Master Thesis

Reusable Mathematical Models

Tobias K. Widmer
widmer@id.ethz.ch

Supervisors:

Prof. Dr. B. Meyer
B. Schoeller

Chair of Software Engineering
Department of Computer Science, ETH Zurich

July 13, 2004
Abstract

In this thesis we develop a framework of reusable mathematical models intended to be used in EIFFEL contracts. As first steps towards this goal we identify the necessary mathematical concepts and compare already existing approaches to formal software development to the current specification-related facilities provided by the EIFFEL language.

Equipped with this theoretical background, we derive a new contract language named Intermediate Functional Language (IFL), which is specifically designed to support high-level mathematical constructs from set theory and relation algebra. To give the proof of concept, we extend some existing classes of the EIFFELBASE library with mathematical model contracts.

The principal contribution of this thesis consists of a set of library classes bundled in the Mathematical Model Library (MML), providing the necessary support to write powerful and expressive model contracts in IFL. A language processing tool is presented that helps in automating the process of extracting the model contracts of an IFL annotated class and generating the associated proof obligations.
## Contents

1 Introduction ........................................... 7

2 Background ........................................... 9
   2.1 The B Method ......................................... 9
   2.2 The JML Specification Language ...................... 10
   2.3 The Eiffel Method ..................................... 10

3 Introducing Mathematical Models ...................... 13
   3.1 Overview ........................................... 13
   3.2 Mathematical Model Terminology ....................... 13
   3.3 Specification with Mathematical Models ................ 15
   3.4 Implications of Mathematical Models ................... 18
   3.5 Conclusion ......................................... 20

4 Mathematical Concepts ................................ 21
   4.1 Overview ........................................... 21
   4.2 Syntax and Notation .................................. 22
   4.3 Fundamental Logic .................................... 22
      4.3.1 Quantifiers ..................................... 23
   4.4 Sets ................................................. 23
      4.4.1 Set Equality/Set Membership ...................... 23
      4.4.2 Set Containment ................................ 24
      4.4.3 Set Union/Set Intersection ....................... 24
      4.4.4 Set Difference ................................ 25
   4.5 Tuples .............................................. 25
      4.5.1 Ordered Pairs .................................. 25
      4.5.2 Tuple Projection ................................ 26
   4.6 Relations ........................................... 26
      4.6.1 Definition by Extension ......................... 26
      4.6.2 Definition by Comprehension ...................... 26
      4.6.3 Binary Relations ................................ 27
         4.6.3.1 Identity Relation ............................ 27
         4.6.3.2 Domain/Range Projection .................... 27
      4.6.4 Relation Composition ............................. 28
      4.6.5 Relation Restriction ............................. 28
      4.6.6 Relation Inversion ............................... 29
## CONTENTS

### 10 Summary of this Document

### 11 Conclusion

#### A IFL Grammar

- A.1 Syntax and Notation .......................................................... 115
- A.2 The Grammar Rules ............................................................ 115

#### B MML Library Features

- B.1 IFL Language Concepts ....................................................... 119
- B.2 MML Library Classes ......................................................... 119
  - B.2.1 MML_ANY ................................................................. 120
  - B.2.2 MML_PAIR .............................................................. 120
  - B.2.3 MML_SET ............................................................... 120
  - B.2.4 MML_POWERSET ...................................................... 121
  - B.2.5 MML_RELATION ....................................................... 121
  - B.2.6 MML_FUNCTION ...................................................... 122
  - B.2.7 MML_BAG ............................................................... 122
  - B.2.8 MML_SEQUENCE ...................................................... 123
  - B.2.9 MML_GRAPH ........................................................... 123
- B.3 MML Generic Operations .................................................. 124
- B.4 MML Conversion Operations .............................................. 124

#### C EiffelBase Class Specifications

- C.1 EiffelBase Model Structures ............................................. 125
  - C.1.1 LINEAR_SUBSET .................................................... 125
  - C.1.2 BAG ................................................................. 128
  - C.1.3 DYNAMIC_LIST .................................................... 130
  - C.1.4 STACK .............................................................. 131
  - C.1.5 QUEUE .............................................................. 132
- C.2 EiffelBase Class Hierarchy ............................................... 133

#### D Mathematical Definitions

- D.1 Fundamental Logic .......................................................... 135
- D.2 Sets ............................................................................. 135
- D.3 Tuples ........................................................................ 136
- D.4 Relations .................................................................. 136
- D.5 Functions .................................................................. 136
- D.6 Sequences .................................................................. 137
- D.7 Multisets .................................................................. 137
- D.8 Functionals ................................................................ 137
Chapter 1

Introduction

Proving programs in a fully mathematical and rigorous way is hard and time-consuming. Although there exist a number of formal approaches to software engineering like the Z specification language or the B method, these methods strongly focus on refinement-based programming and do not directly support true object-oriented programming as the EIFFEL method does. On the other side, the JML specification language is an object-oriented extension to the JAVA programming language which allows annotating existing JAVA programs with specification-related attributes and contracts. But neither of these approaches really take full advantage the most beneficial concept in software engineering: Reuse of already available software elements. It should not only be possible to reuse units of implemented functionality, but also to rely on units of already proved software components. Unfortunately, the traditional formal methods as B only gradually support the reuse of proved components and their associated proof obligations.

Unlike in JML, contracts are first-class citizens of the EIFFEL programming language [Mey92, Mey97]. Designed as taking an integral part in the software specification process and providing semiformal definitions of class interfaces, they turn out not to be powerful enough to express the complicated relationships in object structures of modern object-oriented environments. Contracts in the EIFFEL programming language are Boolean expressions written in EIFFEL itself, thus providing no support for higher-level expressions of first-order or higher-order predicate calculus. We are faced with several shortcomings, meaning on the one hand the monolithic and inflexible proof process of formal methods like B, and on the other hand the restricted possibilities of formal specification using Design by ContractTM [Mey97] of the EIFFEL method. A combination of the reusability and extendibility of object-oriented methods with the rigorous and powerful mathematical methods used in formal development seems therefore reasonable. The idea of applying mathematical models to prove contract-equipped EIFFEL classes has been initiated by Meyer in [Mey03]. A nice overview is given by Schoeller in [Sch03]. As part of the ongoing effort to prove contract-equipped EIFFEL classes, an open question is how the necessary mathematical models can be modeled as software components to assist client programmers in writing even more expressive and powerful contracts, using ordinary Boolean expressions in EIFFEL assertions.

The ultimate goal of this thesis is therefore to create a collection of extendible and reusable software components that allow EIFFEL programmers to write powerful contracts using abstract mathematical models.
Chapter 2

Background

In this chapter we give a short overview of existing approaches to formal or semi-formal software development that are of direct interest for the design of the Intermediate Functional Language (IFL) and the Mathematical Model Library (MML). The overview is designed to support the decision what particular mathematical concepts are crucial for the further discussions of this report and how they can be integrated into a set of powerful model classes. We choose three unique and hopefully representative approaches to software construction.

Section 2.1 presents B, a fully mathematical method of software development. Section 2.2 provides an overview of the JML Specification Language. Finally, in section 2.3 we discuss a semi-formal approach, the EIFFEL method, which the following presentation is based on.

2.1 The B Method

The B Method [Abr96] is a formal mathematical approach to software specification and development based on the Z specification language from the same author. It has been successfully applied in industry, and has robust, commercially available tool support for the entire development life-cycle, from specification through refinement down to code generation.

Each stage in the development process has a verification step. Specifications in B are mathematical models of the required behavior of a system. These specifications are then transformed through a sequence of formally defined refinement steps towards a concrete mathematically proved implementation, for which source code can be generated. During this process, the developer has to discharge a number of proof obligations.

B uses the Abstract Machine Notation [B-C] for the specification of abstract machines, the unit of proof in the B Method. Abstract machines contain state (a set of variables constrained by an invariant) and applicable operations (operations that may change the state, while maintaining the invariant, and may return a sequence of results). The foundations of B machines are the concepts of substitution and a large set of basic mathematical constructs from set theory and relation algebra.

In the following discussions of this report, we are not taking the rigorous mathematical road of software engineering as the B Method does, but would like to use the broad foundation of set theory and relation algebra used in the B Method for our mathematical models. Further, we would like to design the MML class library in order to support mechanical proofs on mathematical models as the B Method allows it on abstract machines.
2.2 The JML Specification Language

The JML language \([LPC^+, \ LBR]\) is a \textit{behavioral interface specification language} (BISL) tailored to \textsc{java}, taking the main characteristics of the \textsc{eiffel} \cite{Mey92} and the \textsc{Larch} \cite{Lea96, CL94} approach, blending them to specification purposes. The language uses the \textsc{javadoc} commenting facilities of the \textsc{Java} programming language to annotate existing \textsc{Java} classes with JML specifications.

JML is designed to be used by working software engineers and requires only basic mathematical education. To achieve this, the JML language follows the \textsc{eiffel}-style assertion syntax combined with a \textit{model-based approach} to specification as pioneered by VDM and \textsc{Larch}. However, JML not only supports Boolean expressions, but also features quantifiers, specification-only variables, frame conditions and other enhancements that make it more expressive for specification than \textsc{eiffel}. The language also allows assertions to be intermixed with \textsc{Java} code as an aid to verification and debugging.

Leavens and Cheon \cite{LC} show how Design by Contract\textsuperscript{TM} \cite{Mey97} can be simulated in JML, and Breuuesse and Poll \cite{BP03} present how JML specifications can be verified using model fields. Labeth, Meyer, Müller and Poetzsch-Heffter \cite{LMMPH00} do a formal verification of a doubly linked list specified with JML using the \textsc{jive} system.

The JML language and its \textit{model-based approach} to specification provides the basis for the development of a library of \textit{reusable mathematical models} (MML) and the associated \textit{contract language} \textsc{ifl}.

2.3 The Eiffel Method

The following text has been adapted and shortened from \cite{Wik}.

\textsc{eiffel} \cite{Mey92} is a pure \textit{object-oriented programming language} which emphasizes the production of \textit{robust software}. It is strongly statically typed and furnished with \textit{automatic memory management} (typically implemented by \textit{garbage collection}).

Created in 1985, \textsc{eiffel} has become a mature programming language with development tools available from multiple suppliers. Characteristics of \textsc{eiffel} include Design by Contract\textsuperscript{TM} \cite{Mey97}, the sometimes too liberal use of \textit{implementation inheritance} including \textit{multiple inheritance}, and \textit{genericity}. Design by Contract\textsuperscript{TM} is a method to annotate classes with assertions, preconditions, postconditions, loop variants and invariants.

Assertions in the \textsc{eiffel} language are simple \textit{Boolean expressions} of first-order predicate logic written in the language itself. \textit{Existential} and \textit{universal quantifiers} are partially provided by library classes. However, no higher-order functions and related constructs are supported. \textsc{eiffel} has a \textit{unified type system}: All types in \textsc{eiffel} are classes, making it possible to create \textit{descendants} of basic classes such as \textsc{character}. Additionally, the \textsc{eiffel} language allows \textit{operator overloading}, including the ability to define new operators, but does not have \textit{method overloading}.

The \textsc{eiffel} language aims to promote clear and elegant coding. It emphasizes \textit{declarative statements} over \textit{procedural code} and intentionally limits stylistic expression, leaving much freedom for optimizations to the compiler. The simplicity of the language not only makes the code more readable, but also allows a programmer to concentrate on the important aspects of a program without getting distracted by implementation details. \textsc{eiffel}’s simplicity is intended to promote simple, readable, usable, reusable and reliable solutions to business problems.

The \textsc{eiffel} language will certainly influence the design of the \textsc{ifl} language (see chapter 5). We adopt the idea of using \textsc{eiffel}, or a subset of it, as the \textit{contract language} for specification with
2.3. **THE EIFFEL METHOD**

mathematical models. We also reuse the support for *run-time assertion checking* offered by all EIFFEL run-time systems. The class library presented in chapter 6 is supposed to fill in the missing concepts of EIFFEL assertions, complementing the IFL language with a set of reusable model classes.
Chapter 3

Introducing Mathematical Models

This chapter introduces mathematical models as a tool to rigorously specify EIFFEL classes with expressive contracts.

3.1 Overview

The presentation of mathematical models is laid out as follows: Section 3.2 introduces the basic notions and concepts of mathematical models. We distinguish between two types of contract assertions, regular assertions and model assertions. Section 3.3 is dedicated to applying model contracts to existing software classes. We present the process of choosing the most suitable mathematical model, and discuss how existing EIFFEL classes can be extended with abstract mathematical models. Section 3.4 discusses the benefits of specifications with mathematical models, using illustrative examples where appropriate.

Finally, section 3.5 summarizes the discussions of the previous sections and lists the most important consequences of applying mathematical models to EIFFEL classes.

3.2 Mathematical Model Terminology

Before discussing the benefits and implications of mathematical models (see section 3.4), it seems convenient to introduce a few important concepts which are needed to fully describe the process of annotating existing EIFFEL classes with mathematical models. We proceed in the order of definition, giving for each concept a rigorous description, complementing it with an illustrative example where appropriate.

The basic concepts of mathematical models comprise the following definitions:

- **Regular Assertion**: Boolean EIFFEL expression used as assertion in EIFFEL contracts. Regular assertions are the set of assertions appearing in contracts before any annotation with mathematical models is made. They correspond to normal EIFFEL assertions as used by the EIFFEL language to support the Design by Contract™ [Mey97] method. EIFFEL assertions have the following characteristic properties:
• The expressions must conform to Boolean types. Each expression therefore yields a result conforming to type BOOLEAN when evaluated.

• The expressions should be pure. A pure expression is an expression that does not change the state of the current object (in EIFFEL denoted by the predefined entity ‘Current’). In the current EIFFEL standard, purity of expressions is not enforced. Consequently, pure expressions are only allowed to use attributes or pure queries.

• The expressions can be verified at run-time. Their execution can be monitored, triggering an exception if the expression is evaluated to ‘False’ and resulting in the abortion of the program execution.

• Abstraction Function: General function of type $f : X \rightarrow Y$, where $X$ is the base type of the EIFFEL class $C$ being specified (or in the case where $C$ is a generic class, the family of types represented by class $C$) and $Y$ represents the mathematical abstraction computed by $f$. Let us illustrate this concept with an example:

Example: Consider an instance of a class LINKED_LIST $[G]$. The instance contains a possibly empty set of elements at certain indexes, depending on the state of computation. Suppose the linked list contains the elements $a_1, a_2, ..., a_n$ at a certain point of time during the computation. A possible implementation of an abstraction function $f : \text{LINKED\_LIST} \rightarrow \text{SEQUENCE}$ iterates through the list instance and constructs the mathematical equivalent of a linked list, a mathematical sequence $[a_1, a_2, ..., a_n]$ (see section 4.8.1).

• Abstraction Invariant: EIFFEL class invariant describing the relation between the current object state and the mathematical abstraction computed by an abstraction function.

• Model Assertion: Regular assertion conforming to the restrictions necessary to express properties about mathematical models. These restrictions are discussed in chapter 5 where the Intermediate Functional Language (IFL) is presented. Additionally, model assertions have to contain only feature calls to model features (see definition 3.2).

• Model Feature: EIFFEL feature designed to represent the current state of an object in an abstract mathematical view. A model feature is a pure query which extracts the full state of an object at run-time and captures one or more mathematical facets of the state of computation at a certain point of time. A model feature must be designed according to the following rules:

  • Every model feature is the EIFFEL implementation of an abstraction function.
  
  • Every model feature implementation in EIFFEL must be pure, thus not changing the current state of the computation on the current object.
  
  • Every model feature computes the mathematical abstraction of the current object’s state before the execution of the model feature.

• Regular Contract: EIFFEL contracts consisting of regular assertions only.

• Model Contract: EIFFEL contracts where only feature calls to model features are allowed. Model contracts are playing an important role in the proof process of contracts, which is described in detail in chapter 8.
3.3 Specification with Mathematical Models

- **Model Class**: Regular EIFFEL classes containing *model contracts*.

- **Gluing Invariant**: EIFFEL class invariant describing the relation between two or more *model features* of a class. A gluing invariant states properties and constraints involving several *model features*, describing global constraints of the mathematical models in an EIFFEL class. Consider the following example:

**Example**: The EIFFEL class `LINKED_LIST [G]` represents sequential lists with an *internal cursor*. Two basic *mathematical abstractions* can easily be identified: A mathematical *sequence* representing the ordered elements of the *linked list*, and an index pointing to the element where the cursor is currently located. Therefore we introduce two *model features* `sequence` and `cursor` of type `LINKED_LIST → SEQUENCE` and `LINKED_LIST → INTEGER`, respectively. To relate the two *model features*, we can specify which cursor positions are valid. The most obvious way to do this is to extend the *class invariant* with the following invariant clause:

\[
\text{cursor} \geq \text{sequence.lower\_bound} \text{ and } \text{cursor} \leq \text{sequence.upper\_bound} \quad (3.1)
\]

Expression 3.1 states that the cursor always stays on a existing element in the *sequence model*.

### 3.3 Specification with Mathematical Models

The acquired insights of section 3.2 allow us now to give an exact procedure to annotate an existing EIFFEL class with mathematical models. This procedure is divided into the following steps:

1. **Model Selection**: Select the appropriate mathematical models for a class. We give two basic criteria: *Mathematical Abstraction* and *Equivalence Partitions*.

   (a) To determine mathematical models of a class by using *mathematical abstraction*, we have to extract the mathematical notion represented by the class implementation. In many cases, EIFFEL classes are implementations of *Abstract Data Types* (ADT’s). In this case, the selection of the necessary mathematical models is straightforward in most cases and just requires a translation from the ADT syntax to pure mathematical notation. Let us consider an example to illustrate this:

   **Example: Undirected Weighted Graph**

   Consider an EIFFEL class `GRAPH [W]` which represents undirected graphs with an associated weighting of type `W` defined on its edges. The class provides features to add and remove nodes from the graph, change the weight of an edge, and others. From a mathematical point of view, a graph `G` can simply be defined by a vertex set `V` and a relation `E` defined on `V` indicating the edges of the graph. Therefore relation `E` has signature `E : V ↔ V`. In the case of a weighted graph, we have an additional weighting function `W : V × V → \mathbb{R}` which is defined on the subset `E` of the *Cartesian product* of the vertex set. Finally, a weighted graph `G` is completely defined by a 3-tuple of sets `G ≜ (V, E, W)` with `E : V ↔ V` and `W : V × V → \mathbb{R}`. Therefore we use three different mathematical models to represent the node set, the edge relation and the weighting function. Note that the above-mentioned tuple could be represented as a single model as well. However, this would yield more complicated formulas for *model assertions*, and *gluing invariants* in particular. Therefore we decide to use three separate models in favor of simplicity.
(b) The other criterion, *equivalence partitions*, is based on the *object equality relation* defined on every instance of an Eiffel class. For every class annotated with mathematical models, consider its features in the *flat form*. For each of these features, list all the attributes of the class that are changed during the execution of the corresponding feature, and similarly, the attributes that are not affected by its execution. This results in a partition of *class attributes* with respect to a certain feature; a set of attributes that is changed by the feature and a set of unchanged attributes. This partition can be helpful as a hint which *class attributes* a particular model should cover, or in other words, what the domain (the *Cartesian product* of a subset of all *class attributes*) of the *abstraction function* of that particular model is. Let us illustrate this concept with a simple example:

**Example: Linked List with Internal Cursor**

Consider the Eiffel class `LINKED_LIST [G]` implementing a list using linked cells, and providing an *internal cursor* which can be used to iterate over the list elements. Additionally, the cursor has two positions before and after, denoting positions before the first element and after the last element, respectively. Usually, *linked lists* provide features to append or insert new elements, features to remove elements and in this case features to move the cursor over the list elements. As an example, let us take a feature `put_left (element : G)` with the following semantics:

- If the cursor is *before*, this feature is not applicable
- If the cursor is *after*, the element is inserted at the end of the *linked list*, and the cursor stays *after*.
- In all other cases, the implementation inserts the element to the left of the current cursor position and let the cursor point to the same element as before.

If we consider this feature now with respect to *equivalence partitions*, it is evident that in all the defined cases for the feature `put_left` the cursor stays on the same element (or in other words the *cursor values* of the pre-state and the post-state are *equivalent*), whereas the *linked list elements* are changed by the operation. Note however that *equivalence* of the *cursor position* does not necessarily mean *immutability* of the *cursor value* itself, since all indexes of the elements to the right hand side of the old cursor position are increased. The arguments above naturally suggest to separate *cursor index* and *linked list elements* and to use separate mathematical models for each one of them.

2. **Model Implementation:** Implement the selected mathematical model (or the *model feature*, respectively) as a *pure* Eiffel query. The implementation of a *model feature* \( f \) must satisfy the following conditions:

(a) Feature \( f \) must not have a *precondition*. Since *model features* are evaluated in a global context, they have to be applicable at all times during the execution of a system.

(b) The *postcondition* of feature \( f \) states the *abstraction invariant* of \( f \). This is not mandatory, since the *abstraction invariant* is expressed anyway in the invariant clause of the corresponding class. For reasons of better readability and documentation we recommend stating it again in the *postcondition*.

(c) The implementation of \( f \) must not change the state of the current object, nor must it change any feature arguments received by feature calls of features having *model assertions* involving \( f \). See section 5.3 for a definition of *pure expressions*. 
(d) The mathematical model constructed by \( f \) reflects the state of the current object at the time of the feature call. More details on how to construct mathematical models from the object state can be found in chapter 6 and in particular in section 6.4.

3. **Contract Specification**: Specify contracts of a feature \( f \) in terms of the implemented mathematical models and its model features. There are only a few general rules how model contracts are specified. In most cases, the specification is feature-dependent and cannot be systematically determined. However, we would like to give a few general rules of advice:

(a) If a model \( m \) is changed by \( f \), introduce a model assertion in the postcondition of \( f \) ensuring object equality of the model \( m \). In EIFFEL, there is a convenient way to use the pre-state value of an object in a postcondition, which we can use in this case. This special kind of assertion is called immutable model assertion:

\[
m \text{.is_equal (old } m)\]

(b) Sometimes, we may also want to specify that a feature always causes changes in the model \( m \), which can be expressed using:

\[
\text{not m.is_equal (old } m)\]

(c) Similar to immutable model assertions, we introduce another kind of assertion to state that an actual feature argument is not changed by the feature itself. This assertion type is called immutable argument assertion (see also section 7.2) and is used to declare a feature as side-effect free with respect to the feature arguments.

(d) However, the most important advice to contract specification is to annotate pure features with a special pure feature assertion (see also section 7.2). The EIFFEL programming method strongly recommends to follow the Command/Query Separation Principle [Mey97], a design rule which states that a feature is either a command (a procedure which changes the state of the current object) or a query (a pure function). A more precise definition of pure features can be found in section 5.2.2. We use a pure feature assertion to express that a feature is indeed a pure query which does not change the state of the current object:

\[
\text{is_equal (old } Current)\]

This is obviously also valid for model features which justifies a pure feature assertion for every model feature.

**Note**: Due to a problem in the language specification of the EIFFEL programming language, the semantics of pure feature assertions are not the ones we desire. Unfortunately, entities of reference type are evaluated only as reference value in the pre-state. Therefore, pure feature assertions always evaluate to 'True'. Since these kinds of assertions are tautologies, we use them for documentation purposes only. The desired semantics for the code snippet 3.4 would require an adapted object equality relation. For further information see section 6.6.

4. **Equality Relation**: Redefine the feature \( \text{is_equal (other: like Current)} \) in order to provide a new implementation involving the model \( m \). With the introduction of a new model \( m, object
equality also requires the two corresponding models of the compared objects to be equal, which
has to be checked by any new implementation. Additionally, the postcondition of \texttt{is\_equal} has
to be updated accordingly to account for model equality.

5. \textbf{Gluing Invariants}: For every new model \( m \) introduced in a class, state a corresponding \textit{gluing}
invariant relating the model directly or indirectly to all other models in the flat form of the
class. Let us again give an example using the previously introduced class \texttt{LINKED\_LIST [G]} to
illustrate the concept of gluing invariants:

\begin{itemize}
  \item \textbf{Example: Linked List with Sequence and Cursor Model}
\end{itemize}

Consider the \texttt{EIFFEL} class \texttt{LINKED\_LIST [G]} introduced in step (1), but extended with two
immediate model features called \texttt{sequence} and \texttt{cursor}. A natural property of the cursor is to
have a valid position during the life-time of a linked list instance, e.g. a position that always
 corresponds to a element in the list. We use this property now to express a \textit{gluing invariant}
relating the \texttt{cursor model} to the \texttt{sequence model}:

\begin{equation}
\texttt{cursor} \geq \texttt{sequence.lower\_bound} \quad \text{and} \quad \texttt{cursor} \leq \texttt{sequence.upper\_bound}
\end{equation}

\section{3.4 Implications of Mathematical Models}

We have introduced mathematical models in the last few sections of chapter 3. After this theoretical
background it is important to realize the consequences of mathematical models and what benefits and
improvements models can bring when using it within EIFFEL contracts. The following properties are
noteworthy:

\begin{itemize}
  \item \textbf{State Change Specification}: Mathematical models give the developer the ability to rigorously
express which parts of the state space of an object change. Let us illustrate this statement with
our linked list example:

\begin{itemize}
  \item \textbf{Example: Appending Element to Linked List}
\end{itemize}

Consider again the \texttt{EIFFEL} class \texttt{LINKED\_LIST [G]} of section 3.3 having two immediate model
features called \texttt{sequence} and \texttt{cursor}. If we want to specify the state changes of a feature
\textit{extend} (\texttt{element: \texttt{G}}) which appends a new element to the linked list, we use the two models to express
these state changes separately:

\begin{align}
\texttt{sequence.is\_equal (old sequence.extended (element))} & \quad (3.6) \\
\texttt{cursor} \leq \texttt{sequence.upper\_bound} & \text{ implies } \texttt{cursor.is\_equal (old cursor)} \quad (3.7) \\
\texttt{cursor} > \texttt{sequence.upper\_bound} & \text{ implies } \texttt{cursor.is\_equal (old cursor + 1)} \quad (3.8)
\end{align}

Assertions 3.6, 3.7 and 3.8 express the following properties: After the execution of the feature,
the \texttt{sequence model} is simply the \texttt{old sequence model} extended with the new element. Regarding
the cursor model, we have to differentiate between two cases: If the cursor is not in the \textit{after}
position, it points to the same element as before (which implies that the cursor value does not change). If the cursor is behind the last element in the \textit{after} position, the index of the \textit{after}
position itself is increased due to the insertion of the new element.
• **Minimal Contract Specification**: Mathematical models allow complicated relationships in the object structure of a system to be expressed with very few assertions. Experience has shown (see section 7.3) that the definition of a feature stated in its *postconditions* in most cases comprises only one line:

**Example**: *Sequence Concatenation*

Consider class `LINKED_LIST [G]` from the previous example. The specification of an imaginary feature `concatenate` (*other*: `like Current`) that concatenates two sequences is a simple feature call expression:

\[
\text{sequence.is_equal (old sequence.concatenated (other))} \quad (3.9)
\]

\[
\text{cursor.is_equal (old cursor)} \quad (3.10)
\]

• **High-Level Contract Specification**: Mathematical models allow the programmer to write high-level specifications without losing mathematical preciseness. With the support of a powerful class library providing EIFFEL classes for the most common *mathematical abstractions* (see chapter 6), even contracts for general graph problems of discrete mathematics can be expressed:

**Example**: *Graph Strong Connectivity Property*

Consider an EIFFEL class `DIRECTED_GRAPH` representing *strongly-connected directed graphs* with the associated mathematical models `nodes` and `edges`. To express the strong connectivity property we use a *model invariant* (or in this case a *gluing invariant*) which comprises only one line:

\[
\text{edges.reflexive_transitive_closure.is_subset (nodes.cart_product (nodes))} \quad (3.11)
\]

• **Correctness Proofs of Contracts**: Mathematical models not only make it possible to write more expressive contracts, but also provide a basis for *correctness proofs* of EIFFEL contracts. The concept of *contract flattening* can be directly used to generate *proof obligations* for assertions. Correctness proofs of contracts will be described in more detail in chapter 8.

It is evident that the above-mentioned statements are only conjectures and can only be confirmed by performing actual specification of EIFFEL classes. The experience gained in the specification of EIFFELBASE data structures (see chapter 7) has shown that these statements are indeed justified.
3.5 Conclusion

In this chapter, we have introduced mathematical models as an expressive tool to write more formal and powerful contracts in EIFFEL classes. In section 3.2 we have given definitions for the most important concepts used later on in the discussion. Section 3.3 is dedicated to a thorough discussion of mathematical models applied to EIFFEL classes. We have laid out the procedure of annotating existing classes have presented the following steps as a guideline to introduce mathematical models for an EIFFEL class:

- **Model Selection**: How are mathematical models chosen?
- **Model Implementation**: How are the corresponding model features implemented?
- **Contract Specification**: How are EIFFEL contracts specified using mathematical models?
- **Equality Relation**: How is the object equality relation implemented?
- **Gluing Invariants**: How are the necessary gluing invariants expressed?

