant}\, I\, \texttt{variant}\, V\, \texttt{until}\, S\, \texttt{end}, s) = \\ \mathbf{UseSetI}(S, s) \cup R\end{array}\right]} \tag{7.28}$$

$$\frac{s_E(C) = \text{TRUE}}{\mathbf{UseSetI}(\,\texttt{from}\,\texttt{until}\, C\, \texttt{invariant}\, I\, \texttt{variant}\, V\, \texttt{until}\, S\, \texttt{end}, s) = \varnothing} \tag{7.29}$$

$$\frac{B = \mathsf{body}(\mathsf{version}(\mathsf{typeof}(s_E(l)))(f))}{\left[\begin{array}{c}\mathbf{UseSetI}(l.f(a^1, \ldots, a^n), s) = \\ \mathbf{UseSetI}(B, (s_H, \langle \texttt{Current} := s_E(l), arg^1 := s_E(a^1), \ldots, arg^n := s_E(a^n)\rangle, s_G))\end{array}\right]} \tag{7.30}$$

$$\mathbf{UseSetI}(\texttt{check}\, l\, \texttt{end}, s) = \varnothing \tag{7.31}$$

7.10.2 *Modify sets*

The modify set of an execution of an instruction I for a given state s is the set of all memory locations that have been changed during the execution. The modify set is an over-approximation; it ignores the possibility that the same value is written again into the memory location, which thus does not change.

Only the assignment instruction can modify memory locations. We also assume full command-query separation. This makes it possible to define modify sets only on instructions, not on expressions, as the later have no side-effects.

$$\mathbf{ModifySet}(\_,\_) : \mathit{Instr} \times \mathit{State} \to \mathbb{P}\ \mathit{Obj} \tag{7.32}$$

Again, we define modify sets over the syntax of instructions.

$$\mathbf{ModifySet}(\epsilon, s) = \varnothing \tag{7.33}$$

$$\frac{\langle S^1, s\rangle \rightsquigarrow s'}{\mathbf{ModifySet}(S^1 S^2, s) = \mathbf{ModifySet}(S^1, s) \cup \mathbf{ModifySet}(S^2, s')} \tag{7.34}$$

$$\mathbf{ModifySet}(l := E, =)\varnothing \tag{7.35}$$

$$\langle a := l, s\rangle \rightsquigarrow = \{\mathbf{Current}\} \tag{7.36}$$

$$\frac{s_E(C) = \text{TRUE}}{\mathbf{ModifySet}(\texttt{if}\ C\ \texttt{then}\ S1\ \texttt{else}\ S2\ \texttt{end}, s) = \mathbf{ModifySet}(S1, s)} \tag{7.37}$$

$$\frac{s_E(C) = \text{FALSE}}{\mathbf{ModifySet}(\texttt{if}\ C\ \texttt{then}\ S1\ \texttt{else}\ S2\ \texttt{end}, s) = \mathbf{ModifySet}(S2, s)} \tag{7.38}$$

$$\frac{\left[\begin{array}{c} s_E(C) = \text{FALSE} \\ \langle S, s\rangle \rightsquigarrow s' \\ R = \mathbf{ModifySet}(\texttt{from}\ \texttt{until}\ C\ \texttt{invariant}\ I\ \texttt{variant}\ V\ \texttt{until}\ S\ \texttt{end}, s')\end{array}\right]}{\left[\begin{array}{c}\mathbf{ModifySet}(\texttt{from}\ \texttt{until}\ C\ \texttt{invariant}\ I\ \texttt{variant}\ V\ \texttt{until}\ S\ \texttt{end}, s) = \\ \mathbf{ModifySet}(S, s) \cup R\end{array}\right]} \tag{7.39}$$

$$\frac{s_E(C) = \text{TRUE}}{\mathbf{ModifySet}(\texttt{from until}\ C\ \texttt{invariant}\ I\ \texttt{variant}\ V\ \texttt{until}\ S\ \texttt{end}, s) = \varnothing} \tag{7.40}$$

$$\frac{\left[\begin{array}{c} B = \mathsf{body}(\mathsf{version}(\mathsf{typeof}(s_E(l)))(f)) \\ E = \langle \texttt{Current} := s_E(l), arg^1 := s_E(a^1), \ldots, arg^n := s_E(a^n) \rangle \end{array}\right]}{\mathbf{ModifySet}(l.f(a^1, \ldots, a^n), s) = \mathbf{ModifySet}(B, (s_H, E, s_G))} \tag{7.41}$$

$$\mathbf{ModifySet}(\texttt{check}\ l\ \texttt{end}, s) = \varnothing \tag{7.42}$$

7.10.3 Correctness

A given implementation satisfies the frame specification if the result of the evaluation of use or modify expression in the pre-state of the execution is a superset of the actual use or modify set.

$$\forall f : \mathit{Feature}, s : \mathit{State} |$$
$$(\mathbf{UseSetI}(\mathsf{body}(f), ()s) \cap \mathit{Allocated}(s)) \subseteq \mathbf{eval}(\mathsf{use}(f), s) \tag{7.43}$$
$$(\mathbf{ModifySet}(\mathsf{body}(f), ()s) \cap \mathit{Allocated}(s)) \subseteq \mathbf{eval}(\mathsf{modify}(f), s) \tag{7.44}$$

Computing the use and modify frames is not modular. To compute UseSetI and ModifySet, we need to look at the body of all suppliers of the routine.

In modular verification, the correctness of all suppliers can be assumed to show the correctness of a routine. The solution is to use the transitivity of the subset relation: if the supplier is correct, then its use and modify frame has to be a subset of the evaluation of its **`use`** and **`modify`** clause. Excluded from this subset relation are all objects that were allocated by the routine itself or its suppliers.

- An implementation is correct with respect to its **`modify`** clause, evaluated to the set M, if there is an assignment to an attribute, **`Current`** is an element of M and
- if an other routine f is called in state s and N contains the objects that were created during the execution of the routine, then

  $$\mathbf{eval}(\mathsf{modify}(f), s) \subseteq (M \cup N)$$

  has to hold.

- An implementation is correct with respect to its **use** clause, evaluated to the set M, If an other routine f is called in state s and N contains the objects that were created during the execution of the routine, then
$$\mathbf{eval}(\mathsf{use}(f), s) \subseteq (U \cup N)$$
has to hold.

- The **use** and **modify** frames of every attribute must contain **Current**.

All features, commands and queries need to have a use frame. If no explicit use frame is specified, then "the set of all living objects" ($Allocated$, see (7.10)) is assumed, making any implementation automatically correct, as $U \cup N$ and $M \cup N$ contain all living objects at state s and the frame always contains only living objects.

7.11 Reasoning with frames

The key rule behind frame based reasoning, defined as an axiomatic, Hoare style rule, is

$$\frac{q\,\mathtt{use}\,qf \quad c\,\mathtt{modify}\,cf \quad \{Q\}c\{R\}}{\{Q \wedge P(q) \wedge qf \cap cf = \varnothing\}c\{R \wedge P(q)\}} \tag{7.45}$$

This rule states: if a query q is framed by qf as its use frame and a command c is framed with cf as its modify frame, and we know that if the two frames are disjoint at the pre-state of the execution of c, then a predicate $P(q)$ is retained over the execution of c.

7.11.1 Soundness

The following theorem is the translation of rule (7.45) into the operational semantics as presented here:

$$\begin{gathered}\forall c : Routine, q, qf, cf : Query, s : State| \\ \langle c, s\rangle \rightsquigarrow s' \wedge P(\mathbf{eval}(q, s)) \wedge (\mathbf{eval}(qf, s) \cap \mathbf{eval}(cf, s) = \varnothing) \Rightarrow \\ P(\mathbf{eval}(q, s'))\end{gathered} \tag{7.46}$$

To prove the soundness of this important property, we first prove that any allocated memory location outside of the modify frame cf will keep

its value the execution of c. We have implied the validity of the call with respect to precondition and invariant to shorten the proof text.

$$\forall c : Routine, s, s' : State, o : Obj, a : Attribute|$$
$$\langle \mathsf{body}(c), s\rangle \rightsquigarrow s' \wedge o \in Allocated(s_H) - \mathbf{eval}(\mathsf{modify}(c), \Rightarrow) s_H(o.a) = s'_H(o.a) \tag{7.47}$$

Proof: The theorem is proved inductively over all possible c. Because we know that the modify set needs to be a subset of the actual specification, we it is sufficient to prove

$$\forall B : Instr, s, s' : State, o : Obj, a : Feature|$$
$$\langle B, s\rangle \rightsquigarrow s' \wedge o \in Allocated(s_H) - \mathbf{ModifySet}(B, s) \Rightarrow s_H(o.a) = s'_H(o.a)$$

which is also the induction hypothesis, which can be assumed to hold for any subexpression of B, abbreviated as IH in the following proof.

Skip:

$$\begin{aligned} & \langle , s\rangle \rightsquigarrow s', o \in Allocated(s) - \mathbf{ModifySet}(T, s) \vdash s_H(o.a) = s'_H(o.a) \\ (5.121) \Rightarrow\ & \langle , s\rangle \rightsquigarrow s, o \in Allocated(s_H) - \mathbf{ModifySet}(T, s) \vdash s_H(o.a) = s_H(o.a) \end{aligned} \quad \square$$

Sequential composition:

$$\begin{aligned} & \langle S1S2, s\rangle \rightsquigarrow s', o \in Allocated(s_H) - \mathbf{ModifySet}(S1S2, s) \\ & \vdash s_H(o.a) = s'_H(o.a) \\ (5.122) \Rightarrow\ & \langle S1, s\rangle \rightsquigarrow s'', \langle S2, s''\rangle \rightsquigarrow s', o \in Allocated(s_H) - \mathbf{ModifySet}(S1S2, s) \\ & \vdash s_H(o.a) = s'_H(o.a) \\ (7.34) \Rightarrow\ & \langle S1, s\rangle \rightsquigarrow s'', \langle S2, s''\rangle \rightsquigarrow s', o \in Allocated(s_H) - (\mathbf{ModifySet}(S1, s) \cup \\ & \mathbf{ModifySet}(S1, s'')) \\ & \vdash s_H(o.a) = s'_H(o.a) \\ \Rightarrow\ & \langle S1, s\rangle \rightsquigarrow s'', \langle S2, s''\rangle \rightsquigarrow s', o \in Allocated(s_H) - \mathbf{ModifySet}(S1, s), \\ & o \in Allocated(s_H) - \mathbf{ModifySet}(S1, s'')) \\ & \vdash s_H(o.a) = s'_H(o.a) \\ IH \Rightarrow\ & s_H(o.a) = s''_H(o.a), s''_H(o.a) = s'_H(o.a) \vdash s_H(o.a) = s'_H(o.a) \end{aligned}$$

Assignment to local:

$$
\begin{aligned}
& \langle l := E, s\rangle \rightsquigarrow s', o \in \mathit{Allocated}(s_H) - \mathbf{ModifySet}(l := E, s) \\
& \vdash s_H(o.a) = s'_H(o.a) \\
(5.123)\ \Rrightarrow\ & [\![E, s]\!] \rightsquigarrow s'', (s''_H = s'_H), o \in \mathit{Allocated}(s_H) - \mathbf{ModifySet}(l := E, s) \\
& \vdash s_H(o.a) = s'_H(o.a) \\
(5.137)\ \Rrightarrow\ & (s_H(o.a) = s''_H(o.a)), (s''_H = s'_H), o \in \mathit{Allocated}(s_H) - \mathbf{ModifySet}(l := E, s) \\
& \vdash s_H(o.a) = s'_H(o.a) \\
\Rrightarrow\ & s_H(o.a) = s''_H(o.a), s''_H(o.a) = s'_H(o.a) \vdash s_H(o.a) = s'_H(o.a) \quad \square
\end{aligned}
$$

Assignment to attribute:

$$
\begin{aligned}
& \langle a := l, s\rangle \rightsquigarrow s', o \in \mathit{Allocated}(s_H) - \mathbf{ModifySet}(a := l, s) \\
& \vdash s_H(o.a) = s'_H(o.a) \\
(5.124)\ \Rrightarrow\ & s'_H = s_H\langle s_E\texttt{Current}.b := s_E(l)\rangle, o \in \mathit{Allocated}(s_H) - \mathbf{ModifySet}(a := l, s) \\
& \vdash s_H(o.a) = s'_H(o.a) \\
(7.36)\ \Rrightarrow\ & s'_H = s_H\langle s_E(\texttt{Current}).b := s_E(l)\rangle, o \in \mathit{Allocated}(s_H) - \{s_E(\texttt{Current})\} \\
& \vdash s_H(o.a) = s'_H(o.a) \\
(5.72)\ \Rrightarrow\ & (\forall p \in \mathit{Allocated} - \{\mathit{Current}\} | s_H(p.a) = s'_H(p.a)), \\
& o \in \mathit{Allocated}(s_H) - \{s_E(\texttt{Current})\} \\
& \vdash s_H(o.a) = s'_H(o.a) \\
\Rrightarrow\ & s_H(o.a) = s'_H(o.a) \vdash s_H(o.a) = s'_H(o.a) \quad \square
\end{aligned}
$$

True case:

$$
\begin{aligned}
& \langle \mathbf{if}\ C\ \mathbf{then}\ S1\ \mathbf{else}\ S2\ \mathbf{end}, s\rangle \rightsquigarrow s', s_E(C) = \mathrm{TRUE}, \\
& o \in \mathit{Allocated}(s_H) - \mathbf{ModifySet}(\mathbf{if}\ C\ \mathbf{then}\ S1\ \mathbf{else}\ , s) \\
& \vdash s_H(o.a) = s'_H(o.a) \\
(5.125)\ \Rrightarrow\ & \langle S1, s\rangle \rightsquigarrow s', o \in \mathit{Allocated}(s_H) - \mathbf{ModifySet}(\mathbf{if}\ C\ \mathbf{then}\ S1\ \mathbf{else}\ , s) \\
& \vdash s_H(o.a) = s'_H(o.a) \\
(7.37)\ \Rrightarrow\ & \langle S1, s\rangle \rightsquigarrow s', o \in \mathit{Allocated}(s_H) - \mathbf{ModifySet}(S1, s) \\
& \vdash s_H(o.a) = s'_H(o.a) \\
IH\ \Rrightarrow\ & s_H(o.a) = s'_H(o.a) \vdash s_H(o.a) = s'_H(o.a) \quad \square
\end{aligned}
$$

(false case the same way)

Loop continuation:

$$
\begin{aligned}
& \langle \textbf{from until } C \textbf{ invariant } I \textbf{ variant } V \textbf{ loop } S \textbf{ end}, s\rangle \rightsquigarrow s', \\
& o \in \mathit{Allocated}(s_H)-\\
& \textbf{ModifySet}(\textbf{from until } C \textbf{ invariant } I \textbf{ variant } V \textbf{ loop } S \textbf{ end}, s), \\
& s_E(C) = \text{FALSE}, \\
& \vdash s_H(o.a) = s'_H(o.a) \\
(5.127) \Rrightarrow\ & \langle S, s\rangle \rightsquigarrow s', \\
& \langle \textbf{from until } C \textbf{ invariant } I \textbf{ variant } V \textbf{ loop } S \textbf{ end}, s''\rangle \rightsquigarrow s', \\
& o \in \mathit{Allocated}(s_H)-\\
& \textbf{ModifySet}(\textbf{from until } C \textbf{ invariant } I \textbf{ variant } V \textbf{ loop } S \textbf{ end}, s) \\
& s_E(C) = \text{FALSE} \\
& \vdash s_H(o.a) = s'_H(o.a) \\
(7.39) \Rrightarrow\ & \langle S, s\rangle \rightsquigarrow s', s_E(C) = \text{FALSE}, \\
& \langle \textbf{from until } C \textbf{ invariant } I \textbf{ variant } V \textbf{ loop } S \textbf{ end}, s''\rangle \rightsquigarrow s', \\
& o \in \mathit{Allocated}(s_H) - \textbf{ModifySet}(S, s), \\
& o \in \mathit{Allocated}(s_H)-\\
& \textbf{ModifySet}(\textbf{from until } C \textbf{ invariant } I \textbf{ variant } V \textbf{ loop } S \textbf{ end}, s'') \\
& \vdash s_H(o.a) = s'_H(o.a) \\
\mathit{IH} \Rrightarrow\ & s_H(o.a) = s''_H(o.a), s''_H(o.a) = s'_H(o.a) \vdash s_H(o.a) = s'_H(o.a) \quad \square
\end{aligned}
$$

Loop termination:

$$
\begin{aligned}
& \langle \textbf{from until } C \textbf{ invariant } I \textbf{ variant } V \textbf{ loop } S \textbf{ end}, s\rangle \rightsquigarrow s', \\
& o \in \mathit{Allocated}(s_H)-\\
& \textbf{ModifySet}(\textbf{from until } C \textbf{ invariant } I \textbf{ variant } V \textbf{ loop } S \textbf{ end}, s) \\
& s_E(C) = \text{TRUE} \\
& \vdash s_H(o.a) = s'_H(o.a) \\
(5.128) \Rrightarrow\ & s_E(C) = \text{TRUE}, s = s', \\
& o \in \mathit{Allocated}(s_H)-\\
& \textbf{ModifySet}(\textbf{from until } C \textbf{ invariant } I \textbf{ variant } V \textbf{ loop } S \textbf{ end}, s), \\
& \vdash s_H(o.a) = s'_H(o.a) \\
(7.40) \Rrightarrow\ & s_E(C) = \text{TRUE}, s = s', o \in \mathit{Allocated}(s_H) - \varnothing \\
& \vdash s_H(o.a) = s'_H(o.a) \\
\Rrightarrow\ & \vdash s_H(o.a) = s_H(o.a) \quad \square
\end{aligned}
$$

Feature invocation:

$$
\begin{aligned}
& \langle l.f(a^1,\ldots,a^n), s\rangle \rightsquigarrow s', \\
& o \in \mathit{Allocated}(s_H) - \mathbf{ModifySet}(l.f(a^1,\ldots,a^n), s) \\
& \vdash s_H(o.a) = s'_H(o.a) \\
(5.129) \Rrightarrow\ & B = \mathsf{body}(\mathsf{version}(\mathsf{typeof}(s_E(l)))(f)) \\
& \langle B, (s_H, \langle \texttt{Current} := s_E(l), arg^1 := s_E(a^1), \ldots, arg^n := s_E(a^n)\rangle, s_G)\rangle \rightsquigarrow s'', \\
& s''_H = s'_H, o \in \mathit{Allocated}(s_H) - \mathbf{ModifySet}(l.f(a^1,\ldots,a^n), s) \\
& \vdash s_H(o.a) = s'_H(o.a) \\
(7.41) \Rrightarrow\ & B = \mathsf{body}(\mathsf{version}(\mathsf{typeof}(s_E(l)))(f)) \\
& \langle B, (s_H, \langle \texttt{Current} := s_E(l), arg^1 := s_E(a^1), \ldots, arg^n := s_E(a^n)\rangle, s_G)\rangle \rightsquigarrow s'', \\
& s''_H = s'_H, \\
& o \in \mathit{Allocated}(s_H) - \\
& \mathbf{ModifySet}(B, (s_H, \langle \texttt{Current} := s_E(l), arg^1 := s_E(a^1), \ldots, \\
& \quad arg^n := s_E(a^n)\rangle, s_G)) \\
& \vdash s_H(o.a) = s'_H(o.a) \\
IH \Rrightarrow\ & \vdash s_H(o.a) = s_H(o.a) \quad \square
\end{aligned}
$$

The second theorem show that the result of a boolean query will only change if a memory location in the use-set of the computing routine is changed.

$$
\begin{aligned}
& \forall q : \mathit{Query}, s, s' : \mathit{State}, o, r : \mathit{Obj}, a : \mathit{Attribute} | \\
& \quad \mathbf{eval}(q, s) = res \wedge o \notin \mathbf{eval}(\mathsf{use}(c), s) \Rightarrow \\
& \quad\quad \mathbf{eval}(q, (s_H\langle o.a := r\rangle, s_E, s_G))
\end{aligned}
\tag{7.48}
$$

Proof: The theorem is proved inductively over all possible q under a specific state s. Because we know that the use set needs to be a subset of the actual specification, we it is sufficient to prove

$$
\begin{aligned}
& \forall E : \mathit{Expr}, s, s' : \mathit{State}, o, r, res : \mathit{Obj}, a : \mathit{Attribute} | \\
& \quad \mathbf{eval}(E, s) = res \wedge o \notin \mathbf{UseSetE}(E, s) \Rightarrow \\
& \quad\quad \mathbf{eval}(E, (s_H\langle o.a := r\rangle, s_E, s_G)) = res
\end{aligned}
\tag{IH1}
$$

which is also the induction hypothesis, which can be assumed to hold for any subexpression of B (with a corresponding sub-state), abbreviated as $IH1$ in the following proof.

In parallel, we have to prove that the following property holds for all instructions I and states s:

$$\begin{array}{c} \forall I : Instr, s, s' : State, o, r : Obj, a : Attribute| \\ \langle I, s\rangle \rightsquigarrow s' \wedge o \notin \mathbf{UseSetI}(I, s) \Rightarrow \\ \langle I, (s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \end{array} \quad \text{(IH2)}$$

Also, we have to prove that the following property holds for all instructions E and states s:

$$\begin{array}{c} \forall E : Expr, s, s' : State, o, r : Obj, a : Attribute| \\ [\![E, s]\!] \rightsquigarrow s' \wedge o \notin \mathbf{UseSetE}(E, s) \Rightarrow \\ [\![E, (s_H\langle o.a := r\rangle, s_E, s_G)]\!] \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \end{array} \quad \text{(IH3)}$$

Proofs for (IH1)

Local read:

$$\begin{array}{ll} & \mathbf{eval}(l, s) = res, o \notin \mathbf{UseSetE}(l, s) \\ & \vdash \mathbf{eval}(l, (s_H\langle o.a := r\rangle, s_E, s_G)) = res \\ (5.104) \Rightarrow & s_E(l) = res, o \notin \mathbf{UseSetE}(l, s) \\ & \vdash s_E(l) = res \quad \square \end{array}$$

Reference equality:

$$\begin{array}{ll} & \mathbf{eval}(l = l', s) = res, o \notin \mathbf{UseSetE}(l = l', s) \\ & \vdash \mathbf{eval}(l = l', (s_H\langle o.a := r\rangle, s_E, s_G)) = res \\ (5.105) \Rightarrow & (s_E(l) = s_E(l')) = res, o \notin \mathbf{UseSetE}(l = l', s) \\ & \vdash (s_E(l) = s_E(l')) = res \quad \square \end{array}$$

Expression assignment:

$$
\begin{array}{rl}
 & \mathbf{eval}(l := E1; E2, s) = res, o \notin \mathbf{UseSetE}(l := E1; E2, s) \\
 & \vdash \mathbf{eval}(l := E1; E2, (s_H\langle o.a := r\rangle, s_E, s_G)) = res \\
(5.106) \Rrightarrow & \langle l := E1, s\rangle \rightsquigarrow s', \mathbf{eval}(E2, s') = res, o \notin \mathbf{UseSetE}(l := E1; E2, s) \\
 & \vdash \langle l := E1, (s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow s'' \wedge \mathbf{eval}(E2, s'') = res \\
(7.16) \Rrightarrow & \langle l := E1, s\rangle \rightsquigarrow s', \mathbf{eval}(E2, s') = res, \\
 & o \notin \mathbf{UseSetE}(E1, s), o \notin \mathbf{UseSetE}(E2, s) \\
 & \vdash \langle l := E1, (s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow s'' \wedge \mathbf{eval}(E2, s'') = res \\
(5.123) \Rrightarrow & \mathbf{eval}(E2, (s_H, s_E\langle l := \mathbf{eval}(E1, s)\rangle, s_G)) = res, \\
 & o \notin \mathbf{UseSetE}(E1, s), o \notin \mathbf{UseSetE}(E2, s) \\
 & \vdash \mathbf{eval}(E2, (s_H\langle o.a := r\rangle, s_E\langle l := \mathbf{eval}(E1, s\langle o.a := r\rangle)\rangle, s_G)) = res \\
(IH1) \Rrightarrow & \mathbf{eval}(E2, (s_H, s_E\langle l := res'\rangle, s_G)) = res, \\
 & o \notin \mathbf{UseSetE}(E1, s), o \notin \mathbf{UseSetE}(E2, s) \\
 & \vdash \mathbf{eval}(E2, (s_H\langle o.a := r\rangle, s_E\langle l := res'\rangle, s_G)) = res \\
(IH1) \Rrightarrow & \text{TRUE} \quad \square
\end{array}
$$

Create Expressions:

$$
\begin{array}{rl}
 & \mathbf{eval}(\mathsf{create}c(l^1, \ldots, l^n), s) = res, \\
 & o \notin \mathbf{UseSetE}(\mathsf{create}T.c(l^1, \ldots, l^n), s) \\
 & \vdash \mathbf{eval}(\mathsf{create}c(l^1, \ldots, l^n), (s_H\langle o.a := r\rangle, s_E, s_G)) = res \\
(5.107) \Rrightarrow & \mathsf{new}(s_H) = res, o \notin \mathbf{UseSetE}(MC, s) \\
 & \vdash \mathsf{new}(s_H\langle o.a := r\rangle) = res \\
(5.74) \Rrightarrow & \mathsf{new}(s_H) = res \vdash \mathsf{new}(s_H) = res \quad \square
\end{array}
$$

Manifest Constants:

$$
\begin{array}{rl}
 & \mathbf{eval}(MC, s) = res, o \notin \mathbf{UseSetE}(MC, s) \\
 & \vdash \mathbf{eval}(MC, (s_H\langle o.a := r\rangle, s_E, s_G)) = res \\
(5.108) \Rrightarrow & \mathsf{new}(s_H) = res, o \notin \mathbf{UseSetE}(MC, s) \\
 & \vdash \mathsf{new}(s_H\langle o.a := r\rangle) = res \\
(5.74) \Rrightarrow & \mathsf{new}(s_H) = res \vdash \mathsf{new}(s_H) = res \quad \square
\end{array}
$$

Attribute read:

$$
\begin{array}{rl}
 & \mathbf{eval}(l.a, s) = res, o \notin \mathbf{UseSetE}(l.a, s) \\
 & \vdash \mathbf{eval}(l.a, (s_H\langle o.a := r\rangle, s_E, s_G)) = res \\
(5.109) \Rrightarrow & s_H(s_E(l).a) = res, o \notin \mathbf{UseSetE}(l.a, s) \\
 & \vdash s_H\langle o.a := r\rangle(s_E(l).a) = res \\
(7.19) \Rrightarrow & s_H(s_E(l).a) = res, o \neq s_E(l) \\
 & \vdash s_H\langle o.a := r\rangle(s_E(l).a) = res \\
(5.72) \Rrightarrow & \vdash \mathrm{TRUE} \quad \square
\end{array}
$$

Query call:

$$
\begin{array}{rl}
 & (f \notin \mathit{OnceFunction} \vee \neg \mathsf{stored}(s_G, f)), \\
 & \mathbf{eval}(l.f(l^1, \ldots, l^n), s) = res, o \notin \mathbf{UseSetE}(l.f(l^1, \ldots, l^n), s) \\
 & \vdash \mathbf{eval}(l.f(l^1, \ldots, l^n), (s_H\langle o.a := r\rangle, s_E, s_G)) = res \\
(5.110) \Rrightarrow & (f \notin \mathit{OnceFunction} \vee \neg \mathsf{stored}(s_G, f)), \\
 & B = \mathsf{body}(\mathsf{version}(\mathsf{typeof}(s_E(l)))(f)), \\
 & E = \langle \mathtt{Current} := s_E(l), arg^1 := s_E(l^1), \ldots, arg^n := s_E(l^n)\rangle, \\
 & \mathbf{eval}(B, s) = res, o \notin \mathbf{UseSetE}(l.f(l^1, \ldots, l^n), s) \\
 & \vdash \mathbf{eval}(B, (s_H\langle o.a := r\rangle, s_E, s_G)) = res \\
(7.20) \Rrightarrow & (f \notin \mathit{OnceFunction} \vee \neg \mathsf{stored}(s_G, f)), \\
 & B = \mathsf{body}(\mathsf{version}(\mathsf{typeof}(s_E(l)))(f)), \\
 & E = \langle \mathtt{Current} := s_E(l), arg^1 := s_E(l^1), \ldots, arg^n := s_E(l^n)\rangle, \\
 & \mathbf{eval}(B, s_H, E, s_G) = res, o \notin \mathbf{UseSetE}(B, s, E, s_G) \\
 & \vdash \mathbf{eval}(B, s_H\langle o.a := r\rangle, E, s_G) = res \\
(IH1) \Rrightarrow & \vdash \mathrm{TRUE} \quad \square
\end{array}
$$

Query call of old once value:

$$
\begin{array}{rl}
 & f \in \mathit{OnceFunction}, \mathsf{stored}(s_G, f), \\
 & \mathbf{eval}(l.f(l^1, \ldots, l^n), s) = res, o \notin \mathbf{UseSetE}(l.f(l^1, \ldots, l^n), s) \\
 & \vdash \mathbf{eval}(l.f(l^1, \ldots, l^n), (s_H\langle o.a := r\rangle, s_E, s_G)) = res \\
(5.111) \Rrightarrow & f \in \mathit{OnceFunction}, \mathsf{stored}(s_G, f), \\
 & s_G(f) = res \\
 & \vdash s_G(f) = res \quad \square
\end{array}
$$

Proofs for (IH2)

Skip:

$$
\begin{array}{rl}
& \langle , s\rangle \rightsquigarrow s', o \notin \mathbf{UseSetI}(, s) \\
& \vdash \langle , (s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(5.121) \Rrightarrow & o \notin \mathbf{UseSetI}(, s) \\
& \vdash \langle , (s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow (s_H\langle o.a := r\rangle, s_E, s_G) \\
(5.121) \Rrightarrow & \text{TRUE} \quad \square
\end{array}
$$

Sequential composition:

$$
\begin{array}{rl}
& \langle S1\, S2, s\rangle \rightsquigarrow s', o \notin \mathbf{UseSetI}(S1\, S2, s) \\
& \vdash \langle S1\, S2, (s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(5.122) \Rrightarrow & \langle S1, s\rangle \rightsquigarrow s'', \langle S2, s''\rangle \rightsquigarrow s', o \notin \mathbf{UseSetI}(S1\, S2, s) \\
& \vdash \langle S1\, S2, (s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(7.23) \Rrightarrow & \langle S1, s\rangle \rightsquigarrow s'', \langle S2, s''\rangle \rightsquigarrow s', \\
& o \notin \mathbf{UseSetI}(S1, s), o \notin \mathbf{UseSetI}(S2, s'') \\
& \vdash \langle S1\, S2, (s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(5.122) \Rrightarrow & \langle S1, s\rangle \rightsquigarrow s'', \langle S2, s''\rangle \rightsquigarrow s', \\
& o \notin \mathbf{UseSetI}(S1, s), o \notin \mathbf{UseSetI}(S2, s'') \\
& \vdash \langle S1, (s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow (s''_H\langle o.a := r\rangle, s''_E, s''_G) \wedge \\
& \langle S2, (s''_H\langle o.a := r\rangle, s''_E, s''_G)\rangle \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(IH2) \Rrightarrow & \text{TRUE} \quad \square
\end{array}
$$

Assignment to local:

$$
\begin{array}{rl}
& \langle l := E, s\rangle \rightsquigarrow s', o \notin \mathbf{UseSetI}(l := E, s) \\
& \vdash \langle l := E, (s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(5.123) \Rrightarrow & [\![E, s]\!] \rightsquigarrow s'', s' = (s''_H, s''_E\langle l := \mathbf{eval}(E, s)\rangle, s''_G), \\
& o \notin \mathbf{UseSetI}(l := E, s) \\
& \vdash \langle l := E, (s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(7.24) \Rrightarrow & [\![E, s]\!] \rightsquigarrow s'', s' = (s''_H, s''_E\langle l := \mathbf{eval}(E, s)\rangle, s''_G), \\
& o \notin \mathbf{UseSetE}(E, s) \\
& \vdash \langle l := E, (s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G)
\end{array}
$$

$$
\begin{array}{rl}
(5.123) \Rrightarrow & [\![E, s]\!] \rightsquigarrow s'', s' = (s''_H, s''_E\langle l := \mathbf{eval}(E, s)\rangle, s''_G), \\
& o \notin \mathbf{UseSetE}(E, s), \\
& [\![E, s_H\langle o.a := r\rangle, s_E, s_G]\!] \rightsquigarrow s''' \\
& \vdash (s'\langle o.a := r\rangle, s'_E, s'_G) = (s'''_H, s'''_E\langle l := \mathbf{eval}(E, s_H\langle o.a := r\rangle, s_E, s_G)\rangle, s'''_G)) \\
(IH1) \Rrightarrow & [\![E, s]\!] \rightsquigarrow s'', s' = (s''_H, s''_E\langle l := \mathbf{eval}(E, s)\rangle, s''_G), \\
& o \notin \mathbf{UseSetE}(E, s), \\
& [\![E, s_H\langle o.a := r\rangle, s_E, s_G]\!] \rightsquigarrow s''' \\
& \vdash (s'\langle o.a := r\rangle, s'_E, s'_G) = (s'''_H, s'''_E\langle l := \mathbf{eval}(E, s)\rangle, s'''_G)) \\
(IH3) \Rrightarrow & [\![E, s_H\langle o.a := r\rangle, s_E, s_G]\!] \rightsquigarrow s'', \\
& (s'\langle o.a := r\rangle, s'_E, s'_G) = (s''_H, s''_E\langle l := \mathbf{eval}(E, s)\rangle, s''_G), \\
& o \notin \mathbf{UseSetE}(E, s), \\
& [\![E, s_H\langle o.a := r\rangle, s_E, s_G]\!] \rightsquigarrow s''' \\
& \vdash (s'\langle o.a := r\rangle, s'_E, s'_G) = (s'''_H, s'''_E\langle l := \mathbf{eval}(E, s)\rangle, s'''_G)) \\
\Rrightarrow & \text{TRUE} \quad \square
\end{array}
$$