In section 3.4, the benefits and consequences of mathematical models in general are presented. Four key benefits of class specification using mathematical models have been identified:

- **State Change Specification**: Partial state changes can be formally expressed
- **Minimal Specification**: Preciseness of mathematical models ensures minimal specification
- **High-Level Specification**: Encapsulation of complicated object relationships in class libraries makes high-level specifications possible
- **Correctness Proofs**: Mathematical models help in the verification process of regular assertions

The discussions of chapter 3 have shown that mathematical models are a powerful tool for specification, provided a broad mathematical background exists, complemented with a flexible, reusable and extendible class library of mathematical models. The mathematical background is discussed in the next chapter (chapter 4), and in chapter 6 we present a library of reusable mathematical model classes.
Chapter 4

Mathematical Concepts

After the introduction on mathematical models in the previous chapter, we present the associated mathematical objects and concepts. Mathematically inclined readers may skip the following sections of chapter 4 on the first reading and continue directly with the introduction to the Intermediate Functional Language (IFL) in chapter 5.

4.1 Overview

The chapter is structured as follows: For each mathematical concept, a rigorous definition is given associated with an informal description of the concept and how it is related to our goal of creating a powerful class library of mathematical models. To top the presentation of selected concepts off, we give some simple but hopefully instructive examples.

Section 4.3 presents the most fundamental properties of first-order predicate logic to build the basis for all following discussions. In section 4.4, the concept of a mathematical set is introduced. Next, section 4.6 gives the concept of a mathematical relation based on the notion of ordered pairs, having their first and second element in a specified domain and range set, respectively.

Building upon these two concepts, we study the generalized notion of a mathematical function in section 4.7. Following are the different notions of generalized sets in section 4.8, manifested by sequences (section 4.8.1) and multisets (section 4.8.2), which heavily rely on the previous concepts. Higher-order functions (section 4.9) build the powerful cornerstones for a modular and generic collection of mathematical facilities to manipulate not only sets, but also functions itself. We study the concepts of function composition (section 4.9.1), function currying (section 4.9.2) and function iteration (section 4.9.3) and present a set of useful generic functionals to manipulate collections in section 4.9.4.

During the course of this chapter we strongly obey the rule of definition before use, except for two cases: The relation symbol ↔ and the function symbol →. Relations and functions are used in the definition parts of the concept descriptions. For the sake of completeness, the definitions of these two symbols are given below:

\[ X \leftrightarrow Y \triangleq \mathbb{P}(X \times Y) \]  \hspace{1cm} (4.1)

\[ X \rightarrow Y \triangleq \{ f : X \leftrightarrow Y \mid \forall x : X \Rightarrow \exists_1 y : Y \Rightarrow (x, y) \in f \} \]  \hspace{1cm} (4.2)
4.2 Syntax and Notation

Before starting with the definitions of the most fundamental mathematical concepts, let us review the typography and notation used in the following discussions. The most important symbol is the definition symbol \( \doteq \). Contrary to mathematical conventions, where most of the time the same symbol \( = \) is used to denote either definition or equality, we would like to stress the defining nature of the following statements and therefore use the symbol \( \doteq \) to make this difference evident.

Single elements are written in lower case letters \( x \) and can denote both single elements or subsets. The exact meaning is defined by the surrounding context. Sets are usually denoted by capital letters \( X \). The powerset of a set \( X \) is written as \( \mathcal{P}(X) \). Further, we use some predefined sets that are listed below without further explanation:

- \( \mathbb{Z} \): Integers
- \( \mathbb{N} \subset \mathbb{Z} \): Non-Negative Integers
- \( \mathbb{R} \): Reals
- \( \mathbb{B} \doteq \{true, false\} \): Booleans

In our notation, the cardinality of a set is attached as an annotation to the set name. If we want to express the set of all subsets of \( X \) that have cardinality greater than one, we simply write \( \mathcal{P}(X) \).

Relations (see section 4.6) will also denoted by upper case letters \( R \), whereas functions are named with lower case letters \( f \).

The following sections of chapter 4 explain the necessary mathematical concepts using conventions introduced above.

4.3 Fundamental Logic

We begin with the most fundamental logic operations. Assuming two arbitrary elements \( x \) and \( y \) of set \( \mathbb{B} \), the following properties hold:

**Definitions:**

- \( x \land y \doteq true \) if \( x = true, y = true \)
- \( x \lor y \doteq false \) if \( x = false, y = false \)
- \( \neg x \doteq false \) if \( x = true \)
- \( x \Rightarrow y \doteq \neg x \lor y \)

**Description:**

The \( \land \) operator denotes logical conjunction. \( x \land y \) is only true if both operands are true. Similarly, the \( \lor \) operator denotes logical disjunction, yielding true if one or both operands are true. The logical negation \( \neg \) inverts the value of its operand, turning true into false and vice-versa.

The last operator symbolizes logical implication. \( x \Rightarrow y \) is true except if \( x \) is true and \( y \) is false at the same time. Note in particular that \( x \Rightarrow y \) is true whenever \( x \) is false. This is intuitively justified by the fact that anything can be concluded from wrong assumptions.
4.3.1 Quantifiers

For most of the following definitions, it is convenient to be able to express properties of several elements belonging to a certain class. To this end, we use the existential and universal quantifiers of predicate calculus.

**Notation:**
- $\exists$: Existential Quantifier
- $\forall$: Universal Quantifier

**Definitions:**
- $\exists x : X \bullet P(x) \triangleq true$ if there exists an $x$ in $X$ such that $P(x)$ is true
- $\forall x : X \bullet P(x) \triangleq false$ if there exists an $x$ in $X$ such that $P(x)$ is false

**Description:**
The existential quantifier $\exists$ expresses the fact that an element of a ground set $X$ contains at least one element $x$ such that the predicate $P(\cdot)$ is satisfied. Similarly, the universal quantifier $\forall$ expresses the property that all elements $x$ of a set $X$ satisfy a predicate $P(\cdot)$.

As an important consequence, the universal quantifier $\forall$ yields true for any empty set $X$, independently of the predicate. In this case, any existential quantification yields false. Obviously, the opposite holds too: If an expression $\exists$ is true for a predicate $P(\cdot)$ over a ground set $X$, the universal quantification $\forall x : X \bullet \neg P(x)$ yields always false.

4.4 Sets

The first mathematical structure we introduce is the elementary notion of a mathematical set.

4.4.1 Set Equality/Set Membership

The most fundamental properties of sets are set membership and set equality:

**Notation:**
- $=, \neq$: Set Equality, Set Inequality
- $\in, \notin$: Set Membership, Set Non-Membership

The properties $\neq$ and $\notin$ are the complements of the set equality and set membership properties.

**Definitions:**
- $x \neq y \triangleq \neg(x = y)$
- $x \notin Y \triangleq \neg(x \in Y)$

**Description:**
A set $X$ is equal to a set $Y$ if and only if it has the same cardinality and equality holds for all members of the sets. The type of the equality properties is therefore $X \leftrightarrow Y$. The type of the membership properties is similarly deduced as $X \leftrightarrow \mathcal{P}X$. 
CHAPTER 4. MATHEMATICAL CONCEPTS

4.4.2 Set Containment

Most of the remaining set relationships of the category set containment can be described using the subset property.

Notation:
- ∅: Empty Set
- \( \mathbb{P}_1 \): Non-Empty Subset
- \( \subseteq, \subset \): Subset, Proper Subset

Definitions:
\[
\begin{align*}
\emptyset X & \triangleq \{ x \in X \mid \text{false} \} \\
\mathbb{P}_1 X & \triangleq \{ S \in \mathbb{P}X \mid S \neq \emptyset X \} \\
S \subseteq T & \triangleq \{ x \in X \mid S \in \mathbb{P}X, T \in \mathbb{P}X, x \in S \Rightarrow x \in T \} \\
S \subset T & \triangleq \{ x \in X \mid S \in \mathbb{P}X, T \in \mathbb{P}X, S \subseteq T \land S \neq T \}
\end{align*}
\]

Description:
The empty set \( \emptyset \) is a set which contains no elements. Therefore its cardinality is zero.

Informally, a set \( S \) is a subset of a set \( T \) if every member of \( S \) is also a member of \( T \). More strongly, a proper subset \( S \) is a subset which is not equal to the set \( T \). Additionally, for any set \( X \), \( \mathbb{P}_1 X \) denotes the set of all subsets of \( X \) that have cardinality greater than one (or in other words, are not equal to the empty set \( \emptyset \)).

4.4.3 Set Union/Set Intersection

Set union and set intersection are binary operations on sets. We directly use the generalized form of these operations for the following definitions.

Notation:
- \( \cup \): (Generalized) Set Union
- \( \cap \): (Generalized) Set Intersection

Definitions:
\[
\begin{align*}
\cup S & \triangleq \{ x \in X \mid S \in \mathbb{P}(\mathbb{P}X), \exists T \in S \bullet x \in T \} \\
\cap S & \triangleq \{ x \in X \mid S \in \mathbb{P}(\mathbb{P}X), \forall T \in S \bullet x \in T \}
\end{align*}
\]

Description:
The generalized form of union and intersection operates on sets of sets. Assuming \( S \) is a set of sets, then \( \cup S \) is its generalized union, containing all elements which are members of some sets contained in \( S \). The generalized intersection \( \cap S \) is defined in an analogous way: it contains precisely those members contained in all sets belonging to \( S \). Therefore the type of both operations is \( \mathbb{P}(\mathbb{P}X) \rightarrow \mathbb{P}X \).

The definition of union and intersection naturally follows from the above descriptions allowing \( X \) to contain one element sets only.
4.4.4 Set Difference

Like set union and set intersection, set difference is a binary set operation. For any two sets $S$ and $T$, it removes all elements from $S$ which are also elements of $T$.

Notation:

- $\setminus$: Set Difference

Definition:

$$S \setminus T \triangleq \{ x \in S \mid x \notin T, S \in \mathcal{P}X, T \in \mathcal{P}X \}$$

Description:

The members of $X \setminus Y$ are exactly those elements that are members of $X$, but not of $Y$. The type of the operation is therefore $\mathcal{P}X \times \mathcal{P}X \rightarrow \mathcal{P}X$.

4.5 Tuples

For the discussions following in the next few sections we need the notion of a collection of elements whose order is significant. For this purpose, the mathematical concept of an $n$-tuple is used. The number $n$ can be replaced by any positive integer and is constant. Therefore an $n$-tuple represents an ordered collection of $n$ elements, each of them possibly being of different type and value. However, the elements do not have to be unique. Special instances of $n$-tuples in mathematics are complex numbers (or more general ordered pairs) which have an $n$ of 2, or quaternions as a generalization of complex numbers with dimension 4.

A general $n$-tuple is: $(a_1, a_2, \ldots, a_n) = (b_1, b_2, \ldots, b_n)$ if and only if $a_1 = b_1, a_2 = b_2, \ldots, a_n = b_n$. Formally, an $n$-tuple can either be defined in terms of sets as

$$ (a_1, a_2, \ldots, a_n) = \{ a_1, \{ a_1, a_2 \}, \{ a_1, a_2, a_3 \}, \ldots, \{ a_1, a_2, \ldots, a_n \} \} $$

or by induction:

- a 1-tuple $(a_1)$ is defined as $a_1$
- if $t$ is an $n$-tuple, then $(t, a_{n+1})$ is an $(n + 1)$-tuple.

It is trivial to show that either of the above definitions implies the other, which proves their equivalence. For the following concepts, relations and functions, we restrict ourselves to the special notion of ordered pairs, mathematical objects that correspond to 2-tuples.

4.5.1 Ordered Pairs

An ordered pair $(x, y)$ is a collection of two elements such that $x$ can be distinguished as the first element and $y$ as the second element.
4.5.2 Tuple Projection

The projection operations are needed to access the elements of an ordered pair.

Notation:

- first, second : Pair Projection Operations

Definition:

\[ \text{first} \ (x, y) \triangleq x \]
\[ \text{second} \ (x, y) \triangleq y \]

Description:

The projection operation first is used to extract the first element of an ordered pair. Similarly, the operation second extracts the second element of an ordered pair. The type of this operations is therefore deduced as \( X \times Y \rightarrow X \).

4.6 Relations

Based on the previously introduced notion of tuples and ordered pairs, we are able to formally define the concept of a relation. Since ternary relations and even higher-dimensional relations are not of particular interest when we are attempting to specify data structures (see chapter 7), we only consider the special case of binary relations. Since relations are sets of ordered pairs, all operations applicable to sets, including union, intersection and difference are also defined for relations.

4.6.1 Definition by Extension

One way to define a relation is to list for all possible elements in the domain set the related element in the range set (see section 4.6.3.2). This is similar to the way of defining a set just by stating all its members. Therefore any relation \( R \) on a ground set \( X \) can easily be defined by enumeration all pairs that are element of the relation, e.g. \( R = \{(x_i, x_j), (x_m, x_n), \ldots\} \). The order in which the pairs are listed is not important, since a relation is a set of ordered pairs. Definition by extension is particularly convenient for finite relations involving only sets of small cardinality.

4.6.2 Definition by Comprehension

The other way to define a relation is not by extension, meaning by listing all the contained members, but by comprehension, e.g. through a characteristic property of its members. This is clearly applicable to sets too. Definition by comprehension is particularly interesting for functions (see section 4.7). The following example shows the relation that relates each element to plus or minus the other one:

\[ \text{double} \triangleq \{(x, y) \in \mathbb{N} \times \mathbb{Z} \mid (y = 2 \times x) \lor (y = -2 \times x)\} \] (4.4)

This relation contains the elements \((0, 0), (3, -6), (3, 6), \ldots\), and so on, and could be defined equivalently by extension:

\[ \text{double} \triangleq \{(0, 0), (3, -6), (3, 6), \ldots\} \] (4.5)

If the domain set or range set of the relation involves countable but infinite sets, definition by comprehension proves to be more convenient, since no infinite enumerations are needed.
4.6.3 Binary Relations

A binary relation relates elements of a domain set to elements of a range set.

Notation:

- \( \leftrightarrow \) : Binary Relation
- \( \mapsto \) : Maplet

Definition:

\[
X \leftrightarrow Y \triangleq \mathcal{P}(X \times Y)
\]

\[
x \mapsto y \triangleq (x, y)
\]

Description:

If \( X \) and \( Y \) are sets, then the set of binary relations between \( X \) and \( Y \) is exactly the powerset of the Cartesian product between \( X \) and \( Y \), written \( \mathcal{P}(X \times Y) \). Note that this definition corresponds to the one already given in section 4.1. The maplet is a simple alternative notation to express the graphical nature of ordered pairs \((x, y)\) and elements of binary relations.

4.6.3.1 Identity Relation

An important special case of a relation is the identity relation \( \text{id} \).

Notation:

- \( \text{id} \) : Identity Relation

Definition:

\[
\text{id} X \triangleq \{ x \in X \mid x \mapsto x \}
\]

Description:

The identity relation on a set \( X \) relates each member of \( X \) to itself and is therefore of type \( X \rightarrow Y \).

4.6.3.2 Domain/Range Projection

Two functions in particularly related to binary relations are the projection functions \( \text{domain} \) and \( \text{range} \).

Notation:

- \( \text{dom, ran} \) : Domain Projection/Range Projection

Definition:

\[
\text{dom } R \triangleq \{ x \in X \mid y \in Y, R \in X \leftrightarrow Y, (x \mapsto y) \in R \}
\]

\[
\text{ran } R \triangleq \{ x \in X \mid y \in Y, R \in X \leftrightarrow Y, (x \mapsto y) \in R \}
\]
Description:

Assuming $R$ being a binary relation from a domain set $X$ to a range set $Y$, the domain of $R$ consists of the set of all members of $X$ which are related to at least one member of $Y$ by $R$. Similarly, the range of $R$ is the set of all members of $Y$ to which $R$ relates at least one member of $X$.

Note that the above projection functions for binary relations are generalizations of the functions first and second for ordered pairs. Contrary to ordered pairs, relation projection functions are of type $(X \times Y) \rightarrow \mathbb{P}X$, indicating a set valued result, whereas the type of first and second obviously is $X \times Y \rightarrow X$.

4.6.4 Relation Composition

Most of the power and usefulness of relations comes from the fact that a relation can be expressed as a composition of more general relations. The relation composition serves this purpose.

Notation:

- $\circ$: Relation Composition

Definition:

$$R \circ S \triangleq \{ x \mapsto z \mid x \in X, y \in Y, z \in Z, S \in X \leftrightarrow Y, R \in Y \leftrightarrow Z, (x \mapsto y) \in R \land (y \mapsto z) \in S \}$$

Description:

The relation composition $R \circ S$ of two relations $R \in Y \leftrightarrow Z$ and $S \in X \leftrightarrow Y$ relates a member $x \in X$ to a member $z \in Z$ if and only if there is at least one element $y \in Y$ to which $x$ is related by $S$ and which is itself related to $z$ by $R$.

4.6.5 Relation Restriction

Similar to the projection functions dom and ran which project a relation in its vertical dimension, we define two other operations on relations: the domain restriction and the range restriction. Contrary to the domain and range projection functions, the relation restriction projects a relation in the horizontal dimension, meaning that the restriction functions yield a subset of the original relation.

Notation:

- $\leftarrow$: Domain Restriction
- $\rightarrow$: Range Restriction

Definition:

$$S \leftarrow R \triangleq \{ x \mapsto y \mid x \in X, y \in Y, S \in \mathbb{P}X, R \in X \leftrightarrow Y, x \in S \land (x \mapsto y) \in R \}$$

$$R \rightarrow T \triangleq \{ x \mapsto y \mid x \in X, y \in Y, R \in X \leftrightarrow Y, T \in \mathbb{P}Y, y \in T \land (x \mapsto y) \in R \}$$
Description:

The *domain restriction* $S ◁ R$ of a relation $R$ to a set $S$ relates an element $x$ from the domain set $\text{dom } R$ to elements of the range set $\text{ran } R$ if and only if $x$ is a member of $S$. The type of the domain restriction is $\mathbb{P}X \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Y)$. Similarly, the *range restriction* $R ⊿ T$ of $R$ relates elements from the domain set to an element $y$ of the range set if and only if $y$ is a member of $T$. Its type is $(X \leftrightarrow Y) \times \mathbb{P}X \rightarrow (X \leftrightarrow Y)$.

Notation:

- $\triangleleft$: Domain Anti-Restriction
- $\triangleright$: Range Anti-Restriction

Definition:

\[
S \triangleleft R \triangleq \{ x \mapsto y \mid x \in X, y \in Y, S \in \mathbb{P}X, R \in X \leftrightarrow Y, x \notin S \land (x \mapsto y) \in R \}
\]

\[
R \triangleright T \triangleq \{ x \mapsto y \mid x \in X, y \in Y, R \in X \leftrightarrow Y, T \in \mathbb{P}Y, y \notin T \land (x \mapsto y) \in R \}
\]

Description:

The anti-restriction operations are the complemented counterparts of the previous restriction operations. An element $x$ from the domain set $\text{dom } R$ is related to elements of the range set $\text{ran } R$ by $S \triangleleft R$ if and only if $x$ is not a member of $S$. The type of the domain anti-restriction is $\mathbb{P}X \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Y)$. The relation $R \triangleright T$ of type $(X \leftrightarrow Y) \times \mathbb{P}X \rightarrow (X \leftrightarrow Y)$ relates elements from the domain set $\text{dom } R$ to an element $y$ of the range set if and only if $y$ is not a member of $T$.

4.6.6 Relation Inversion

Considering a relation $R$ as a set of maplets, we can define the *inverse relation* as the set of *inversed maplets* of $R$. The inverse of a maplet $x \mapsto y$ is the maplet $y \mapsto x$ with its elements just reversed, in graphical terms the back edge of a directed edge in a graph.

Notation:

- $R^{-1}$: Inverse Relation

Definition:

\[
R^{-1} \triangleq \{ y \mapsto x \mid x \in X, y \in Y, R \in X \leftrightarrow Y, (x \mapsto y) \in R \}
\]

Description:

An element $y$ of the range set of $R$ is related to an element $x$ of the domain set by the inverse relation $R^{-1}$ of type $(X \leftrightarrow Y) \rightarrow (Y \leftrightarrow X)$ if and only if $R$ relates $x$ to $y$. 
4.6.7 Relation Image

The relation image is another projection operation applicable to relations. It is a combined operation of domain restriction and range projection.

**Notation:**
- $R[S]$ : Relation Image

**Definition:**

$$R[S] \triangleq \{ y \in Y \mid x \in X, R \in X \leftrightarrow Y, S \in \mathbb{P}X, (x \mapsto y) \in R \land x \in S \}$$

**Description:**

The relation image $R[S]$ of a set $S$ through a relation $R$ is the set of all elements of the range set $\text{ran} R$ to which $R$ relates some member of $S$. The type of this image operation is therefore $(X \leftrightarrow Y) \times \mathbb{P}X \rightarrow \mathbb{P}Y$.

4.6.8 Relation Closure

Another important category of operations applicable to relations are relation closures.

**Notation:**
- $R^{+}$: Transitive Closure
- $R^{*}$: Reflexive-Transitive Closure

**Definition:**

$$R^{+} \triangleq \{ Q \in X \leftrightarrow X \mid \cap \{ R \in X \leftrightarrow X, R \subseteq Q \land Q \circ Q \subseteq Q \} \}$$

$$R^{*} \triangleq \{ Q \in X \leftrightarrow X \mid \cap \{ R \in X \leftrightarrow X, \text{id} X \subseteq Q \land R \subseteq Q \land Q \circ Q \subseteq Q \} \}$$

**Description:**

The transitive closure $R^{+}$ of a relation $R$ with identical domain and range set is the strongest or smallest relation containing the relation $R$ which is transitive. Similarly, the reflexive-transitive closure $R^{*}$ is the strongest relation containing $R$ which is both reflexive and transitive. Both relation closures are of type $(X \leftrightarrow X) \rightarrow (X \leftrightarrow X)$.

These closures can also be defined in terms of relation iteration: Two elements $x$ and $y$ are related by the relation closure $R^{k}$ of $R$ with $k \geq 0$ if and only if there exist $k + 1$ elements $z_0, z_1, \ldots, z_k$ with $x = z_0$ and $z_k = y$, such that $0 \leq i < k$. The inverse relation closure $R^{-k}$ is defined as $(R^{1})^{k}$. 
4.7 Functions

Given a relation \( R \) and an element \( x \) of its domain set, there may be zero or more elements \( y \) such that \( R \) contains the pair \( x \mapsto y \). In fact, there may be infinitely many such elements in the range set of \( R \). A relation \( f \) that relates each \( x \) of its domain to at least one \( y \) from its range is called a function or sometimes also a mapping.

Notation:

- \( \not\rightarrow \): Partial Function
- \( \rightarrow \): Total Function

Definition:

\[
X \not\rightarrow Y \triangleq \{ f : X \not\leftrightarrow Y \mid x \in X, y_1 \in Y, y_2 \in Y, (x \mapsto y_1) \in f \land (x \mapsto y_2) \in f \Rightarrow y_1 = y_2 \}
\]

\[
X \rightarrow Y \triangleq \{ f : X \rightarrow Y \mid \text{dom} \ f = X \}
\]

Description:

Relations that relate at most one element of its range set to elements of its domain set are called functional relations or just functions. Functions are a particularly interesting kind of relations since they guarantee that for any element of its domain the associated range element is unique.

If a function \( f : X \rightarrow Y \) relates each element \( x \) of the domain \( X \) to an element in the range \( Y \), the function is called a total function. Otherwise it is called a partial function. Note that the definition of a function corresponds exactly to the one given in the introduction to this chapter.

We further distinguish between the following subtypes of functions: A partial or total function \( f \) that maps different elements of its domain to different elements of its range is called an injective function. Functions that have type \( f : X \rightarrow Y \) with \( \text{ran} \ f = Y \) are called surjective functions or onto-mappings. Finally, functions that are both injective and surjective are called bijective functions or one-to-one mappings.

Since functions are special instances of relations, all operations applicable to relations are also applicable to functions. The only exception to this are function intersection (section 4.7.2) and function union (section 4.7.3), which are presented in later sections. Special care must be taken when defining the inverse of a function (section 4.7.4).

4.7.1 Finite Functions

Finite functions are particularly suitable for modeling how computers deal with information. The natural finiteness of computer resources guarantees in many cases, such as memory or disk resources, that the domain of a function is finite. Some authors also prefer the term finite mapping.

Notation:

- \( \not\rightarrow \): Finite Function

Definition:

\[
X \not\rightarrow Y \triangleq \{ f : X \not\rightarrow Y \mid \text{dom} \ f \ \text{is finite} \}
\]

Description:

The set of finite functions contains all functions with a domain set \( X \) that is finite.
4.7.2 Function Intersection

Since functions are specialized relations and relations themselves specialized sets, the intersection operation is also applicable to functions. The intersection of two functions is the set of ordered pairs from the domain set to the range set that belong to both functions. Since the intersection operation should yield a function and not a relation, we have to adapt its definition accordingly.

**Notation:**
- \( \cap \): Function Intersection

**Definition:**
\[
f \cap g \triangleq \{ f : X \to Y, g : X \to Y, h : X \to Y \}
\]
\[
dom h = \{ x \in \dom f \cap \dom g \mid f(x) = g(x) \}
\]
\[
ran h = \{ x \in \dom h \mid h(x) = f(x) = g(x) \}
\]

**Description:**
The intersection of two functions \( f \) and \( g \) yields a function \( h \) that yields the value of \( f \) and \( g \) wherever \( f \) and \( g \) yield the same value. The type of the intersection operation is therefore \((X \to Y) \times (X \to Y) \to (X \to Y)\).

4.7.3 Function Union

Although the union operation can be applied to functions as well, the result of this operation does not have to be necessarily a function as well. This is the case for elements of the function domains where both functions are defined, but yield another value. This results in a relation in general.

One possible way out of this problem would be to restrict this operation to pairs of functions that yield the same value for all common elements in their domain set. Obviously, this operation does not make much sense.

The other option is to define the union operation for any function pairs and making it non-commutative to always get a function as result. We choose the second option in the following discussions:

**Notation:**
- \( \sqcup \): Function Union

**Definition:**
\[
f \sqcup g \triangleq \{ f : X \to Y, g : X \to Y, h : X \to Y \}
\]
\[
dom h = \{ x \in \dom f \cup \dom g \mid f(x) = g(x) \}
\]
\[
ran h = \{ x \in \dom h \mid h(x) = f(x) \leftrightarrow x \in \dom f \land x \notin \dom g, h(x) = g(x) \leftrightarrow x \in \dom g \}
\]

**Description:**
The union operation on functions is defined as a function union which is not commutative anymore. Wherever the two functions \( f \) and \( g \) yield different values, the value of \( g \) takes precedence, resulting in a function and not a relation. An alternative definition for function union is given by \(((\dom g) \subseteq f) \cup \allowbreak g\). The type of this operation is therefore \((X \to Y) \times (X \to Y) \to (X \to Y)\).
4.7.4 Function Inversion

Inverting a function does not necessarily result in a function, but generally yields a relation (see section 4.6.6). The only kind of functions that guarantee its inverse to be again a function is the special category of injective functions. Since injective functions map different elements from its domain to different elements in its range, the inverse function is a function as well. In all other cases, the inverse of a function is usually a relation.

4.8 Generalized Sets

In the previous definitions we always used unordered sets which contain each element exactly once, in no particular order. To extend our collection of mathematical concepts, we introduce further generalizations of sets, allowing on the one hand elements to occur multiple times, and on the other hand to define a particular order on its elements.

4.8.1 Sequences/Lists

Sequences are sets that contain elements in a particular order.

**Notation:**

- $\text{seq}$: Finite Sequences
- $\text{seq}\_1$: Non-empty Finite Sequences
- $\text{rank}$: Finite Rankings/Finite Injective Sequences

**Definition:**

$$\text{seq } X \triangleq \{ f : \mathbb{N} \rightarrow X \mid \text{dom } f = 1 \ldots \| f \| \}$$

$$\text{seq}_1 X = \{ f \in \text{seq } X \mid \| f \| > 0 \}$$

$$\text{rank } X = \text{seq } X \cap ( f : \mathbb{N} \rightarrow X \mid \forall x \in X, \forall y \in X \bullet x \neq y \Rightarrow f(x) \neq f(y))$$

**Description:**

The notation seq $X$ designates the set of finite sequences over $X$. Sequences are finite partial functions from $\mathbb{N}$ to the set $X$ whose domain are exactly the indexes $1 \ldots n$ for any natural number $n$. We use the notation $(x_1, x_2, \ldots, x_n)$ to define the sequence that contains the elements $x_1, x_2, \ldots, x_n$ by extension. The set seq$_1 X$ is exactly seq $X$ without the empty sequence, which we denote by $\langle \rangle$ in the following discussions. The empty sequence symbol $\langle \rangle$ is an alternative notation for the function $\emptyset : \mathbb{N} \rightarrow X$, the empty function.

Further, rank $X$ is the set of injective sequences over $X$. Injective sequences are sequences which whose elements are unique, what is already suggested by the nature of the sequence function $f$. Injective sequences are sometimes also called rankings. There are a number of important operations defined for sequences which we are presenting in the following sections.
4.8.1.1 Sequence Concatenation

The concatenation operation on sequences combines two sequences over the same ground set \( X \). We use the \( \triangledown \) operator to denote sequence concatenation. Like in the discussion of set union and set intersection (section 4.4.3), we also consider the generalized form of sequence concatenation. The \( \triangledown \) operator is reused to denote both the simple (as binary operator) and the generalized form (as unary prefix operator) of concatenation. The exact semantics of this operator are evident from the surrounding context.

Notation:
- \( \triangledown \): (Generalized) Sequence Concatenation

Definition:

\[
\triangledown () \triangleq ()
\]

\[
\triangledown (S) \triangleq \{ S \mid S \in \text{seq } X \}
\]

\[
\triangledown (S \triangledown T) \triangleq \{ (S \triangledown T) \mid S \in \text{seq } (\text{seq } X), T \in \text{seq } (\text{seq } X) \}
\]

\[
U \triangledown V \triangleq \{ U \cup \{ n \in \text{dom } V \mid V \mid \rightarrow V (n) \} \mid U \in \text{seq } X, V \in \text{seq } X \}
\]

Description:

If \( U \) is a sequence of sequences of elements in \( X \), then \( \triangledown U \) yields a single sequence containing all the elements of the sequences in \( U \), in the order of the concatenated sequences. The type of the generalized concatenation is \( \text{seq } (\text{seq } X) \rightarrow \text{seq } X \). The simple concatenation of two sequences over the elements of a ground set \( X \) consists of all elements of the first sequence followed by the elements of the second sequence. The type of this operation is therefore \( \text{seq } X \times \text{seq } X \rightarrow \text{seq } X \).

4.8.1.2 Sequence Decomposition

Apart from the sequence concatenation operation there are several decomposition operations applicable to sequences of elements in \( X \).

Notation:
- \( \text{first} \): First Element
- \( \text{last} \): Last Element
- \( \text{tail} \): Sequence Tail
- \( \text{front} \): Sequence Front
4.8. GENERALIZED SETS

Definition:

first \( S \) \( \triangleq \{ S \in \text{seq } X \mid S(1) \} \)

last \( S \) \( \triangleq \{ S \in \text{seq } X \mid S(\| S \|) \} \)

tail \( S \) \( \triangleq \{ S \in \text{seq } X \mid \lambda n : 1 \ldots \| S \| - 1 \cdot S(n + 1) \} \)

front \( S \) \( \triangleq \{ S \in \text{seq } X \mid (1 \ldots \| S \| - 1) \triangleleft S \} \)

Description:

The operations first and last return the first and last element of any sequence \( S \), respectively. The resulting values are exactly the results of evaluating the sequence function \( S \) with the according argument 1 and \( \| S \| \). The type of these operations is seq\(_1\) \( X \to X \), meaning first and last are only defined on non-empty sequences.

The sequences tail \( S \) and front \( S \) contain all the elements of the sequence \( S \) except for the first and last one, respectively. tail and front are only defined on sequences of more than one element as well. Both operations are of type seq\(_1\) \( X \to \text{seq } X \).

4.8.1.3 Sequence Reversal

The most important permutation operation for sequences is the sequence reversal operation.

Notation:

- \( S^\sim \) : Reverse Sequence

Definition:

\( S^\sim \) \( \triangleq \{ S \in \text{seq } X \mid \lambda n : \text{dom } S \cdot S(\| S \| - n + 1) \} \)

Description:

The sequence reversal operator \( \sim \) applied to a sequence \( S \) returns a sequence \( S^\sim \) containing the same elements, but in reverse order. Obviously, the type of the sequence reversal operation is seq\(_1\) \( X \to \text{seq } X \).

4.8.1.4 Sequence Filtering

In the special case of unordered sets, a filter operation could simply be defined using the set difference operation. In the context of sequences, special care must be taken to preserve the order of the elements.

Notation:

- \( \uparrow \) : Sequence Filtering
Definition:
\[
\langle \rangle \upharpoonright U \triangleq \{ \langle \rangle \mid U \in \mathbb{P}X \}
\]
\[
\langle x \rangle \upharpoonright U \triangleq \{ x \in X, U \in \mathbb{P}X \mid \langle x \rangle \leftrightarrow x \in U, \langle \rangle \leftrightarrow x \notin U \}
\]
\[
(S \bowtie T) \upharpoonright U \triangleq \{ ((S \upharpoonright U) \bowtie (T \upharpoonright U)) \mid S \in \text{seq } X, T \in \text{seq } X, U \in \mathbb{P}X \}
\]

Description:
The formal definition given above is an inductive definition based on sequence concatenation. If \( S \) is a sequence of elements in \( X \) and \( U \) is a subset of \( X \), then \( S \upharpoonright U \) is a sequence which contains exactly those elements of \( S \) which are not members of \( U \), in the same order as in \( S \). The type of the filtering operation is \( \text{seq } X \times \mathbb{P}X \to \text{seq } X \).

4.8.1.5 Sequence Disjointness/Sequence Partitioning

The previously encountered concepts of set intersection and set containment can naturally be generalized and extended to sequences. To this end, we use the concept of indexed families of sets. An indexed family of sets is a set of type \( \mathbb{P}(I \rightarrow \mathbb{P}X) \), where \( I \in \mathbb{N}^* \) is the set of positive indexes used to identify the members of the set family. Each member of the set family has therefore a uniquely assigned index.

Notation:
- \( \bowtie \): Disjointness
- \( \uplus \): Partitioning

Definition:
\[
\bowtie S \triangleq \{ S \in I \rightarrow \mathbb{P}X, T \in \mathbb{P}X \mid \forall i, j \in \text{dom } S, i \neq j \rightarrow S(i) \cap S(j) = \emptyset \}
\]
\[
S \uplus T \triangleq \{ S \in I \rightarrow \mathbb{P}X, T \in \mathbb{P}X \mid \bowtie S \land \bigcup \{ i \in \text{dom } S \cdot S(i) \} = T \}
\]

Description:
We say that a family of sets \( S \) (in our case an indexed family of sets) is disjoint if and only if each pair of non-equal sets in the family has empty intersection. In addition to the disjointness property, the disjoint family \( S \) partitions a set \( T \) if the union of all \( S(i) \) is exactly \( T \).

4.8.1.6 Sequence Closure

A third type of closures which is only defined on functions is the sequence closure. The sequence closure \( f^{**} \) of a function \( f : X \rightarrow X \) is defined such that for any \( x \in X \), \( f^{**} \) yields the sequence \( x, f(x), f(f(x)), \ldots \), up to the first element that is not in \( \text{dom } f \) anymore. We use \( f^{**} \) to denote the sequence closure including \( x \), and \( f^{++} \) to express that the sequence starts with \( f(x) \). Note that \( f^{++} \) is equivalent to applying \( f \) to the sequence closure \( f^{**} \). An important remark on the finiteness of the sequence closure construct: It is obvious that only acyclic functions \( f \) yield a finite sequence closure.

In the definition for the sequence closure we use the iterate functional presented in section 4.9.3. Let us denote \( f^{**}_n \) and \( f^{++}_n \) as the sequence closures of \( f \) with length \( n \). Then the recursive definition for the sequence closure is the following:
4.8. GENERALIZED SETS

Notation:

- $f^{**}, f^{++}$: Sequence Closure

Definition:

\[
f^{**}_{n+1} x \triangleq f^{**}_n x \bowtie \text{iterate } f x n
\]
\[
f^{**}_0 x \triangleq \text{iterate } f x 0
\]
\[
f^{++}_{n+1} x \triangleq f^{++}_n x \bowtie \text{iterate } f x n
\]
\[
f^{++}_0 x \triangleq f(x)
\]

Description:

The sequence closure is a special case of the more general transitive closures defined on relations. But in order to get a sequence as result, the underlying structure is required to have the functional property. Therefore sequence closures are only defined on functions with identical domain and range. To ensure the finiteness of the sequence, the function $f$ has to be acyclic.

4.8.2 Multisets/Bags

A direct generalization of sets are multisets, sometimes also called bags. Multisets do not have the restriction of containing unique elements only, but allow each element to occur more than once. Like sets, multisets have no defined order on their elements.

Notation:

- $\text{bag}$: Finite Multisets
- $\text{bag}_1$: Non-Empty Finite Multisets

Definition:

\[
\text{bag } X \triangleq f : X \rightarrow \mathbb{N}_1
\]
\[
\text{bag}_1 X \triangleq \{ f \in \text{bag } X \mid \| f \| > 0\}
\]

Description:

A multiset $\text{bag } X$ over a set $X$ is an unordered set whose elements can occur multiple times. Similar to section 4.6, there are two basic ways of defining a bag: Definition by extension and definition by comprehension. In the following presentation we take the somehow simpler form of definition by extension.

We use the notation $\llbracket [x_1, x_2, \ldots, x_n] \rrbracket$ to denote a bag that contains the elements $x_1, x_2, \ldots, x_n$. A more complete definition which also takes the multiplicity of the elements into account uses the previously introduced maplet notation $\{ x_1 \mapsto k_1, x_2 \mapsto k_2, \ldots, x_n, \mapsto k_n \}$, where for each $i$ the element $x_i$ occurs exactly $k_i$ times. The empty bag is denoted by $\llbracket \rrbracket$, similar to the empty sequence.
4.8.2.1 Multiset Membership

The membership property for multisets can be defined similar to the definition for set membership.

**Notation:**

- $\in, \notin$: Multiset Membership, Multiset Non-Membership

The property $\notin$ is the complement of the multiset membership property.

**Definition:**

\[
x \in Y \triangleq \{ x \in \text{dom} Y \mid x \in X, Y \in \text{bag} X \}
\]

\[
x \notin Y \triangleq \neg(x \in Y)
\]

**Description:**

An element $x$ is a member of a multiset $Y$ if and only if the function $Y$ evaluates to a non-negative $x$.

4.8.2.2 Multiset Multiplicity

Since elements can occur multiple times in a multiset, the properties of a set have to be extended with a notion of multiplicity.

**Notation:**

- $\text{mult}$: Multiset Multiplicity

**Definition:**

\[
\text{mult} Y x \triangleq \{ x \in X, Y \in \text{bag} X \mid \text{ran} ((\lambda x : X \cdot 0) \cup (\{ x \} \triangleleft Y)) \}
\]

**Description:**

The multiplicity of an element $x$ in a multiset $Y$ denoted by $\text{mult} Y x$ is simply defined as the range of the function $Y$. To prevent the case where an element $x$ does not occur at all in the multiset, we use the function union.

4.9 Functionals

So far, functions only take arguments of simple set and tuple types. In order to provide more flexibility and to increase the power and expressiveness of function application, we use the concept of higher-order functions, sometimes also called functionals. Higher-order functions are generalizations of conventional functions allowing not only set and tuple types as argument types and result types, but higher-order functions itself. By allowing functions as arguments, we can construct functions that operate on other functions or even return higher-order functions as the result of a computation over functional arguments.
4.9. FUNCTIONALS

4.9.1 Function Composition

The most basic application of higher-order functions is the composition operation of functions. The concept of function composition is similar to relation composition (see section 4.6.4). However, for functions we would like not to give up the functional property of the composition, meaning that function composition should always yield a function, and not a relation.

Notation:

- \( \circ : \) Function Composition

Definition:

\[ g \circ f \triangleq \{ f : X \rightarrow Y, g : Y \rightarrow Z, h : X \rightarrow Z \} \]

\[ \text{dom } h = \{ x \in \text{dom } f, f(x) \in \text{dom } g \mid x \} \]

\[ \text{ran } h = \{ x \in \text{dom } f \mid g(f(x)) \} \]

Description:

The function composition \( g \circ f \) is a function \( h \) which has the same domain as \( f \). The range of \( h \) consists of all elements \( z \in Z \) which are the result of applying \( f \) to \( x \) first, and then applying \( g \) to that result. The composition operator obviously is a higher-order function taking two functions as arguments and resulting in a function yielding the same values as the nested application of \( f \) and \( g \) to an argument \( x \). The type of the functional composition is therefore \( (Y \rightarrow Z) \times (X \rightarrow Y) \rightarrow (X \rightarrow Z) \).

4.9.2 Function Currying

The other important functional which we will use in the following discussions is the higher-order function \( \text{curry} \). This functional is termed after the mathematician H.B. Curry. It transforms any two-argument function into a one-argument function. For a number of considerations it proves to be convenient to allow only one-argument functions.

Given a function \( f \) with a two arguments, function currying considers \( f \) as a one-argument function whose result is itself a function taking one argument. Naturally, this concept can be extended to functions with more than two arguments. For simplicity, we define \( \text{curry} \) for total functions only.

Notation:

- \( \text{curry} : \) Function Currying

Definition:

\[ \text{curry } f \triangleq \{ x \in X, y \in Y, f : X \times Y \rightarrow Z, g : X \rightarrow (Y \rightarrow Z) \mid g(x)(y) = f(x, y) \} \]

Description:

What seems not so clear at first sight is that \( \text{curry } f \) in fact represents the same function as \( g \). Assuming \( z \) to be the result of applying \( f \) to the arguments \( x \) and \( y \), applying \( \text{curry } f \) to the first argument \( x \) yields a total function from \( Y \) to \( Z \), which exactly yields \( z \) for any second argument \( y \).

The purpose of the \( \text{curry} \) higher-order function is to achieve partial evaluation of a function. It is itself a total function of type \( ((X \times Y) \rightarrow Z) \rightarrow (X \rightarrow (Y \rightarrow Z)) \).
4.9.3 Function Iteration

A functional that is particularly useful for the definition of the sequence closure in section 4.8.1.6 is the iterate functional. This functional is closely related to function composition and in fact is a generalization of this concept. Iterate applies a function a certain number of times to an argument. In other words, iterate can be expressed as $f \circ f \circ \ldots \circ f \circ f$, with $f$ applied $n$ times. Therefore, the type of the function argument has to be $f : X \rightarrow X$. It follows that the type of the iterate functional is $(X \rightarrow X) \times \mathbb{N} \rightarrow (X \rightarrow X)$. As for the curry functional, we define iterate for total functions only. For convenience reasons, we use a recursive curried form of definition:

Notation:

- iterate: Function Iteration

Definition:

\[
\text{iterate } f \ x \ (n + 1) \triangleq f \ \text{iterate } f \ x \ n \\
\text{iterate } f \ x \ 0 \triangleq x
\]

Description:

It is obvious from the style of definition that function iteration is a recursive concept. Iterate applied to an argument of 0 yields exactly the identity function. The $n$-th iterate of $f$ is then defined as the function application of $f$ to the $(n - 1)$-th iterate of $f$.

4.9.4 Generic Functionals

Apart from the powerful concepts of function composition and function currying, we introduce some generic higher-order functions that seem very useful as basic building blocks of functionality for sets and sequences.

4.9.4.1 map

Map, as the name already suggests, is a generic function which applies a functional to all elements of a set or sequence. The function map incorporates the concept of a transformation from a domain set to a range set. Its type is therefore deduced to be $((X \rightarrow Y) \times P.X) \rightarrow P.Y$, or $X \rightarrow Y \rightarrow P.X \rightarrow P.Y$ in the more natural curried form. This function has numerous practical examples thanks to its generic functional argument.

Example:

\[
\text{cardinality} : X \rightarrow \mathbb{N}_0^+ \triangleq \|X\| \\
\text{map (cardinality, [[3, 4, 5], [6, -1]])} = [3, 2]
\]
4.9. FUNCTIONALS

4.9.4.2 filter

Another common application of higher-order functions is filtering of set or sequence elements. For this purpose we define the following function filter, which uses a unary predicate as filter criterion. Applying the predicate to a domain set $X$ results in a range set of the same type. Therefore the type of filter is $\mathbb{P}X \times (X \rightarrow \mathbb{B}) \rightarrow \mathbb{P}X$.

Examples:

$even : \mathbb{Z} \rightarrow \mathbb{B} \triangleq \{z \in \mathbb{Z} \mid \exists k \in \mathbb{Z} \cdot z = 2 \times k\}$

$\text{filter } (\{-1, 3, 4, 6, 9, 13\}, even) = \{4, 6\}$

4.9.4.3 accumulate

The higher-order function accumulate is also known under the name reduce. It is used to calculate a result incrementally over a set (sequence). The main idea of accumulate, hence the name, is to store intermediate values from the computation over a subset (subsequence) in an accumulator, and compute successive values from the current element and the previously stored result. Therefore this function needs three arguments: The accumulation function $a$, a domain set $X$ over which to compute the result and an initial value for the accumulator, called $c$. From the above description it is evident that the type of accumulate is $((X \times Y) \rightarrow Y) \rightarrow Y \rightarrow \mathbb{P}X \rightarrow Y$, expressed in the more natural curried form. The accumulate function can be applied in several contexts.

Examples:

$sum : \mathbb{P}\mathbb{Z} \rightarrow \mathbb{Z} \triangleq \text{accumulate (operator +)}0$

$sum (\{1, 2, 3, 4\}) = 10$

4.9.4.4 forall

A function which can be defined in an elegant way using the previously encountered accumulate function is the Boolean function forall. This function tests whether a certain property $p$ holds for all elements in a set or sequence. It is the functional counter part of the universal quantifier. The extensive usage of this function in the EIFFELBASE library justifies a separate definition. Again, the property is specified using a unary predicate, which yields true if and only if the required property is satisfied. The deduced type therefore is $(X \rightarrow \mathbb{B}) \rightarrow \mathbb{P}X \rightarrow \mathbb{B}$. An elegant way to define forall uses the Boolean operator $\wedge$ together with the accumulate function and the initial value true for its accumulator: forall $\triangleq \text{accumulate } p \circ \text{operator } \wedge \text{true}$.

Examples:

$positive : \mathbb{Z} \rightarrow \mathbb{B} \triangleq \{z \in \mathbb{Z} \mid x > 0\}$

$forall (positive, \{-1, 0, 5, 7\}) = false$

$forall (positive, \{1, 2, 3, 4\}) = true$
4.9.4.5 \textit{exists}

Since we have defined the \textit{universal quantifier} as a functional \textit{forall}, it is natural to also specify the \textit{existential quantifier} as a corresponding functional \textit{exists}. Similar to the \textit{forall} functional, \textit{exists} tests whether a certain property $p$ holds for at least one element in a \textit{set} or \textit{sequence}. As above, the property is specified using a \textit{unary predicate}, which yields \textit{true} if and only if the required property is satisfied. The deduced type therefore is $\textit{exists} : (X \to \mathbb{B}) \to \mathcal{P}X \to \mathbb{B}$. Note that these two functionals are equivalent, meaning that each one can be expressed by the other, by negating the \textit{functional} and the \textit{predicate}.

\textbf{Examples:}

$positive : \mathbb{Z} \to \mathbb{B} \triangleq \{ z \in \mathbb{Z} \mid x > 0 \}$

$exists (positive, \{-1, 0, 5, 7\}) = true$

$exists (positive, \{-1, -2, -3, -4\}) = false$
Chapter 5

The Intermediate Functional Language (IFL)

This chapter introduces the IFL contract language used to specify EIFFEL contracts with mathematical models.

5.1 Overview

In section 5.2, we present our approach on how to design the IFL language. We explain the reasons for the additional restrictions that are introduced to adapt the language of EIFFEL Boolean expressions to meet the strict requirements of mathematical modeling.

The important concept of model features which has been introduced in chapter 3 is revisited in section 5.2.1. A summary describes which necessary conditions have to be met by EIFFEL classes with annotated mathematical models, and in particular we discuss model features in the context of the previously defined concepts of chapter 3, namely abstraction functions, abstract contracts and model contracts. Additionally, we also provide definitions of clean and pure features, which help in the following elaboration of a rigorous language definition for the IFL contract language.

In section 5.3, we proceed with the definition of valid IFL language expressions. We discuss the concrete restrictions which have been identified in section 5.2, and finally give a list of semi-formal descriptions of IFL validity errors, designed as reference for the implementation of IFL language verifiers.

The last section of this chapter, section 5.5, is dedicated to the description of the IFL language grammar, which is derived from the upcoming EIFFEL standard [Mey].

5.2 Devising A Language Specification

Devising a specification for a powerful functional language requires a careful examination of the existing language constructs, in particular where the existing EIFFEL language grammar can be changed and adapted to favor a more restricted IFL language grammar, modeling the restrictions of the IFL language as accurate as possible, while leaving as few as possible semantic checks to the semantic analysis phase of the translation process. We are first presenting the necessary restrictions that have to be met by model features in order to comply with the IFL language and then discuss the important concept of pure expressions.
CHAPTER 5. THE INTERMEDIATE FUNCTIONAL LANGUAGE (IFL)

5.2.1 Specification of Model Features

Since any implementation of model features has to provide a result computed by the abstraction function of the corresponding model, it has to conform to the properties of mathematical concepts and structures. We list the necessary language rules for model features, and for every rule, we give a justification why it is necessary in our view.

The language rules for model features $f$ introduced in a model class $C$ are the following:

- **Pure Query Rule**: Model features must be implemented as pure queries only. Since model features represent the abstraction function of the corresponding model in $C$, they are not allowed to change the state of the current object at all. In other words, a model feature $f$ can be called as many times during the evaluation of a model contract, each time yielding the exact same non-void result. This property is essential for mathematical reasoning about model contracts. Note that deferred model features are perfectly valid and even sometimes needed (consider for example a deferred class under specification, whose internal implementation details have not yet been fixed, thus leaving the construction of the mathematical models open for descendants).

Additionally, we forbid the following language constructs of the EIFFEL language:

1. **External Query Rule**: External features are normally used to interface EIFFEL code with existing code written in another programming language. Most conventional programming languages provide no notion of pure features in their type systems. Therefore feature purity could not be guaranteed anymore when allowing external features.

2. **Once Query Rule**: A language construct which is unique in the EIFFEL programming language is the notion of once features, a special kind of features that are only executed once during the life-time of a software system. If the once feature is a command, then the instructions of this feature are not executed anymore during subsequent calls to this feature. In the case of a query, subsequent function calls use the result computed during the first function call.

- **Model Argument Rule**: Model features cannot depend on arguments other than the current state of computation. Every abstraction function of a mathematical model is of the form $f : X \rightarrow Y$, where $X$ is the base type of the EIFFEL class $C$ being annotated with models (or in the case where $C$ is a generic class, the family of types represented by class $C$). Therefore forbidding any arguments for model features does not decrease the power of the IFL language, since the domain of the abstraction function is the base type of the class itself, automatically providing access to all necessary information on the current state of computation.

- **Model Type Rule**: Model features may only return results of basic numeric types like CHARACTER, INTEGER, REAL, BOOLEAN and reference types of classes of a model class library like the Mathematical Model Library (MML) presented in chapter 6. If using the MML class library, the reference types have to conform to MML_ANY.

- **Model Implementation Rule**: Model features are only allowed to return a result which is not 'Void'. This restriction should be stated both in the postcondition of the corresponding model feature and in the class invariant of the model class which introduces the model feature. The implementation of a model feature is required to comply with the abstraction invariant.
5.2. DEVISING A LANGUAGE SPECIFICATION

- **Model Feature Declaration Rule**: Model features have to be declared in a feature clause which exports its features both to the root class of the EIFFEL type system, class ANY, and to a special marker class which is designed to allow IFL language processing tools to easily extract model features from the flat form of an EIFFEL class. In the MML class library, this marker class is called MML_SPECIFICATION. The reason why model features have to be exported to ANY lies in the export validity rule VAPE of the EIFFEL language [Mey92], which states that every feature occurring in feature calls of a precondition of a feature exported to class C in the model class M must be at least exported to class C.

- **Model Class Inheritance Rule**: Model classes which introduce model features must conform to the marker class mentioned in the Pure Query Rule. Again, if model contracts are expressed using the MML class library facilities, then the model class has to conform to class MML_SPECIFICATION. The reason for this rule is the need to access models of feature arguments. The following example presents such a case:

**Example: Set Intersection**

Consider the EIFFEL class SUBSET [G] representing the mathematical notion of a set. This class introduces a feature intersect (other: SUBSET [G]) which removes all elements in the current set which are not in the other set. The definition of the feature can be easily stated in its postcondition using mathematical models:

\[
\text{set.is\_equal (old set.intersected (other.set))} \tag{5.1}
\]

Assertion 5.1 needs access to the set model of the feature argument. In order to grant this access, the set model must be exported to ANY. Since models do not have any precondition that can fail nor do they change the state of the current object (see section 5.2.2), exposing the mathematical models to the public does not raise any problems.

5.2.2 Pure Features/Pure Expressions

It has already been mentioned that model features do not change the state of the current object. We would like now to formalize this property and to give a precise definition of feature purity. The first important definition for our further discussions is the definition of pure feature:

- **Definition**: Pure Feature

An effected feature \( f \) of a class \( C \) is pure if and only if the following two conditions are met:

- \( f \) contains only assignments to local variables
- \( f \) contains only feature calls to features which are also pure

This simple definition guarantees that objects instantiated from a class which offers pure features only are immutable objects, what the following definition states:

- **Definition**: Immutable Object

An object \( o \) instantiated by a class \( C \) is immutable if and only if class \( C \) provides pure features only.
The two preceding definitions naturally lead to the important definition of pure expressions:

- **Definition**: Pure Expression
  
  An expression $e$ is called pure if and only if $e$ involves immutable objects only.

Note that from a mathematical point of view expressions are stateless and pure anyway, however in the IFL language, which is designed as a subset of the EIFFEL programming language, we have to make sure that model assertions do not change the state of the current object.

However, there is a small problem with the definitions mentioned above. The EIFFEL language, among many others, offers the concept of deferred features, a mechanism to postpone a feature implementation to descendant classes. Unfortunately, the definition for pure features is dependent on feature implementations, which makes the concept of pure features inapplicable for deferred features.

We therefore introduce a more general concept to capture also deferred features, which is called clean features:

- **Definition**: Clean Feature
  
  A feature $f$ appearing in a class $C$ of a system $S$ is called clean if and only if one of the following conditions is met:

  - $f$ is a pure feature in class $C$
  - $f$ is an immediate deferred feature in class $C$ and all redefinitions in $S$ are clean as well
  - $f$ has been undefined in class $C$ or any ancestor class in $S$.

An obvious observation following from the above definitions is that an expression which contains only feature calls to clean features is a pure expression. This property is crucial with respect to polymorphism and will be used in the following discussion of IFL expressions.

### 5.3 IFL Expressions

This section is dedicated to the presentation of IFL expressions, the kind of expressions that is used in the model assertions of model contracts. Since the IFL contract language is based on a subset of the EIFFEL language, our starting point for the following discussions are expressions of Boolean type as defined by the most recent EIFFEL standard [Mey]. Therefore, unless otherwise stated, IFL expressions are described exactly by the same language grammar as EIFFEL expressions.

However, there are some necessary restrictions that have to be introduced in order to let IFL expression reflect the mathematical world more closely. We identify three categories of additional rules in IFL expressions: Type-related issues that have to be handled by type rules, changes to the EIFFEL language making IFL a language directly mappable to pure mathematics, specification-oriented rules for features annotated with mathematical models, and finally rules governing the use of binary operators in IFL expressions. We are going to describe each category in detail.
5.3. IFL EXPRESSIONS

5.3.1 Types in IFL Expressions

We have already seen the Model Type Rule in section 5.2.1 which constrains the possible types of model features. The restrictions are similar for IFL expressions. All features which are involved in a feature call of an IFL expression have to be of basic types CHARACTER, INTEGER, REAL, and BOOLEAN, or of reference types from mathematical model class libraries. Obviously, the type of the expression itself is BOOLEAN, since IFL expressions are used in model assertions.

All the type-related restrictions serve only one purpose: To model mathematics as accurately as possible. By restricting ourselves to basic numeric types from the EIFFELBASE class library, we allow mathematical reasoning to be performed directly on the models. Depending on the reasoning language, minor translation effort has to be done, which is purely syntactical. Classes from mathematical model libraries are specifically designed to only instantiate mathematical objects that are immutable and stateless. In conjunction with the Pure Query Rule, these two type restrictions therefore guarantee immutability of objects and therefore pure expressions as desired.

5.3.2 Specification with IFL Expressions

This category of restrictions on IFL expressions gives rules about their usage in normal EIFFEL features. These rules ensure specification completeness of feature contracts in a model class. We identify two basic criteria for a complete specification:

1. Each feature assertion must contain assertion clauses involving all introduced or inherited model features

2. Each function must specify the computation of the result with a special definition assertion (see assertion 5.2).

The reasons for the first rule are very simple. If we assume that all the mathematical aspects of an EIFFEL class and its associated ADT are captured by its set of mathematical models, it is natural now to claim that a specification is complete if and only if a feature assertion involves all model features of that model class. This rule can easily checked by any language processing tool like the IFL Contract Flattener (see chapter 9). It has to be said that this rule is only necessary, but in no way sufficient. Enforcing each model feature to occur at least once in a feature assertion does only help not to forget the specification with respect to inherited models.

The second rule is basically an accentuation of the first rule. We additionally require every post-condition of EIFFEL functions to contain an exact specification of the result they compute. This kind of assertion is called a definition assertion (see also section 7.2) and has the following form:

\[ \text{Result.is_equal} (...) \]  

(5.2)

The right-hand side of assertion 5.2 is exactly the expression of the feature implementation that is assigned to the predefined entity 'Result'. At least one definition assertion is required. It is also possible to state more than one definition assertion in more complicated function specifications.
5.3.3 Mathematical Notions of IFL

The EIFFEL programming language has been designed as a pure, object-oriented general-purpose programming language. However, we have to carefully examine whether the language constructs involved in Boolean expressions are indeed compatible with mathematics. The following language constructs of the EIFFEL language have been proved incompatible with a mathematical specification language:

- **External Features**: Features with an implementation written in another language
- **Once Features**: Features that are executed only once during the life-time of a system
- **Impure Features**: Features that change the current object state

These kinds of features are prohibited by the **Pure Query Rule** and its sub-rules.