Assignment to attribute:

$$
\begin{array}{rl}
& \langle a := l, s\rangle \rightsquigarrow s', o \notin \mathbf{UseSetI}(l := E, s) \\
& \vdash \langle a := l, s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G \\
(5.124) \Rrightarrow & o \notin \mathbf{UseSetI}(l := E, s) \\
& \vdash (s_H\langle o.a := r\rangle\langle s_E(\texttt{Current}).a := s_E(l)\rangle, s_E, s_G) = \\
& \quad (s_H\langle s_E(\texttt{Current}).a := s_E(l)\rangle\langle o.a := r\rangle, s_E, s_G) \\
(7.25) \Rrightarrow & o \neq s_E(\texttt{Current}) \\
& \vdash (s_H\langle o.a := r\rangle\langle s_E(\texttt{Current}).a := s_E(l)\rangle, s_E, s_G) = \\
& \quad (s_H\langle s_E(\texttt{Current}).a := s_E(l)\rangle\langle o.a := r\rangle, s_E, s_G) \\
(5.72) \Rrightarrow & o \neq s_E(\texttt{Current}) \\
& \vdash (s_H\langle o.a := r\rangle\langle s_E(\texttt{Current}).a := s_E(l)\rangle, s_E, s_G) = \\
& \quad (s_H\langle o.a := r\rangle\langle s_E(\texttt{Current}).a := s_E(l)\rangle, s_E, s_G) \quad \square
\end{array}
$$

True case:

$$
\begin{array}{l}
s_E(C) = \text{TRUE}, \\
\langle \texttt{if}\ C\ \texttt{then}\ S1\ \texttt{else}\ S2\ \texttt{end}, s\rangle \rightsquigarrow s', \\
o \notin \mathbf{UseSetI}(\texttt{if}\ C\ \texttt{then}\ S1\ \texttt{else}\ S2\ \texttt{end}, s) \\
\vdash \langle \texttt{if}\ C\ \texttt{then}\ S1\ \texttt{else}\ S2\ \texttt{end}, s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow \\
\quad (s'_H\langle o.a := r\rangle, s'_E, s'_G)
\end{array}
$$

$$
\begin{aligned}
(5.125) \Rrightarrow\ & s_E(C) = \text{TRUE},\\
& \langle S1, s\rangle \rightsquigarrow s',\\
& o \notin \mathbf{UseSetI}(\texttt{if}\ C\ \texttt{then}\ S1\ \texttt{else}\ S2\ \texttt{end}, s)\\
& \vdash \langle S1, s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G)\\
(7.26) \Rrightarrow\ & s_E(C) = \text{TRUE}\\
& \langle S1, s\rangle \rightsquigarrow s',\\
& o \notin \mathbf{UseSetI}(S1, s)\\
& \vdash \langle S1, s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G)\\
(IH2) \Rrightarrow\ & \text{TRUE} \quad \square
\end{aligned}
$$

(false case the same way)

Loop continuation:

$$
\begin{aligned}
& s_E(C) = \text{FALSE},\\
& \langle \mathbf{from\ until}\ C\ \mathbf{invariant}\ I\ \mathbf{variant}\ V\ \mathbf{loop}\ S\ \mathbf{end}, s\rangle \rightsquigarrow s',\\
& o \notin \mathbf{UseSetI}(\mathbf{from\ until}\ C\ \mathbf{invariant}\ I\ \mathbf{variant}\ V\ \mathbf{loop}\ S\ \mathbf{end}, s)\\
& \vdash \langle \mathbf{from\ until}\ C\ \mathbf{invariant}\ I\ \mathbf{variant}\ V\ \mathbf{loop}\ S\ \mathbf{end}, s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow\\
& \quad (s'_H\langle o.a := r\rangle, s'_E, s'_G)\\
(5.127) \Rrightarrow\ & s_E(C) = \text{FALSE},\\
& \langle S, s\rangle \rightsquigarrow s'',\\
& \langle \mathbf{from\ until}\ C\ \mathbf{invariant}\ I\ \mathbf{variant}\ V\ \mathbf{loop}\ S\ \mathbf{end}, s''\rangle \rightsquigarrow s',\\
& o \notin \mathbf{UseSetI}(\mathbf{from\ until}\ C\ \mathbf{invariant}\ I\ \mathbf{variant}\ V\ \mathbf{loop}\ S\ \mathbf{end}, s),\\
& \langle S, s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow s'''\\
& \vdash \langle \mathbf{from\ until}\ C\ \mathbf{invariant}\ I\ \mathbf{variant}\ V\ \mathbf{loop}\ S\ \mathbf{end}, s'''\rangle \rightsquigarrow\\
& \quad (s'_H\langle o.a := r\rangle, s'_E, s'_G)\\
(7.28) \Rrightarrow\ & s_E(C) = \text{FALSE},\\
& \langle S, s\rangle \rightsquigarrow s'',\\
& \langle \mathbf{from\ until}\ C\ \mathbf{invariant}\ I\ \mathbf{variant}\ V\ \mathbf{loop}\ S\ \mathbf{end}, s''\rangle \rightsquigarrow s',\\
& o \notin \mathbf{UseSetI}(S, s),\\
& o \notin \mathbf{UseSetI}(\mathbf{from\ until}\ C\ \mathbf{invariant}\ I\ \mathbf{variant}\ V\ \mathbf{loop}\ S\ \mathbf{end}, s'),\\
& \langle S, s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow s'''\\
& \vdash \langle \mathbf{from\ until}\ C\ \mathbf{invariant}\ I\ \mathbf{variant}\ V\ \mathbf{loop}\ S\ \mathbf{end}, s'''\rangle \rightsquigarrow\\
& \quad (s'_H\langle o.a := r\rangle, s'_E, s'_G)\\
(IH2) \Rrightarrow\ & s_E(C) = \text{FALSE},
\end{aligned}
$$

$$
\begin{array}{ll}
& \langle S, s\rangle \rightsquigarrow s'', \\
& \langle \textbf{from until}\, C\, \textbf{invariant}\, I\, \textbf{variant}\, V\, \textbf{loop}\, S\, \textbf{end}, s''\rangle \rightsquigarrow s', \\
& o \notin \textbf{UseSetI}(S, s), \\
& o \notin \textbf{UseSetI}(\textbf{from until}\, C\, \textbf{invariant}\, I\, \textbf{variant}\, V\, \textbf{loop}\, S\, \textbf{end}, s'), \\
& \langle S, s_H, s_E, s_G)\rangle \rightsquigarrow s''' \\
& \vdash \langle \textbf{from until}\, C\, \textbf{invariant}\, I\, \textbf{variant}\, V\, \textbf{loop}\, S\, \textbf{end}, s'''\rangle \rightsquigarrow \\
& \quad (s'_H, s'_E, s'_G) \\
\Rrightarrow & \text{TRUE} \quad \square
\end{array}
$$

Loop termination:

$$
\begin{array}{ll}
& s_E(C) = \text{TRUE} \\
& \langle \textbf{from until}\, C\, \textbf{invariant}\, I\, \textbf{variant}\, V\, \textbf{loop}\, S\, \textbf{end}, s\rangle \rightsquigarrow s', \\
& o \notin \textbf{UseSetI}(\textbf{from until}\, C\, \textbf{invariant}\, I\, \textbf{variant}\, V\, \textbf{loop}\, S\, \textbf{end}, s) \\
& \vdash \langle \textbf{from until}\, C\, \textbf{invariant}\, I\, \textbf{variant}\, V\, \textbf{loop}\, S\, \textbf{end}, s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow \\
& \quad (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(5.127)\ \Rrightarrow & s_E(C) = \text{TRUE}, \\
& \langle , s\rangle \rightsquigarrow s', \\
& o \notin \textbf{UseSetI}(\textbf{from until}\, C\, \textbf{invariant}\, I\, \textbf{variant}\, V\, \textbf{loop}\, S\, \textbf{end}, s) \\
& \vdash \langle , s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow \\
& \quad (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(5.121)\ \Rrightarrow & (s_H\langle o.a := r\rangle, s_E, s_G) = (s_H\langle o.a := r\rangle, s_E, s_G) \\
\Rrightarrow & \text{TRUE} \quad \square
\end{array}
$$

Feature invocation:

$$
\begin{array}{ll}
& \langle l.f(l^1, \ldots, l^n), s\rangle \rightsquigarrow s', \\
& o \notin \textbf{UseSetI}(l.f(l^1, \ldots, l^n), s) \\
& \vdash \langle l.f(l^1, \ldots, l^n), s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(5.129)\ \Rrightarrow & B = \mathsf{body}(\mathsf{version}(\mathsf{typeof}(s_E(l)))(f)), \\
& \langle B, (s_H, \langle \texttt{Current} := s_E(l), arg^1 := s_E(a^1), \ldots, arg^n := s_E(a^n)\rangle, s_G)\rangle \rightsquigarrow \\
& \quad (s'_H, s''_E, s'_G), \\
& o \notin \textbf{UseSetI}(l.f(l^1, \ldots, l^n), s) \\
& \vdash \langle B, s_H\langle o.a := r\rangle, \\
& \qquad \langle \texttt{Current} := s_E(l), arg^1 := s_E(a^1), \ldots, arg^n := s_E(a^n)\rangle, s_G)\rangle \rightsquigarrow \\
& \quad (s'_H\langle o.a := r\rangle, s''_E, s'_G)
\end{array}
$$

$$
\begin{array}{rl}
(7.30) \Rrightarrow & B = \mathsf{body}(\mathsf{version}(\mathsf{typeof}(s_E(l)))(f)),\\
& \langle B, (s_H, \langle \texttt{Current} := s_E(l), arg^1 := s_E(a^1), \ldots, arg^n := s_E(a^n)\rangle, s_G)\rangle \rightsquigarrow\\
& (s'_H, s''_E, s'_G),\\
& o \notin \mathbf{UseSetI}(B, (s_H,\\
& \quad \langle \texttt{Current} := s_E(l), arg^1 := s_E(a^1), \ldots, arg^n := s_E(a^n)\rangle, s_G))\\
& \vdash \langle B, s_H\langle o.a := r\rangle,\\
& \quad \langle \texttt{Current} := s_E(l), arg^1 := s_E(a^1), \ldots, arg^n := s_E(a^n)\rangle, s_G)\rangle \rightsquigarrow\\
& (s'_H\langle o.a := r\rangle, s''_E, s'_G)\\
(IH2) \Rrightarrow & \text{TRUE} \quad \square
\end{array}
$$

Proofs for (IH3)

Local read:

$$
\begin{array}{rl}
& [\![l, s]\!] \rightsquigarrow s', o \notin \mathbf{UseSetE}(l, s)\\
& \vdash [\![l, (s_H\langle o.a := r\rangle, s_E, s_G)]\!] \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G)\\
(5.112) \Rrightarrow & o \notin \mathbf{UseSetE}(l, s)\\
& \vdash (s_H\langle o.a := r\rangle, s_E, s_G) = (s_H\langle o.a := r\rangle, s_E, s_G) \quad \square
\end{array}
$$

Reference equality:

$$
\begin{array}{rl}
& [\![l = l', s]\!] \rightsquigarrow s', o \notin \mathbf{UseSetE}(l = l', s)\\
& \vdash [\![l = l', (s_H\langle o.a := r\rangle, s_E, s_G)]\!] \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G)\\
(5.113) \Rrightarrow & o \notin \mathbf{UseSetE}(l = l', s)\\
& \vdash (s_H\langle o.a := r\rangle, s_E, s_G) = (s_H\langle o.a := r\rangle, s_E, s_G) \quad \square
\end{array}
$$

Expression assignment:

$$
\begin{array}{rl}
& [\![l := E1; E2, s]\!] \rightsquigarrow s', o \notin \mathbf{UseSetE}(l := E1; E2, s)\\
& \vdash [\![l := E1; E2, (s_H\langle o.a := r\rangle, s_E, s_G)]\!] \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G)\\
(5.115) \Rrightarrow & \langle l := E1, s\rangle \rightsquigarrow s'', [\![E2, s'']\!] \rightsquigarrow s', o \notin \mathbf{UseSetE}(l := E1; E2, s)\\
& \langle l := E1, (s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow (s'''_H, s'''_E, s'''_G)\\
& \vdash [\![E2, s'''']\!] \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G)\\
(7.16) \Rrightarrow & \langle l := E1, s\rangle \rightsquigarrow s'', [\![E2, s'']\!] \rightsquigarrow s',
\end{array}
$$

$$
\begin{array}{rl}
& o \notin \mathbf{UseSetI}(l := E1, s) \\
& o \notin \mathbf{UseSetE}(E2, s'') \\
& \langle l := E1, (s_H\langle o.a := r\rangle, s_E, s_G)\rangle \rightsquigarrow s''' \\
& \vdash [\![E2, s''']\!] \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(IH2) \Rrightarrow & \langle l := E1, s\rangle \rightsquigarrow s'', [\![E2, s'']\!] \rightsquigarrow s', \\
& o \notin \mathbf{UseSetE}(E2, s'') \\
& \vdash [\![E2, s''_H\langle o.a := r\rangle, s''_E, s''_G)]\!] \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(IH3) \Rrightarrow & \vdash \mathrm{TRUE} \quad \square
\end{array}
$$

Create Expressions:

$$
\begin{array}{rl}
& [\![\mathsf{create}\, c(l^1, \ldots, l^n), s]\!] \rightsquigarrow s', \\
& o \notin \mathbf{UseSetE}(\mathsf{create}\, c(l^1, \ldots, l^n), s) \\
& \vdash [\![\mathsf{create}\, c(l^1, \ldots, l^n), (s_H\langle o.a := r\rangle, s_E, s_G)]\!] \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(5.119) \Rrightarrow & E = \langle \mathtt{Current} := \mathsf{new}(s), arg^1 := s_E(l^1), \ldots, arg^n := s_E(l^n)\rangle \\
& \langle \mathsf{body}(c), (s_H\langle \mathsf{defIn}(c)\rangle, E, s_G)\rangle \rightsquigarrow (s'_H, E, s'_G) \\
& o \notin \mathbf{UseSetE}(\mathsf{create}\, c(l^1, \ldots, l^n), s) \\
& \vdash \langle \mathsf{body}c, (s_H\langle \mathsf{defIn}(c)\rangle\langle o.a := r\rangle, E, s_G)\rangle \rightsquigarrow (s'_H\langle o.a := r\rangle, E, s'_G) \\
(7.17) \Rrightarrow & E = \langle \mathtt{Current} := \mathsf{new}(s), arg^1 := s_E(l^1), \ldots, arg^n := s_E(l^n)\rangle \\
& \langle \mathsf{body}(c), (s_H\langle \mathsf{defIn}(c)\rangle, E, s_G)\rangle \rightsquigarrow (s'_H, E, s'_G) \\
& o \notin \mathbf{UseSetE}(\mathsf{body}(c), (s_H\langle \mathsf{defIn}(c)\rangle, E, s_G)) \\
& \vdash \langle \mathsf{body}c, (s_H\langle \mathsf{defIn}(c)\rangle\langle o.a := r\rangle, E, s_G)\rangle \rightsquigarrow (s'_H\langle o.a := r\rangle, E, s'_G) \\
(IH2) \Rrightarrow & \vdash \mathrm{TRUE} \quad \square
\end{array}
$$

Manifest Constants:

$$
\begin{array}{rl}
& [\![MC, s]\!] \rightsquigarrow s', o \notin \mathbf{UseSetE}(MC, s) \\
& \vdash [\![MC, (s_H\langle o.a := r\rangle, s_E, s_G)]\!] \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(5.120) \Rrightarrow & o \notin \mathbf{UseSetE}(MC, s) \\
& \vdash (s_H\langle o.a := r\rangle, s_E, s_G) = (s_H\langle o.a := r\rangle, s_E, s_G) \quad \square
\end{array}
$$

Attribute read:

$$
\begin{array}{rl}
& [\![l.a, s]\!] \rightsquigarrow s', o \notin \mathbf{UseSetE}(l.a, s) \\
& \vdash [\![l.a, (s_H\langle o.a := r\rangle, s_E, s_G)]\!] \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(5.120) \Rrightarrow & o \notin \mathbf{UseSetE}(MC, s) \\
& \vdash (s_H\langle o.a := r\rangle, s_E, s_G) = (s_H\langle o.a := r\rangle, s_E, s_G) \quad \square
\end{array}
$$