External features are not allowed since feature purity cannot be guaranteed when interfacing procedures and functions written in a programming language that does not support the declaration of pure methods.

The reason why once features are not allowed in the IFL language is that they are not state independent. All executions of a once feature have the same effect as the first one. In case of a once procedure, the action is only executed the first time and in case of a once function, all feature calls return the result computed during the first execution of the feature. This property is certainly not desired in a mathematical environment and has therefore been prohibited in the IFL language.

The third kind of incompatible EIFFEL features are impure features, that is to say features that change the state of the current object. To model the functional nature of mathematical specifications and to completely prevent abstract or concrete side-effects of IFL expressions, only feature calls to pure features are allowed in IFL expressions. Note that this applies to creation procedures as well; except that they are allowed to initialize class attributes of the target object.

To summarize, IFL expressions occurring in a class C of a system S are allowed only to contain feature calls to clean features which are clean with respect to S.

5.3.4 Binary Operators in IFL

Another incompatibility of the EIFFEL language (and of any non-functional programming language) with mathematical specifications is the concept of reference equality. In mathematics, if two objects are considered equal, then the two object have indeed the same identity. This is contrary to object-oriented programming, where objects with different identities can be considered equal. This mismatch has an impact on the binary operators that are involved in reference equality comparisons. Unary operators of the EIFFEL language do not have to be checked for mathematical compatibility, since there are no unary operators for reference equality comparisons in the language specification.

We declare the following constructs of the EIFFEL language to be incompatible with the IFL language:

- **Reference Equality Operator** ‘\texttt{=}’
- **Reference Inequality Operator** ‘\texttt{=/=}’
However, for better usability and easier use we would like to make an exception and allow reference (in)equality operators in expressions with a right-hand operand equal to \texttt{Void}'. The reasons for this exception are twofold:

- \texttt{Void}' is a special singleton object of the EIFFEL type system. In some implementations like the EIFFELBASE library, \texttt{Void}' is a frozen attribute of type NONE, whereas others treat \texttt{Void}' as a language keyword or predefined entity. Reference equality and object equality comparisons therefore always yield the exact same result. Consequently, it is justified to introduce this convenient syntactical shortcut.

- In EIFFEL, object equality comparison is supported by the three features \texttt{is_equal (other: like Current)}, \texttt{equal (some: ANY; other: like some)} and its counter part for deep equality comparison, \texttt{deep_equal (some: ANY; other: like some)} of class ANY. Shallow equality comparison is provided in two forms, a feature taking only the peer object as argument and a feature which takes both objects to be compared as arguments. The advantage of the second form is the possibility to deal with \texttt{_void}' arguments.

In order to provide a consistent approach to \texttt{comparison} in the IFL language, we only use \texttt{object equality} comparison provided by the feature \texttt{is_equal (other: like Current)} and take the reference operators \texttt{=}' and \texttt{=/=} as a shortcut for \texttt{equal (object, Void)}.

One concept that is less important in our case are non-strict Boolean operators like \texttt{and then}' or \texttt{or else}'. As described in chapter 7, these operators are seldom used in specifications involving mathematical models, which is the reason why we decide not to incorporate them into the IFL specification. The \texttt{implies}' operator however proves to be crucial to \texttt{case differentiation} in \texttt{model postconditions} and is therefore supported by the IFL language.

## 5.4 IFL Validity Errors

In this section we formalize the observations of the previous sections of chapter 5. We describe the language restrictions of IFL by giving a set of definitions for \textit{validity errors}. For the following discussions, we assume that the reader is familiar with the validity errors of the current EIFFEL language standard [Mey92].

We present the \textit{validity errors} of the IFL language in four categories: The first category presented in section 5.4.1 lists the \textit{validity errors} related to the EIFFEL class being annotated. Section 5.4.2 presents the second category containing all \textit{validity errors} related to \textit{model features} in an EIFFEL class, and the third category specifies validity errors resulting from \textit{incorrect IFL expressions} (section 5.4.3). Finally, the fourth category presented in section 5.4.4 defines \textit{validity errors} related to \textit{incomplete specifications} of IFL contracts.

All \textit{validity errors} have an associated \textit{error code}, starting with the prefix \texttt{IFL}', a \textit{short summary} of the constructs involved in the validity statement, and a set of \textit{necessary} and \textit{sufficient conditions} for the IFL language construct under definition.
5.4.1 Model Class Validity Errors

The following conditions must be met by classes having a model feature $f$ either declared in the class itself or contributed by a proper ancestor:

- **IFLVMC**: A model class $C$ introducing a model feature $f$ is model-valid if and only if the class satisfies the following condition:
  - Class $C$ must directly or indirectly inherit from $MML\_SPECIFICATION$

5.4.2 Model Feature Validity Errors

A valid model feature $f$ declared in a model-annotated class $C$ has to adhere to the following requirements:

- **IFLVMF**: A model feature $f$ declared in a model class $C$ is model-valid if and only if the feature $f$ is either:
  - a class attribute
  - a pure query

- **IFLVMF-x**: A model feature $f$ declared in a model class $C$ is model-valid if and only if the feature $f$ is:
  - **IFLVMF-1**: a pure feature
  - **IFLVMF-2**: not an external feature
  - **IFLVMF-3**: not a once feature

- **IFLVMA**: A model feature $f$ declared in a model class $C$ is model-valid if and only if the feature $f$ is either:
  - a frozen class attribute
  - a possibly deferred function with an empty list of formal arguments.

- **IFLVMT**: A model feature $f$ declared in a model class $C$ is model-valid if and only if the feature $f$ conforms to at least one of the following types:
  - MML\_ANY
  - INTEGER, CHARACTER, REAL
  - BOOLEAN

- **IFLVME**: A model feature $f$ declared in a model class $C$ is model-valid if and only if the feature $f$ is explicitly exported to the following classes:
  - MML\_SPECIFICATION
  - ANY
5.4. IFL VALIDITY ERRORS

5.4.3 Model Expression Validity Errors

Valid IFL expressions occurring in *model contracts* are subject to the following additional rules:

- **IFLVEF-x**: A feature call $f$ in a *model expression* $E$ is *model-valid* if and only if the call $f$ is either:
  - **IFLVEF-1**: a call to a *pure feature*
  - **IFLVEF-2**: not a call to an *external feature*
  - **IFLVEF-3**: not a call to a *once feature*

- **IFLVEC**: A creation call to a *creation procedure* $c$ on a object $o$ in a *model expression* $E$ is *model-valid* if and only if the procedure $c$ has no *side-effects* except on the object $o$ under creation.

- **IFLVEE-x**: A *model expression* $E$ is *model-valid* if and only if it does not involve the following *reference operators* with a right-hand side expression unequal to 'Void':
  - **IFLVEE-1**: the reference equality operator ‘==$'
  - **IFLVEE-2**: the reference inequality operator ‘/=’

- **IFLVEB-x**: A *model expression* $E$ is *model-valid* if and only if it does not involve the following *non-strict Boolean operators*:
  - **IFLVEB-1**: the Boolean operator ‘and then’
  - **IFLVEB-2**: the Boolean operator ‘or else’

5.4.4 Model Specification Validity Errors

A specification of an Eiffel class using IFL and MML models is *complete* if and only if:

- **IFLVSM**: All features in the *flat form* $F$ of a class $C$ have an associated *postcondition* containing an IFL expression for each of the *model features* appearing in $F$.

- **IFLVSD**: All functions in the *flat form* $F$ of a class $C$ are required to contain at least one *definition assertion* in their *postconditions*.

For further discussions on specification completeness consult section 8.4.
5.5 IFL Language Grammar

The last section of chapter 5 is dedicated to the IFL language grammar. Since the IFL language is a subset of the EIFFEL programming language, we start from the existing grammar describing the upcoming EIFFEL standard [Mey]. Consequently, the IFL language grammar could also be designed as a subset of the EIFFEL language grammar.

The reason why we do not follow this approach here are the various language restrictions that IFL introduces for Boolean expressions. The overall idea behind the definition of any language grammar for type-safe programming languages is to move as much error processing to the parsing phase of a language processing tool, in order to have a grammar which is as strict as possible and which simplifies the semantic analysis. In other words, as many errors in IFL expressions should be processed as syntax errors. We therefore carefully examine occasions to further restrict the EIFFEL grammar for the IFL language. The following section describes the approach we have taken to design the IFL grammar.

5.5.1 Design of the Language Grammar

The basis of our IFL grammar is the EIFFEL grammar used in the GOBO EIFFEL TOOLS [Beza] enriched with the language constructs of the upcoming EIFFEL standard [Mey]. Since IFL is a pure functional language containing pure expressions as basic language concept, it suffices to restrict ourselves to the grammar rules for assertions in the EIFFEL language. We can expect a far less complicated language grammar leading to additional improvements with respect to a simpler semantic analysis.

However, enforcing language rules by syntax sometimes can lead to an explosion of grammar rules. Since the IFL language is as a contract language, the suggestion to enforce the Boolean nature of its expressions by syntax is certainly justified. In conjunction with inline agents, expressions of arbitrary type have to be handled as well, which results in extensive duplication of grammar rules. For simplicity and better readability, we decided to move type checking to the semantic analysis phase.

Similarly, postconditions in the EIFFEL language can contain special expressions referring to the pre-state of the feature (’old’ expressions). Obviously, ’old’ expressions are only valid in postconditions of features. If we would want to enforce this by syntax, all grammar rules describing EIFFEL expressions would have to be duplicated as well, yielding a very complicated language grammar.

The same is also valid for the ’Precursor’ keyword, a mechanism to refer to the version of a feature before it has been redefined. To avoid infinite recursion, calls to ’Precursor’ are not allowed in assertions. This validity rule is also checked during semantic analysis to avoid an explosion of grammar rules.

The following steps lead to the definition of the language grammar:

- **Definition of Start Symbol:**

  The start symbol of the grammar is ’CONTRACT’ because the IFL language is a purely functional language defined for assertions in EIFFEL contracts.
• **Introduction of Inline Agents:**
  Introduce inline agents as described in the upcoming EIFFEL standard [Mey]. This only requires an additional alternative in the rule 'Agent', which handles the formal argument list of the agent and its associated routine body:

```
Agent ::= agent [ AgentTarget ] AgentUnqualified | [ ( Declarations ) ] [ : Type ] Routine
```

The rules 'Declarations', 'Type' and 'Routine' are the same ones that are used to describe formal argument lists, result types and routine bodies.

• **More Strict Call Definition:**
  The definition of the 'Call' rule is more strict in the IFL language grammar than in the upcoming EIFFEL standard [Mey]. The problem with the version from the standard is that the syntax accepts constructs which are not correct EIFFEL feature calls such as `foo.Result` and `Current (foo)`. The IFL language specification given in appendix A has been designed not to accept these erroneous feature calls:

```
Call ::= [ CallTarget . ] CallChain
CallArguments ::= Expression | CallArguments , Expression | empty
CallChain ::= CallUnqualified | CallChain . CallUnqualified
CallTarget ::= Current | Result | ( Expression )
CallUnqualified ::= Identifier [ ( CallArguments ) ]
```

• **Removal of Reference Equality:**
  Based on the discussions of section 5.3 and section 5.4, all occurrences of *reference equality* operators have been removed. Additionally, a new rule 'Void' has been introduced to handle the case where the right-hand side of an expression is equal to 'Void' (see section 5.3.4):

```
Void ::= Expression (= | /=) Void
```

• **Removal of Non-Strict Boolean Operators:**
  All *non-strict Boolean operators* except the 'implies' operator have been removed according to the IFL language specification in section 5.3.4. The specifications of EIFFELBASE classes presented in chapter 7 do not use these operators. Experience shows that the applications of these operators are quite limited and therefore it is not justified to incorporate them into the IFL language grammar.
5.5.2 The Grammar Rules

The last section is dedicated to the design process of the IFL grammar. In this section, we formally describe the language grammar based on the design process and the observations made earlier in this chapter. The grammar is specified in the widely accepted EBNF syntax, which basically uses regular expressions to describe the language constructs.

The start symbol of the IFL grammar is the rule ‘CONTRACT’. A complete description of the grammar can be found in appendix A.
Chapter 6

The Mathematical Model Library (MML)

This chapter presents the principal contribution of this thesis, the Mathematical Model Library (MML).

6.1 Overview

In chapter 4 we have presented the mathematical notions we need to formally specify EIFFEL classes, and in chapter 5 the Intermediate Functional Language (IFL) has been presented to express EIFFEL contracts in a more rigorous mathematical way based on the concepts developed above. This chapter introduces the library of mathematical models (MML), a set of classes designed to model mathematics more closely than already existing implementations of **tuples**, **sets**, **lists** and others, data structures which can be found in many class libraries.

Section 6.2 presents the principal design ideas behind the MML library. In section 6.3 we list the classes provided by the library and explain their purpose. The usage of the MML library is described in section 6.4. The following sections mention the limitations of the current library version (section 6.5) and the problems encountered during implementation (section 6.6). Finally, in section 6.7 we provide an outlook on work that can be done to further improve the class library.

6.2 Design of the MML Library

Designing a library of mathematical notions in a pure object-oriented fashion is not easy. It seems that there is an *impedance mismatch* between object-oriented analysis and design and the structure of mathematical concepts used in set theory and relation algebra. We have identified the following problems:

- **Inheritance Hierarchy Problem**: The inheritance hierarchy of a class library modeling mathematical concepts does not necessarily correspond to the natural taxonomy of mathematical structures. This mismatch is illustrated best by the following example:

  **Example: Inheritance Relation**

  Consider a deferred class `MML_SET [G]` modeling the concept of a mathematical set. Further, we have other deferred classes `FUNCTION [G, H]` representing discrete finite functions and `MML_PAIR [G, H]` corresponding to 2-*tuples*. From a mathematical point of view, it seems
natural to let \texttt{MML\_FUNCTION} \([G, H]\) inherit from \texttt{MML\_SET} \([\texttt{MML\_PAIR} [G, H]]\) to emphasize the fact that \textit{discrete functions} are nothing else than \textit{functional relations} which are themselves described as a set of \textit{maplets} or \textit{pairs}. From the object-oriented point of view taken by the \textsc{Eiffel} method, an \textit{inheritance relation} between the two classes \texttt{MML\_SET} \([G]\) and \texttt{MML\_FUNCTION} \([G, H]\) implies also a \textit{sub-typing relation}. \textit{Polymorphism} and \textit{sub-typing} therefore suggest that every \textit{function} can be viewed as a \textit{set} as well. However, there exist some operations that perfectly valid on sets as \textit{union}, \textit{intersection} or \textit{element insertion}, which potentially break the \textit{functional property} of a \textit{function}.

- \textbf{Object-Orientatio vs. Mathematics Problem}: The core concept of \textit{object-oriented programming} is the notion of \textit{feature call} or \textit{message passing}. In order to initiate a computation, we call a feature on a target object (sometimes denoted as \texttt{"this"}, \texttt{"self"} or in the \textsc{Eiffel} language \texttt{"Current"}). The target object which is an instance of a certain class in the system "knows" how to react to this message (by changing its own state, or by sending further messages to other arbitrary objects). In a mathematical world however, there is no such concept of \textit{message passing} and moreover, objects are \textit{immutable} as already discussed in section 5.2.2. Any implementation of a class library for mathematical models has to deal with these restrictions.

- \textbf{Object Identity Problem}: In mathematics there is no concept of \textit{object identity} and, closely related, no notion of \textit{reference} and \textit{reference equality}. Contrary to object-oriented methods, if two objects are mathematically considered equal, then they have the same identity (they are the same object in fact). We certainly have to account for this substantial difference.

- \textbf{Type System Problem}: The type system of the \textsc{Eiffel} language was not designed to closely model mathematics, but to support \textit{object-oriented design} and \textit{analysis}. It has to be investigated to what extent the mathematical taxonomy of objects can be mapped onto a type system where inheritance relations automatically imply sub-typing and polymorphism. The main priority however is to model the mathematical concepts as closely as possible, if necessary also with a loss in type information.

Based on the main problems identified above, we present the principle design concepts behind the Mathematical Model Library (MML) in the next sections.

### 6.2.1 Facets vs. Abstract Data Types

The recommended method of \textit{system decomposition} promoted by the \textsc{Eiffel} method is the search of \textit{abstract data types} (ADT’s) in the problem domain. It seems not that clear whether this approach will lead to a satisfactory design of a class library modeling mathematical concepts. The reason is mainly the \textbf{Inheritance Hierarchy Problem} mentioned above. We therefore use a slightly different approach to \textit{system decomposition}: The search of \textit{mathematical facets} in the problem domain. The following definition describes the concept of a \textit{facet}:

- \textbf{Definition}: Facet

A \textit{facet} is a \textit{set of features} consolidated in a \textit{class interface} which models a certain aspect of an \textit{abstract data type} (ADT).
Based on the definition of facets, the process of *system decomposition* can be described as follows:

1. **Identify Mathematical Objects**
   
   Step (1) of the design process we identify the basic mathematical notions that have to be modeled in the class library. The focus point for the selection of mathematical constructs *tuples* and *sets*. All other mathematical constructs of *relation algebra* and *set theory* are derived from these concepts. We further introduce relations as sets of tuples describing a mapping between a *domain* and a *range* set. *Functions*, *sequences*, *multisets* and other high-level mathematical structures can all be expressed in terms of *relations*. We select a function \( f : \mathbb{N} \rightarrow X \) which maps the non-negative indexes to corresponding elements as representation for *sequences*. Similarly, we choose a function \( f : X \rightarrow \mathbb{N} \) to associate the *multiplicity* of a *multiset* element with the element itself. Other mathematical structures that are of interest during specification are *powersets*, *Cartesian product*, or even more complex concepts like *graphs* and *networks*. It is not clear however to what extent a library of reusable mathematical models should support such high-level constructs. Whether the mathematical concept justifies introducing a separate class is decided based on the new functionality offered by the construct.

2. **Enumerate Applicable Operations**
   
   Step (2) of the design process consists of finding the necessary operations applicable on the mathematical structures. We use an *incremental* approach: If an operation is considered crucial for the mathematical structure, it is accepted into the set of *applicable operations*. Otherwise it is discarded. This approach guarantees a *minimal* and *easy to use class interface* of the mathematical model. Later during the implementation phase, it is likely that some operations are missing for the structure. If we can justify to introduce these operations, we include them as well.

3. **Find Facets in Operations**
   
   Having identified the most important mathematical abstractions with their associated operations, we are trying to find *intersections* of the sets of *applicable operations* on the mathematical structures. Such a set is called a *facet* and usually yields a separate EIFFEL class, capturing a certain property of objects that exhibit this *facet*. For the set of *mathematical abstractions* found in step (1) (*Mathematical Object Identification Step*), the following *facets* can be found:

   - **Convertible Facet**: Mathematical objects that can be *converted* to other objects which do not necessarily have to be *mathematically compatible*. As an example, a *multiset* can be converted to a *random sequence*.
   - **Extendible Facet**: Container objects where elements can be *inserted* or *removed*, usually collections of elements.
     
     **CAUTION**: This facet provides features which are not *pure* and therefore can only be used *internally*.
   - **Filterable Facet**: Container objects that can be *filtered* according to a certain criterion, usually given as a *unary predicate* (based on the functional *filter*).
   - **Mappable Facet**: Mathematical objects that can be *mapped* to arbitrary elements using a *mapping function* (based on the functional *map*).
CHAPTER 6. THE MATHEMATICAL MODEL LIBRARY (MML)

- **Processable Facet**: Container objects which offer *iteration* facilities to traverse container structures.
- **Quantifiable Facet**: Mathematical objects over which *universal* and *existential quantifiers* are defined.
- **Reducible Facet**: Container objects over which a value can be computed by *accumulation* (based on the functional *reduce*).
- **Typeable Facet**: Mathematical objects that provide *type information*.

Each of the facets listed above will result in a separate potentially deferred EIFFEL class. Classes needing these facilities will therefore establish an *inheritance relation*.

4. **Identify Mathematical Taxonomy**

The last step in the design process is the development of a clean and simple understandable *inheritance hierarchy* which models the mathematical taxonomy of *set theory* and *relation algebra* as accurate as possible. To support facilities which are common to all mathematical objects (see section 6.2.7), we introduce a common *ancestor class* with the name MML_ANY. The sole purpose of this class is to introduce a mathematical *equality relation*, a mathematical *compatibility relation* which states whether an object can be substituted by another one, and *type information* such as whether an object is set-valued, function-valued, sequence-valued and so forth.

Further, based on the discussion in step (1) (*Mathematical Object Identification Step*), we create a class MML_PAIR [G, H] and a class MML_SET [G] for the notions of *ordered pairs* and *unordered collections* of elements. Relations are modeled by class MML_RELATION [G, H] which naturally inherits from MML_SET [MML_PAIR [G, H]] to express the mathematical nature of relations. Further, we use the functionality provided by the *relation model* to implement a class MML_FUNCTION [G, H], which inherits from MML_RELATION [G, H] and models the notion of a discrete finite function. The sub-typing relation induced by the inheritance relation closely models the mathematical relationship between the two concepts.

The notion of a *multiset* is captured by class MML_BAG [G], a descendant of MML_FUNCTION [G, INTEGER]. Similarly, MML_SEQUENCE [G] represents the mathematical notion of *sequences*, ordered collections derived from MML_FUNCTION [INTEGER, G]. Note the inherent similarity between *bags* and *sequences* in the MML library. We expect that both classes can benefit from the rich facilities provided by the common ancestor class MML_FUNCTION [G, H] and its implementations. However, the polymorphic relation between multisets/sequences and functions is not existent and should therefore be eliminated. Luckily, the upcoming EIFFEL standard offers the concept of expanded inheritance which allows sub-typing without polymorphic use.

The mathematical notions described in the paragraphs above are only basic constructs and in many cases not powerful enough to model the complicated relationships of object structures in object-oriented software systems. We therefore implement two other classes MML_POWERSET [G] and MML_GRAPH [G] which represent *powersets* (sets of sets) and *graphs*, respectively. It can be argued whether the notion of powersets justifies the introduction of another class. Powersets provide specialized *union* and *intersection* operations for *generalized union* and *intersection*. The wide use of these operations makes a separate class very convenient. Moreover,
the mathematical notion of a graph is a crucial concept and therefore integrated into the MML library, if not for its original operations, then for its convenience in contract specifications. Another important concept of set theory is the Cartesian product. Its relationship with relations and power sets would make it an ideal candidate for inclusion into the class library. After some investigations on what kind of operations are applicable to the Cartesian product we decided not to implement such a class.

![Figure 6.1: The MML Model Class Hierarchy](image)

### 6.2.2 Strict Interface/Implementation Separation

The relatively high impedance mismatch between the pure mathematical objects and their procedural implementations suggests a strong information hiding policy. We are following this design rule and create two kind of classes:

- **Model Interface Classes**: Interfaces and facets of a mathematical model.
- **Model Implementation Classes**: An implementation of the associated model interface class.

Each mathematical model consists of a deferred class MML_XXX representing the model interface of the mathematical model. Associated with this interface, the MML library provides at least one implementation of the corresponding model interface. The MML library in its current form provides a default implementation with the name MML_DEFAULT_XXX for each model interface. The advantage of this interface/implementation separation is the strong distinction between specification and verification as described in section 6.2.5.
6.2.3 Default Implementation of Interfaces

Strong interface/implementation separation facilitates the use of several model interface implementations tailored to specific needs. This option is especially important with respect to run-time assertion checking. Mathematical models make it possible to specify complex graph problems with very little effort. Most of the problems related to graphs and networks require sophisticated algorithms to solve them, if they are solvable in polynomial time at all. The MML library easily supports the extension of the library by specific algorithms for hard problems, just by implementing the desired model interface. But most of the time, the default implementations are sufficient, since in this library the most efficient algorithms applicable are used anyway, e.g. the Floyd-Warshall [CLRS01] algorithm to calculate the transitive closures of relations.

![Figure 6.2: The MML Implementation Class Hierarchy](image)

Additionally, the MML library also facilitates the creation of mathematical models with additional facets. The following example shows an implementation of the Reducable Facet:

**Example**: Consider the MML class `MML_REDUCEABLE_SET [G -> NUMERIC, NUMERIC]` representing the notion of a mathematical set containing numeric elements only. The set class should provide a generic operation which accepts a binary operator defined on numeric types and an initial numeric value. The result of this operation is the numeric value computed incrementally on the set elements. The model interface of this class can therefore be composed from `MML_SET [G -> NUMERIC]` and `MML_REDUCEABLE [NUMERIC, NUMERIC]`. We use the default implementation `MML_DEFAULT_SET [NUMERIC]` of the MML library, and are only required to implement the feature `reduced`:
6.2. DESIGN OF THE MML LIBRARY

deferred class
  REDUCEABLE_SET [G -> NUMERIC, H]

inherit
  MML_DEFAULT_SET [H]
  undefine
    copy
  end

  MML_REDUCEABLE [G, H]
  undefine
    copy
  end

end -- class REDUCEABLE_SET

6.2.4 Feature Declaration/Feature Implementation Separation

Another consequence from the interface/implementation separation is the principle of feature declaration/feature implementation separation. Model interfaces and model implementations have different tasks to fulfill:

- **Model Interface**: Specification of Class Interfaces, Regulation of Feature Export and Feature Renaming

- **Model Implementation**: Provider of Feature Implementations and Redefinitions

The features declared in the model interface determine the class interface of the mathematical model. The export status of the features and the renaming of feature names is controlled in the interface. The implementation of the class is provided by the model implementation. Any redefinitions of existing feature implementations are performed in the model implementation as well.

6.2.5 Specification vs. Implementation

The last consequence following from the interface/implementation separation principle concerns the difference between specification and implementation. The deferred model interface classes can be used for the sole purpose of specification and documentation, whereas the implementation classes are designed to support run-time assertion checking as provided by any EIFFEL run-time system. This ensures that deferred classes which are annotated with mathematical models for specification purposes only do not have to reference model implementations.
6.2.6 Type Safety vs. Generality

During the implementation of the MML class library we have come across situations where the type system of the EIFFEL language is not expressive enough to express the correct types of mathematical models. An important shortcoming of the EIFFEL type system are formal generic parameters. In EIFFEL, genericity is provided on a class basis, and not on a feature basis as e.g. in C++. This property makes it impossible to introduce anonymous types in features like composed of the class MML_RELATION [G, H]:

\[
\text{composed}(\text{other} : \text{MML}\_\text{RELATION}[H, I]) : \text{MML}\_\text{RELATION}[G, I] \quad (6.1)
\]

The formal generic parameter I cannot be declared on a class basis, since it is not known what type the argument other has at compile-time. The signature of the feature composed displayed above would not cause any problems if genericity is introduced on a feature basis. We therefore have to search for an alternative solution. There are at least two ways to solve this problem:

- **Resort to Untyped Arguments**: Use ANY for arguments whose type involves anonymous types. The signature of the composed feature would then look like:

\[
\text{composed}(\text{MML}\_\text{RELATION}[H, \text{ANY}]) : \text{MML}\_\text{RELATION}[G, \text{ANY}] \quad (6.2)
\]

The disadvantage of this approach is that IFL expressions loose type information. However, it is seldom necessary to access type information of elements in mathematical models, what makes strong typing less important on the level of elements in mathematical structures.

- **Use Auxiliary Classes**: Define auxiliary classes with additional formal generic parameters corresponding to the anonymous types of feature arguments. Consider the following example which introduces a compositor class for relation composition:

```eiffel
deferred class
    COMPOSITOR_EXAMPLE [G, H, I]

feature -- Composition
        -- The relation composed of 'first' and 'second'.
        deferred
    end
end -- class COMPOSITOR_EXAMPLE
```

We use the first approach for the MML library. IFL expressions are usually short and do not depend on types. Introducing auxiliary classes for this problem could result in an explosion of classes.

Another problem where a compromise between type safety and generality has to be found manifests itself in the feature is_identity of class MML_RELATION [G, H]. A necessary condition for a relation to be the identity relation is the equality of this domain and range sets. Unfortunately, the simple expression domain.is_equal (range) is not valid, since the formal generic parameters G and H are unrelated. This problem also arises at various other places.

To remedy this problem, we introduce the concept of type stripping. The **Typeable Facet** includes a feature untyped, which has a type derived from the current type of 'Current', except that all formal generic parameters of the base class are replaced by ANY. The following example illustrates this:
deferred class
PAIR_EXAMPLE [G, H]

inherit
MML_ANY
  undefined
  untyped
end

MML_TYPEABLE [PAIR_EXAMPLE [G, H]]

feature {MML_ANY} −− Type
  untyped: PAIR_EXAMPLE [ANY, ANY] is
    −− The object ‘current’ as typeless object.
    do
      Result := Current
    end
end −− class PAIR_EXAMPLE

Therefore, the general strategy for type safety is the following: Start with a type-less implementation using only ANY as type, and then incrementally infer types until typing problems occur (see feature composed). This approach has been proven very successful and has led to the discovery of a number of errors in the MML class library.

6.2.7 Object Equality/Mathematical Equality

The last section presenting the design decisions behind the MML library is dedicated to object equality. We have already discussed why reference equality is not suitable in our context. However, even object equality might be too strong in some cases. Consider an instance of class MML_SET [MML_SET [G]] and another instance of class MML_POWERSET [G]. Should these two instances considered to be equal if they contain the same set-valued elements? In a mathematical context they should, since the two instances represent the same abstraction and also the same object.

That is the reason for another equality relation called mathematical equality. This relation states whether two objects are mathematically equivalent. The common ancestor class MML_ANY introduces the feature equals (other: MML_ANY) which implements this relation. Note that contrary to is_equal (other: like Current) the signature is not declared using result type covariance to better capture the nature of mathematical equality. It can be questioned whether two equality relations are indeed necessary in the MML library. For reasons of simplicity, MML objects do not support the object equality relation anymore. The implementation of is_equal in class MML_ANY has been designed to replace object equality by mathematical equality:
deferred class
  EQUALITY_EXAMPLE
inherit
  ANY
  redefine
    is_equal
  end
feature  --  Comparison
  is_equal (other: like Current): BOOLEAN is
    --  Is ‘other’ considered equal to ‘current’?
    do
      Result := equals (other)
    ensure then
      implementation_of_equal: Result.is_equal (equals (other))
    end

  equals (other: ANY): BOOLEAN is
    --  Is ‘other’ mathematically equivalent to ‘current’?
    deferred
end  --  class EQUALITY_EXAMPLE

To implement the mathematical equality relation we have to perform the following steps:

1. Defer is_equal in every model interface
2. Undefine is_equal in every feature adaptation clause of the model implementation
3. Implement is_equal in every model implementation if necessary

These steps ensure that the feature adaptation clauses of the MML library remain as short as possible. Experience has shown that almost any model implementation needs a specialized implementation of the equality relation anyway, thus causing not much extra work to be done.

6.3 The Classes of the MML Library

After the last section describing the general design, we present the mathematical model classes provided by the MML library. There are three basic class categories: mathematical model classes (section 6.3.1), facet classes (section 6.3.2), and auxiliary classes (section 6.3.3). Each of these categories is described in detail in the following sections.

6.3.1 Mathematical Model Classes

This section describes the mathematical model classes of the MML library.

- MML_SPECIFICATION: This special marker class is used to export model features. It does not provide any features at all, but has only documentation purposes. Newly introduced model features are declared in a feature clause which exports its features to MML_SPECIFICATION and ANY. See section 6.4 for a detailed role description of this class.
6.3. **THE CLASSES OF THE MML LIBRARY**

- **MML\_ANY**: The common ancestor class of the MML library is responsible for a default implementation of the mathematical equality relation and for the introduction of mathematical type information. In class **MML\_ANY** the object equality relation is replaced by its mathematical counterpart. For a list of applicable operations see appendix B.2.1 and B.4.

- **MML\_PAIR**: Class **MML\_PAIR** \([G, H]\) represents the mathematical notion of a 2-tuple, or pair. It provides different operations to access the tuple components (features `first`, `second` and `element`) and features conversion operations like `inversed` and `identity_pair`. Additionally, class **MML\_PAIR** implements the **Typeable**, **Quantifiable** and **Processable** Facet (see section 6.2.1). The operations provided by **MML\_PAIR** are listed in appendix B.2.2.

- **MML\_SET**: Class **MML\_SET** \([G]\) models the notion of a mathematical set. It provides operations for set membership (feature `is_member`), functions to randomly access the set elements (features `any_element` and `any_item`), queries for set containment like `is_superset`, `is_subset` and others, plus basic set operations as `set union` (feature `united`) and `set intersection` (feature `intersected`). Additionally, class **MML\_SET** implements the **Typeable**, **Quantifiable**, **Processable**, **Filterable**, **Convertible**, **Mappable** and **Extendible** Facet (see section 6.2.1). Operations applicable to **MML\_SET** instances are listed in appendix B.2.3.

- **MML\_POWERSET**: Class **MML\_POWERSET** \([G]\) is an extension to the **MML\_SET** \([G]\) class which implements the notion of a set of sets (a powerset). Apart from all the functionality inherited from **MML\_SET**, **MML\_POWERSET** introduces the concept of generalized set union and intersection. The class therefore implements the same facets as **MML\_SET**. Its applicable operations are listed in appendix B.2.4.

- **MML\_RELATION**: Class **MML\_RELATION** \([G, H]\) introduces the concept of a mathematical relation based on a set of pairs. It features operations to query the type of the relation (features `is_reflexive`, `is_symmetric`, and others) and implements conventional projection operations such as `domain`, `range` and `image`, associated with several domain restriction and range restriction functions. Further, class **MML\_RELATION** introduces the composition operation which combines two relations and the `inversion` operation for relations. Another category of operations provided by relations are transitive closures. Class **MML\_RELATION** provides a feature `transitive_closure` and a feature `reflexive_transitive_closure` for transitive closures and reflexive transitive closures, respectively. This class implements the **Typeable** and **Mappable** Facet (see section 6.2.1). All available operations are listed in appendix B.2.5.

- **MML\_FUNCTION**: Based on class **MML\_RELATION**, the MML library provides the notion of functions in class **MML\_FUNCTION** \([G, H]\). This class declares features to query the type of a function (features `is_partial`, `is_total`, and others). Like class **MML\_RELATION**, class **MML\_FUNCTION** also defines function intersection and function union, as well as function composition and inversion. Additionally, the class introduces a special type of closure, the `sequence_closure`. Function models implement the same facets as relations. The operations available on functions are listed in appendix B.2.6.

- **MML\_BAG**: The notion of multisets is represented by class **MML\_BAG** \([G]\). Similar to class **MML\_SET** \([G]\), multisets also offer features for bag containment (features `is_superbag`, `is_subbag` and others). It also provides a very convenient function to convert multisets into random sequences of the elements. The remaining operations are similar to those from the set model
**MML_SET** \([G]\). Additionally, multisets also implement the **Typeable**, **Quantifiable** and **Processable Facet** (see section 6.2.1) to make it possible to iterate over the same element multiple times according to its **multiplicity**. The multiset operations are listed in appendix B.2.7.

- **MML_SEQUENCE**: Class **MML_SEQUENCE** \([G]\) represents a mathematical sequence. This class provides features to insert and remove elements from a sequence (features **appended**, **prepended**, **extended_at**, **pruned_at**, and others), features for **sequence containment** (features **is_subsequence**, **is_supersequence**, and others). Additionally, class **MML_SEQUENCE** also offers operations like **sequence reversal**, **sequence partition** and **sequence decomposition**. A complete list of features applicable to sequence objects can be found in appendix B.2.8.

- **MML_GRAPH**: The last **model class** in the MML library models the notion of a **directed graph**. Graphs are represented as pairs of a **node set** and an **edge relation** in the MML library. class **MML_GRAPH** \([G]\) naturally inherits from **MML_PAIR** \([MML_SET\ [G], MML_RELATION\ [G, G]]\). This class offers features to insert and remove new nodes from the graph (features **node_extended** and **node_pruned**) and similarly, features to change the edges of a graph (features **edge_extended** and **edge_pruned**). Further, class **MML_GRAPH** also provides features to query for **graph containment** (features **is_supergraph**, **is_subgraph**, and others), very similar to sets. Additionally, it supports also simple queries for **graph connectivity** and **graph completeness**. Its applicable features are listed in appendix B.2.9.

### 6.3.2 Facet Classes

This section describes the different **facet classes** of the MML library, based on the previously introduced **facets** in section 6.2.1. All operations provided by the following facets are listed in appendix B.3.

- **MML_CONVERTIBLE**: This **facet class** offers features to convert mathematical objects into other objects. Currently, class **MML_CONVERTIBLE** offers a feature **identity_pair** to create a pair with its first and second element equal to the current object. This facility is used at various places in class **MML_RELATION** and its implementations.

- **MML_EXTENDIBLE**: Class **MML_EXTENDIBLE** is an **internal class** which provides an interface for mathematical objects which support the insertion and removal of elements. This class provides two features **extend** and **prune**, which both are not pure. It is used in all **model classes** of the MML library, but should not be used by any clients of the class library.

- **MML_FILTERABLE**: This class provides the concept of a mathematical structure whose elements can be filtered according to a **unary predicate**. Feature **filtered** has signature **filtered** (FUNCTION [ANY, TUPLE [G], BOOLEAN]): like **Current**. It is used by nearly all classes in the MML library.

- **MML_MAPPABLE**: Class **MML_MAPPABLE** represents the concept of a structure containing transformable elements and offers a feature **mapped** which performs the transformation. It has signature **mapped** (FUNCTION [ANY, TUPLE [G], H]): **MML_SET** [H]. The only argument of this feature is the **mapping function**, mapping an element of type **G** to a corresponding element of type **H**. This class is primarily used in class **MML_RELATION** and its implementations.
6.3. THE CLASSES OF THE MML LIBRARY

- **MML_PROCESSABLE**: The **Processable Facet** provides two kinds of features: **Pure features** designed to be used by clients of the MML class library, and **impure features** for internal use. Feature *applied* can be used by clients to apply an action to all elements of a mathematical structure. Its signature is *applied (PROCEDURE [ANY, TUPLE [G]])*. The other features *do_all* and its conditional variant *do_if* are used internally and are not exported to clients of the class library.

- **MML_QUANTIFIABLE**: This class represents the notions of *existential* and *universal quantifiers*. Both quantifiers take an argument of type *FUNCTION [ANY, TUPLE [G], BOOLEAN]* representing a *unary predicate*. This class is used throughout the whole MML class library, often in conjunction with **MML_FILTERABLE**.

- **MML_REDUCEABLE**: Feature *reduced* of class **MML_REDUCEABLE** implements the functional *accumulate* introduced in section 4.9.4.3. This functional takes three arguments: An initial value of type *G*, an operator function of type *FUNCTION [ANY, TUPLE [G, G], G]* and a mapping function *FUNCTION [ANY, TUPLE [H], G]*. The mapping function can be used if the values to be accumulated have to be computed first from the element of the structure. If this is not the case, the *identity function* can be used as argument.

- **MML_TYPEABLE**: The **Typeable Facet** provides the two important concept of *mathematical type information* and *mathematical compatibility*. The feature which implements mathematical compatibility, *is_compatible*, is primarily used in the implementations of *is_equal* of class **MML_ANY**. The features providing *mathematical type information* are designed to be only used by clients of the MML library.

### 6.3.3 Auxiliary Classes

In this section we present the **auxiliary classes** of the library.

- **MML_FUNCTIONALS**: Class **MML_FUNCTIONALS** is an **auxiliary class** used by various *model implementation* classes. It offers *higher-order agents* and other *functionals* which are very convenient to use in *iterations* over collection and *quantifiers* from first-order logic. Other features offered by this class are *Boolean operators* (features *anded*, *ored*, *negated*, and others) and functionals for *function currying*, *function composition* and *function iteration*. All available features are listed in appendix B.3.
6.4 Usage of the Class Library

The last section has described the different types of classes in the MML library. Here in this section we describe how to use the MML library. Section 6.4.1 describes the compilation process of the MML library. In section 6.4.2 we revisit the specification process of section 3.3 and explain it again in the context of the MML class library.

6.4.1 Compiling the Class Library

The MML class library in its current version does not need any additional libraries to compile correctly. The following components are required:

- ISE EIFFELSTUDIO 5.3
- ISE EIFFELBASE Class Library

The class library depends on the EIFFELBASE library shipped with EIFFELSTUDIO 5.3. To compile the MML library, open EIFFELSTUDIO and create a new project from one of the ACE files in the root of the MML code base. All files use relative path names only. In order to resolve the cluster paths in the MML code base correctly, the developer has to set the environment variable `$MML` to the path of the code base root.

The ACE files itself are completely platform-independent and can be used on all platforms supported by ISE EIFFELSTUDIO.

6.4.2 Specification with the Class Library

This section gives a summary on how to use the MML library for specification of EIFFEL classes. The actual specification process using mathematical models is described in detail in section 3.3. The following steps have to be performed to introduce a new model if using the MML library:

1. **Feature Clause Declaration**: Declare a feature clause exported both to MML_SPECIFICATION and ANY.

   Each class introducing new model features is required to declare a new feature clause with appropriate export status. The library provides a special marker class MML_SPECIFICATION for this purpose. By convention, model feature clauses are placed at the beginning of the class text, annotated with the comment 'Model'. More details can be found in section 5.2.1, in particular the Model Feature Declaration Rule.
6.4. Usage of the Class Library

2. Inheritance Clause Declaration: Declare an inheritance clause for MML_SPECIFICATION if the class does not already conform to MML_SPECIFICATION.

In case the model class under specification does not already conform to MML_SPECIFICATION, let the class inherit from MML_SPECIFICATION according to the Model Class Inheritance Rule of section 5.2.1. The example below shows the newly introduced inheritance clause:

```plaintext
defered class SET_EXAMPLE [G]
inherit
  MML_SPECIFICATION
feature {MML_SPECIFICATION, ANY} -- Model
  set_model: MML_SET [G] is
    -- The set model
    deferred
    ensure
      pure_feature: is_equal (old Current)
      result_not_void: Result /= void
    end
end -- class SET_EXAMPLE
```

3. Model Selection: Select the appropriate model implementations from the MML library to implement the identified model features.

Each one of the model features representing a particular abstraction function has to be mapped to corresponding classes of the MML library. Use a specific MML class which represents the appropriate abstraction. Note that combinations of several models are possible like MMLPAIR [MML_SET [NODE], MML_RELATION [NODE, NODE]], which represents a directed graph.

4. Model Implementation: Implement each model feature such that a valid model instance is returned.

According to the Model Implementation step of section 3.3 and the Model Implementation Rule of section 5.2.1, model features have to provide a non-void result representing a model object. If the model feature is an attribute, this step can be neglected. In case the model feature is a function, any implementation is required to create the corresponding 'Result' entity and initialize it in such a way that the postcondition and the abstraction invariant of the model feature, and any gluing invariant involving that model, are satisfied. The code snippet below shows a possible implementation of a model feature representing a mathematical multiset:
deferred class
  BAG_EXAMPLE [G]

inherit
  LINEAR [G]

MML_SPECIFICATION

feature \{MML_SPECIFICATION, ANY\} -- Model

  bag: MML_BAG [G] is
    -- The bag model
    do
    create \{MML_DEFAULT_BAG [G]\} Result.make_empty
    from
    start
    until
    after
    loop
      Result := Result.extended (item)
      forth
    end
    ensure
      pure_feature: is_equal (old Current)
      result_not_void: Result /= void
    end

end -- class BAG_EXAMPLE

5. **Contract Specification**: Specify the contracts of the *model class*.  
   This step is analogous to the one described in section 3.3.

6. **Equality Relation**: Implement the *equality relation* of the *model class* accordingly.  
   As described in section 3.3, the *equality relation* is an important cornerstone of the specification of classes with mathematical models. After the previous discussions, the rule is now very simple to determine: Any *model class* introducing one or more *model features* has to redefine existing implementations of *is_equal* or must introduce a new *equality relation*. Each introduced *model feature* has to be tested for equality, and the results have to be `and-ed` with the result of the existing implementation, if any. This ensures that an object is equal if and only if all its models are equal.

7. **Gluing Invariants**: Directly or indirectly relate all the introduced *model features* to the existing ones.  
   This step is analogous to the one described in section 3.3.
6.5 Limitations of the Class Library

Some of the current limitations of the MML class library are listed in this section.

The EIFFEL type system in the current standard [Mey92] has some shortcomings which have turned out to be relevant for the design of the MML class library. Multiple inheritance of generic classes can cause unpredictable behavior. If a class \( A \) inherits twice or more times from the same class \( B \), but with different actual generic parameters, polymorphic feature calls to features of class \( B \) cannot be correctly resolved. Renaming of feature names resolves all name clashes. But for the resolution of the correct feature implementation, we have to consider two separate cases: If the actual generic parameters conform to each other, without overloading of features there is no possibility to ever resolve these feature calls correctly. Since the EIFFEL language does not offer overloading, cases with conforming actual generic parameters have to be forbidden.

For non-conforming actual generic parameters there is no reason to disallow multiple inheritance of the same class. All feature calls can be correctly resolved from the static type of the call target. Unfortunately, no EIFFEL compiler supports this at the moment. The tendency in the upcoming EIFFEL standard [Mey] is to disallow multiple inheritance of the same generic class with different actual generic parameters in general. More about this problem can be found in category Eiffel Type System of section 6.6.

Another limitation of the MML class library is partly related to its mathematical nature and partly to its 'functional style' implementation. Specifications using mathematical models offered by the MML library cannot express constraints involving reference equality. The interface classes of the MML library model pure mathematical models, which are in no way related to the notion of references. Additionally, the design of the MML library only allows immutable objects, objects whose applicable operations are all pure. These two main characteristics of the MML library imply that MML classes are not able to model reference equality. In object-oriented systems, reference equality is an integral part. It has to be investigated to what extent reference equality can be simulated by mathematical models.

The EIFFEL type system does not seem powerful enough to express mathematical models in a type-safe way. The missing option to declare anonymous generic parameters as in feature composed (other: MML_RELATION [H, I]): MML_RELATION [G, I] of class MML_RELATION [G, H] and the inability to declare the base class of a type using anchored declaration require a lot of patches in the implementation to gain as much type-safety as possible. The approach during the design of the class library has been to use as much type information as possible. For cases where the type system of the EIFFEL language is not able to give a solution, where have introduced the concept of type stripping. The Typeable Facet (see section 6.2.1) provides a feature to convert a mathematical object to an untyped object. In these cases, type safety is not guaranteed anymore on the level of the elements in a mathematical structure, but on the object level, it is of course guaranteed.

Last, the current version of the MML library does not support continuous functions. The implementation class MML_DEFAULT_FUNCTION can only model finite discrete functions. However, the interface class MML_FUNCTION fully supports continuous functions. Clients of the MML library which need support for continuous functions therefore only have to provide a default implementation for continuous functions.
6.6 Implementation Problems

The last section of this chapter about the MML library lists some of the problems that have been encountered during the implementation of the class library. We are presenting the problems in different categories. All problems are referring to the MML implementation developed with ISE EIFFELSTUDIO 5.3.

- **Inline Agents:**

  - **Functional Programming:** Experience has shown that *inline agents* are a necessary and powerful tool for *functional programming*, as required in IFL *model contracts*. Unfortunately, only the upcoming EIFFEL standard [Mey] supports *inline agents*. In the current version of the MML class library, we have to resort to *wrapper features*. Additionally, *operator features* are not supported by the ISE compiler. All *wrapper features* have been declared with an ‘*obsolete*’ clause stating that these wrappers can be removed as soon as *inline agents* are implemented in the ISE EIFFEL distribution.

- **Missing Compiler Support:** The ISE compiler currently does not support the creation of agents from attributes or constants. Luckily, *attributes* to be used as *agents* can be declared as functions returning the value of an *internal implementation attribute*. The same also applies to *agents* on *open targets* of *basic types* and class *NONE*. The ISE compiler is not able to generate code for these language constructs, although the standard explicitly allows these constructs.

- **Class Interface:**

  - **Declarations in Eiffel:** The EIFFEL language has been designed to be strict with respect to *feature declaration*. Redefinitions of features have to be explicitly stated in a *feature adaptation clause*. Together with *feature renaming* and *feature undefinition*, the adaptation clauses can become tedious to create and to maintain. A possible solution to this problem is given in category *Eiffel Language Specification*.

  - **Missing Tool Support:** This is a rather general problem. The EIFFEL community misses some powerful tools for *renaming of class features* and *attributes*, and in general support for *refactoring*. Tools which are able to handle the complex *feature renaming operations* in EIFFEL classes would ease the work of the developer significantly, and *seamless development* would be put into practice finally.

- **Design Decisions:**

  - **Common Ancestor Class:** Most of the time, the design of a class library requires the choice for a *common ancestor class*. After the initial design phase, it was clear that a class *MML_ANY* should capture all properties common to mathematical objects such as *mathematical equality*, *type information* and *compatibility*. The question has arisen whether the elements in the mathematical structures also have to agree to a *common ancestor class*. As it turned out, the *object equality relation* induced by class *ANY* is sufficient.
6.6. IMPLEMENTATION PROBLEMS

- **Auxiliary Functionals**: The MML classes are implemented using many generic functionals and higher-order agents. The first thought has been to include these functionals into the implementation classes of MML_SET. However, classes like MML_PAIR which do not inherit from MML_SET also need the facilities provided by higher-order functions. Moreover, these facilities are probably also needed by clients of the class library, which is the reason why these features have to be available in the model interfaces. We therefore decided to create a new facility class MML_FUNCTIONALS which provides access to functionals and higher-order agents. Client classes simply inherit from MML_FUNCTIONALS by facility inheritance.

- **Generic Tuples vs. Pairs**: The implementation of generic n-tuples is difficult if not impossible to achieve with the current facilities offered by the Eiffel language and run-time environment (see category Eiffel Kernel Classes). We have therefore decided to only implement a specialized variant of a tuple, 2-tuples or pairs. Experience has shown that pairs are sufficient for most uses. Additionally, the MML class library also offers sequences (class MML_SEQUENCE) which could be used as untyped tuples in the form of MML_SEQUENCE[ANY] instances.

- **Covariant Result Types**: Covariant redefinition of feature result types sometimes can lead to code duplication when used in conjunction with multiple inheritance and the strict interface/implementation separation principle (see section 6.2.2) applied to the mathematical models in the MML library (see classes MML_RELATION and MML_FUNCTION). Nevertheless, we have decided to use covariance where appropriate.

- **Types vs. Set of Elements**: The modeling of mathematical relations for the MML library has raised the question whether a relation should be defined by a domain and range set with a pair set describing the relationship between the two sets, or just by the pair set itself. More generally, it is unclear whether the type of the domain and range sets can be described by the set of elements (the domain and range, respectively) itself. The advantage of the first approach is the ability to define a relation on parts of a domain set. Experience however has shown that this is rarely needed. We have therefore decided to describe a relation by a pair set, and to derive the domain and range sets from the pair set itself.

- **Eiffel Type System**:

  - **Declaration of Formal Generic Parameters**: During the implementation of the model class MML_RELATION [G, H], different approaches have been implemented. One approach uses formal generic set parameters to parameterize the domain and range set of relations. The formal parameter list looks like MML_RELATION[D -> MML_SET[G], R -> MML_SET[H]]. The class itself inherits consequently from MML_SET[MML_PAIR[G, H]]. Unfortunately, the ISE compiler seems to dislike this construct and complains about unknown formal generic parameters G and H. This limitation can be circumnavigated by stating the generic parameters before declaring the set parameters as in MML_RELATION [G, H, D -> MML_SET[G], R -> MML_SET[H]]. In the current library version, a simpler solution has been found (see class MML_DEFAULT_RELATION).
• Anonymous Types: As already discussed in section 6.2.6, the EIFFEL language does not allow anonymous types. Types derived from generic classes are based on these classes and not on single features. The feature composed of class MML_RELATION [G, H] which has signature composed (MML_RELATION [H, I]): MML_RELATION [G, I] is not valid in EIFFEL. All known EIFFEL compilers complain about an unknown class name I and do not recognize that I is a formal generic parameter. There are two solutions to this problem: First, one could write an auxiliary class MML_COMPOSITION [G, H, I] which would implement the feature composed described above and perform the necessary type casts. Second, anonymous types could be replaced by ANY, thus loosing the relationship between the argument type and the result type of feature composed. In order to avoid an explosion of classes in the MML library, we have chosen the second approach.

• Generic Anchored Type Declaration: Feature composed of class MML_RELATION poses even another problem. In order to ensure that relations can only be composed with relations, and functions only with functions, we have to state that the feature arguments are either of relation or function type, but not mixed. Unfortunately, anchored declaration is only possible on a type basis, but in our case the base class of the type should be covariantly declared; the formal generic parameters cannot be declared using anchored declaration. The signature of feature composed would look like compose (like Current [H, ANY]): like Current [G, ANY]. Note that this is not a valid EIFFEL feature signature. Our proposal is to investigate the extension of covariant redefinition to base classes of types. The formal generic parameters could be covariantly constrained as in FOO [G -> like Current], and the base class could be covariantly declared as (like Current) [G]. In every case, the usage of this extended form of covariant redefinition is not limited to the example above and would improve the expressiveness of the EIFFEL type system further.

• Constrained Genericity for Reference Types: Currently, the EIFFEL language does not provide means to specify that formal generic parameters should be instantiated with reference or expanded types only. Sometimes this can lead to problems when using expressions of generic types on the right-hand side of an assignment instruction. The solution is to use assignment attempt in conjunction with a creation constraint of the form FOO [G -> create make end]. However, this can yield circular reference in the inheritance clauses.

• Multiple Inheritance of Generic Classes: The current EIFFEL standard [Mey92] allows a descendant class to inherit twice or more times from the same generic class with different actual parameters. Apart from the very complex feature adaptation clauses with large ‘rename’ and ‘select’ clauses to resolve name clashes and other ambiguities, this can also cause unpredictable behavior. Consider the following example from the MML class library: class MML_SET [G] implements the Quantifiable Facet (see section 6.2.1) by inheriting from MML_QUANTIFIABLE [G]. Among the descendant classes of MML_SET, class MML_POWERSET [G] provides specific operations on sets of sets and inherits naturally from MML_SET [MML_SET [G]]. The inherited Quantifiable Facet is only defined on sets, but ideally we would like another one on the elements in the sets. The introduction of another inheritance relation with MML_QUANTIFIABLE [G] (the other one is now defined on sets because of inheritance) causes problems: The ISE compiler is not able to resolve feature calls to quantifier features correctly anymore. Theoretically, after resolving any name clashes, the settings are unambiguous. Type G and MML_SET [G] never conform to each other, which makes it possible for the compiler to always deduce the correct feature from the static type of the feature call target.
6.6. IMPLEMENTATION PROBLEMS

• Eiffel Kernel Classes:

• TUPLE: This class does not offer a feature to reverse a tuple. The reversed tuple of a tuple \((a_1, a_2, \ldots, a_n)\) is the tuple \((a_n, a_{n-1}, \ldots, a_1)\), with its indexes sorted in descending order. Unfortunately, this class does not offer a way to dynamically create a tuple with the necessary number of elements and their associated types. The number of types has to be known at compile-time, which can not always be guaranteed. Consider the feature \textit{inversed} of class \textit{MML\_RELATION} \([G, H]\): Since a relation is a set of pairs (2-tuples), inversing a relation means inversing the tuples contained in the relation. For the special case of pairs, the implementation of \textit{inversed} for class \textit{MML\_PAIR} \([G, H]\) is easy. However, for general n-tuples, there is no way to implement an \textit{inversed} query because of the missing run-time reflection support. Since most of the functionality requiring tuples can be implemented using pairs, it has been decided to only implement 2-tuples instead of generic n-tuples.

• TUPLE: Another problem related to class \textit{TUPLE} concerns object equality. Tuple objects can only be created as objects using the built-in \textit{reference equality} comparison. If we would want to store manifest integer constants (which are of type \textit{INTEGER\_REF} surprisingly), we get wrong results if we compare the content of the tuple with entities of type \textit{INTEGER}. The reason for this is the default \textit{reference equality} comparison which cannot be changed in manifest tuples. A better policy would be to use \textit{object equality} comparison as default, at least for \textit{basic numeric types}.

• Eiffel Language Specification:

• \textbf{Old Expressions}: A unique and very useful feature of the EIFFEL language are \textit{’old’} expressions, Boolean expressions in \textit{postconditions} which refer to the \textit{pre-state} of the computation, that is the state before the feature call. Unfortunately there is a problem in the language specification. An expression \textit{’old e’} of type \textit{T} is evaluated before the statements of the feature implementation are executed. The result of the evaluation is stored in a reference entity of type \textit{T}. This approach works fine with expanded types like \textit{INTEGER, REAL} and other basic numeric types, but does not yield the desired result with reference types. Recall the special assertion type \textbf{Pure Feature Assertion} (see section 3.3). Usually, these assertions are of the form \textit{is\_equal (old Current)}. Since \textit{Current} is a read-only entity whose reference value never changes, \textit{’old Current’} is a tautology. In this case, the \textit{’old’} expression only compares the two reference values instead of comparing the two object pointed to by the references. It has to be noted that neither \textit{shallow equality} nor \textit{deep equality} would be sufficient. A special \textit{object equality relation} would have to be introduced, which does not consider the object structure itself, but rather the logical object structure. Extended type systems like the \textit{Universe Type System} [MPH99] may provide a way out of this dilemma.

• \textbf{Assignment Attempt}: In EIFFEL, type casts (and in particular \textit{dynamic down casts}) are implemented using a special form of assignment \textit{’?’} called \textit{assignment attempt}. The semantics of this construct are the following: If the reference on the right-hand side is not \textit{’Void’} and conforms to the dynamic type of the left-hand side, the value is assigned to the entity of the left-hand side. Otherwise, the assignment target receives the value \textit{’Void’}. Unfortunately, \textit{assignment attempt} in EIFFEL is an instruction rather than an expression, thus prohibiting its use in assertions. The solution would be to implement \textit{assignment
**CHAPTER 6. THE MATHEMATICAL MODEL LIBRARY (MML)**

*attempt* as an expression or introduce an *‘instanceof’* operator as provided by the JAVA language.

- **Undefining Features:** In the current EIFFEL standard [Mey92] only effected features can be undefined in feature adaptation clauses. However, in our opinion this language rule is too restrictive for one important reason: If a descendant class effects a deferred feature inherited from an ancestor class, and later on the ancestor class decides to provide a general implementation for this feature, all clients are broken. Allowing to undefine deferred features would help in this case and improve seamless development promoted by the EIFFEL method.