Query call:

$$
\begin{aligned}
& f \notin \mathit{OnceFunction} \\
& [\![l.f(l^1,\ldots,l^n), s]\!] \rightsquigarrow s', o \notin \mathbf{UseSetE}(l.f(l^1,\ldots,l^n), s) \\
& \vdash [\![l.f(l^1,\ldots,l^n), (s_H\langle o.a := r\rangle, s_E, s_G)]\!] \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(5.116) \Rrightarrow\ & B = \mathsf{body}(\mathsf{version}(\mathsf{typeof}(s_E(l)))(f)), \\
& E = \langle \texttt{Current} := s_E(l), arg^1 := s_E(l^1), \ldots, arg^n := s_E(l^n)\rangle, \\
& [\![B, (s_H, E, s_G)]\!] \rightsquigarrow (s'_H, s'_E, s'_G), o \notin \mathbf{UseSetE}(l.f(l^1,\ldots,l^n), s) \\
& \vdash [\![B, (s_H\langle o.a := r\rangle, E, s_G)]\!] \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(7.20) \Rrightarrow\ & B = \mathsf{body}(\mathsf{version}(\mathsf{typeof}(s_E(l)))(f)), \\
& E = \langle \texttt{Current} := s_E(l), arg^1 := s_E(l^1), \ldots, arg^n := s_E(l^n)\rangle, \\
& [\![B, (s_H, E, s_G)]\!] \rightsquigarrow (s'_H, s'_E, s'_G), o \notin \mathbf{UseSetI}(l.f(l^1,\ldots,l^n), (s_H, E, s_G)) \\
& \vdash [\![B, (s_H\langle o.a := r\rangle, E, s_G)]\!] \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(7.20) \Rrightarrow\ & \mathrm{TRUE} \quad \square
\end{aligned}
$$

Query call of new once value:

$$
\begin{aligned}
& f \in \mathit{OnceFunction}, \neg\mathsf{stored}(s_G, f), \\
& [\![l.f(l^1,\ldots,l^n), s]\!] \rightsquigarrow s', o \notin \mathbf{UseSetE}(l.f(l^1,\ldots,l^n), s) \\
& \vdash [\![l.f(l^1,\ldots,l^n), (s_H\langle o.a := r\rangle, s_E, s_G)]\!] \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(5.117) \Rrightarrow\ & B = \mathsf{body}(\mathsf{version}(\mathsf{typeof}(s_E(l)))(f)), \\
& E = \langle \texttt{Current} := s_E(l), arg^1 := s_E(l^1), \ldots, arg^n := s_E(l^n)\rangle, \\
& [\![B, (s_H, E, s_G)]\!] \rightsquigarrow (s'_H, s'_E, s'_G), o \notin \mathbf{UseSetE}(l.f(l^1,\ldots,l^n), s) \\
& \vdash [\![B, (s_H\langle o.a := r\rangle, E, s_G)]\!] \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(7.20) \Rrightarrow\ & B = \mathsf{body}(\mathsf{version}(\mathsf{typeof}(s_E(l)))(f)), \\
& E = \langle \texttt{Current} := s_E(l), arg^1 := s_E(l^1), \ldots, arg^n := s_E(l^n)\rangle, \\
& [\![B, (s_H, E, s_G)]\!] \rightsquigarrow (s'_H, s'_E, s'_G), o \notin \mathbf{UseSetI}(l.f(l^1,\ldots,l^n), (s_H, E, s_G)) \\
& \vdash [\![B, (s_H\langle o.a := r\rangle, E, s_G)]\!] \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(7.20) \Rrightarrow\ & \mathrm{TRUE} \quad \square
\end{aligned}
$$

Query call of old once value:

$$
\begin{aligned}
& f \in \mathit{OnceFunction}, \mathsf{stored}(s_G, f), \\
& [\![l.f(l^1,\ldots,l^n), s]\!] \rightsquigarrow s', o \notin \mathbf{UseSetE}(l.f(l^1,\ldots,l^n), s) \\
& \vdash [\![l.f(l^1,\ldots,l^n), (s_H\langle o.a := r\rangle, s_E, s_G)]\!] \rightsquigarrow (s'_H\langle o.a := r\rangle, s'_E, s'_G) \\
(5.120) \Rrightarrow\ & o \notin \mathbf{UseSetE}(MC, s) \\
& \vdash (s_H\langle o.a := r\rangle, s_E, s_G) = (s_H\langle o.a := r\rangle, s_E, s_G) \quad \square
\end{aligned}
$$

Through the repeated application of the proved theorem, we can show that the attributes of all objects outside of the use frame can change without changing the result of the expression:

$$\begin{gathered}\mathbf{eval}(E,s) = res \wedge \\ \forall o \in \mathbf{UseSetE}(E,s), a \in Attribute | s_H(o.a) = s'_H(o.a) \\ \Rightarrow \mathbf{eval}(E,(s'_H, s_E, s_G)) = res\end{gathered}$$

The two theorems just proved are used to show the main property of the frame rule (7.45): if the frame of a command and a query are disjoint, then calling the command will not change the result value of the query.

$$\begin{aligned}
& \forall c : Routine, q, qf, cf : Query, s : State | \\
& \langle c, s\rangle \rightsquigarrow s' \wedge P(\mathbf{eval}(q,s)) \wedge (\mathbf{eval}(qf,s) \cap \mathbf{eval}(cf,s) = \varnothing) \Rightarrow \\
& P(\mathbf{eval}(q,s')) \\
(7.44) \Rrightarrow\ & \langle c, s\rangle \rightsquigarrow s' \wedge P(\mathbf{eval}(q,s)) \wedge \\
& (\mathbf{eval}(qf,s) \cap \mathbf{ModifySet}(cf,s) \cap Allocated(s) = \varnothing) \Rightarrow \\
& P(\mathbf{eval}(q,s')) \\
(7.43) \Rrightarrow\ & \langle c, s\rangle \rightsquigarrow s' \wedge P(\mathbf{eval}(q,s)) \wedge \\
& (\mathbf{UseSetE}(qf,s) \cap \mathbf{ModifySet}(cf,s) \cap Allocated(s) = \varnothing) \Rightarrow \\
& P(\mathbf{eval}(q,s')) \\
Theorem1 \Rrightarrow\ & \langle c, s\rangle \rightsquigarrow s' \wedge P(\mathbf{eval}(q,s)) \wedge \\
& (\mathbf{UseSetE}(qf,s) \cap \mathbf{ModifySet}(cf,s) \cap Allocated(s) = \varnothing) \wedge \\
& (\forall o : Obj, a : Attribute | \mathbf{eval}(q,s) = res \wedge o \notin \mathbf{eval}(\mathsf{use}(c), s) \Rightarrow \\
& \quad \mathbf{eval}(q, (s_H\langle o.a := r\rangle, s_E, s_G)) = res) \Rightarrow \\
& P(\mathbf{eval}(q,s')) \\
(5.110) \Rrightarrow\ & \langle c, s\rangle \rightsquigarrow (s'_H, s_E, s_G) \wedge P(\mathbf{eval}(q,s)) \wedge \\
& (\mathbf{UseSetE}(qf,s) \cap \mathbf{ModifySet}(cf,s) \cap Allocated(s) = \varnothing) \wedge \\
& (\forall o : Obj, a : Attribute | \mathbf{eval}(q,s) = res \wedge o \notin \mathbf{eval}(\mathsf{use}(c), s) \Rightarrow \\
& \quad \mathbf{eval}(q, (s_H\langle o.a := r\rangle, s_E, s_G)) = res) \Rightarrow \\
& P(\mathbf{eval}(q, (s'_H, s_E, s'_G))) \\
(5.138) \Rrightarrow\ & \langle c, s\rangle \rightsquigarrow (s'_H, s_E, s_G) \wedge P(\mathbf{eval}(q,s)) \wedge \\
& (\mathbf{UseSetE}(qf,s) \cap \mathbf{ModifySet}(cf,s) \cap Allocated(s) = \varnothing) \wedge \\
& (\forall o : Obj, a : Attribute | \mathbf{eval}(q,s) = res \wedge o \notin \mathbf{eval}(\mathsf{use}(c), s) \Rightarrow \\
& \quad \mathbf{eval}(q, (s_H\langle o.a := r\rangle, s_E, s_G)) = res) \Rightarrow
\end{aligned}$$

$$
\begin{aligned}
& P(\mathbf{eval}(q,(s'_H,s_E,s_G))) \\
\mathit{Theorem2} \Rrightarrow\ & \langle c,s\rangle \rightsquigarrow (s'_H,s_E,s_G) \wedge P(\mathbf{eval}(q,s)) \wedge \\
& (\mathbf{UseSetE}(qf,s) \cap \mathbf{ModifySet}(cf,s) \cap \mathit{Allocated}(s) = \varnothing) \wedge \\
& (\forall o : \mathit{Obj}, a : \mathit{Attribute} | \mathbf{eval}(q,s) = res \wedge o \notin \mathbf{eval}(\mathsf{use}(c),s) \Rightarrow \\
& \quad \mathbf{eval}(q,(s_H\langle o.a := r\rangle, s_E, s_G)) = res) \wedge \\
& \mathbf{eval}(q,s) = \mathbf{eval}(q,(s'_H,s_E,s_G)) \Rightarrow \\
& P(\mathbf{eval}(q,(s'_H,s_E,s_G))) \quad \square
\end{aligned}
$$

7.12 Frames and Inheritance

Frame specifications work well together with inheritance. There is a clear understanding of what makes up a *stronger* specification when it comes to frames: the smaller the frame, the stronger the contract.

Respecting the frames is an obligation of the supplier. Subtyping rules enforces the strengthening of supplier obligations in subtypes. This means that the frame specifications of subtypes have to be subsets of the frame specifications in supersets.

As already with existing contracts, proving that a given frame is a subset of another frame $f \subseteq g$ might not be feasible. Instead we suggest to use a similar mechanisms of combining new frame specifications with existing specifications that will always result in a stronger contract.

We make use of the property of frames that the intersection of two sets is always a subset of each of these sets:

$$\forall A, B : \texttt{FRAME} | A \cap B \subseteq A$$

Whenever we inherit a frame specification, the unfolded contract form is created by intersecting the new frame with the inherited one. To make it easier for the developer to see that the inherited frame needs to be combined with the frame of the parent, the `only` keyword introduced by ECMA is reused. Listing 7.6 illustrates this. The unfolded form of `B`s contract is shown in listing 7.7.

7.13 Applying frames

Reasoning with frames is driven by establishing and maintaining the disjointness of frames. Kassios introduced and proved a number of important rules about frames (see [37, sec. 5.3]). These rules shown in section 7.5.3.

Translated into the notation of dynamic frame contracts, we get a rule specific for object creation:

$$\frac{c\,\mathbf{ensure\,confined}\,f \quad c\,\mathbf{modify},f \quad g\,\mathbf{use}\,g}{\{true\} \quad n := \mathbf{create}\,c \quad \{t \neq n \Rightarrow t.g \cap n.f = \varnothing\}} \tag{7.49}$$

and a rule for command invocations:

$$\frac{c\,\mathbf{ensure\,confined}\,f \quad c\,\mathbf{modify}\,f \quad g\,\mathbf{use}\,g}{\{g \cap f = \varnothing\} \quad t.c(a) \quad \{g \cap f = \varnothing\}} \tag{7.50}$$

All premises of these rules are state independent. Using these two rules, it is very easy to reason about encapsulated data structures (see section 7.14).

7.13.1 Contracts for frames

As for any other feature, the supplier may support the client in using a component by specifying contracts for frame queries.

- A *precondition* for a frame query suggests that this frame can only accessed in specific situations.
- A *postcondition* gives information about the content of the frame to the user. If the frame query has not arguments, the postcondition can also easily be turned into a class invariant.
- A *use frame* limits the abstraction function to only use the resources specified in its use set.

The self-framing frame (`f use f`) is probably the contract of a frame query encountered most often. Adding preconditions is dangerous: frame are only used in contracts and partial functions in contracts are difficult to resolve (see section 3.4).

Postconditions or invariants that give a lower bound for the frame (for example "at least x is contained in the frame g") are useless to the client. The client is interested in an upper bound, as he has to try to maintain disjointness of frames.

For example, lets consider the `LINKABLE[G]` class from EiffelBase. The purpose of this class is to construct the cells that make up a linked list. If we describe the frame of the cell as

```
feature -- Frame

 representation: FRAME
  use
    representation
```

we provide a very weak contract, describing that the cell can use an arbitrary amount of objects to implement its contract and it is the job the client (the actual linked-list object) to keep these frames disjoint.

If we instead explicitly state that the frame of the `LINKABLE[G]` is limited to the current object, using the following contract

```
feature -- Frame

 representation: FRAME
  ensure
    only_current: Result = { Current }
```

it becomes trivial to derive that the frames of two cells are disjoint by comparing the references:

$$c1 \neq c2 \Rightarrow c1.\text{representation} \cap c2.\text{representation} = \varnothing$$

This contract limits all implementations of cells to only use fields of the current object and no "helper objects".

7.14 Patterns of frames

After applying frames on numerous problems, it seems obvious that there are a number of patterns that are encountered repeatedly. We call these pattern *Framing Patterns*, similar to the concept of Design Patterns [30]. This section gives an overview about frame patterns with examples, without any claim of completeness.

7.14.1 Encapsulation

The most common pattern that we have identified is the encapsulation pattern. In the encapsulation pattern, there is a part of the state that is the *internal representation* of the object state. In terms of models as introduced in chapter 6, they are the concrete implementation of the model.

The internal representation has the following properties:

- There is a frame, called the *representation frame*. This frame is self-framing, meaning its size is defined by the objects it frames.

- On object creation, the representation frame is only populated with new object, including the object that is just being created.
- Queries that only depend on the model are only using the representation frame.
- Commands that only change the model are framed to only modify the representation frame.
- All commands of the class confine the representation by specifying **`confined`** *`representation`* in the post-condition.

A typical example of a class implementing this pattern is given in listing 7.8. It shows how the command and the query implement are framed by the representation. The return value of *`has`* only depends on the representation, and the command *`extend`* only modifies the representation.

7.14.2 Friend

A friend is a class that may violate the encapsulation pattern given above: it operates on some other class (the subject) and may change the state of it. Friendship can (but does not have to) be expressed by allowing access to the internal representation using export restrictions of the subject.

A typical friend is a cursor. It has a state of it own (a current cursor position) but offers a number of features that depend on or affect the representation of the subject.

Friends are normally created by the subject through a factory query (for example *`new_cursor`*`:`*`CURSOR`*) or the subject is passed to the friend on creation as a target for operations.

- A *friend* has an *internal representation* frame and a query to access the subject. The own representation and the representation of the subject are disjoint.
- The internal representation behaves as described in the encapsulation pattern.
- Some features only use or modify the own representation. These are *internal features*. Calling these internal features does not effect the representation of the subject.
- Features that access or modify the representation of the subject are *subject-dependent features*.

Listing 7.9 shows the class *ELEMENT_REMOVER* that will remove a fixed element from the given integer set.

7.14.3 View

A view is very similar to the friend, except that it never modifies the subject. That means that a modify clauses referencing the representation of the subject is not valid. Iterators are typical views.

The listing 7.10 shows the contract of an iterator that operates on an *INTEGER_SET* as described in listing 7.8.

The contract clearly describes how the set and the iterator class relate: any operation on the set will invalidate the iterator, changing its model arbitrarily (**use** frames of the models of the iterator). It is common that the documentation for iterators specifies that during iteration the underlying data structure must not be changed.

Views are always independent from each other, even if they operate on the same subject and thus have an aliased state. Two different iterators can operate independently, as the *forth* operation only modifies the local, encapsulated representation.

7.14.4 Proxy

The proxy is a friend that does not have a state of its own, but instead completely relies on the state of another object.

- The proxy does not define a frame of its own.
- The subject to operate on is passed during creation.
- All commands and queries **use** and **modify** the representation of the subject.

A typical scenario where is proxy is used for is interface adaption. To interface C, many Eiffel libraries first wrap the low-level C interface in one class and then provide a second class as a client to the first one implementing a nice, object-oriented interface.

In the EiffelVision 2 library, all widgets are implemented as proxies through the bridge design pattern.

Proxies are also used if access to the real object is costly. Proxies may implement caching or on-demand techniques to give a fast interface to the clients.

7.14.5 Wrapper

A wrapper completely encloses the subject without revealing that it is actually operating on some enclosed entity. Because the reference to the subject is secret, the wrapper needs to include the subjects representation into its own (if it has the need for an own representationx).

```
feature -- Frame

 representation: FRAME
  do
   Result := subject.representation
  end
```

A complete description of the relation between wrapper and subject is normally given on object creation, but is not visible in the commands and queries of the class.

The wrapper is very commonly used in stream-based class structures. An abstract *INPUT_STREAM* is provided, but the contract does not give information where the stream is read from. A subclass, for example *FILE_INPUT_STREAM* then gets a file handle on creation that it will use as a resource for the data.

The creation procedure of *FILE_INPUT_STREAM* will have a post-condition like the following:

```
feature

 make (a_file: FILE) is
  modify
   own_representation: representation
  ensure
   representation: representation = subject.
      representation
```

This information is not available to clients that use the abstract version of *INPUT_STREAM*. If potential aliasing should be avoided, clients have to establish this information through different means, for example through a precondition (as it was done in listing 2.9)

7.15 Runtime monitoring

It is possible to runtime monitor dynamic frame contracts. This section contains a sketch how this can be achieved.

At the start of a routine execution, the frame expressions have to be evaluated and stored in temporary variables. The runtime systems needs to record all objects that were created during the execution of the routine.

On every call to a query, the use frame of the called query, minus the set of new objects, has to be a subset of the use frame of the calling feature. On every call to a command, the use and modify frames of the called command, minus the set of new objects, have to be subsets of the use and modify frame of the calling feature.