- **Ternary Boolean Expressions:** Assertions in EIFFEL frequently use case differentiation in the form of *‘a implies b and not a implies c’*. Depending on the complexity of the expression $a$, assertions can become very complex and verbose. A ternary Boolean expression *‘if c then s else t end’* as provided by JAVA/C like languages, with both $s$ and $t$ being Boolean expressions themselves, would be a nice shortcut to make case differentiation assertions easier to read.

- **Complex Feature Adaptation Clauses:** The feature adaptation clauses of some MML classes are quite complex and become hard to manage. The EIFFEL language should provide higher-level operations on feature adaptation clauses. Valuable operations would be: Prefix all feature names which yield a name clash with a common prefix $p$; Always select the most specialized implementation in case of multiple inheritance; Automatically redefine all features which are reimplemented in the class. The manual management of feature adaptation clauses has proven to be too tedious and error-prone. A feature adaptation language based on regular expressions could help in cases of very complex adaptation clauses.

- **Grammar of Agent Call Targets:** In the current EIFFEL standard [Mey92], the allowed constructs for agent targets are quite limited. The language grammar states that only parenthesized expressions are allowed. Unfortunately, this restriction prohibits feature call chains such as *‘agent a.b.c.d.e’*. Feature implementations are not affected by this restriction, since temporary variables can be declared anytime. In assertions however, this restriction makes it impossible to specify many contracts with the use of agents. Inline agents introduced in the upcoming EIFFEL standard [Mey] probably alleviate this problem.

- **Object Equality Relation:**

  - **ANY:** The equality relation $is\_equal$ induced by class ANY raises interesting problems in the context of mathematical models. On the one hand, the static typing and the type safety of the EIFFEL language has to be preserved in model implementation classes, but on the other hand, covariant redefinition of the equality relation $is\_equal$ (other: like Current) is too strong for many cases. Imagine an object of type $MML\_RELATION [G, H]$ which has to be compared to an object of type $MML\_PAIR [MML\_PAIR [G, H]]$. In case the two objects contain the same pairs with the same elements (compared using object equality), these objects should be considered equal. Covariant redefinition of the argument is therefore too strong. But a signature $is\_equal (ANY)$ is neither desirable. The current implementation of the MML class library therefore does not use anchored declaration for the feature argument, but instead manually redefines the argument type according to the implementation class.
• **TUPLE**: Class *TUPLE* uses *reference equality* comparison as default. In some cases, the default setting is not sufficient and would have to be changed. For further information see category *Eiffel Kernel Classes*. 
6.7 Further Work

The last section of this chapter provides an overview of the future development of the MML class library. In our opinion, the main topics are:

- **High-Level Mathematical Models**: The current version of the MML library only provides models for the most fundamental mathematical notions like *pairs*, *sets*, *relations*, *functions*, *multisets* and *sequences*. *Graphs* are supported by class `MML_GRAPH`. Based on this class, models for *networks* and *trees* are easily implemented. Other more complex mathematical structures can be composed from the existing models.

- **Support for Continuous Functions**: The model interface class `MML_FUNCTION` already supports *continuous functions*. However, the current version of the class library does not provide an implementation for *continuous functions* or for *infinite functions*. It has to be investigated whether these types of functions are indeed necessary, and how easily implementations for these concepts can be created.

- **Introducing Inline Agents**: The upcoming EIFFEL standard [Mey] introduces *inline agents*, the ability to use an *anonymous feature implementation* as agent. Currently, the MML library uses a lot of *wrapper features* to compensate for the missing concept of *inline agents*. Hopefully, the *model implementation classes* can be cleaned up significantly using *inline agents*.

- **Introducing Operators**: The current model interface classes of the MML library do not support *prefix* and *infix operators*. The limited support of the EIFFEL language for *free operator names* restricts the use of operators significantly. In the upcoming EIFFEL standard [Mey], *free operator names* can consist of a larger set of characters, making the selection of operator names easier.
Chapter 7

Specifications of Fundamental Data Structures

In this part of the thesis report we apply the acquired mathematical concepts to the most fundamental data structures of the EIFFELBASE library, using contracts written in the IFL language supported by the MML library.

7.1 Overview

We have treated the necessary mathematical background in chapter 4. Based on these insights and the presentations of the IFL contract language (chapter 5) and the MML class library (chapter 6), we give the proof of concept that a collection of mathematical notions in conjunction with an expressive functional contract language and a powerful class library are indeed sufficient to specify advanced data structures. In section 7.2 we describe our approach to annotate data structures of the EIFFELBASE library with mathematical models. Based on this specification experience, we are listing eventual problems encountered in the specifications of EIFFELBASE classes in section 7.4. Section 7.6 discusses how to further use the annotated EIFFEL classes. In the last section of this chapter (section 7.5) we also shed some light on the benefits of mathematical models for specification of class libraries.

7.2 The Specification Process

The specification process for the EIFFELBASE library should be efficient and finally produce class specifications which are as complete as possible. Since the data structure classes of the EIFFELBASE library usually have a relatively large class interface, we are going to restrict ourselves to the most commonly used abstractions, in order to favor completeness of the specifications. We therefore try to choose a set of classes which is as representative as possible and which shows most of the problems that can occur during specification. Our approach to specify the data structure classes involves the following steps:
Data Structure Specification Process:

1. Determine the most generic common ancestor classes $A$.

2. Introduce model features $f$ for the current class $C$ if necessary.

3. Specify the contracts of all features in the flat form of $C$ using all previously introduced model features $f$.

4. Determine all direct descendants of $C$ which are ancestors of at least one EIFFELBASE data structure class and continue with step (2).

We now provide a more detailed description of each step of the data structure specification process:

Step (1) is used to determine the common ancestors of the class set chosen to be specified with mathematical models. In the EIFFEL type system, computing a common ancestor class is a trivial task: All EIFFEL classes automatically inherit from the predefined class ANY. We therefore start the specification process in class ANY, meaning that the current class is set to class ANY.

Step (2) describes how to introduce new model features designed to represent a new facet or mathematical abstraction implemented by the current class. If the abstraction represented by class $C$ is not yet concrete enough to be captured by mathematical models, we leave all existing contracts and try to complement them with the different contract types identified earlier (section 5.3.2).

Otherwise, if we can identify a mathematical abstraction which can be represented by a mathematical model, we introduce one or more model features of appropriate type. The process of introducing new models cannot be formally described, but has to be rather guided by experience and intuition. More on this topic is listed under the Model Type Rule of section 5.2.1.

In step (3) we perform the actual specification of the current class $C$. To this purpose, we use the flat form $F$ of $C$. The introduction of new model assertions is done incrementally. We consider the feature contracts in $F$. If no model assertion exists already for a particular model feature $f$, we introduce a new one, describing the contract with respect to the particular mathematical abstraction represented by $f$. We use the following kinds of assertions for the EIFFELBASE class specifications:

- **Definition Assertions**: To define the result computed by functions.
- **Pure Feature Assertions**: To express that an EIFFEL feature does not change the state of the current object.
- **Immutable Argument Assertion**: To state that objects passed as actual feature arguments are not changed by an EIFFEL feature.
- **Other Regular Assertions**: Regular EIFFEL assertions describing preconditions and postconditions.
For class invariants, we use the following kinds of assertions:

- **Abstraction Invariants**: Invariants describing *abstraction functions* of mathematical models.
- **Gluing Invariants**: Invariants that relate several *model features* together.
- **Model Invariants**: Invariants specifying *global constraints* on *model features*.
- **Other Regular Invariants**: Regular EIFFEL assertions describing *class invariants*.

Note that some kinds of assertions have to be updated with every new *model features* that is introduced. This is the case with *pure feature assertions*, where *immutability* of the model has to be verified, or with *definition assertions* and various other kinds of *invariant assertions*.

Step (4) finally describes the process of moving upward in the EIFFELBASE class hierarchy. We consider all the *direct descendents* of C and check whether they are *ancestors* of one of the *data structures* that we want to specify. These classes are exactly the candidates to continue with the *specification process*. After having determined the set of valid classes V for another iteration, we start the **Data Structure Specification Process** on all the candidates, but beginning with step (2). The final result of this process is a *class hierarchy* for the *data structure classes* using mathematical models.

In the next section, this process is applied to a representative set of *data structure classes* belonging to the EIFFELBASE class library.

### 7.3 The EiffelBase Class Specifications

We introduce the *data structures* to be specified in this chapter. The framework developed so far, in particular the IFL *contract language* and the mathematical models from the MML *class library*, is applied to *data structures* of the EIFFELBASE class library. We choose the classes to be specified according to the following criteria:

1. **Usage of the Data Structure**: Choose *data structures* from the EIFFELBASE that are widely used in common applications
2. **Diversity of Concepts**: Select classes that represent *abstract concepts* which are as original as possible
3. **Sufficient Documentation**: Choose classes whose abstraction can be derived from the *feature comments* as easily as possible

As it turns out, criterion (3) is the one which is satisfied the least. Most EIFFELBASE classes contain only shortened *feature comments* which range from one to three lines of text. The real shortcoming however are the rudimentary class *header comments* in the ‘indexing’ clauses. All the properties of a class that cannot be expressed in a *class invariant* should be stated in the ‘indexing’ clause. The same also applies to *feature comments*, which are designed to list *preconditions* and *postconditions* that are not stated in a ’*require*’ clause and an ’*ensure*’ clause, respectively.
We therefore have to search for data structure classes which are sufficiently documented and commonly used. Based on the criteria mentioned above, we choose the following classes:

- **BAG**: Deferred class implementing the concept of a mathematical multiset (see section 4.8.2)
- **LINEAR_SUBSET**: Deferred class representing mathematical sets (see section 4.4) which additionally can be traversed in a linear order.
- **DYNAMIC_LIST**: Deferred class representing the concept of a linearly traversable sequence of elements which can be dynamically modified.
- **STACK**: Deferred class representing a last-in first-out dispenser without any concrete implementation.
- **QUEUE**: Deferred class representing a first-in first-out dispenser, also without any concrete implementation.

Figure 7.1: The EIFFELBASE Class Specifications
The specifications for the five classes \textit{BAG}, \textit{LINEAR\_SUBSET}, \textit{DYNAMIC\_LIST}, \textit{STACK} and \textit{QUEUE} are given in appendix C. The complete collection of specifications can be found in the MML code base in the following clusters:

- \textit{specification/hierarchy}: The EIFFELBASE class hierarchy specifications
- \textit{specification/structures}: The EIFFELBASE data structure specifications

7.4 Specification Problems

The specification of the set of chosen data structure classes sometimes was not easy and lead to the discovery of heavy underspecification, contradictions in contracts and flaws in the taxonomy of the EIFFELBASE library. We discuss the encountered problems in two sections. First, we list general problems not involving a particular class in section 7.4.1. Section 7.4.2 lists problems which are directly related to a specific EIFFEL class. If possible, we would also like to give a possible solution for the problem being discussed.

All problems mentioned in the next two sections are referring to the implementation of EIFFELBASE shipped with ISE EIFFELSTUDIO 5.3.

7.4.1 General Problems of EiffelBase

The section of general EIFFELBASE-related problems is divided into several problem categories. Each category describes an aspect of EIFFELBASE which caused problems with respect to specification with mathematical models.

- \textbf{Active Data Structures}:

- \textbf{Iteration Facilities}: Internal cursors of active data structures and functionals such as \textit{for\_all}, \textit{there\_exists} and \textit{do\_all} do not represent the same concept and should be strictly distinguished. Class \textit{TRAVERSABLE} introduces both cursor features \textit{start} and \textit{after} as well as support for quantifiers \textit{for\_all}, \textit{do\_all}, and so on. However, functionals like \textit{do\_all} are already applicable in class \textit{CONTAINER}. The linearity of the \textit{TRAVERSABLE} class is not necessary for an implementation of logic quantifiers.

This problem manifests itself in class \textit{TRAVERSABLE\_SUBSET}. Besides that the class does not inherit from \textit{TRAVERSABLE} (despite its apparent similarity), it represents a wrong abstraction. We cannot define a current iteration element of a set, since sets to not have a defined order on their elements by definition. Hence features implementing operations on a cursor do not make much sense in such classes. However, the functional \textit{do\_all} and the quantifiers \textit{for\_all} and \textit{there\_exists} are well defined on sets and should therefore belong to the class interface of \textit{TRAVERSABLE\_SUBSET}.

One solution to this problem are facets as presented in section 6.2.1.
• **Equality Relation:** The equality relation of active data structures involves not only the elements of the structure, but also a cursor position and other internal data. Unfortunately, it is nowhere specified whether two active data structures with the same elements but different cursor positions are considered equal. We have decided to neglect cursor positions and to only compare the elements of the structures.

• **Polluted Code:** Generally, the EIFFEL BASE classes representing active data structures contain a lot of code related to structure traversal (internal iterators). It can be argued whether a cleaner separation of data structure- and iteration code is desired. In our view, the use of external iterators could bring many advantages from the developer’s point of view. External iterators are concurrent, group iteration-related code together and can be part of an independent class hierarchy of iterators. The other less radical solution is to follow more the interface/implementation principle and to use facets (see sections 6.2.1 and 6.2.2).

• **Composite Features:** Many high-level operations like the feature merge_left of class DYNAMIC_CHAIN internally use a cursor to search and update elements in the structure. This is very convenient from an implementation viewpoint, but dangerous from an operational point of view. Although the EIFFEL BASE class library has not been designed for concurrent use, feature occurrences of class LINEAR demonstrates why this approach is sometimes not the best one: The implementation of occurrences is not pure because of the calls to feature search. This feature is inherently impure, thanks to the Command/Query Separation Principle [Mey97]. The common understanding of the semantics of feature occurrences however suggests that it should have a pure implementation.

• **Class Hierarchy:**

  • **Abstraction of Classes:** The EIFFEL BASE class library contains many classes which do not introduce features for themselves, but instead inherit from a number of distinct classes representing totally different concepts. The problems with this approach once more lie in the the contracts: Not all features of the original concept also make sense in the derived concept. Let use show this problem with an example: class INTEGER_INTERVAL inherits from SET to express the collection-character of an interval. This inheritance relationship seems unnatural for two reasons: First, intervals have a total order and could therefore better be modeled by sequences. Second, even though class SET is deferred, it already contains to much functionality such as element removal which is not applicable here. A clear distinction of immutable and mutable structures could help in this case.

  • **Feature Introduction vs. Facet Accumulation:** This is a rather general problem and involves two common design strategies: On the one hand an inheritance hierarchy of implementation classes which introduce new features as soon as the become necessary, and on the other hand an inheritance hierarchy of classes which implement a set of interface classes while choosing partial implementations from a set of other implementation classes. A good example to illustrate these two different methods is the deferred class BOX: It only introduces a new feature full, but inherits from class CONTAINER as well. Clients which want to use the full aspect automatically inherit a container behavior, which is not necessarily desired. Class BOX would be a lot more useful if it did not inherit from CONTAINER, and clients of BOX would inherit from the corresponding facet classes modeling the desired container behavior.
7.4. SPECIFICATION PROBLEMS

- **Linearity/Bi-linearity**: Are indeed both concepts needed in the EIFFELBASE? Class \textit{BILINEAR}, which inherits twice from \textit{LINEAR}, is not easy to specify, since it is not clear which \textit{iteration features} are contributed by which \textit{inheritance relation} with \textit{LINEAR}. Using a more general implementation using appropriate status queries such as \textit{forward_traversable}, \textit{backward_traversable} and adapted \textit{preconditions}, \textit{BILINEAR} could also be used as a replacement for \textit{LINEAR}.

- **Read Only Operations**: Related to the problem of class \textit{INTEGER_INTERVAL} with its \textit{SET} implementation, it has to be investigated to what extent classes representing \textit{immutable structures} could be introduced in the EIFFELBASE library. Immutable sets would solve the above-mentioned problem. Generally, an implementation for \textit{mutable structures} should inherit from an \textit{immutable structure}. The EIFFELBASE class \textit{COLLECTION} demonstrates this dilemma quite well: It provides too many operations for element change which are not used anymore in descendant classes modeling \textit{immutable structures} like \textit{TUPLE} or \textit{INTEGER_INTERVAL}.

- **Specification Style**:
  - **Void References**: The designers of the EIFFELBASE class library use a tolerant style for preconditions: References to 'Void' are implicitly allowed, but the MML library cannot handle void references. There is no concept of references in mathematics, but only object identities. The goal of the MML library is to model mathematics as closely as possible, which prohibits void references as elements in data structures. Nevertheless references to 'Void' can be modeled by a dummy void object without any loss of generality.
  - **Indexing of Arrays**: The mapping of model indexes to structure indexes is not explicit. In order to be as general as possible, we would have to define a mapping function which takes care of index conversion. Since the EIFFELBASE library uses the canonical non-zero based indexing, we use this indexing style in the MML class library as well.

- **Feature Naming**:
  - **Uniform Feature Naming**: The EIFFEL style guidelines suggest uniform naming of features which perform similar or analogous operations. However, the plan of emphasizing commonality is sometimes pursued too careless. Different concepts like the current iteration element of a data structure (class \textit{TRAVERSABLE}) and the result of an agent function call (class \textit{FUNCTION}) both use the feature name \textit{item}. In this case, the emphasis on commonality is not justified in our opinion and yields a class interface which is unnecessarily hard to understand.

- **Higher-Order Functions**:
  - **Linearity and Functionals/Quantifiers**: Class \textit{LINEAR} introduces the higher-order functions \textit{for_all}, \textit{there_exists}, \textit{do_all}. These functionals have nothing to do with the concept of linearity and should be moved up to the level of collections (class \textit{COLLECTION}). See also category \textbf{Active Data Structures}. 


• Pure Objects:
  
  • Pure Methods: The EIFFELBase library strictly follows object-oriented design principles. Most feature calls are of the form \( x.f \) where \( x \) is the current object (the target of the feature call) and \( f \) is the name of the feature to call. These features are commands which apply the operation to the target, but do not return a new object. The problem with this approach is that all these features are not pure. However, in mathematics there exists no concept of mutable objects. Additionally, feature calls involving commands cannot be used in expressions, thus prohibiting the use of these features in assertions. A good example for this discrepancy is class \( \text{SET} \) with its features \( \text{intersect, subtract, merge} \).
  
  • Specification of Axioms: Since the EIFFELBase library does not adhere to mathematical standards (there are no functions which return the changed object, but leave the target of the feature call intact), it is not possible to state important axioms without resorting to cloning or creation expressions. Consider the class invariant from an imaginary class \( \text{RELATION} [G, H] \):

\[
\text{inversed.inversed.is_equal} \ ((\text{Current})) \tag{7.1}
\]

Most classes of the EIFFELBase do not allow to express axioms like the above one, since they only offer impure features which would change the state of the current object during the evaluation of assertions. The solution is to resort to cloning (calling the feature on the cloned object) or creation expressions (calling the feature on an anonymously created object), both of which lacks simplicity and clarity.

• Reusability of Contracts:
  
  • Sequence Model Contracts: During the specification of the data structures, it has become apparent that too many sequence models were used. As mentioned in section 7.5, this could be the sign of a design problem in the class hierarchy of EIFFELBase. Either the abstraction of a sequence has been overused, or the designers failed to extract a sufficiently general concept of an immutable sequence.

7.4.2 Class-Specific Problems of EiffelBase

The section for class-specific problems of the EIFFELBase library lists the problems in the order of the class they appear. Each problem is associated with one or more features, or marked as a conceptual problem. We explain the problem in short and try to give a possible solution if possible, similar to section 7.4.1. The following list of problems is certainly not complete, but gives an overview of the most important issues and the diversity of problems encountered during specification.

• ARRAY
  
  • prunable, extendible: These queries always return \( \text{False} \), since the size of an array cannot be changed. In the descendant class \( \text{ARRAYED_LIST} \), they always return \( \text{True} \), because arrayed lists automatically grow upon inserting new elements. This can lead to somehow confusing contracts such as the ones in features \( \text{extend} \) and \( \text{prune} \).
7.4. SPECIFICATION PROBLEMS

- **wipe_out**: Obsolete clauses are certainly not the right way to declare a feature as inapplicable. Since the number of elements in an array cannot be changed by definition, class `ARRAY` should not inherit from `COLLECTION`, or `COLLECTION` should factor out the operations applicable to immutable collections to a separate class, and let `ARRAY` inherit from it. Moreover, the implementation of `wipe_out` calls `discard_items`, which has a contract with completely different semantics.

- **BAG**

  - **Conceptual Problem**: Class `BAG` is located too high in the class hierarchy of EIFFEL-BASE. All implementation classes modeling the concept of sets such as `ARRAYED_SET` also inherit from class `BAG`, thus also inheriting feature `occurrences`. Although set implementations of occurrences always return 1, sets usually do not offer such features. It seems that class `BAG` is a too general class, not representing the mathematical counterpart of a multiset, but rather the concept modeled by the Extendible Facet (see section 6.2.1).

- **CHAIN**

  - **index_of**: Class `CHAIN` redefines this feature and implements a wrapper which remembers the current cursor position before a call to the impure implementation provided by feature `sequential_index_of`, and restores it afterwards. It is not clear why the ancestor version of `index_of` is not a pure feature in the first place.

  - **index_of, sequential_index_of**: These features are heavily underspecified: It is not apparent what happens if the element whose index has to be retrieved is not in the chain. Only the feature comments provide further help, but in a rather minimalistic way.

  - **remove**: This feature needs a precondition which states that the cursor should be at a valid position. Unfortunately, that cannot be covered by the existing feature `prunable`. For operations like `remove`, one more general feature similar to `extendible` and `prunable` is required to express assertions related to cursor positions.

  - **start, finish, forth, back**: These features do not specify their effect on an empty chain.

  - **Conceptual Problem**: Class `CHAIN` inherits twice from `SEQUENCE`. Class `SEQUENCE` itself inherits from `BILINEAR`, which inherits twice from `LINEAR`. It is not clear why a chain should implicitly inherit four times from class `LINEAR`. It neither seems justified to introduce an inheritance relation to reuse the partial implementations of `LINEAR` and `BILINEAR`, respectively. Due to the duplication of features in class `CHAIN`, many features have to be renamed and hidden, leading to potential CAT calls [Mey97].

  - **Conceptual Problem**: Since class `CHAIN` inherits twice from `SEQUENCE`, it also inherits two sequence models. At first sight, it seems like these two features have to be kept distinct. However, from a physical point of view, they represent the same mathematical sequence (the data structure represented by class `CHAIN`) and can therefore be merged. From a mathematical point of view, this is an argument not to inherit more than once from `SEQUENCE`, or more general, from a class providing the same model feature.
COLLECTION

- fill: This feature has been proven very problematic to specify in all its occurrences in the class hierarchy of the EIFFELBASE library. The feature comment states that the implementation "[fills] the current object with as many elements from the other structure as possible". The only precondition of fill expresses that extendible must be 'True'. Vague specifications as this one make it nearly impossible to apply mathematical model specifications.

CURSOR, CURSOR_STRUCTURE

- Conceptual Problem: The intent of classes CURSOR and CURSOR_STRUCTURE is not obvious. The EIFFELBASE class library uses active data structures with internal cursors. It is not clear why class CURSOR and a whole class hierarchy of cursor classes have been introduced in the first place.

DYNAMIC_CHAIN

- put_front, put_left, put_right and others: The feature comment of these features states: "Do not move cursor". There are at least two ways of interpreting this sentence. First, not moving the cursor could mean that the cursor stays on the same element after the feature call as before. Second, it could denote that the cursor position itself is not changed. In our opinion, the first interpretation makes more sense. Additionally, it is not clear what happens when the cursor is at an undefined position. The tolerant contract style of EIFFELBASE complicates a rigorous specification of such features with mathematical models.

DYNAMIC_LIST

- Conceptual Problem: Class DYNAMIC_LIST inherits twice from DYNAMIC_CHAIN. Like class CHAIN, it seems not justified to inherit from DYNAMIC_CHAIN just to reuse its (partial) implementation. Obviously, mismatches in the contracts prohibited the use of simple redefinition and the 'Precursor' keyword, which is a sign of an unjustified inheritance relation.

INTEGER_INTERVAL

- capacity: This feature has hidden assertion checks which are not implementation-related. The 'check' clause contains assertions requiring defined lower and upper bounds. In fact, these assertions are hidden contracts, which should be rather declared in a precondition.

- extend, put: These features are inherited from class INDEXABLE and renamed. Class INTEGER_INTERVAL then introduces new features extend and put with implementations to extend the range of the interval. All feature contracts are also inherited, but mean something completely different in this context: The contracts of INTEGER_INTERVAL state that the bounds of the interval have to be set, whereas the contracts of INDEXABLE express that a key-value pair is inserted into a container. Moreover, the contracts also contradict those inherited from class SET.
• **wipe_out**: The inherited postcondition of `wipe_out` states that `is_empty` should be `True`, but the interval bounds yield a capacity of two: the attribute `lower_internal` is set to 0 and `upper_internal` to 1, therefore the capacity is `upper_internal − lower_internal + 1`. This contradicts with the definition assertion of feature `is_empty`.

• **prune, indexable_put**: These features have been commented as inapplicable, their contracts however state a change of the models of this class. Although the implementation of these features is empty, this could potentially lead to contradictions in the contracts.

• **Conceptual Problem**: The notion of logic quantifiers is reimplemented in class `INTEGER_INTERVAL`. This class defines its own features `for_all` and `exists` to implement universal and existential quantifiers. Actually, class `SET` should have introduced this concept already, or class `INTEGER_INTERVAL` could implement the **Quantifiable Facet** (see section 6.2.1).

• **Conceptual Problem**: Why should an integer interval of class `INTEGER_INTERVAL` be indexable by integers itself? Intervals are completely defined by bounds, and checks whether an integer is in the interval are simple range checks on its bounds. Therefore the implementation as an indexable data structure is hardly justified.

• **LINEAR**

  • **has**: The implementation of this feature uses `start` which is not pure, and does not revert the cursor position. If the implementation of `has` needs to move the internal cursor, it should at least remember the cursor position and restore it later.

  • **occurrences**: The same problem as in feature `has` also arises in feature `occurrences`: The implementation uses the internal cursor and calls feature `search`, which is impure by definition (Command/Query Separation Principle [Mey97]).

  • **Conceptional Problem**: Class `LINEAR` introduces the concept of a linear sequence with a cursor position after and the query off. As already discussed in category **Class Hierarchy** of section 7.4.1, it can be argued whether the two concepts linearity/bi-linearity could be merged. In our opinion, the more general concept of bilinear sequences is preferable. Two additional queries to determine whether the cursor can be moved forward and backward have to be introduced. The preconditions of the corresponding features would call these queries if the cursor has to be moved.

• **LINEAR_SUBSET**

  • **extend**: The precondition of this feature simply states that the element is a member of the set after the feature call. However, nothing is said about where the element is inserted if not already in the structure. Since `LINEAR_SUBSET` has a defined order on the set members, this is an important point which has to be stated in the feature contracts.

  • **merge**: What happens to the sequence model of the current object when merging two sets? In this case one can only guess, and is therefore not able to give a rigorous specification.

  • **wipe_out**: A similar problem occurs in feature `wipe_out`. It is not clear what happens to the cursor after a call to `wipe_out`. As it turned out, some implementations of `LINEAR_SUBSET` move the cursor to after and some move it to before.
CHAPTER 7. SPECIFICATIONS OF FUNDAMENTAL DATA STRUCTURES

- **Conceptual Problem:** Class **LINEAR_SUBSET** represents an ambiguous concept. Mathematical sets do not have a *partial order* by definition. It can be doubted whether class **LINEAR_SUBSET** represents a valid concept. Indeed, as it turns out in descendant classes, the fact that **LINEAR_SUBSET** always has a current *iteration element* is not compatible with the notion of a set. Moreover, an iteration over a linear set which changes the set itself has undefined results. The result of these operations is not dependent from the order of the elements, but from the non-determinism in the data structure. Such a dependency is clearly not desired.

- **Conceptual Problem:** Class **LINEAR_SUBSET** provides the feature *before* and all other features of class **BILINEAR**, but does not inherit from it. This problem actually raises already in class **TRAVERSABLE_SUBSET**. See also **TRAVERSABLE_SUBSET**.

- **LIST**

  - **Conceptual Problem:** Class **LIST** contains only feature implementations which could have been introduced already in **CHAIN**. In order to simplify the class hierarchy of the **EIFFELBASE** library, this class could be merged with **CHAIN**.

- **QUEUE**

  - *extend, put, force:* These features have been declared as *synonyms* in class **QUEUE**. However, *ancestor versions of extend* do not move the cursor position, whereas every *extend* in a queue sets the new tail element (and therefore changes also the cursor position). This could lead to contractions in *model contracts*.

  - **Conceptual Problem:** The concept of an queue contains no current element, but only a *head* and a *tail element*. What effect do the implementations of *item, prune, remove, replace* and others, have on the current *iteration element*? Once more, we have to make an assumption that the current *iteration element* is equal to the *head element*, the element that is going to be dequeued next. Therefore *remove* and *replace* operate on the *head element* and *force* sets the new *tail element*.

- **SEQUENCE**

  - *prune, prune_all:* It is not clear why these features should move the cursor to *off* in case the element to be pruned is not found. These kind of side-effects are dangerous and unnecessary, since the occurrence of an element can be verified before. Feature *prune* should leave the cursor unchanged in this case.