Assigning to an attribute requires that `Current` is contained in the modify set. Accessing an attribute requires that `Current` is contained in the use set.

Should any of these subset relations become violated, a runtime assertion violation is raised.

Runtime monitoring of frames has not been implemented as part of this thesis, as it requires modifications to the runtime system to keep track of newly created objects.

7.16 Summary

Dynamic frame contracts are a powerful technique to solve the problems exposed by the frame problem. They are a technique that allows to add information about possible interference without any violation of information hiding. With frames, modular reasoning and verification of classes that are using multiple other classes for their implementation becomes possible.

Frames do not prevent any aliasing. They do not restrict the object structure. They expose knowledge about read and write effects of features.

We have identified a number of common patterns of using dynamic frame contracts. There is potential on improving the usability of dynamic frame contracts by adding extra language constructs that implement the patterns, possibly using the type system.

```
class A

feature
  f
    use
      a
    modify
      a
    deferred
    end

  a: FRAME
end

class B

inherit
  A
    redefine f end

feature
  f
    use only
      b
    modify only
      b
    deferred
    end

  b: FRAME
end
```

Listing 7.6: Frame specifications and inheritance

```
class B

feature
  f
    use
      b ∩ a
    modify
      b ∩ a
    deferred
    end

  a: FRAME
  b: FRAME
end
```

Listing 7.7: Unfolded inherited frame specification

```
class INTEGER_SET
create make_empty

feature -- Initialization
  make_empty
    modify
      representation
    ensure
      model_definition: model.is_empty
      frame_confined: confined representation

feature -- Model and Frame
  model: MML_SET[INTEGER]

  representation: FRAME
    use
      self_framing: representation

feature -- Access
  has (v: INTEGER)
    use
      representation
    ensure
      model_definition: Result = model.has (v)

feature -- Modification
  extend (v: INTEGER)
    require
      not_contained: not has (v)
    use
      representation
    modify
      representation
    ensure
      model_definition: model = old model.extended (v)
      frame_confined: confined representation
end
```

Listing 7.8: Encapsulation of an integer set

```
class ELEMENT_REMOVER
create make

feature -- Initialization
  make (a_subject: INTEGER_SET, a_value:INTEGER)
    modify
      only_local: representation
    ensure
      subject_set: subject = a_subject
      frame_confined: confined representation

feature -- Models and Frames
  model: INTEGER
    use
      only_local: representation
  representation: FRAME
    use
      self_framing: representation

feature -- Modification
  remove_element
    use
      reads_subject: subject.representation.united (
          representation)
    modify
      subject_changed: subject.representation
    ensure
      elements_removed: subject = old subject.pruned (model)
end
```

Listing 7.9: Class that removes elements from sets

```
class ITERATOR
create make

feature -- Initialization
  make (a_subject: INTEGER_SET)
    modify
      only_local: representation
    ensure
      subject_set: subject = a_subject
      model_index_set: model_index = 1
      frame_confined: confined representation

feature -- Model and Frame
  model_sequence: MML_SEQUENCE[INTEGER]
    use
      reads_subject: subject.representation.united(
          representation)
  model_index: INTEGER
    use
      reads_subject: subject.representation.united(
          representation)
  representation: FRAME
    use
      self_framing: representation

feature -- Access
  subject: INTEGER_SET
  item: INTEGER
    require
      index_valid: model_sequence.domain.has (model_index)
    use
      reads_subject: subject.representation.united(
          representation)
    ensure
      result_computed: Result = model_sequence.item (
          model_index)

feature -- Modification
  forth
    use
      reads_subject: subject.representation.united(
          representation)
    modify
      only_local: representation
    ensure
      cursor_moved: model_index = old model_index + 1
      sequence_not_changed: model_sequence = old
          model_sequence
invariant
  model_sequence_definition: model_sequence.range = subject
      .representation
end
```

CHAPTER 8

BALLET: A VERIFIER FOR EIFFEL

Ballet is an experimental implementation of an automatic verifier for Eiffel. It takes a single class and tries to verify the implementation of the features of the class based on the contracts of the features used by the contract and implementation.

This chapter summarizes the design and implementation of Ballet and the translation of contracted Eiffel into a representation that is passed on to an existing theorem prover. The full source code is available as a branch on EiffelStudio in its public subversion repository.

8.1 Related work

An early chain of systems for the automatic verification of code was the Larch toolchain [33][13][40].

SPARK[3] is a verification system for an ADA-like language. It mostly targets verification of properties of information-flow.

An automatic verification system for Eiffel called *ES-Verify* was developed by Ostroff et. al. [68]. This verifier also uses a model library for the verification of unbounded data structures, similar to the work presented here. All types are value types to avoid aliasing. It uses *Perfect Developer*[15] as the underlying proof engine.

The ESC/Java[28] and ESC/Java-2[12] systems are systems for the verification of contracted Java classes. Though they are neither sound nor complete, they offer the developer support the developer by offering hints at possible contract violations. They also provide the developer with models to support writing contracts.

The Spec# verification system[4] and the underlying Boogie[19] are the

main sources of the work presented here. Other than Spec#, there is also a formal translation of Java and JML to BoogiePL[11][39].

8.2 Verification technology

For software development, it is important to avoid conceptual breaks and to provide fast round-trip between implementation and validation.

In software verification, we avoid conceptual breaks by providing a single consistent environment and language for development and verification. The developer learns a single language for both tasks.

Such a conceptual break exists if the system takes the development language, transforms it into some other language (for example of the underlying prover), and then shows this representation to the developer for an interactive proof or to report verification errors. The developer has to learn a second language and needs to match expressions of the second language to the corresponding expressions of the first language to understand the verification problem.

The smaller the code changes are between two verification steps, the easier it is for the developer to understand the problems raised during verification and to fix them. Many issues discovered during the verification require a change in the code or specification. If development and verification go hand-in-hand, the developer can constantly use the verification infrastructure to check his progress and the consistency and correctness of the overall system.

On-the-fly verification is a verification technique that works in the background of the development environment and reacts to change by issuing its result as unobtrusive information to the developer. The Spec# programming pioneered on-the-fly verification[4]. The prover works constantly in the background and will directly (after a delay of a few seconds) mark unprovable assertions and contracts with green squiggly lines. Error messages are presented in tool-tips.

Ballet is integrated into EiffelStudio and tries to achieve similar goals. EiffelStudio implements an innovative recompilation strategy called *melting-ice technology*[26]. This allows very fast recompilations of system and vx-erification cycles.

While the recompilation is very fast, it still needs to be triggered explicitly by the developer. Ballet requires the result of recompilation, making use of the compiler-generated intermediate representation. Thus, Ballet needs to be called explicitly by the developer, though a typical Ballet run

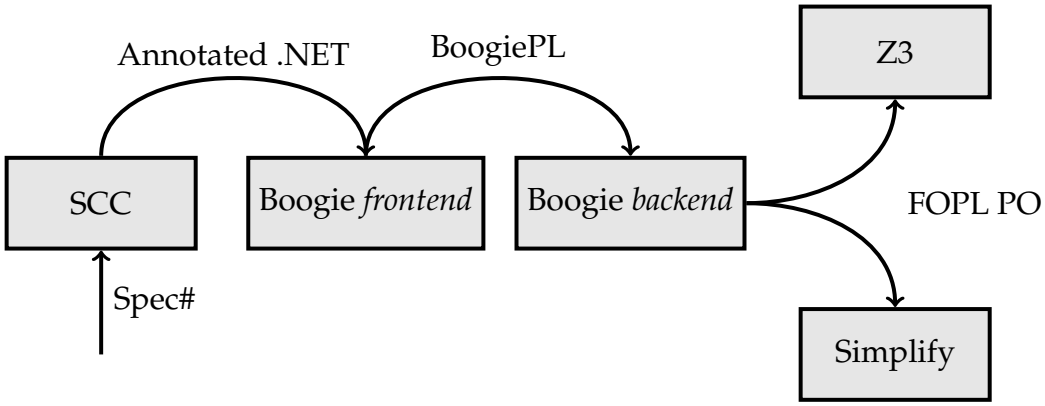

Figure 8.1: The Boogie tool chain.

only takes a few seconds (depending on the speed of the underlying Simplify theorem prover).

8.3 Boogie and BoogiePL

Boogie is the verifier underlying the Spec# programming system. The Spec# code is compiled to .NET byte-code using `ssc` (the Spec# compiler). The resulting .NET assembly is annotated with the Spec# contracts.

The actual Boogie tool takes the annotated .NET assembly and first transforms it to an intermediate representation called *BoogiePL* [19]. Then, BoogiePL is transformed to produce verification conditions for a first-order predicate prover and runs the prover. Traditionally, Boogie was relying on *Simplify* [20] to discard proof obligations. Recently, the better maintained prover Z3[18] has been added and is now the default. An overview of the Boogie tool chain is given in figure 8.1.

BoogiePL is an imperative programming language targeted for compilation. It includes a simple mathematical language to express axioms in first-order predicate logic with an appropriate theory for arrays, booleans and integers. New types can be introduced and defined by means of function declarations and axioms to define the underlying ADT[45].

The program state is defined by a set of variables; procedures define state transformations. Each procedure is declared through a signature. The declaration can contain preconditions and postconditions, both declared by an first-order expression on the given state. The pre-state of the computation in the postcondition of a procedure can be referenced using

the **old** keyword.

Implementation clauses provide code for declared procedures. The main imperative instructions are **assert** and **assume**. The only control-flow construct available is a non-deterministic **goto**. Together with assert and assume, this can be used to model all other control-flow instructions.

The main task of the Boogie backend is to derive proof obligations from the imperative code given in the procedure implementations. This is done by using weakest-precondition reasoning [22][23].

BoogiePL takes away the burden of implementing a weakest-precondition generator. Although it was designed for the Spec# programming language, it does not enforce any specific details of the object model, not even object-oriented code. This makes is well suited for the verification of Eiffel, whose object model is significantly different from the .NET/Java object model.

Listing 8.1 shows an example implementation of a `max` function that computes the maximum value from two given input integers. The "." and "$" are just normal characters in identifiers, with no special meaning.

We first define the mathematical definition of the maximum through the `math_max` function. Then we specify that the result of the procedure needs to match the mathematical definition.

8.4 Eiffel byte-code

EiffelStudio offers two intermediate representations. The first one is the standard abstract syntax tree generated from the Eiffel parser. During the traversal of the type checker, a second intermediate representation is generated called the *byte-code IR*. The byte-code IR is the representation of Eiffel that is used for the different backends for the Eiffel compiler (finalized C code, workbench C code, melted interpretation, .NET CIL).

The byte-code IR significantly simplifies the representation of Eiffel. It is similar to the static model presented in chapter 5. All types (including anchored declarations) are resolved, identifiers for local variables and arguments are replaced by abstract values. All creation instructions are replaced by equivalent creation expressions. The flattened version of the contracts is made accessible.

Each feature is represented in the byte-code IR as a tree of *byte nodes*, implementing a composite pattern. An abstract iterator for this tree is provided through the class `BYTE_NODE_VISITOR`. Most generation of BoogiePL in Ballet is implemented using the visitor pattern.

```
function math_max(a:int,b:int) returns (int);
axiom (forall a:int,b:int :: (math_max(a,b) == a)
                    || (math_max(a,b) == b));
axiom (forall a:int,b:int :: math_max(a,b) >= a);
axiom (forall a:int,b:int :: math_max(a,b) >= b);

procedure max (a:int,b:int) returns (Result:int);
  ensures Result == math_max(a,b);

implementation max (a:int,b:int) returns (Result:int) {
  start:
    goto a_is_max,b_is_max;
  a_is_max:
    assume (a >= b);
    Result := a;
    goto end;
  b_is_max:
    assume (!(a >= b));
    Result := b;
    goto end;
  end:
    return;
}
```

Listing 8.1: Maximum of two values as BoogiePL

8.5 Verification strategy

Ballet is implemented similarly to Boogie. An overview is shown in figure 8.2. The Eiffel code is read and analyzed by the standard Eiffel compiler. The compiler performs the necessary syntax and type checks. Then the generated intermediate representation is passed on to Ballet. Ballet generates the BoogiePL code used by the Boogie backend to create the verification conditions. Simplify or Z3 will then try to verify the code.

Ballet always verifies a single class at a time. The following phases structure the generation of BoogiePL in Ballet:

- The *analysis phase* traverses the list of features in the class and separates them into attributes, functions, and procedures.
- In the *attribute phase*, the necessary field definitions are generated for each attribute.

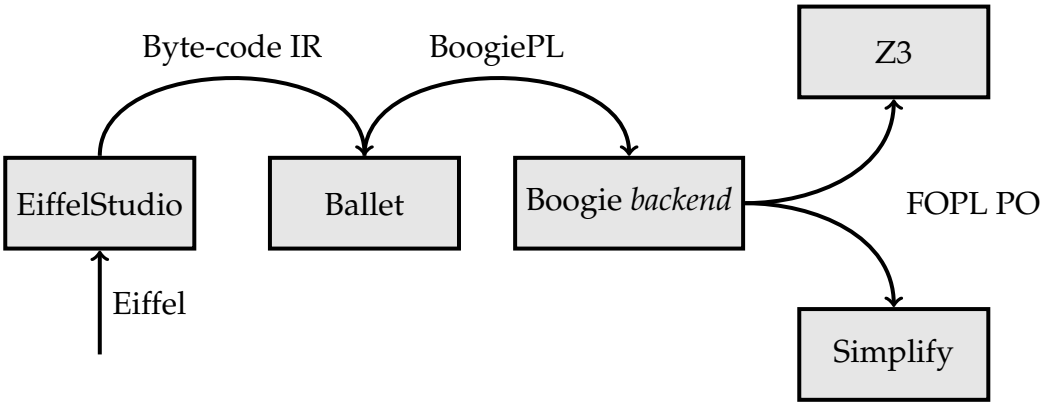

Figure 8.2: Overview of the Ballet tool chain.

- A functional version of every function (query that is not an attribute) is generated during the *function phase*.
- The BPL procedures declaration for each routine (function or procedure) is generated during the *signature phase*. This includes the generation of the contracts.
- The actual BPL implementation for each feature is generated during the *implementation phase*.
- During all previous phases, Ballet kept track of all features from other classes that were used in the class to verify. The *use phase* declares the required functional and procedural version of these features.

The BoogiePL code generated in each phase, except for the trivial analysis phase, is explained in detail in section 8.7.

8.6 Background theory

The background theory is a set of declarations that is prepended to every generated BoogiePL file. It includes the definition of the state and the axiomatization of the set theory required for models and frames. It also includes a number of support definitions for some axiomatic classes like `STRING`, `BOOLEAN`, `INTEGER` and for `ANY`.

8.6.1 Memory model

The memory model is a simplified version of the one used by Boogie for the verification of .NET assemblies. It defines the heap to be a two-dimensional array of references and names, containing values, that satisfies a certain heap condition:

```
function IsHeap(h:[ref,name]any)
  returns (bool);
var Heap: [ref,name]any where IsHeap(Heap);
```

A special field `$allocated` marks an object as currently allocated. The `IsAllocated` function accesses this field. If an object is allocated, all reachable objects are allocated as well.

```
const unique $allocated:name;
function IsAllocated(h:[ref,name]any,o: any)
  returns (bool);
axiom (forall h: [ref,name]any, o: ref ::
  IsAllocated(h,o) <==> cast(h[o,$allocated],bool));
axiom (forall h:[ref,name]any,o:ref,f:name ::
      { IsAllocated(h,h[o,f]) }
      IsHeap(h) ==> IsAllocated(h,h[o,f]));
```

Two function describe object creation: `new` allocates a new, unallocated object and returns it. `X` returns the heap modified by the allocation.

```
function new([ref,name]any) returns (ref);
function X([ref,name]any) returns ([ref,name]any);
axiom (forall h:[ref,name]any ::
      {IsAllocated(X(h),new(h))}
      IsAllocated(X(h),new(h)));
axiom (forall h:[ref,name]any ::
      !IsAllocated(h,new(h)));
axiom (forall h:[ref,name]any, o:ref, f:name ::
      {X(h)[o,f]}
      IsAllocated(h,o) ==> (X(h)[o,f] == h[o,f]));
```

Finally, we declare **`Void`**, which is a predefined reference called `null` in BoogiePL, to be always allocated.

```
axiom (forall h: [ref,name]any ::
      { IsAllocated(h,null) }
      IsAllocated(h,null));
```

Remark: The clauses in curly { and } are *triggers*. Triggers are hints to the prover, telling when to apply a rule. While they do not carry semantic information, they are critical for the prover to be efficient.

8.6.2 Set theory

The background theory defines sets using abstract data types. The sets are used as models as well as dynamic frames. While the background theory currently only includes sets, other data type like relations, bags and sequences can be added later.

Sets are introduced as a new type with the definition of a constant $\varnothing$ and the possibility to add a single element to the set $S \cup \{x\}$.

```
type set;

const set.make_empty: set;
function set.extended (set,any) returns (set);
```

Other features defined in the *MML_SET* data type have corresponding functions. All axioms try to simplify the given set to a form that only contains `set.make_empty` and `set.extended`. The full background theory is listed in appendix C.

8.7 Translations

This section contains descriptions of the different phases of the BoogiePL generation and the translation rules that are implemented.

Eiffel contracts contain executable code. BoogiePL does not allow procedure invocations in the contract language. Only first-order predicates are allowed. This makes it necessary to create two versions in BoogiePL for every query that may occur in the contract.

The first version is the regular *procedure* implementation that is used when query is used inside the body of a routine. The second version is a definition of a BoogiePL *function* that mimics the abstraction function implemented by the query.

8.7.1 Types

The current implementation of Ballet only supports three expanded types: *BOOLEAN*, *CHARACTER* and *INTEGER*. Booleans are mapped to the BoogiePL boolan type, characters and integers to the BoogiePL integer type. All other types have to be reference types. Table 8.1 shows the mapping of the Eiffel to the corresponding BoogiePL types.

Eiffel	BoogiePL
ANY	`ref`
BOOLEAN	`bool`
INTEGER, *INTEGER_**, *INTEGER_*_REF*	`int`
CHARACTER, *CHARACTER_**, *CHARACTER_*_REF*	`int`
FRAME, *MML_SET*, *MML_DEFAULT_SET*	`set`
other reference type	`ref`

Table 8.1: Translation of Eiffel to BoogiePL types

8.7.2 Attribute phase

In the attribute phase, we are defining a name for the field and generate the functional version to read the attribute. For example, the following definition of an attribute

```
feature -- Access

  x: INTEGER
```

in class *SIMPLE_COUNTER* is translated into the following BoogiePL code:

```
const unique field.SIMPLE_COUNTER.x:name;
function fun.SIMPLE_COUNTER.x([ref, name]any, ref) returns
    (int);
  axiom (forall H:[ref,name]any, C:ref :: {fun.
      SIMPLE_COUNTER.x(H, C)} fun.SIMPLE_COUNTER.x(H, C) ==
      H[C,field.SIMPLE_COUNTER.x]);
```

The **unique** keyword makes sure that the field name is different from all other field names. BoogiePL allows . (dot) and $ (dollar) to be a regular part of an identifier. This makes it possible to map single Eiffel identifiers to multiple BoogiePL identifiers. We use the prefix `field.` for field names, `fun.` for function names and (as we will see later) `proc.` for the procedural version of an identifier. Arguments are prefixed with `arg.`. Also, we include the class name of the feature in the name to BoogiePL avoid name clashes, as BoogiePL assumes a global name space.

8.7.3 Function phase

For each feature we generate the functional version by first declaring a function that takes two or more arguments: the heap of the pre-state, the reference to the target object and the arguments (if any).

Then, we declare an axiom quantified over all states, current objects and arguments. The axiom expresses that the precondition implies the postcondition, where we replace all occurrences of **`Result`** by the function application.

For example, the following function

```
plus (y:INTEGER): INTEGER is
   -- The attribute 'x' plus the argument 'y'
  do
    Result := x + y
  ensure
    added: x = Result - y
  end
```

is translated into the following function declaration and axiom:

```
function fun.SIMPLE_COUNTER.plus([ref, name]any, ref, arg.y
   :int) returns (int);
axiom (forall H:[ref, name]any, C:ref, arg.y:int:: (C !=
   null) ==> fun.SIMPLE_COUNTER.x(H, C) == fun.
   SIMPLE_COUNTER.plus(H, C, arg.y) - arg.y);
```

We can see how the function definition makes use the previous function definition for the attribute to compute the value.

There is an inherit problem with translating contracts this way. If we introduce an unsound contract like the following

```
strange: INTEGER
   -- Impossible contract
  ensure
    added: false
```

we get the following functional definition

```
function fun.SIMPLE_COUNTER.strange([ref, name]any, ref)
   returns (int);
axiom (forall H:[ref, name]any, C:ref :: false);
```

which introduces the expression **`false`** into the set of hypothesis. This makes it possible to prove everything, making the whole theory unsound. It is a known weakness of the overall approach and present in Spec# and Boogie.

Features that belong to axiomatic classes (expanded classes and model classes) are treated specially: they are translated into the corresponding expressions in the background theory.

The Eiffel code `a + b`, where `a` and `b` are of type *INTEGER* is conceptually a call to the feature **infix "+"**. But as *INTEGER* is an axiomatic class, Ballet translates the call to the BoogiePL addition (`a + b`).

Calls to features of the class *FRAME* or *MML_SET* are translated into the corresponding function calls of the background theory. The feature invocation `s.united(t)` on *FRAME* is translated to the application of the function `set.united(s,t)` and not to a function invocation `fun.FRAME.united(Heap,s,t)`. Also, the feature *united* is not marked to be generated in the *use phase* (see section 8.7.6).

8.7.4 Signature phase

In the signature phase, Ballet generates the signatures of all features. The signature of the feature *plus* from the previous section produces the following signature definition:

```
procedure proc.SIMPLE_COUNTER.plus(Current: ref, arg.y:int)
    returns (Result:int);
  requires Current != null;
  free ensures (fun.SIMPLE_COUNTER.plus(Heap, Current, arg.
     y) == Result);
  ensures (fun.SIMPLE_COUNTER.x(Heap, Current) == Result -
     arg.y);
```

Every feature call requires that the target of the call is not **Void**, putting an obligation on the caller. Once *attached types* [56] are in place, the `require` clause can be rewritten into a `free require` clause, that gives extra information to the body of a routine, but will assume that the correctness of the call has already been taken care of by the type system.

8.7.5 Implementation phase

In the implementation phase, Ballet generates the body (implementation) of every routine.

All control flow instructions are translated into a version only using non-deterministic goto statements. The condition

```
if p1 then
  b1
elseif p2 then
  b2
```

```
  else
    b3
  end
```

```
    goto true1,false2;
  true1:
    assume(p1);
    b1
    goto endif5;
  false2:
    assume(!(p1));
    goto true3,false4;
  true3:
    assume(p2);
    b2
    goto endif5;
  false4:
    assume(!(p2));
    b3
    goto endif5;
  endif5:
```

Listing 8.2: Translation of a condition into BoogiePL

is translated to BoogiePL code shown in listing 8.2 and the loop statement

```
  from b1 invariant p1 until p2 loop
    b2
  end
```

```
  b1
  assert(p1);
  goto loop1,exit2;
loop1:
  assume(!(p2));
  b2
  assert(p1);
  goto loop1,exit2;
exit2:
  assume(p2);
```

Listing 8.3: Translation of a loop into BoogiePL

is translated to the BoogiePL code shown in listing 8.3. Calls to commands and assignments are translated using the BoogiePL `call` and `:=` syntax.

There are numerous locations where expressions are used in Eiffel: right side of assignments, arguments to commands, in conditions (for branches and loops), loop invariants and check instructions.

We have to make sure to satisfy the precondition of queries used in expressions before the call. This is done by creating calls to the procedure version of the query before the use of the function version as an argument.

The Eiffel code *incr_n_times(x)* (where *incr_n_times* and *x* are a queries of the class *SIMPLE_COUNTER*) is translated into the following BoogiePL code:

```
call attr1 := proc.SIMPLE_COUNTER.x(Current);
call proc.SIMPLE_COUNTER.incr_n_times(Current,fun.
   SIMPLE_COUNTER.x(Heap, Current));
```

The contents of the variable `attr1` are not used after the call. BoogiePL requires that the result of a call to a procedure with a return value needs to be assigned.

8.7.6 Use phase

During previous phases, Ballet kept track of all features used in the contracts and the code. Only feature that were neither part of the analyzed class nor of axiomatic classes were recorded.

The use phase will use the generation routines from the functional phase and signature phase to generate the function and procedure versions of all used features. The implementations are not generated; we assume that they are correctly implemented.

While processing the used features and generating the BoogiePL code, new features will be encountered that are used in the contracts of the used features. These are indirect clients to the current class. Such features are also added to the set of used features. Generation will terminate once a fixpoint has been reached and the set of used and generated features is the same.

8.7.7 Translating frames

BoogiePL provides framing using the `modifies` keyword. This keyword list a set of variables that might be modified by the procedure invocation. All other variables will remain untouched.

For a object-oriented language such as Eiffel, this is useless: the state is defined completely by a the single variable `Heap`. Any feature that mod-

ifies the heap (including queries due to weak purity[16]) will need to list the heap in the modifies clause.

A direct translation of the frame axiom into the BoogiePL background theory would require the quantification over all commands or queries. This cannot be expressed in first-order predicate logic.

It is possible to express dynamic frame contracts as regular contracts resolving the quantification during generation. We generate an assertion for every command and every query in form of a `free ensures` in the postcondition of the command that describes the interference with the query. An assertion needs to be added for every query that might interfere with the command.

The generated postcondition states that the result of the query will not have changed by the feature call if its use frame of the query and the modify frame of the command were disjoint. For a command c and a query q, with c framed by the modify frame cf and q framed by the use frame qf, we generate the following assertion in the postcondition:

$$\forall o : Obj :: \mathbf{old}(cf(o) \cap qf(o) = \varnothing) \Rightarrow q(o) = \mathbf{old}(q(o))$$

Translated to BoogiePL, this postcondition becomes a little more difficult to read:

```
free ensures (forall o:ref ::
  ((set.is_disjoint_from (fun.C.cf(old(Heap),o),
    fun.C.qf(old(Heap),Current))) ==>
  (fun.C.q(Heap,o) == fun.C.q(old(Heap),o))));
```

Such a clause is generated for every command that carries a modifies clause, for every query with a use frame that is used in the system. The extra overhead could be reduced if BoogiePL would include dynamic frames into the language.

The current version of Ballet does not include assertions to check that a given implementation satisfies a given frame. It can be implemented by storing the use and modify frame, as well as the set of all allocated object at the beginning of the execution in local variables. Assume that we have named these variables U, M and A. Then, the following assertions are added to the code:

- Every assignment to a local variable requires that **`Current`** is contained in M.
- Access to a local attribute requires **`Current`** to be contained in U.

- Every call to a command requires that the use frame of the command intersected A is a subset of U and the modify frame of the command intersected with A is a subset of M.
- Every call to a query requires the use frame of the query intersected with A is a subset of U.

These assertions check that the implementation adheres to its use and modify sets (see section 7.10.1 and 7.10.2) and — as the use and modify sets are always subsets of theirs frames — its frame specification.

8.8 Error reporting

Ballet implements a very simple error reporting strategy that adds the line number, the column number and the assertion tag of any assertion as a comment to the generated BoogiePL code.

Boogie will report the line number with the assertion (assert or post-condition) that could not be proved. Ballet then searches the BoogiePL code for the corresponding line and extracts the position and tag from the BoogiePL comment. It then generates an error message in the EiffelStudio error dialog that describes the error, error position and tag.

8.9 Experiments

The following sections contain examples of code fragments that we were able to verify for correctness using Ballet.

8.9.1 Regular Eiffel

The following counter class implements many features of the Eiffel language that can be proved without the help of models or frames. The comments describe what kind of language construct was tested with in the feature.

```
class
  SIMPLE_COUNTER

feature -- Access

  x: INTEGER
```

```
increase is
   -- Simple contract
  do
    x := x + 1
  ensure
    increase_ok: x = old x + 1
  end

move_to_zero is
   -- if-then-elseif-else condition
  do
    if x > 0 then
      x := x - 1
    elseif x < 0 then
      x := x + 1
    else
      x := 0
    end
  ensure
    close_to_zero: (x.abs < (old x.abs)) or (x = 0)
  end

increase_twice is
   -- Sequential feature calls
  do
    increase
    increase
  ensure
    double_increase_ok: x = old x + 2
  end

increase_n_times (i: INTEGER) is
   -- Loop and feature call
  require
    i_not_neg: i >= 0
  local
    old_x: INTEGER
  do
    from
      old_x := x
    invariant
      loop_inv: x <= old_x + i
    until
```

```
      x >= old_x + i
    loop
      increase
    end
  ensure
    increase_ok: x = old x + i
  end

 add (other:SIMPLE_COUNTER) is
    -- Simple aliasing problem
  require
    not_void: other /= Void
    not_same: other /= Current
  do
    x := x + other.x
  ensure
    x_added: x = old x + other.x
  end

 move_one (other:SIMPLE_COUNTER) is
    -- Complex aliasing problem with feature invocation
  require
    not_void: other /= Void
    not_same: other /= Current
  local
    tmp: INTEGER
  do
    tmp := x
    other.increase_n_times (1)
    x := tmp - 1
  ensure
    current_decr: x = old x - 1
    other_increase: other.x = old other.x + 1
  end
end
```

The code is correctly translated into BoogiePL and proved to be correct within a few seconds on Pentium 4 3.6 GHz machine.

Feature *add* and *move_one* shows that simple aliasing problem can be verified without the need for models or frames. In the case of *move_one*, temporary variables were needed to enforce the frame separations.

8.9.2 Model-based contracts

The following scenario describes a specification that cannot be contracted with regular Eiffel: we want the class *TOKEN_DISPENSER* to issue tokens, but never the same token twice. The problem with this contract is that it needs to talk about an unbounded set of tokens: all tokens that have already been issued by the dispenser.

We regard the set of all tokens already issued as the model of the *TOKEN_DISPENSER*. The token dispenser then asserts that the new token generated was not in the old model, but has been added to the updated model. The Eiffel code is shown in listing 8.4.

With the model-based contract, the following code is provable:

```
 local
    x: TOKEN
    y: TOKEN
    z: TOKEN
  do
    t.generate_token
    x := t.last_token
    t.generate_token
    y := t.last_token
    t.generate_token
    z := t.last_token
    check not_same1: x /= y end
    check not_same2: y /= z end
    check not_same3: x /= z end
  end
end
```

Without models, assertion `not_same3` could not be verified. With models, the verification is made possible.

8.9.3 Dynamic frame contracts

For dynamic frame contracts, we were able to prove the *INT_STORE* example as discussed in section 2.7. For *set_item*, Ballet generated the following procedure definition:

```
procedure proc.FRAME_TEST.set_item(Current: ref, arg.
   a_value:int);
 requires Current != null;
 modifies Heap;
 ensures (fun.FRAME_TEST.item(Heap, Current) == arg.
    a_value);
```

```
class TOKEN_DISPENSER

feature -- Access

  last_token: TOKEN

feature -- Creation

  generate_token is
     -- Generate a new token
    do
      create last_token.make
    ensure
      model_expanded: model = old model.extended (last_token
         )
      new_token: not (old model).contains(last_token)
      different_token: last_token /= old last_token
    end

feature -- Model

  model: MML_SET[TOKEN]

end
```

Listing 8.4: Example of a contract based on models

```
  ensures (set.equals(fun.FRAME_TEST.representation(Heap,
     Current), old(fun.FRAME_TEST.representation(Heap,
     Current))));
  free ensures (forall o:ref :: ((set.is_disjoint_from (fun
     .FRAME_TEST.representation (old (Heap), o), fun.
     FRAME_TEST.representation (old (Heap), Current))) ==>
     (fun.FRAME_TEST.representation (Heap, o) == fun.
     FRAME_TEST.representation (old (Heap), o))));
  free ensures (forall o:ref :: ((set.is_disjoint_from (fun
     .FRAME_TEST.representation (old (Heap), o), fun.
     FRAME_TEST.representation (old (Heap), Current))) ==>
     (fun.FRAME_TEST.item (Heap, o) == fun.FRAME_TEST.item
     (old (Heap), o))));
```

The generated free ensure clauses make it possible for a client of *INT_STORE* to prove the *copy* feature that was not provable before (see listing 2.9).

8.10 Summary

Ballet demonstrates how Eiffel can be verified. It shows how contracts are translated to BoogiePL and what is provable in Eiffel as it is today.

We showed how a background theory formalizes a theory for models and how model-based contracts are translated directly into this background theory. The automatic provers underlying the Boogie tool chain are able to discard proof obligations generated by model-based contracts.

BoogiePL only offers very primitive constructs for framing. For reference-based languages that uses a heap-like memory model, such *modifies* clauses are insufficient to express dynamic frame contracts. Instead, we had to generate explicit free postconditions that express the framing axiom (7.45).

The current implementation of ballet misses the definition of supplier obligations for dynamic frames. We have sketched how these proof obligations can be translated into BoogiePL.

Overall, once the correct BoogiePL was generated, the experiments were able to prove the correctness of the code. The most difficult part was to add the right triggers to the background theory for the proof to go through. From discussions with the researchers behind Boogie, this seems to be similar in the case of verifying.

CHAPTER 9

CONCLUSIONS

Modern provers seem to be powerful enough to prove real-world code. More difficult than the proofs are the specifications: getting the specifications sufficiently strong for the client of a class to be able to reason about his own code requires new ways to express contracts.

Modularity is curse and blessing. It is difficult devise specifications that are complete enough for the users to reason about a software module and at the same time open enough to retain the benefits of functional decomposition and information hiding. Once we have managed to overcome these difficulties, modular verification can help to manage the combinatorial explosions that are inherit in formal verification and make proofs reusable.

9.1 Contributions

This thesis improves the Design by Contract specification language to become powerful enough for the modular verification of object-oriented code. We do this by improving the specification language in two ways:

- Express the state of an object explicitly by providing models.
- Express the possible interference of software components with other components though frame specifications.