- **TABLE**

  - *bag_put:* Feature *put* of class **BAG** is not applicable to tables. It is renamed and hidden in order to not occupy the feature name for the feature *put* of class **TABLE**, which is used to associate a value with a key. Once more, feature hiding rises the question whether the *inheritance relation* between these two classes is indeed justified.

  - *extend:* Class **TABLE** inherits the feature *extend* of class **BAG**. This feature is not hidden and therefore part of the class interface. It is not clear which *key* the *value* passed as *actual feature argument* is associated with. Unfortunately, the feature comments do not provide further information.
7.4. SPECIFICATION PROBLEMS

- **put**: The feature `put (value: G; key: K)` is not specified at all. A possible postcondition would be `item (key).is_equal (value).

- **prune**: Index structures like `TABLE` usually contain `values` which occur multiple times in the structure. However, feature `prune` does not specify which `value` is removed from the structure, or whether all `values` associated with any `key` are removed.

- **CONCEPTUAL PROBLEM**: We already mentioned this problem regarding the `inheritance relation` between `INTEGER_INTERVAL` and `INDEXABLE`, a descendant class of `TABLE`. In addition to the problem stated there, class `TABLE` is sometimes used in places where the `keys` in the table have to be occurring in strictly ascending order, like in sequences. Class `TABLE` seems to be the wrong abstraction in these cases.

- **Conceptual Problem**: The `keys` of a table are stored as a `set`, since they have to be unique within the structure. Class `TABLE` however inherits from `BAG`. It is not clear whether the `multiset-character` of class `TABLE` should refer to the values in the structure. A more natural approach would be to express the `set-character` of the keys in the table.

- **TRAVERSABLE, TRAVERSABLE_SUBSET**

  - **is_empty**: This feature is still deferred, but the class invariant already demonstrates a correct implementation: `Result = (count = 0)`. Therefore, the feature could be effected earlier in the inheritance chain, e.g. already in class `SET`, which provides the necessary feature `count`.

  - **CONCEPTUAL PROBLEM**: Class `TRAVERSABLE_SUBSET` does not inherit from class `TRAVERSABLE`, even though class `TRAVERSABLE_SUBSET` implements all features offered by `TRAVERSABLE`. This design decision prohibits any polymorphic use of instances of class `TRAVERSABLE` and yields to extensive code duplication. The reason for this decision is apparent. `TRAVERSABLE` is a data structure class and not a facet class. Modeling class hierarchies more facet-oriented as the MML class library does helps to avoid these kind of problems.

  - **CONCEPTUAL PROBLEM**: Since class `TRAVERSABLE_SUBSET` does not inherit from class `TRAVERSABLE`, all iteration-related features and quantifiers have to be reimplemented. Surprisingly, `TRAVERSABLE_SUBSET` provides no quantifiers and only a subset of the iteration features introduced in `TRAVERSABLE`. In our view, the last two conceptual problems arise because of the lack of agents at the time of the design of the `EIFFELBASE`, which could have been used in features like `do_all`.

  - **CONCEPTUAL PROBLEM**: All descendants from `TRAVERSABLE_SUBSET` suffer from the mismatch of linearity and mathematical sets. The descendant class `LINEAR_SUBSET` even forces a total order on the clients of this class. `put_left` and `move_item` are operations which cannot be defined on sets. The situation is even aggravated in descendants like `LINKED_SET` or `ARRAYED_SET`, which introduce several other features involving cursor movements or are dependent on a total order.
CHAPTER 7. SPECIFICATIONS OF FUNDAMENTAL DATA STRUCTURES

7.5 Benefits of Mathematical Models

We have shown a relatively extensive list of problems which turned to surface during the specification (see section 7.2) of the EIFFELBASE classes. Based on the experience gained during the specification phase and its occurring problems, we try to compile a list of the benefits that mathematical models could bring if applied to data structure classes on a regular basis:

- **Partial State Change Specification**: Ability to specify which parts of the state space change

  Mathematical models ease the difficulty of specifying that certain aspects of an object are changed by a feature call. Let us take an example from the EIFFELBASE class DYNAMIC_CHAIN to illustrate this point:

  **Example**: Sequence Element Extension

  ```eiffel
  deferred class interface
  DYNAMIC_CHAIN_EXAMPLE [G]
  feature -- Element change
    put_front (other: like item)
      -- Add 'other' at beginning.
      -- Do not move cursor.
    require
      other /= void
    ensure
      sequence.is_equal (old sequence.prepended (other))
      (old cursor >= sequence.lower_bound) implies cursor.is_equal (old cursor + 1)
      sequence.first.is_equal (other)
      (old cursor < sequence.lower_bound) implies cursor.is_equal (old cursor)
      other.is_equal (old other)
      bags.is_equal (old bag.extended (other))
      indexable.cardinality.is_equal (old indexable.cardinality + 1)
      (old not sequence.is_defined (cursor)) implies not sequence.is_defined (cursor)
  end -- class DYNAMIC_CHAIN_EXAMPLE
  ```

  In this example the state space representing the sequence with its elements is changed, whereas the cursor state remains the same under this operation.

- **Definition Assertions**: Assertions to specify computation results in postconditions

  Mathematical models facilitate the specification of features that perform complex tasks. The following example again is an extract from the EIFFELBASE library, showing a query that determines whether another set is disjoint with the current one:
Example: Set Disjointness Property

defered class interface

SUBSET_EXAMPLE [G]

feature — Disjointess

disjoint (other: SUBSET_EXAMPLE [G]): BOOLEAN

—- Do current set and 'other' have no items in common?

require

other /= void

object_comparison.is_equal (other.object_comparison)

ensure

Result.is_equal (set.is_disjoint (other.set))

Result.is_equal (set.intersected (other.set).is_empty)

(set.is_empty or other.set.is_empty) implies Result.is_equal (old Current)

other.is_equal (old other)

set.is_equal (old set)

end —- class SUBSET_EXAMPLE

• Specification of Abstract Axioms: Feasibility to state axioms as class invariants

Mathematical models provide a high-level view of specification that favors short and expressive contracts. Expressing strong class invariants using conventional EIFFEL facilities is sometimes impossible since class interfaces normally are not designed to be side-effect free. However, model class libraries like the MML library which are particularly designed for specification using mathematical models guarantee side-effect freeness, making the specification of axioms a lot easier.

Example: Complete Graph

Consider a class COMPLETE_GRAPH with two model features nodes and edges representing the set of nodes belonging to this graph and its associated edge set expressed as a relation over the node set, respectively. The obvious class invariant for COMPLETE_GRAPH can be expressed in one line:

\[
\text{nodes.cartesian_product}(\text{nodes}).\text{is_superset}(\text{edges})
\]

The class in the EIFFELBASE library which represents the notion of sets (class SUBSET) unfortunately contains few side-effect free operations. This undesirable property makes many EIFFELBASE classes difficult, if not impossible, to specify without mathematical models.

• Specification of Composite Operations: Feasibility to easily specify composite operations using shared assertions

Model assertions using class libraries like the MML library usually are quite verbose due to the feature naming recommendations of the EIFFEL style guidelines. Moreover, complex operations may also need many case differentiations. Luckily, common subexpressions in model contracts can be factored out and help to keep model contracts as short as possible. Boolean expressions that are used at several places in the specification are simply transformed into a corresponding predicate. Thanks to the side-effect freeness of the IFL language, the approach of assertion outsourcing is perfectly valid.
• **Design of Class Hierarchies**: Ability to check whether a class hierarchy models the concepts of a problem domain without redundancy.

Mathematical models not only provide a good basis for the specification of EIFFEL classes, but also have other advantages. They can help to validate a class hierarchy with respect to a minimal number of classes and a non-redundant modeling of the concepts in the problem domain. The fact that several model features of the same type exist in a class interface can be a hint that too many classes are modeling the same concept or a similar one. Additionally, inheritance relations can be verified by taking the flattened contract form of the ancestor class. The model contracts of a class which wants to inherit from this ancestor must comply with the flattened contract form of the ancestor (yielding no contradictions in its contracts).

### 7.6 Usage of Model Specifications

After the last two sections on EIFFELBase-related problems (section 7.4) and a discussion about the benefits of mathematical models (section 7.5), we would like to clarify what class specifications using mathematical models can be used for. Three basic use cases for annotated EIFFEL classes are presented below:

- **Runtime Assertion Checking**: Create proxy classes implementing the specification class and extending the EIFFELBase implementation class.

  There are two possibilities to use model-specified EIFFEL classes for runtime assertion checking. First, regular EIFFEL classes can directly be annotated with mathematical models using effected model features. The disadvantage of this option is that we do not consider the flat form of the class, thus potentially missing important inherited features during the specification process. But on the other hand, we do not have to create any additional classes. Second, we can generate the contract form of the class and create a deferred interface class. This class is the basis for our specification process described in section 7.2. After having specified the specification class, we create a proxy class class which inherits both from the EIFFELBASE implementation class and the created specification class. The advantage of this approach is a very flexible assertion checking mechanism. Depending on whether assertion checking is desired, simply inherit from the specification class.

- **Class Interface Documentation**: Use model-annotated classes for annotation and documentation purposes.

  Mathematical model not only provide a solid specification of EIFFEL classes, but also serve to document a class in an efficient way. The clarity and expressiveness of mathematical models surpass feature comments easily. The EIFFEL method uses very few documentation anyway (usually about two or three lines per feature). The advantage of documentation by mathematical models is that the promise of always up-to-date and complete documentation by Design by Contract™ [Mey97] is kept, which is demonstrated by the following example:
Example: Source Code Documentation for CHAIN Class:

defered class interface
CHAIN_EXAMPLE [G]

feature — Element change
merge_left (other: like Current)
— Merge ‘other’ into current structure before cursor
— position. Do not move cursor. Empty ‘other’.
require
cursor >= sequence.lower_bound
other /= void
not is_equal (other)
ensure
(old cursor.is_equal (sequence.lower_bound)) implies
sequence.is_equal (old other.sequence.concatenated (sequence))
(old cursor > sequence.upper_bound) implies sequence.is_equal (old sequence.concatenated (other.sequence))
(old cursor > sequence.lower_bound and cursor <= sequence.upper_bound) implies
sequence.is_equal (old sequence.interval (sequence.lower_bound, cursor — 1),
collapsed (other.sequence.concatenated (sequence.interval (cursor, sequence.upper_bound))))
cursor.is_equal (old cursor + 1)
bags.is_equal (old bags.added (other.bag))
indexable.cardinality.is_equal (old indexable.cardinality + other.indexable.cardinality)
other.sequence.is_empty and other.bags.is_empty and other.indexable.is_empty
end — class CHAIN_EXAMPLE

• Contract Correctness Proofs: Prove that the regular assertions of a contract are implied by its model assertions.

Specification classes are the basis for contract correctness proofs. The following chapter 8 describes the proof process in detail.

7.7 Conclusion

This chapter has shown that specification of class libraries using mathematical models is possible and can bring substantial benefits to both developers and clients. In section 7.2 we have presented our approach to specify EIFFELBASE classes. Section 7.3 describes the process of choosing the most suitable classes for specification. We have identified the following criteria:

• Usage of the data structure in common applications
• Diversity of concepts represented by the classes
• Sufficient documentation available to developers
The following classes have been selected for specification using mathematical models:

- **BAG** (see appendix C.1.2)
- **LINEAR_SUBSET** (see appendix C.1.1)
- **DYNAMIC_LIST** (see appendix C.1.3)
- **STACK** (see appendix C.1.4)
- **QUEUE** (see appendix C.1.5)

In section 7.4 we have listed the problems encountered during the specification of these classes. Section 7.4.1 covers general problems of the EIFFELBASE class library, whereas class-specific problems are listed in section 7.4.2. The most important benefits of using mathematical models are listed in section 7.5:

- Partial State Change Specification: The ability to specify partial changes of the state space of a software system.
- Definition Assertions: The ability to give precise and expressive definitions of feature implementations in its postcondition.
- Specification of Abstract Axioms: The ability to specify class invariants.
- Specification of Composite Operations: The ability to factor out shared assertions.
- Design of Class Hierarchy: The ability to check whether a class hierarchy models the concepts of the problem domain accordingly.

In the section 7.6 we have presented the most important uses cases of annotated EIFFEL classes:

- Runtime Assertion Checking
- Class Interface Documentation
- Contract Correctness Proofs

Finally, based on the experience of the specification phase and the problems that turned to surface, we would like to mention a few other issues which we have become aware of.

As listed in section 7.4.1 and section 7.4.2, there are quite a few mismatches in the inheritance hierarchy of the EIFFELBASE library. During specification, this kind of problem manifests itself in contradictions of contracts. Regular assertions or model assertions which appear in the flattened contract of a feature can contain contradictions, meaning the contract can never be satisfied by any client or feature implementation. Common sources of these problems are misunderstandings in state changes or indexing problems in array-like data structure (see section 7.4.2).

Another characteristic of the EIFFEL language are feature name synonyms, the ability to associate more than one name with a feature implementation. Despite its usefulness, synonymous feature names can bring a lot of trouble when combined with inheritance. Consider a feature \( f \) in a class \( A \) and a
feature $g$ in a class $B$. Assuming that $f$ and $g$ possess contracts with compatible semantics, we can redefine both features in a descendant class $C$ of $A$ and $B$, using $f$ and $g$ as synonyms of the redefined implementation. Different semantics are too easily mixed by using synonyms, in particular if the features are poorly specified with regular assertions. In conjunction with language processing tools such as the contract flattener, which are able to extract the flattened contract form of a model class, mathematical models can help to discover this type of problem. A good example for this policy is the class $STACK$ with the synonyms $put$, $extend$ and $force$.

A related problem of the Eiffel method is the concept of feature hiding, the ability to make the export status more restricted in descendant classes. Many classes in the EiffelBase library (e.g. $TUPLE$) make heavy use of feature hiding. The problem of this concept is that it breaks polymorphism. The rule that every feature applicable to an ancestor base type is also applicable in the base type of descendant classes is not valid anymore, leading to possible run-time errors. The specification of Eiffel classes using mathematical models prohibits most such export restrictions. Usually, features inherited from an implementation class are hidden in order to not pollute the class interface of the client class. The class $INTEGER_INTERVAL$ demonstrates this quite well: The inherited feature $put$ is certainly not applicable anymore to the abstraction of a modifiable interval of integers. The question therefore raises whether the selected inheritance relation (class $SET$) is indeed justified. This kind of problems is flagged by the contract flattener (see chapter 9), which indicates a potential CAT call [Mey97] in these cases.

The last remark we would like to make is the growing importance of reusability. The EiffelBase class library provides a class $TRAVERSABLE$ which represents the notion of a structure whose elements can be traversed in a linear order, and each element is visited exactly once. Similarly, class $TRAVERSABLE_SUBSET$ represents sets that can be traversed according to the description given before. However, it is not clear why $TRAVERSABLE_SUBSET$ does not inherit from $TRAVERSABLE$, but instead implements all the necessary functionality itself. It has to be said that $TRAVERSABLE_SUBSET$ is a deferred class, so descendant classes can provide an implementation via inheritance. Although this solves the code redundancy problem, the reusability of specifications (and its associated proof obligations) is not guaranteed. It seems that class libraries should also be designed with reusability of specification in mind. The interface/implementation separation principle (section 6.2.2) which has been applied to the MML library fully supports the reuse of contracts and proof obligations.
Chapter 8

Proving Contract Correctness

In this chapter we are investigating how to prove contract correctness of regular assertions as an application of mathematical models.

8.1 Overview

This chapter consists of three sections. Section 8.2 introduces a process called contract flattening which is used to extract a class interface consisting of model contracts only. This concept will be used in section 8.3 where we describe how to obtain proof obligations for correctness proofs of regular assertions. In section 8.4 we are discussing when a specification can be considered complete.

8.2 The Contract Flattening Algorithm

To be able to formally introduce proof obligations for regular assertions in the next section, we need to introduce the concept of contract flattening. Contract flattening is a process which eliminates regular assertions and transforms them into equivalent model assertions. The final state of the contract flattening process is a class interface containing model assertions only. The algorithm for contract flattening of a class C consists of the following steps:

**Contract Flattening Algorithm:**

1. Create the contract form of C.
2. Replace every occurrence of a function call expression E involving a non-model feature f by the contract of f.
3. Repeat step (2) until no such expressions E exist anymore in the contract form of C.

We provide now a detailed description of each of the steps of the contract flattening algorithm.

Step (1) creates the contract form of the class C. The contract form is a flat form of a class with no associated implementation. It is important to realize that contract flattening is only defined on the set of preconditions, postconditions and invariants of a class.
In order to work properly, we have to take a flat view of a class interface. The contract form is then easily created by replacing all feature implementations by the ‘\textit{deferred}’ keyword and make class \textit{C} itself deferred.

Step (2) is a little bit more complicated. We first have to give a definition of non-model features. These features are exactly the functions of the flat form of a class which have not been designed to represent mathematical models according to the definitions of section 3.2 and section 5.2.1. We do not have to consider attributes (they cannot have contracts in the current EIFFEL standard) and procedures (since they do not occur in expressions). Let us now consider a contract \textit{c} with an assertion containing a function call expression \textit{e} on an arbitrary target \textit{t} having \textit{f} as feature name referring to a non-model feature with the (possibly trivial) set of preconditions \textit{P} and postconditions \textit{Q}. We can now distinguish the following cases for every assertion \textit{A} in the postconditions \textit{Q} of \textit{f}:

- \textit{A} is a definition assertion: The call expression \textit{e} is substituted by the actual feature argument of \textit{A} (see section 5.3.2).

- \textit{A} is not a definition assertion: In this case, the assertion \textit{A} is ’and-ed’ with the \textit{c}.

For the preconditions \textit{P}, the situation is similar: Since there are no definition assertions in preconditions, all assertions \textit{A} are ’and-ed’ with the existing contract \textit{c} as well. Note that since the contracts of an EIFFEL class normally are not written in the IFL language, we also have to account for definition assertions with the reference equality operator ‘\textit{=}’.

Step (3) starts the recursive part of the contract flattening algorithm. The replacing of call expressions is repeated until there exists no contract anymore such that step (2) can be applied. The final result of this process is the flattened contract form containing model contracts only. The set of regular assertions that have been flatted in conjunction with the generated assertions during the contract flattening process is called flattened assertions.

The flattened contract form of an EIFFEL class will be used in the next section presenting the construction of proof obligations for contract correctness proofs.

### 8.3 Proof Obligations for Contracts

An important question for EIFFEL classes and the correctness of its contracts is the following: Do the existing regular assertions comply with a class specification using mathematical models? In this chapter we would like to investigate this problem.

The necessary and sufficient condition for contract correctness is that for every precondition, postcondition and invariant the set of model assertions is at least as strong as the set of regular assertions. In other words, the set of model assertions should imply the set of regular assertions. We formalize this in the following definition for a contract correctness proof obligation:
8.4. SPECIFICATION COMPLETENESS

- **Contract Correctness Proof Obligation:**
  
  The proof obligation for a contract $c$ with regular assertions $R$ and model assertions $M$ is the implication $A \Rightarrow C$ with:
  
  - The antecedent $A$ being the conjunction of all assertions in $M$.
  - The consequent $C$ being the flattened assertions $F$ obtained from $R$ by contract flattening.

Two special cases which perhaps are not so obvious have to be considered as well: The case of an empty set of model assertions and similarly, an empty set of regular assertions.

An empty set of model assertions is equivalent to a model assertion 'True'. From the definition of logical implication, it follows that the proof obligation is reduced to the flattened assertions in $c$. The proof obligations therefore consists of showing satisfiability of $F$.

The other case is an empty set of regular assertions. Similar to the case above, it follows that the proof obligation is discharged.

### 8.4 Specification Completeness

Apart from the contract correctness proofs, we have to discuss what conditions have to be satisfied by a specification in order to be complete. A definition of specification completeness for model contracts is quite easy to give:

- **Definition:** Model Class Specification Completeness
  
  A model class $c$ is completely specified if and only if every postcondition in the flat form of $c$ contains an assertion referencing each model feature of $c$ at least once.

Obviously, the definition above does not guarantee specification completeness in the sense that every desired property can be deduced from the class specification. However for our purposes, this definition is sufficient. Additionally, by using this definition, specification completeness can be easily verified by language processing tools like the contract flattener introduced in chapter 9.

The inverse concept of the specification completeness introduced above is the specification completeness for regular contracts. In this case, we cannot give such a simple approximative solution as for the Model Class Specification Completeness. Since we used an approximative definition in the discussions above, we have to find another solution. Fortunately, the proof obligations for contract correctness can take us further: We simply inverse the implication and receive the following definition:

- **Definition:** Eiffel Class Specification Completeness
  
  An EIFFEL class $c$ is completely specified if and only if for every contract $C$ the regular assertions imply the model assertions of $C$.

Note that due to the similarity to the definition of Contract Correctness Proof Obligations, the discussions of section 8.3 also apply to the Eiffel Class Specification Completeness definition.
Chapter 9

The Contract Flattener

In this chapter we present a language processing tool that helps in automating the process of generating the flattened contract form with associated proof obligations for the verification of regular assertions.

9.1 Overview

The first section of this chapter, section 9.2, gives an overview of the functionality offered by the contract flattener application. In section 9.3 we present the design ideas behind this language processing tool and describe the architecture of the application. Finally, section 9.4 provides a short manual on how to use the contract flattener to generate the flattened contract forms of EIFFEL classes with its associated proof obligations.

9.2 Functionality of the Contract Flattener

This section of chapter 9 presents the functionality provided by the contract flattener application. This application provides the following functionality:

- **IFL Language Verification**: 
  Model classes can be checked for correct and complete model contracts.

- **Language Correctness**: 
  The contract flattener application is a language processing tool which verifies model contracts with respect to the IFL language specification. It extracts model features (see section 3.2) and determines the associated model assertions. All model assertions are then subject to the following verification passes:
  - **Model Class Validity Checks**: Verify that a model class is model-valid (see section 5.4.1, and in particular validity error IFLVMC)
  - **Model Feature Validity Checks**: Verify that a model class is feature-valid (see section 5.4.2 and section 3.2)
  - **Model Expression Validity Checks**: Verify that a model class is expression-valid (see section 5.4.3 and section 5.3 for further reference)
If a model class does not pass one of the checks above, all further processing of this class is aborted, and the class is not considered for contract flattening or proof obligation generation. An appropriate error message is displayed in the console window.

- **Specification Completeness:**
  Model classes can be checked for completeness of specification in model contracts.
  
  - **Model Specification Validity Checks:** Verify that a model class has a complete model specification (see section 5.4.4)
  
  If a model class does not pass the validity check described above, an appropriate warning message is displayed in the console. Model specification validity errors are not severe errors which make further processing impossible. Hence the model class is considered for contract flattening or proof obligation generation.

- **Generation of Flattened Contract Form:**
  Model classes can be transformed into the flattened contract form. This form of an EIFFEL class is not only used for the generation of proof obligations, but also serves its purpose as a pure, mathematical specification of an EIFFEL class. See section 8.2 for a description of the flattened contract form and how it is generated.

- **Generation of Proof Obligations:**
  Based on the flattened contract form, the IFL contract flattener is also able to generate the associated proof obligations, as described in section 8.3.

This is a short overview of the facilities offered by the contract flattener application. The usage of this tool is described in section 9.4. The next section presents the design ideas behind the contract flattener.

### 9.3 Design of the Contract Flattener

The contract flattener application has been designed with a few key principles in mind. Similar to the presentation of the design of the MML library (section 6.2), we would like to present the design of the application in categories. The following design ideas have influenced the implementation of this application:

- **Reusing the Gobo Eiffel Tools:**
  Our first key design principle is to reuse the existing facilities of the GOBO EIFFEL TOOLS [Bezb]. No changes should be necessary to the source code of the GOBO libraries, since it is still under development. It turned out that the current release of the GOBO EIFFEL TOOLS, version 3.3, is not sufficient for our purposes. To compute the flattened contract form of an EIFFEL class, we have to create its flat form. The GOBO EIFFEL TOOLS provide a partial implementation to flatten features into the interface of a class, but do not support the flattening of contracts. Unfortunately, even recent snapshots of the repository, taken at the time of writing, do not provide enough functionality. Although the missing functionality could be implemented by the contract flattener application itself, it would require major changes in the GOBO library classes, which is certainly not a good idea at this stage of the development process of the GOBO EIFFEL TOOLS. We have therefore decided to only implement parts of the application. The
pure feature analysis and the some parts of the contract flattening algorithm in particular have to be postponed until the desired functionality is available and the development has become more stable. The application has been developed using ISE EIFFELSTUDIO 5.3.

**Extending the Gobo Eiffel Tools:**

As already mentioned, one of the primary goals has been to not alter any GOBO EIFFEL TOOLS distribution. However, this approach requires a series of extensions that implement the desired extra functionality. Therefore the following extension classes have been introduced:

- **IFL_CLASS**: Extension class of *ET_CLASS* providing the storage for *model features* and different *compilation status flags*, indicating which steps have already been executed and whether they have successfully terminated.

- **IFL_CONTRACT**: Common ancestor class for the contract classes *IFL_PRECONDITIONS*, *IFL_POSTCONDITIONS* and *IFL_INVARIANTS* which are extensions to the corresponding *ET_XXX* assertion classes. The contract class provides storage for *model assertions* and *proof obligations*. Additionally, it keeps track of which models are involved in the corresponding contract.

- **IFL_UNIVERSE**: Extension class of *ET_LACE_UNIVERSE* which extends the compilation steps (4) and (3) (see category Abstract Syntax Tree Visitors) of the Eiffel parser with the Model Collection Steps, Contract Flattening Steps and the Code Generation Step. The universe class *IFL_UNIVERSE* is the driver class of the IFL parser.

- **IFL_FACTORY**: Extension class of *ET_AST_FACTORY* which provides support for IFL classes, IFL contracts and IFL validity error objects.

- **IFL_ERROR_HANDLER**: Extension class of *ET_ERROR_HANDLER* which provides support for the IFL specific validity errors (see section 5.4).

- **IFL_VALIDITY_ERROR**: Extension class of *ET_VALIDITY_ERROR* (see section 5.4).

- **IFL_IMPLEMENTATION_CHECKER**: Extension class which provides a new implementation of *ET_FEATURE_CHECKER* in class *IFL_FEATURE_CHECKER*. This class is described in the step Model Assertion Collection.

- **IFL_ANCESTOR_BUILDER**: Extension class of *ET_ANCESTOR_BUILDER* which uses a somehow more liberal policy to handle errors of language constructs which are not compatible with the upcoming Eiffel standard [Mey]. The internal validity error *IFLVAG* (a class cannot have two *identical ancestor classes with different generic parameters*, see section 6.6) is reported as a warning only.

- **IFL_FEATURE_FLATTENER**: Extension class of *ET_FEATURE_FLATTENER* which implements the contract flattening process during the generation of the flat form of a class.
CHAPTER 9. THE CONTRACT FLATTENER

Abstract Syntax Tree Visitors:
The entire contract flattener application is designed as a specialized EIFFEL compiler which does not produce low-level source code or even object code, but generates different flat forms of a class, in particular the flattened contract form and a set of proof obligations. The application performs several passes over the abstract syntax tree (AST) built from the GOBO EIFFEL Parser. These passes are performed by special visitor classes designed according to the 'Visitor' design pattern [GHJV95]. There are six compilation passes which are equivalent to ISE’s compilation steps of their Melting Ice Technology™ [Eif], called degrees. The actual functionality of the contract flattener is implemented in degrees 4 and 3. Degree 4 determines the model features of a class and degree 3 verifies model assertions and generates the flattened contract form and the proof obligations.

Model Collection Steps:
The following steps describe how model information is extracted from an EIFFEL class. If a class has not been processed in a predecessor compilation step, it will not be considered further by the successor compilation step. The following three steps are executed:

Model Feature Collection:
This compilation step determines the model features of an EIFFEL class. Note that the classes are already available in their flat form (the features have been flattened, but the contracts are not yet processed). A visitor object is used to traverse the AST, searching for features that are directly exported to MML_SPECIFICATION. These model features are marked on a class basis and stored in a set for further processing. The corresponding functionality is implemented by class IFL_FEATURE_COLLECTOR.

Model Assertion Collection:
After the model features of a class have been identified, another visitor object traversed the contracts of the flattened features and searches for occurrences of calls to model features. Similar to the Model Feature Collection step, the resulting model assertions are stored in their corresponding contracts and marked with a tag, indicating that this assertion is subject to contract flattening and the computation of proof obligations. This compilation step is implemented in the extension classes IFL_FEATURE_CHECKER and IFL_IMPLEMENTATION_CHECKER.

Model Verification Step:
The last compilation step of this category is the verification step. It is implemented by class IFL_LANGUAGE_VERIFIER. The rule-based implementation follows closely the structure of section 5.4, where we have listed all validity errors of the IFL language. As
already mentioned at the beginning of this chapter, the pure feature analysis is not yet implemented. Classes which contain invalid model features or invalid model assertions are not considered for the Contract Flattening Steps.

Figure 9.2: The Model Checker Class Hierarchy

- **Contract Flattening Steps:**
  The second series of compilation steps does the actual computation on the AST. The following two steps are performed during degree 3:

  - **Computation of the Flattened Contract Form:**
    All classes in the universe which have been verified as model-valid are candidates for the flattened contract form computation. The algorithm is an implementation of the Contract Flattening Algorithm described in section 8.2 and is executed as a visitor object of class IFL ASSERTION FLATTENER.