Both mechanisms are carefully designed to be conservative extensions to the existing specification language. By conservative, we mean the following properties:

- We minimize the changes to the existing language. For models, we have shown how all can be expressed through a library, understanding the concept of axiomatic class. For frames, we have introduced three new keywords into the language.

- The mechanisms support information hiding. It is possible to express all necessary properties of a class for formal verification without revealing details of the implementation.

- The mechanisms agree with the inheritance relation. There are rules how models and frames can be redefined. The contracts of subtypes are proper refinements of the supertypes.

- If the new specification mechanisms are not used, the semantics of the specified software component is exactly the same as it was before the introduction of the new mechanisms.

Models and frames are real extensions to Design by Contract. They do not only extend the language to make proof possible. Instead, using models and frames creates insights into design decisions and make implicit assumptions explicit.

Models give a clean definition of what the state of an object is. **The state of the object is its model**. Deciding for a model removes ambiguity in abstractions. Our research conducted on the EiffelBase library illustrates the problems of subtyping and maintaining a precise understanding of the features through the inheritance relation. Models define the state of the object even for deferred classes. The quality of contracts is improved by relating to the model.

Frames express the boundaries of an objects state. They can be used to reason about interference. Not many libraries document their possible interference with other libraries.

Most non-interference properties are obvious and the engineers intuition is good enough to avoid potential problems. From personal discussions it seems that few engineers are even aware of the problem.

But there are corner cases where asserting non-interference becomes difficult, for example for Complex data models with a lot of implicit code invocations and shared state.

Already trivial object-oriented concepts such as iterators create problems. The Java documentation uses free form sentences to describe these frame problems:

> *The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.* (Java 5.0 API[81]).

The specification of other behavior, for example what happens if multiple streams try to read from the same source, is not defined at all. Read-ahead and buffering may create unexpected behavior. It is assumed that the developer is aware of the potential problems and avoids using the library this way.

Dynamic frame contracts force us to make these assumptions explicit. They make us think about problems that we can encounter when using references and aliasing. We have shown how typical object structure create patterns of frames.

This thesis demonstrates how the wish to prove software using verification tools forces us to gain a better understanding of our implicit and hidden assumptions underlying software development.

Consequently, the success of our efforts require validation by a rigorous formalization and the extension of verification technology. This thesis includes an operational semantics for Eiffel and an verification tool to validate our improved understanding of software specifications.

9.2 Future work

The operational semantics in chapter 5 explicitly excludes a number of concepts from the Eiffel language.

Exceptions are excluded, justified by the understanding that exceptions in Eiffel manifest bugs, and the successful verification of code with prevent exceptions from being raised.

New concepts of the ISO/ECMA standard were excluded, such as attached types or type tests. These new concepts try to overcome problems of Eiffel with respect to type safety or void calls. A formalization of these concepts would improve their understanding and might improve the overall quality of the standard.

The long term goal is the full formalization of Eiffel and the inclusion of this formalization into the Eiffel standard.

We have described the performance impact of using models for runtime assertion checking in section 6.11. Assertion checking in production code seems to become difficult. Smart caching can improve the situation. We also suggest to explore using a three-valued logic underlying MML

to overcome these problems. Models are only generated up to a certain upper bound, after which an *unknown* value is assumed.

Dynamic frame contracts are described in chapter 7. We have defined the basic rules for frame specifications. We have also observed that the *encapsulation* pattern seems to be very common. Systems like universe types [59] offer a much lighter (in terms of proofs) solution, but create problems with ownership transfer and iterators. Future work should investigate how dynamic frame contracts and universe types relate and if it is possible to move specific frame patterns into the type system.

A problem not explored in this thesis is the relation between frames and class invariants. Eiffel is not sound when invariants rely on the state of other objects or when reentrant calls are executed while the invariant is temporarily broken. Can dynamic frame contracts help to solve these problems?

The Ballet verifier as described in chapter 8 needs further improvements. A number of assumptions were made to simplify proofs, for example limiting the set of expanded types to integers and booleans. Also, being a research vehicle, the tool is unreliable and not ready for productive use.

9.3 Summary

This thesis includes multiple contributions to the state-of-the-art in modular specification and verification of object-oriented programs. We have extended the specification language of an existing development method by two concepts needed to make functional specifications strong enough to prove code. We have formalized the programming language and verified the soundness and practicality of the approach through proofs and experiments.

A major improvement to existing solutions is that we were able to retain the full power of object-orientation, including free aliasing, implicit invocations, bottom-up development and information hiding. Instead of forcing the developer to adopt a certain implementation strategy, we have made the specification language powerful and abstract enough to describe all necessary properties of real-world code for formal verification.

APPENDIX A

MATHEMATICAL SYMBOLS

The following table summarizes all the B and other mathematical symbols used in the thesis as a help to the reader. For a complete, axiomatic definition of the B symbol, please consult the B book[1].

Symbol	*Description*
$\mathrm{card}(S)$	cardinality of the set S
$\{x, y\}$	explicit denotation of a set containing x and y
$(x \mapsto y)$	tuple with the first value x and second value y
$\mathbb{P}(S)$	powerset of S
$A \times B$	set cross procuduct of A and B
$\mathrm{dom}(R)$	domain of the relation R
$\mathrm{ran}(R)$	range of the relation R
id	generic identity relation
R^{-1}	inverse relation of relation R
$A \leftrightarrow B$	set of all relations from A to B
$A \rightarrow B$	set of all total functions from A to B
$A \nrightarrow B$	set of all partial functions from A to B
$A \rightarrowtail B$	set of all partial injections from A to B
$A \mathrel{>\!\!\!\rightarrowtail} B$	set of all total injections from A to B
$A \twoheadrightarrow B$	set of all partial surjections from A to B
$A \mathrel{-\!\!\!\twoheadrightarrow} B$	set of all total surjections from A to B
$A \rightarrowtail\!\!\!\!\twoheadrightarrow B$	set of all total bijections from A to B
$R; S$	relation composition
$S \setminus T$	subtraction of the set T from S

Symbol	*Description*
$R \lhd\!\!+ T$	override the relation T with the relation S
$S \lhd R$	domain restriction — relation R restricted to the tuples where the first element is in S
$S \lhd\!\!\!- R$	domain subtraction — relation R restricted to the tuples where the first element is in S
$S \rhd R$	range restriction — relation R restricted to the tuples where the second element is in S
$S \rhd\!\!\!- R$	range subtraction — relation R restricted to the tuples where the second element is not in S
$R*$	reflexive transitive closure of R

APPENDIX B

MML LIBRARY FUNCTIONS

The following table summarizes all feature available in the MML library, together with precondition and their mathematical counterpart.

Functions marked by $*$ are creation procedures. M is the target model if the call. r is the result value. Not included in this table are redefinitions of signatures for covariant return types and renaming of existing features to avoid name collisions. The prefix *MML_* has been removed.

B.1 MML_ANY

`M.equals(other:ANY)` Value equality	$M = \text{other}$

B.2 MML_PAIR [G,H]

`make_from(one:G,two:H)` $*$ Denotation of a pair	(one, two)
`first:G` Accessing the first element in a pair	M_{first}
`second:H` Accessing the second element in a pair	M_{second}
`is_identity:BOOLEAN` Is the pair element an identity value ?	$M_{first} = M_{second}$
`inversed:PAIR[H,G]` Inverse pair	(M_{second}, M_{first})

B.3 MML_SET [G]

any_item:**G** Non-deterministic selection	$r \in M$
item_where(**pred**:**PREDICATE**[**TUPLE**[**G**]]):**G** Accessing the first element in a pair	$r :\in M \wedge \mathrm{pred}(r)$
identity:**RELATION**[**G**,**G**] Identity relation	$id(M)$
lifted:**POWERSET**[**G**] Set of all possible subsets	$\mathbb{P}\, M$
randomly_ordered:**SEQUENCE**[**G**] Randomly ordered sequence containing M	$\mathrm{ran}(\mathbf{Result}) = M$
count:**INTEGER** Cardinality of the set	$\mathrm{card}(M)$
contains(**v**:**G**):**BOOLEAN** Does M contain v ?	$v \in M$
is_empty:**BOOLEAN** Is the set empty ?	$M = \varnothing$
is_disjoint_from(**other**:**SET**[**G**]):**BOOLEAN** Are M and other disjoint ?	$M \cap \mathrm{other} = \varnothing$
is_superset_of(**other**:**SET**[**G**]):**BOOLEAN** Superset relation (including equality)	$\mathrm{other} \subseteq M$
is_subset_of(**other**:**SET**[**G**]):**BOOLEAN** Subset relation (including equality)	$M \subseteq \mathrm{other}$
is_proper_superset_of(**other**:**SET**[**G**]):**BOOLEAN** Superset relation (excluding equality)	$\mathrm{other} \subset M$
is_proper_subset_of(**other**:**SET**[**G**]):**BOOLEAN** Subset relation (excluding equality)	$M \subset \mathrm{other}$
intersected(**other**:**SET**[**G**]):**SET**[**G**] Intersection	$\mathrm{other} \cap M$
united(**other**:**SET**[**G**]):**SET**[**G**] Union	$\mathrm{other} \cup M$
subtracted(**other**:**SET**[**G**]):**SET**[**G**] Subtraction of one set from another	$M \setminus \mathrm{other}$
difference(**other**:**SET**[**G**]):**SET**[**G**] Symetric subtraction	$(M \cup \mathrm{other}) - (M \cap \mathrm{other})$
extended(**v**:**G**):**SET**[**G**] Set extended by one element	$M \cup \{v\}$

pruned(v:G):SET[G] $M \setminus \{v\}$
Removal of an element from the set
there_exists(pred:PREDICATE[TUPLE[ANY]]):BOOLEAN $\exists x \in M : \text{pred}(x)$
Does any element in M satisfy a given predicate ?
for_all(pred:PREDICATE[TUPLE[ANY]]):BOOLEAN $\forall x \in M : \text{pred}(x)$
Do all elements in M satisfy a given predicate ?
is_partitioned_by(other:SET[SET[G]]):BOOLEAN $\bigcup \text{other} = M \wedge \forall x, y \in \text{other} : x \cap y = \varnothing$
Is other a partion of M ?

B.4 MML_POWERSET [G]

generalized_united:SET[G] $\bigcup M$
Union of all sets contained in M
generalized_intersected:SET[G] $\bigcap M$
Intersection of all sets contained in M
is_generalized_disjoint:BOOLEAN $\bigcap M = \varnothing$
Is there no element contained in all set of M ?

B.5 MML_RELATION [G,H]

is_function:BOOLEAN $M \in G \leftrightarrow H$
Is M a function ?
is_injective:BOOLEAN $M \in G \rightarrowtail H$
Is M an injective relation ?
contains_pair (v1:G,v2:H):BOOLEAN $(v1, v2) \in M$
Does M contain the pair $(v1, v2)$?
extended_by_pair (g:G,h:H):RELATION[G,H] $M \cup \{(g, h)\}$
Relation extended by the pair (g, h)
domain:SET[G] $\text{dom}(M)$
Domain of the relation
range:SET[G] $\text{ran}(M)$
Range of the relation
domain_bag:BAG[G] $\{(x, y) \mid x \in \text{dom}(M), y = \text{card}(\{x\} \lhd M)\}$
Domain of the relation as a bag

range_bag:BAG[G] Range of the relation as a bag	$\{(x, y) \mid x \in \mathrm{ran}(M), y = \mathrm{card}(M \rhd \{x\})\}$
image_of(g:G):SET[H] Projection of the single element g through M	$M[\{g\}]$
image(p:SET[G]):SET[H] Projection of the set p through M	$M[p]$
anti_image_of(h:H):SET[G] Reverse projection of the single element h through M	$M^{-1}[\{h\}]$
anti_image(p:SET[H]):SET[G] Reverse projection of the set p through M	$M^{-1}[p]$
domain_restricted_by (g:G):RELATION[G,H] Domain restricted to the single element g	$\{g\} \lhd M$
domain_restricted (p:SET[G]):RELATION[G,H] Domain restricted to the elements in p	$p \lhd M$
range_restricted_by (h:H):RELATION[G,H] Range restricted to the single element h	$M \rhd \{h\}$
range_restricted (p:SET[H]):RELATION[G,H] Range restricted to the elements in p	$M \rhd p$
domain_anti_restricted_by (g:G):RELATION[G,H] g removed from the domain of M	$\{g\} \lhd\!\!\!\cdot\, M$
domain_anti_restricted (p:SET[G]):RELATION[G,H] Domain restricted to the elements not in p	$p \lhd\!\!\!\cdot\, M$
range_anti_restricted_by (h:H):RELATION[G,H] h removed from the range of M	$M \rhd\!\!\!\cdot\, \{h\}$
range_anti_restricted (p:SET[H]):RELATION[G,H] Range restricted to the elements not in p	$M \rhd\!\!\!\cdot\, p$
item (v:G):H Application of the function M to v	$M(v)$
inversed:RELATION[H,G] Inversion of the relation	M^{-1}

B.6 MML_ENDORELATION [G]

sequenced(other:ENDORELATION[G]):ENDORELATION[G] Sequential composition	$M; \mathrm{other}$
closured:ENDORELATION[G] Transitive closure	$M*$

B.7 MML_BAG [G]

is_superbag_of(other:SET[G]):BOOLEAN
$\forall x : dom(\text{other}) : \text{other}(x) \leq M(x)$
Superbag relation (including equality)

is_subbag_of(other:SET[G]):BOOLEAN
$\forall x : dom(\text{other}) : M(x) \leq \text{other}(x)$
Subbag relation (including equality)

is_proper_superbag_of(other:SET[G]):BOOLEAN
$\forall x : dom(\text{other}) : \text{other}(x) < M(x)$
Superbag relation (excluding equality)

is_proper_subbag_of(other:SET[G]):BOOLEAN
$\forall x : dom(\text{other}) : M(x) < \text{other}(x)$
Subbag relation (excluding equality)

occurrences(v:G):INTEGER $M(v)$
Number of occurrences of v

any_item:G $r \in (\text{dom}(\{0\} \lhd M))$
Non-deterministic element

item_where(pred:PREDICATE[TUPLE[G]]):G
$r \in (\text{dom}(\{0\} \lhd M)) \land \text{pred}(r)$
Any element of M satisfying pred

randomly_ordered:SEQUENCE[G]
A random sequence containing the elements of the bag

count:INTEGER $\sum \text{ran}(M)$
Number of elements contained in the bag

contains(v:G):BOOLEAN $M(v) > 0$
Does the bag contain v ?

is_disjoint_from(other:BAG[G]):BOOLEAN
$(\{0\} \lhd M) \cap (\{0\} \lhd \text{other}) = \varnothing$

extended_n(v:G; n:INTEGER):BAG[G] $\{(v, M(v) + n)\} \lhd\!\!+ M$
Extend the bag by n occurrences of v

B.8 MML_SEQUENCE [G]

any_element:G $r \in \text{ran}(M)$
Any element of the sequence

is_member(v:G):BOOLEAN $v \in \text{ran}(M)$
Is v contained in the sequence ?

is_defined(i:INTEGER):BOOLEAN $i \in \mathrm{dom}(M)$
Does the sequence have an index i ?

is_supersequence_of(other:SET[G]):BOOLEAN
$\exists x : \forall i \in \mathrm{dom}(\mathrm{other}) : \mathrm{other}(i) = M(i + x)$
Supersequence relation (including equality)

is_subsequence_of(other:SET[G]):BOOLEAN
$\exists x : \forall i \in \mathrm{dom}(M) : M(i) = \mathrm{other}(i + x)$
Subsequence relation (including equality)

is_proper_supersequence_of(other:SET[G]):BOOLEAN
$\mathrm{card}(\mathrm{other}) < \mathrm{card}(M) \wedge \exists x : \forall i \in \mathrm{dom}(M) : M(i) = \mathrm{other}(i + x)$
Superbag relation (excluding equality)

is_proper_subsequence_of(other:SET[G]):BOOLEAN
$\mathrm{card}(M) < \mathrm{card}(\mathrm{other}) \wedge \exists x : \forall i \in \mathrm{dom}(\mathrm{other}) : \mathrm{other}(i) = M(i + x)$
Subsequence relation (excluding equality)

is_supersequence_of_at(other:SET[G];x:INTEGER):BOOLEAN
$\forall i \in \mathrm{dom}(\mathrm{other}) : \mathrm{other}(i) = M(i + x)$
Supersequence relation (including equality), starting at x

is_subsequence_of(other:SET[G];x:INTEGER):BOOLEAN
$\forall i \in \mathrm{dom}(M) : M(i) = \mathrm{other}(i + x)$
Subsequence relation (including equality), starting at x

is_proper_supersequence_of(other:SET[G];x:INTEGER):BOOLEAN
$\mathrm{card}(\mathrm{other}) < \mathrm{card}(M) \wedge \forall i \in \mathrm{dom}(M) : M(i) = \mathrm{other}(i + x)$
Superbag relation (excluding equality), starting at x

is_proper_subsequence_of(other:SET[G];x:INTEGER):BOOLEAN
$\mathrm{card}(M) < \mathrm{card}(\mathrm{other}) \wedge \forall i \in \mathrm{dom}(\mathrm{other}) : \mathrm{other}(i) = M(i + x)$
Subsequence relation (excluding equality), starting at x

replaced_at(other:G;x:INTEGER):SEQUENCE[G] $(x, \mathrm{other}) \lhd\!\!+ M$
Replace sequence element at x

first:G $M(1)$
First element of the sequence

last:G $M(\mathrm{card}(M))$
Last element of the sequence

tail:SEQUENCE[G]
Tail of the sequence (without the first element)

front:SEQUENCE[G] $\{\mathrm{card}(M)\} \lhd M$
Front of the sequence (without the last element)

interval(lower:INTEGER;upper:INTEGER):SEQUENCE[G]
Elements of the interval between lower and upper

***extended*(*v*:*G*):*SEQUENCE*[*G*]** $M \cup (\mathrm{card}(M) + 1, v)$ Sequence extended by v
***prepended*(*v*:*G*):*MML_SEQUENCE*[*G*]** Sequence with v prepended
***concatinated*(*other*:*SEQUENCE*[*G*]):*SEQUENCE*[*G*]** Current list with other appended at the end

APPENDIX C

BOOGIEPL BACKGROUND THEORY

```
// Some things are heaps
function IsHeap(h:[ref,name]any) returns (bool);

// Given an heap, some things are allocated
const unique $allocated:name;
function IsAllocated(h: [ref,name]any, o: any) returns (
   bool);
axiom (forall h: [ref,name]any, o: ref :: IsAllocated(h,o)
   <==> cast(h[o,$allocated],bool));

// Every reference stored in the heap is allocated
axiom (forall h:[ref,name]any, o: ref, f: name :: {
   IsAllocated(h,h[o,f]) } IsHeap(h) ==> IsAllocated(h,h[o,
   f]));

function new([ref,name]any) returns (ref);
function X([ref,name]any) returns ([ref,name]any);

axiom (forall h:[ref,name]any :: {IsAllocated(X(h),new(h))}
    IsAllocated(X(h),new(h)));
axiom (forall h:[ref,name]any :: !IsAllocated(h,new(h)));

axiom (forall h:[ref,name]any, o:ref, f:name :: {X(h)[o,f]}
    IsAllocated(h,o) ==> (X(h)[o,f] == h[o,f]));

// Void are always allocated
axiom (forall h: [ref,name]any :: { IsAllocated(h,null) }
   IsAllocated(h,null));
```

```
// The global heap is a heap
var Heap: [ref,name]any where IsHeap(Heap);

// ADT set

type set;

// Constructors
const set.make_empty: set;
function set.extended (set,any) returns (set);

// Content
function set.is_member (set,any) returns (bool);
axiom (forall s:set,x:any :: {set.is_member(set.extended(s,
   x),x)} set.is_member(set.extended(s,x),x));
axiom (forall s:set,x:any,y:any :: x != y ==> set.is_member
   (set.extended(s,y),x) == set.is_member(s,x));
axiom (forall x:any :: !set.is_member (set.make_empty,x));

// Pruning
function set.pruned (set,any) returns (set);
axiom (forall x:any :: {set.pruned(set.make_empty,x)} set.
   pruned(set.make_empty,x) == set.make_empty);
axiom (forall s:set,x:any :: {set.pruned(set.extended(s,x),
   x)} set.pruned(set.extended(s,x),x) == set.pruned(s,x));
axiom (forall s:set,x:any,y:set :: {set.pruned(set.extended
   (s,x),y)} (x != y) ==> set.pruned(set.extended(s,x),y)
   == set.extended(set.pruned(s,y),x));

// Emptiness
function set.is_empty (set) returns (bool);
axiom (forall s:set,x:any :: !set.is_empty(set.extended(s,x
   )));
axiom (set.is_empty (set.make_empty));

// Cardinality
function set.cardinality (set) returns (int);
axiom (set.cardinality(set.make_empty) == 0);
axiom (forall s:set,x:any :: { set.cardinality(set.extended
   (s,x)) } !set.is_member(s,x) ==> set.cardinality(set.
   extended(s,x)) == set.cardinality(s) + 1);
axiom (forall s:set,x:any :: { set.cardinality(set.extended
   (s,x)) } set.is_member(s,x) ==> set.cardinality(set.
```

```
   extended(s,x)) == set.cardinality(s));

// Any element
function set.any_element (set) returns (any);
axiom (forall s:set,x:any :: set.any_element(set.extended(s
   ,x)) == x);

// Subset
function set.is_subset_of (set, set) returns (bool);
axiom (forall s:set :: set.is_subset_of (s, set.make_empty)
   );
axiom (forall s1:set, s2:set, x:any :: set.is_subset_of (s1
   , set.extended(s2, x)) == (set.is_subset_of (s1, s2) &&
   set.is_member (s1, x)));

// Equality
function set.equals (set, set) returns (bool);
axiom (forall s1:set, s2:set :: (s1 == s2) == (set.
   is_subset_of (s1, s2) && set.is_subset_of (s2, s1)));
axiom (forall s1:set, s2:set :: (s1 == s2) == (set.equals (
   s1, s2)));

// Union
function set.united (set,set) returns (set);
axiom (forall s:set :: {set.united (s,set.make_empty)} set.
   united (s,set.make_empty) == s);
axiom (forall s1:set,s2:set,x:any :: {set.united(s1,set.
   extended(s2,x))} set.united(s1,set.extended(s2,x)) ==
   set.extended(set.united(s1,s2),x));

// Intersected
function set.intersected (set,set) returns (set);
axiom (forall s:set :: {set.intersected(s,set.make_empty)}
   set.intersected(s,set.make_empty) == set.make_empty);
axiom (forall s1:set,s2:set,x:any :: {set.intersected(s1,
   set.extended(s2, x))} set.is_member(s1,x) ==> (set.
   intersected(s1,set.extended(s2,x)) == set.extended (set.
   intersected(s1,s2),x)));
axiom (forall s1:set,s2:set,x:any :: {set.intersected(s1,
   set.extended(s2, x))} !set.is_member(s1,x) ==> (set.
   intersected(s1,set.extended(s2,x)) == set.intersected(s1
   ,s2)));
```

```
// Disjoint
function set.is_disjoint_from (set,set) returns (bool);
axiom (forall s:set :: {set.is_disjoint_from (s,set.
   make_empty)} set.is_disjoint_from (s,set.make_empty));
axiom (forall s1:set,s2:set,x:any :: set.is_disjoint_from (
   s1,set.extended(s2,x)) == (!set.is_member(s1,x) && set.
   is_disjoint_from (s1, s2)));
axiom (forall s1:set, s2:set :: set.is_disjoint_from (s1,
   s2) == set.is_disjoint_from (s2, s1));

// Superset
function set.is_superset_of (set, set) returns (bool);
axiom (forall s1:set, s2:set :: {set.is_superset_of (s1, s2
   )} set.is_superset_of (s1, s2) == set.is_subset_of (s2,
   s1));

// proper Subset
function set.is_proper_superset (set, set) returns (bool);
axiom (forall s1:set, s2:set :: {set.is_proper_superset (s1
   , s2)} set.is_proper_superset (s1, s2) == (set.
   is_subset_of (s2, s1) && (s1 != s2)));

// proper Superset
function set.is_proper_subset (set,set) returns (bool);
axiom (forall s1:set,s2:set :: {set.is_proper_subset (s1,
   s2)} set.is_proper_subset (s1,s2) == (set.is_subset_of (
   s1, s2) && (s1 != s2)));

// Substracted
function set.subtracted (set,set) returns (set);
axiom (forall s:set :: {set.subtracted(s,set.make_empty)}
   set.subtracted(s,set.make_empty) == s);
axiom (forall s1:set,s2:set,x:any :: {set.subtracted(s1,set
   .extended(s2,x))} set.subtracted(s1,set.extended(s2,x))
   == set.pruned(set.subtracted(s1,s2),x));

// Difference
function set.difference (set,set) returns (set);
axiom (forall s1:set,s2:set :: {set.difference(s1,s2)} set.
   difference(s1,s2) == set.subtracted(set.united(s1,s2),
   set.intersected(s1,s2)));
```

BIBLIOGRAPHY

[1] Jean-Raymond Abrial. *The B-Book – assigning programs to meanings*. Cambridge University Press, 1996.

[2] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *Proceedings ECOOP'97*, volume 1241 of *LNCS*, pages 32–59. Springer-Verlag, June 1997.

[3] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, March 2003.

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

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

[6] Eric Bezault and Emmanuel Stapf. A free ELKS implementation. `http://sourceforge.net/projects/freeelks/`, April 2007.

[7] Juan Bicarregui. Algorithm refinement with read and write frames. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, pages 148–161. Springer-Verlag, April 1993.

[8] Thomas Bietenhader. Formal semantic specification of a core object-oriented language. Master's thesis, ETH Zurich, 2004.

[9] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *ACM Symposium on Principles of Programming Languages (POPL) 2003*, New Orleans, Louisiana, January 2003.

[10] 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.

[11] Samuel Burri. Translation of object-oriented programs into guarded commands. Master's thesis, ETH Zurich, 2005.

[12] Patrice Chalin, Joseph Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with jml and esc/java2. In *The 4th International Symposium on Formal Methods for Components and Objects (FMCO 2005)*, Amsterdam, The Netherlands, November 2006.

[13] 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.

[14] Keith L. Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*. Plemum Press, New York, 1978.

[15] David Crocker and Judith Carlton.

[16] Á. Darvas and P. Müller. Reasoning About Method Calls in Interface Specifications. *Journal of Object Technology (JOT)*, 5(5):59–85, June 2006.

[17] Á. Darvas and P. Müller. Faithful mapping of model classes to mathematical structures. In *6th Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, New York, NY, USA, 2007. ACM Press.

[18] Leonardo de Moura and Nikolaj Bjørner. *Z3: an efficient SMT solver*. Microsoft Research, November 2007. `http://research.microsoft.com/projects/z3/`.

[19] Robert DeLine and Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, March 2005.

[20] David Detlefts, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories Palo Alto, July 2003.

[21] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[22] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[23] Edsger W. Dijkstra. *A Discipline of Programming*. Tandem Library Books, 1997. ISBN 0-613-92411-8.

[24] Hartmut Ehrig, Bernd Mahr, Martin Große-Rhode, Felix Cornelius, and Philip Zeitz. *Mathematisch-strukturelle Grundlagen der Informatik*. Springer, 2. edition edition, 2001.

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

[26] Eiffel Software Inc., 356 Storke Road, Goleta, California 93117 USA. *Online Eiffel Documentation*, November 2007. `http://docs.eiffel.com/`.

[27] Jean-Christophe Filliâtre, Thierry Hubert, and Claude Marché. *The Caduceus verification tool for C programs*, version 1.05a edition, October 2007.

[28] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, volume 37, pages 234–245, June 2002.

[29] Abraham A. Fraenkel, Yehoshua Bar-Hillel, and Azriel Levy. *Foundations of Set Theory*. North-Holland Publishing Company, 1973.

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[31] Adele Goldberg and David Robson. *Smalltalk-80, the language*. Addison Wesley, September 1989.

[32] Aaron Greenhouse and John Boyland. An object-oriented effects system. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 205–229, London, UK, 1999. Springer-Verlag.

[33] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

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

[35] Tony Hoare and Jay Misra. Vision of a Grand Challenge project. A position paper for the conference Verified software: theories, tools, expriments, Zurich, July 2005.

[36] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM 2006: Formal Methods*, volume 4085 of *LNCS*, pages 268–283. Springer Verlag, Heidelberg, 2006.

[37] Ioannis T. Kassios. *A Theory of Object Oriented Refinement*. PhD thesis, University of Toronto, Department of Computer Science, 2006.

[38] Markus Keller. Catching cats. Master's thesis, Deparement Informatik, ETH Zurich, March 2003.

[39] Erich Laube. Design and implementation of a jml frontend for boogie. Master's thesis, ETH Zurich, 2006.

[40] Gary T. Leavens. An overview of larch/c++: Behavioral specifications for c++ modules. Technical Report 96-01e, Department of Computer Science, Iowa State University, 1996.

[41] 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.

[42] Gary T. Leavens et al. Jml tools. Published by Iowa State University, `http://www.jmlspecs.org/`.

[43] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA '00 Companion*, pages 105–106. ACM, August 2000.

[44] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA '98 Conference Proceedings*, volume 33 of *ACM SIGPLAN Notices*, pages 144–153, October 1998.

[45] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pages 50–59, New York, NY, USA, 1974. ACM Press.

[46] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969. reprinted in McC90.

[47] 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.

[48] Bertrand Meyer. *Eiffel: the language*. Object-Oriented Series. Prentice Hall, New York, NY, 1991.

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

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

[51] Bertrand Meyer. *Reusable software: the Base object-oriented component libraries*. Prentice-Hall, 1994.

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

[53] 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.

[54] Bertrand Meyer. Proving pointer program properties part 1: Context and overview. *Journal of Object Technology*, 2(2):87–108, March–April 2003.

[55] Bertrand Meyer. Proving pointer program properties part 2: The overall object structure. *Journal of Object Technology*, 2(3):77–100, May–June 2003.

[56] Bertrand Meyer, editor. *Eiffel Analysis, Design and Programming Language*. ECMA International, June 2005. As approved as International Standard 367.

[57] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In Eric Jul, editor, *ECOOP '98 — Object-Oriented Programming, 12th European Conference, Brussels, Proceedings*, volume 1445, pages 355–382, July 1998.

[58] Richard Mitchell and Jim McKim. *Design by Contract, by example*. Addison-Wesley, 2002.

[59] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*, pages 131–140. Fernuniversität Hagen, 1999. Technical Report 263.

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

[61] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

[62] 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.

[63] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing, July 1999. `http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html`.

[64] Piotr Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming*. PhD thesis, ETH Zurich, February 2007. No. 17061.

[65] Nonprofit International Consortium for Eiffel (NICE). *The Eiffel Library Standard*, June 1995. TR-EI-48/KL.

[66] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, May 207.

[67] Jonathan S. Ostroff, Chen-Wei Wang, Eric Kerfoot, and Faraz Ahmadi Torshizi. Automated model-based verification of object-oriented code. In *Verified Software: Theories, Tools, Experiments*, pages 18–26. Microsoft Technical Report MSR-TR-2006-117, 2006.

[68] Jonathan S. Ostroff, Chen-Wei Wang, Eric Kerfoot, and Faraz Ahmadi Torshizi. Automated model-based verification of object-oriented code. In *Workshop on Verified Software: Theory, Tools, and Experiments*, number MSR-TR-2006-117. Microsoft Research, August 2006.

[69] Richard Paige and Jonathan Ostroff. The single model principle. *Journal of Object Technology*, 1(5):63–81, November-December 2002. `http://www.jot.fm/issues/issue_2002_11/column6`.

[70] Richard Paige and Jonathan S. Ostroff. ERC — an Object-oriented Refinement Calculus for Eiffel. *Formal Aspects of Computing*, 16:51–79, 2004.

[71] M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. In *Symposium on Principles of Programming Languages (POPL) 2007*, January 2007.

[72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[73] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[74] R. Reiter. On closed world data bases. In *Readings in nonmonotonic reasoning*, pages 300–310. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[75] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, July 2002.

[76] 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.

[77] Bernd Schoeller, Tobias Widmer, and Bertrand Meyer. Making specifications complete through models. In Ralf Reussner, Judith Stafford, and Clemens Szyperski, editors, *Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science*. Springer-Verlog, 2006.

[78] Mark Slagell. Ruby user's guide. `http://www.ruby-lang.org/en/documentation/`, April 2007.

[79] J. M. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, 1989. January.

[80] J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, second edition edition, 1992.

[81] Sun Microsystems, Inc. *Java 2 Platform Standard Ed. 5.0 API*, 2004.

[82] Kim Waldén. *Handbook of Object Technology*, chapter Business Object Notation (BON). CRC Press, 1998.

[83] Kim Waldén and Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture*. Prentice Hall, 1994.

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

[85] Macro Zietzling. Complete contracts for EiffelBase. `http://se.inf.ethz.ch/projects/marco_zietzling/`, 2007. Semester thesis report.

Curriculum Vitae

Name:	Bernd Kurt Alexander Schoeller
Titel:	Diplom-Informatiker
Geburtsdatum:	5. April 1974
Geburtsort:	Aachen, Deutschland
Vater:	Dipl.-Ing. Jochen Schoeller
Mutter:	Ina Schoeller (geb. von Diringshofen)
Nationalität:	Deutsch
Anschrift:	Burstwiesenstrasse 19 8055 Zürich Schweiz

Ausbildung

1980 – 1984	Martin Luther Grundschule, Düren
1984 – 1993	Stiftisches Gymnasium, Düren *Abschluss: Abitur, allgemeine Hochschulreife*
1993 – 2001	Diplomstudium Informatik, Technische Universität Berlin *Abschluss: Diplom-Informatiker*
2002 –	Doktoriatsstudium, ETH Zürich

Beruf

1996 – 1998	Studentische Hilfskraft Technische Universität Berlin
1998 – 1999	Software-Developer Novedia GmbH Systemhaus, Berlin
2002 –	Wissenschaftlicher Assistent, ETH Zürich