  - **Computation of Proof Obligations:**
    Similar to the computation of the flattened contract form, this compilation step is only executed if the class for which proof obligations should be generated is model-valid. The algorithm is an implementation based on the definition of Contract Correctness Proof Obligation given in section 8.3. It is combined with a visitor object instantiated by class IFL IMPLICATION_BUILDER which implements a linear traversal over all contracts of the class being processed.
The last compilation step of the contract flattener application is the code generation step, the output of the computed information to an output directory (see section 9.4.2 for a reference on command-line arguments). For each class which has been successfully processed in one of the Contract Flattening Steps, a separate output file is generated. The files containing the flattened contract form have the extension *.int, files for proof obligations have extension *.pro. The output of both compilation steps is pure EIFFEL code, making it easy to further use the source files (see section 7.6).

Figure 9.3: The Visitor Class Hierarchy

Figure 9.4: The Model Printer Class Hierarchy
9.4 Usage of the Contract Flattener

The contract flattener application is a command-line oriented tool to process universes of EIFFEL classes described by ACE files. The ACE files have to follow the syntactical conventions of the ISE EIFFEL implementation. In section 9.4.1, we describe how to compile the application from source code. We explain the different command-line switches which are used to control the application. Section 9.4.2 shows the synopsis of the application’s command-line interface.

9.4.1 Compiling the Application

The contract flattener application uses advanced facilities of the GOBO EIFFEL TOOLS [Bezb]. The current release of these tools at the time of writing is version 3.3. However, for some algorithms like feature flattening, a more recent developer version is required. Since the GOBO EIFFEL TOOLS are experimental code which has not yet been released, taking a developer version is inevitable in this case. To summarize, the contract flattener application needs the following components for its compilation:

- ISE EIFFELSTUDIO 5.3
- ISE EIFFELBASE Class Library
- GOBO EIFFEL TOOLS Developer Version

The application depends on the EIFFELBASE class library shipped with EIFFELSTUDIO 5.3. A recent snapshot of the GOBO EIFFEL TOOLS can be retrieved from the repository on SourceForge [Bezb]. To be able to compile the contract flattener application, open EIFFELSTUDIO and create a new project from one of the ACE files in the root of the IFL code base. These files use relative path names only. In order to resolve the cluster paths in the IFL code base correctly, the developer has to set the environment variable ‘$IFL’ to the path of the code base.

The ACE files itself are completely platform-independent and can be used on all platforms supported by ISE EIFFELSTUDIO.

9.4.2 Command Line Arguments

The following command-line switches are supported in the current version:

Synopsis:

`IFLC [-?/-HELP]
IFLC [-V/-VERBOSE] [-I/-INTERFACES] [-P/-PROOFS] ace_file out_directory`

`ace_file` The ACE file describing the universe to process
`out_directory` The path of the directory where the output should be generated

`?-HELP` Display help information on the command line
`V/-VERBOSE` Display additional information about the compilation process
`I/-INTERFACES` Generate class interfaces from classes annotated with models
`P/-PROOFS` Generate proof obligations for classes annotated with models
9.5 Further Work

The contract flattener application in its current version does not yet implement all of the desired functionality. The two main improvements which have to be done in future versions of this tool are pure feature analysis and the contract flattening algorithm.

Determining pure features in an EIFFEL universe requires a global analysis of the EIFFEL classes in the universe. The problem with feature purity is that it can only be determined from a concrete implementation. But the EIFFEL language offers the concept of deferred features, operations whose implementation is not available, but delegated to descendant classes, which have the obligation to provide a corresponding implementation. Unfortunately, this concept turns feature purity into a relative notion, which can only be computed relative to a particular universe of EIFFEL classes. The computation of pure features is therefore not continuous (the addition of a class to the universe can result in major changes in the class hierarchy) and cannot be executed incrementally. We have decided to defer the model validity error checks involving pure features until the EIFFEL language provides support for feature purity in the form of the `pure` keyword of the upcoming EIFFEL standard [Mey].

Another shortcoming of the current version of the contract flattener application is the limited support for contract flattening. At the moment, the developer versions of the GOBO EIFFEL TOOLS [Bezb] do not provide an implementation to create the flat form of features which include its contracts. Moreover, facilities to evaluate a feature name occurring in an assertion in the context of a certain type are missing. Our implementation of the contract flattener therefore focuses on the verification of IFL based model contracts. The computation of the flattened contract form has been postponed until the missing facilities are available.
Chapter 10

Summary of this Document

This chapter provides a summary of the work presented in this master thesis report.

In chapter 2 we have compared some of the most representative software development processes of the different approaches to formal software construction. The only representative for a fully mathematical approach which involves mathematical proofs of entire systems is the B Method [Abr96]. A radically different approach which tries to annotate existing code written in JAVA is the JML specification language [LPC+, LBR], supporting textual annotations of class interfaces in the form of JAVADOC comments. Although JML specification expressions can be used as input for translations to theorem prover languages like PVS or ISABELLE [BCC+03], JML remains a purely annotating documentation language. The third approach taken by the EIFFEL Method [Mey92] incorporates semi-formal specifications in the programming language itself, using ordinary EIFFEL expressions to write assertions in contracts.

Chapter 3 provides an introduction to mathematical models. We have provided a detailed description of the mathematical model terminology and describe how the specification of EIFFEL classes is improved using mathematical models. A process consisting of five steps is given as a guideline for the specification process. Section 3.4 discusses what benefits and improvements mathematical models can bring with respect to conventional Design by Contract [Mey97].

The following chapter 4 provides the necessary mathematical background. It starts with a short outline of notational conventions and basic constructs of logic calculus (section 3.2). The following sections 4.4, 4.5, 4.6 and 4.7 present the notions of sets, tuples, relations, functions, respectively, and other mathematical structures as sequences (section 4.8.1) and multisets (section 4.8.2). The last section 4.9 of chapter 4 gives an introduction to higher-order functions and presents some useful generic functionals which are widely used in the Mathematical Model Library (MML).

Chapter 5 introduces the contract specification language IFL. We have described the design process of the language in section 5.2 and created the necessary notation based on chapter 3. Section 5.3 is dedicated to IFL expressions, the basic blocks of contract specification. We have given an overview of the language validity rules and list its validity errors in section 5.4. The last section 5.5 of this chapter provides a complete specification of the IFL language grammar.

Chapter 6 is dedicated to the Mathematical Model Library (MML). We have shown the principal design ideas of the MML library in section 6.2: The concept of facets (section 6.2.1), the principle of interface/implementation hiding (section 6.2.2), default implementations (section 6.2.3), the feature declaration/feature implementation principle (section 6.2.4), specification versus implementation (section 6.2.5), type safety versus generality (section 6.2.6) and the relationship between object equality and mathematical equality (section 6.2.7). The following section 6.3 presents the classes of the
MML library in detail. In section 6.4 we describe how to apply the facilities provided by the MML library to model contracts. We have further noted some limitations of the current MML version in section 6.5 and have discussed the various problems encountered during the implementation phase (section 6.6). The last section of this chapter, section 6.7, provides an outlook of further work that can be done in this area.

Chapter 7 delivers the proof of concept of the acquired insights and gives model specifications of the most fundamental data structures of the EIFFELBASE library. We have presented the process used during specification (section 7.2) and have shown the resulting model-annotated EIFFEL classes (section 7.3). In section 7.6 we have discussed how model specifications can be used for run-time assertion checking. Section 7.4 presents the problems encountered during the specification of the data structures, followed by a discussion of the potential benefits gained by model-annotated EIFFEL classes of class libraries in section 7.5.

The next chapter 8 is a practical application of the concepts presented so far and presents the process of proving contract correctness. We first have introduced the concept of contract flattening in section 8.2, a process used to create the flattened contract form of a class. With the help of a simple algorithm for contract flattening listed in section 8.2, we have shown our approach for contract correctness proofs (section 8.3). A further section of this chapter is dedicated to the question whether a specification is complete (section 8.4).

Chapter 9 presents a language processing tool called contract flattener for the automatic creation of the flattened contract form of EIFFEL classes. In section 9.2, we have given an overview of the functionality offered by this tool and have described the software architecture and design of the application (section 9.3). Section 9.4 is designed as a user’s manual for this tool, and the last section 9.5 of this chapter discusses further work and improvements to be implemented in future versions of this tool.
Chapter 11

Conclusion

In this thesis report we have shown that mathematical models are an expressive tool for specification of EIFFEL classes. The use of a sophisticated class library which provides the necessary mathematical notions as EIFFEL classes makes specifying complicated relationships between objects in object-oriented software systems possible.

The model specification approach is particularly suitable to the specification of advanced data structures. Even complex graph problems can be easily specified using ordinary mathematical constructs of set theory and relation algebra. Classes which have been specified with mathematical models provide a sound basis for contract correctness proofs and documentation. However, it remains open to what extent mathematical models can be used for the specification of complex systems such as graphical user interface classes or distributed software systems. The number of necessary models probably would increase drastically. But the principle concept that is not directly supported by mathematical models is the notion of time. Many applications need to make sure that an action is always executed once before another one gets triggered. Thanks to the Command/Query Separation Principle [Mey97] in conjunction with Abstract Data Types (ADT’s), such constraints can be modeled using status queries answering whether a particular operations has successfully finished. The complexity of such systems may prohibit the specification of such constraints.

But even if mathematical models are probably not powerful enough to express more complex systems, partial specifications help to automatically create source documentation and provide a formal description of subproblems in the problem domain. Much of the specification complexity can be hidden by well-designed model class libraries, resulting in an efficient specification process for classes representing advanced data structures.
Appendix A

IFL Grammar

A.1 Syntax and Notation

The syntax of the following grammar description is based on the EBNF formalism. We adopt the following notational conventions: Names are written in mixed-case letters and usually denote grammar rules. Keywords and other tokens of the IFL language are written in **bold** face. Predefined objects of the EIFFEL language like the entities 'Current', 'Result' or Void are written in mixed-case letters using *italic* face. Comments related to grammar rules are noted in the traditional EIFFEL style.

The start symbol of the IFL language grammar is 'Contract'. The following tokens of the IFL language are *terminal symbols* borrowed from the EIFFEL language. In order to be complete, we list them again:

- Bit: \((0 \mid 1)^+ (b \mid B)\)
- Character: "" \((a - zA - Z)""\)
- Comment: `- - (.)* \((\n (\ r)^* `- - (.)*)\)
- FreeOperator: \((\& | @ | # | '|') (a - zA - Z)^*\)
- Identifier: \((a - zA - Z) (a - zA - Z0 - 9 '|')^*\)
- Integer: \((0 - 9) (0 - 9 '|')^*\)
- Real: \([\text{Integer}] '_' [\text{Integer}] [(e \mid E) [+ -] \text{Integer}]\)
- String: """" \((a - zA - Z)""""

A.2 The Grammar Rules

1. CONTRACT ::= Precondition | Postcondition | Invariant
2. Agent ::= agent [ AgentTarget ] AgentUnqualified | [ ( Declarations ) ] [ : Type ] Routine
3. AgentArgument ::= Expression | { Type } | ?
4. AgentArguments ::= AgentArgument | AgentArguments , AgentArgument | - - empty
5. AgentTarget ::= Entity | { Type | ( Expression )

6. AgentUnqualified ::= ( Identifier | prefix “ Prefix “ | infix “ Infix “ | [] ) | ( AgentArguments ) ]

7. Array ::= << Expressions >>

8. Assignment ::= ( Result | Identifier ) ::= Expression

9. Attempt ::= ( Result | Identifier ) ?= Expression

10. Auxiliary ::= Debug | Check | retry

11. Binary ::= Expression Infix Expression

12. BinaryOperator ::= + | - | * | / | < | > | <= | >= | // | ^ | and | or | xor | implies

13. Body ::= deferred | ( do | once | Keys | pure ) Compound | external String | alias String |

14. Branch ::= inspect Expression [ when When ] [ else Compound ] end

15. Call ::= [ CallTarget . ] CallChain

16. CallArguments ::= Expression | CallArguments , Expression | - - empty

17. CallChain ::= CallUnqualified | CallChain , CallUnqualified

18. CallTarget ::= Current | Result | ( Expression )

19. CallUnqualified ::= Identifier | ( CallArguments )

20. Characters ::= Character .. Character

21. Check ::= check Clauses end

22. Choice ::= Constant | Range | { Type }

23. Choices ::= Choice | Choices , Choice | - - empty

24. Class ::= Identifier [ Types ]

25. Clause ::= [ Identifier : ] Expression

26. Clauses ::= Clause | Clauses ; Clause | - - empty

27. Compound ::= Instruction | Compound ; Instruction | - - empty

28. Condition ::= Expression then Compound

29. Conditional ::= if ( Condition | elseif Condition ) | else Compound | end


31. Control ::= Conditional | Branch | Loop

32. Create ::= create { Type | . CallUnqualified }

33. Creation ::= create { Type } | ( Result | Identifier ) . CallUnqualified
A.2. THE GRAMMAR RULES

34. Debug ::= debug [ Keys ] Compound end

35. Declarations ::= Identifiers : Type | Declarations ; Identifiers : Type | - - empty

36. Entity ::= Current | Result | Identifier

37. Expression ::= Entity | Void | Call | Operator | Agent | Constant | Array | Tuple | Old | Create

38. Expressions ::= Expression | Expressions , Expression | - - empty

39. Header ::= [ Precondition ] [ locale Declarations ]

40. Identifiers ::= Identifier | Identifiers , Identifier

41. Infix ::= BinaryOperator | FreeOperator

42. Initialization ::= Compound [ Invariant ] [ variant Clause ]

43. Instruction ::= Creation | Call | Assignment | Attempt | Control | Compound | Auxiliary

44. Interval ::= [ + | - ] Integer .. [ + | - ] Integer

45. Invariant ::= invariant Clauses

46. Keys ::= String | Keys , String | - - empty

47. Loop ::= from Initialization until Expression loop Compound end

48. Old ::= old Expression

49. Operator ::= ( Expression ) | Unary | Binary

50. Postcondition ::= ensure [ then ] Clauses

51. Precondition ::= require [ else ] Clauses

52. Prefix ::= UnaryOperator | FreeOperator

53. Range ::= Interval | Characters | Strings

54. Routine ::= Header Body [ Postcondition ] [ rescue Compound ] end

55. Strings ::= [ once ] String .. [ once ] String

56. Tuple ::= [ Expressions ]

57. Type ::= Class | expanded Class | reference Class | like ( Current | Identifier )

58. Types ::= Type | Types , Type | - - empty

59. Unary ::= Prefix Expression

60. UnaryOperator ::= not | + | -

61. Void ::= Expression ( = | /= ) Void

62. When ::= Choices then Compound | When Choices then Compound
Appendix B

MML Library Features

B.1 IFL Language Concepts

<table>
<thead>
<tr>
<th>Feature</th>
<th>Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_equal</td>
<td>Object Equality Relation</td>
<td>n.a.</td>
</tr>
<tr>
<td>equals</td>
<td>Mathematical Equality Relation</td>
<td>( a = b )</td>
</tr>
<tr>
<td>is_compatible</td>
<td>Mathematical Compatibility Relation</td>
<td>n.a.</td>
</tr>
<tr>
<td>untyped</td>
<td>Object Type Stripping</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

B.2 MML Library Classes

This appendix lists all the facilities provided by the classes in the Mathematical Model Library (MML). The name of the MML feature is listed in the left-most column, together with a description of the performed operation. In the right-most column, we state the mathematical expression for the library feature. If a feature is not listed in the corresponding class section, it may be inherited or listed in one of the general categories at the end of appendix B.2.

We use the following notational conventions in this appendix:

- **Variables** of arbitrary types are denoted by \( x \) and \( y \).
- **Functions** have names starting with lower-case letters \( f, g, \ldots \)
- **Predicates** start with capital letters \( P, Q, \ldots \)
- **Collections** receive names in lower-case letters \( c, d, \ldots \)
- **Operators** are denoted by lower-case letters \( o, \ldots \)
- **Integer** variables receive the names \( i, \ldots \)
- **Functionals** are referenced by their names in lower-case letters like \( funct, \ldots \)

Some features of the MML class library cannot be directly expressed in mathematical terms and receive an \( n.a. \) to indicate this.
### B.2.1 MML_ANY

<table>
<thead>
<tr>
<th>Feature</th>
<th>Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_pair_valued</td>
<td>Pair Value Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_set_valued</td>
<td>Set Value Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_relation_valued</td>
<td>Relation Value Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_function_valued</td>
<td>Function Value Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_graph_valued</td>
<td>Graph Value Property</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

### B.2.2 MML_PAIR

<table>
<thead>
<tr>
<th>Feature</th>
<th>Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>element</td>
<td>Pair Element Access</td>
<td>n.a.</td>
</tr>
<tr>
<td>first</td>
<td>First Element</td>
<td>n.a.</td>
</tr>
<tr>
<td>second</td>
<td>Second Element</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_identity</td>
<td>Identity Pair Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>inversed</td>
<td>Inversed Pair</td>
<td>$p^{-1}$</td>
</tr>
</tbody>
</table>

### B.2.3 MML_SET

<table>
<thead>
<tr>
<th>Feature</th>
<th>Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>any_element</td>
<td>Non-deterministic Choice</td>
<td>n.a.</td>
</tr>
<tr>
<td>any_item</td>
<td>Constrained Non-deterministic Choice</td>
<td>n.a.</td>
</tr>
<tr>
<td>identity_relation</td>
<td>Identity Relation Conversion</td>
<td>n.a.</td>
</tr>
<tr>
<td>power_set</td>
<td>Powerset Conversion</td>
<td>n.a.</td>
</tr>
<tr>
<td>random_sequence</td>
<td>Random Sequence Conversion</td>
<td>n.a.</td>
</tr>
<tr>
<td>cartesian_product</td>
<td>Cartesian Product Conversion</td>
<td>n.a.</td>
</tr>
<tr>
<td>cardinality</td>
<td>Set Cardinality</td>
<td>$|x|$</td>
</tr>
<tr>
<td>is_member</td>
<td>Set Membership Property</td>
<td>$x \in y$</td>
</tr>
<tr>
<td>is_empty</td>
<td>Empty Set Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_subset</td>
<td>Set Subset Property</td>
<td>$x \supseteq y$</td>
</tr>
<tr>
<td>is_superset</td>
<td>Set Superset Property</td>
<td>$x \subseteq y$</td>
</tr>
<tr>
<td>is_proper_superset</td>
<td>Set Proper Superset Property</td>
<td>$x \supset y$</td>
</tr>
<tr>
<td>is_proper_subset</td>
<td>Set Proper Subset Property</td>
<td>$x \subset y$</td>
</tr>
<tr>
<td>is_partition</td>
<td>Set Partition Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>intersected</td>
<td>Set Intersection</td>
<td>$x \cap y$</td>
</tr>
<tr>
<td>united</td>
<td>Set Union</td>
<td>$x \cup y$</td>
</tr>
<tr>
<td>subtracted</td>
<td>Set Subtraction</td>
<td>$x \setminus y$</td>
</tr>
<tr>
<td>difference</td>
<td>Set Symmetric Difference</td>
<td>$x \oplus y$</td>
</tr>
<tr>
<td>extended</td>
<td>Set Element Insertion</td>
<td>n.a.</td>
</tr>
<tr>
<td>pruned</td>
<td>Set Element Removal</td>
<td>n.a.</td>
</tr>
</tbody>
</table>
### B.2.4 MML_POWERSET

<table>
<thead>
<tr>
<th>Feature</th>
<th>Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>generalized_united</td>
<td>Generalized Set Union</td>
<td>$\cup x$</td>
</tr>
<tr>
<td>generalized_intersected</td>
<td>Generalized Set Intersection</td>
<td>$\cap x$</td>
</tr>
<tr>
<td>is_generalized_disjoint</td>
<td>Generalized Set Disjointness Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_generalized_partition</td>
<td>Generalized Set Partition Property</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

### B.2.5 MML_RELATION

<table>
<thead>
<tr>
<th>Feature</th>
<th>Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_identity</td>
<td>Identity Relation Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_reflexive</td>
<td>Reflexive Relation Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_irreflexive</td>
<td>Irreflexive Relation Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_symmetric</td>
<td>Symmetric Relation Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_asymmetric</td>
<td>Asymmetric Relation Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_antisymmetric</td>
<td>Antisymmetric Relation Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_transitive</td>
<td>Transitive Relation Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>united</td>
<td>Relational Union</td>
<td>$r \cup s$</td>
</tr>
<tr>
<td>intersected</td>
<td>Relational Intersection</td>
<td>$r \cap s$</td>
</tr>
<tr>
<td>extended</td>
<td>Relational Extension</td>
<td>n.a.</td>
</tr>
<tr>
<td>domain</td>
<td>Relation Domain</td>
<td>$\text{dom } r$</td>
</tr>
<tr>
<td>range</td>
<td>Relation Range</td>
<td>$\text{ran } r$</td>
</tr>
<tr>
<td>image_element</td>
<td>Relational Image Element</td>
<td>n.a.</td>
</tr>
<tr>
<td>image</td>
<td>Relational Image</td>
<td>$r \ [s]$</td>
</tr>
<tr>
<td>anti_image_element</td>
<td>Relational Anti Image Element</td>
<td>n.a.</td>
</tr>
<tr>
<td>anti_image</td>
<td>Relational Anti Image</td>
<td>n.a.</td>
</tr>
<tr>
<td>domain_element_restricted</td>
<td>Domain Element Restriction</td>
<td>n.a.</td>
</tr>
<tr>
<td>domain_restricted</td>
<td>Domain Restriction</td>
<td>$r \triangleleft s$</td>
</tr>
<tr>
<td>range_element_restricted</td>
<td>Range Element Restriction</td>
<td>n.a.</td>
</tr>
<tr>
<td>range_restricted</td>
<td>Range Restriction</td>
<td>$r \triangleright s$</td>
</tr>
<tr>
<td>domain_elementAnti_restricted</td>
<td>Domain Element Anti Restriction</td>
<td>n.a.</td>
</tr>
<tr>
<td>domainAnti_restricted</td>
<td>Domain Anti Restriction</td>
<td>$r \triangleleft s$</td>
</tr>
<tr>
<td>range_elementAnti_restricted</td>
<td>Range Element Anti Restriction</td>
<td>n.a.</td>
</tr>
<tr>
<td>rangeAnti_restricted</td>
<td>Range Anti Restriction</td>
<td>$r \trianglerighteq s$</td>
</tr>
</tbody>
</table>
### APPENDIX B. MML LIBRARY FEATURES

#### B.2.6 MML_FUNCTION

<table>
<thead>
<tr>
<th>Feature</th>
<th>Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_continuous</td>
<td>Continuous Function Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_discrete</td>
<td>Discrete Function Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_invertible</td>
<td>Invertible Function Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_finite</td>
<td>Finite Function Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_partial</td>
<td>Partial Function Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_total</td>
<td>Total Function Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_injective</td>
<td>Injective Function Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_surjective</td>
<td>Surjective Function Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_bijective</td>
<td>Bijective Function Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_defined</td>
<td>Argument Defined Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>intersected</td>
<td>Functional Intersection</td>
<td>$f \cap g$</td>
</tr>
<tr>
<td>united</td>
<td>Functional Union</td>
<td>$f \cup g$</td>
</tr>
<tr>
<td>evaluated</td>
<td>Function Evaluation</td>
<td>$f(x)$</td>
</tr>
<tr>
<td>inversed</td>
<td>Functional Inversion</td>
<td>$f^{-1}$</td>
</tr>
<tr>
<td>composed</td>
<td>Functional Composition</td>
<td>$f \circ g$</td>
</tr>
<tr>
<td>sequential_closure</td>
<td>Sequential Closure</td>
<td>$f^{++}$</td>
</tr>
<tr>
<td>transitive_closure</td>
<td>Transitive Closure</td>
<td>$f^+$</td>
</tr>
<tr>
<td>reflexive_transitive_closure</td>
<td>Reflexive-Transitive Closure</td>
<td>$f^*$</td>
</tr>
</tbody>
</table>

#### B.2.7 MML_BAG

<table>
<thead>
<tr>
<th>Feature</th>
<th>Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiplicity</td>
<td>Bag Multiplicity</td>
<td>$\text{mult } x y$</td>
</tr>
<tr>
<td>is_member</td>
<td>Bag Membership Property</td>
<td>$x \in y$</td>
</tr>
<tr>
<td>is_empty</td>
<td>Empty Bag Property</td>
<td>n.a.</td>
</tr>
<tr>
<td>is_subbag</td>
<td>Bag Subset Property</td>
<td>$x \supseteq y$</td>
</tr>
<tr>
<td>is_superbag</td>
<td>Bag Superset Property</td>
<td>$x \subseteq y$</td>
</tr>
<tr>
<td>is_proper_superbag</td>
<td>Bag Proper Superset Property</td>
<td>$x \supset y$</td>
</tr>
<tr>
<td>is_proper_subbag</td>
<td>Bag Proper Subset Property</td>
<td>$x \subset y$</td>
</tr>
<tr>
<td>intersected</td>
<td>Bag Intersection</td>
<td>$x \cap y$</td>
</tr>
<tr>
<td>united</td>
<td>Bag Union</td>
<td>$x \cup y$</td>
</tr>
<tr>
<td>subtracted</td>
<td>Bag Subtraction</td>
<td>$x \setminus y$</td>
</tr>
<tr>
<td>difference</td>
<td>Bag Symmetric Difference</td>
<td>$x \oplus y$</td>
</tr>
<tr>
<td>extended</td>
<td>Bag Element Insertion</td>
<td>n.a.</td>
</tr>
<tr>
<td>pruned</td>
<td>Bag Element Removal</td>
<td>n.a.</td>
</tr>
<tr>
<td>added</td>
<td>Bag Addition</td>
<td>n.a.</td>
</tr>
<tr>
<td>removed</td>
<td>Bag Removal</td>
<td>n.a.</td>
</tr>
</tbody>
</table>
### B.2.8 MML_SEQUENCE

<table>
<thead>
<tr>
<th>Feature</th>
<th>Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>is_member</em></td>
<td>Sequence Membership Property</td>
<td>$x \in s$</td>
</tr>
<tr>
<td><em>isDefined</em></td>
<td>Index Definition Property</td>
<td>n.a.</td>
</tr>
<tr>
<td><em>is_subsequence</em></td>
<td>Sequence Subsequence Property</td>
<td>$s \supseteq t$</td>
</tr>
<tr>
<td><em>is_proper_subsequence</em></td>
<td>Sequence Proper Subsequence Property</td>
<td>$s \supset t$</td>
</tr>
<tr>
<td><em>is_proper_supersequence</em></td>
<td>Sequence Proper Supersequence Property</td>
<td>$s \subset t$</td>
</tr>
<tr>
<td><em>lower_bound</em></td>
<td>Lower Index Bound</td>
<td>lower $s$</td>
</tr>
<tr>
<td><em>upper_bound</em></td>
<td>Upper Index Bound</td>
<td>upper $s$</td>
</tr>
<tr>
<td><em>subtracted</em></td>
<td>Sequence Subtraction</td>
<td>n.a.</td>
</tr>
<tr>
<td><em>extended_at</em></td>
<td>Sequence Element Insertion</td>
<td>n.a.</td>
</tr>
<tr>
<td><em>pruned</em></td>
<td>Sequence Element Removal</td>
<td>n.a.</td>
</tr>
<tr>
<td><em>pruned_at</em></td>
<td>Sequence Element Removal</td>
<td>n.a.</td>
</tr>
<tr>
<td><em>replaced</em></td>
<td>Sequence Element Replacement</td>
<td>n.a.</td>
</tr>
<tr>
<td><em>interval</em></td>
<td>Sequence Subinterval</td>
<td>n.a.</td>
</tr>
<tr>
<td><em>reversed</em></td>
<td>Sequence Reversal</td>
<td>$s$</td>
</tr>
<tr>
<td><em>first</em></td>
<td>First Sequence Element</td>
<td>first $s$</td>
</tr>
<tr>
<td><em>last</em></td>
<td>Last Sequence Element</td>
<td>last $s$</td>
</tr>
<tr>
<td><em>tail</em></td>
<td>Sequence Tail</td>
<td>$tail s$</td>
</tr>
<tr>
<td><em>front</em></td>
<td>Sequence Front</td>
<td>front $s$</td>
</tr>
<tr>
<td><em>appended</em></td>
<td>Sequence Element Appending</td>
<td>n.a.</td>
</tr>
<tr>
<td><em>prepended</em></td>
<td>Sequence Element Prepending</td>
<td>n.a.</td>
</tr>
<tr>
<td><em>concatenated</em></td>
<td>Sequence Concatenation</td>
<td>$s \bowtie t$</td>
</tr>
<tr>
<td><em>filtered</em></td>
<td>Sequence Filtering</td>
<td>$s \setminus t$</td>
</tr>
<tr>
<td><em>is_partition</em></td>
<td>Sequence Partition Property</td>
<td>$s \uplus t$</td>
</tr>
</tbody>
</table>

### B.2.9 MML_GRAPH

<table>
<thead>
<tr>
<th>Feature</th>
<th>Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>edge_extended</em></td>
<td>Graph Edge Insertion</td>
<td>n.a.</td>
</tr>
<tr>
<td><em>edge_pruned</em></td>
<td>Graph Edge Removal</td>
<td>n.a.</td>
</tr>
<tr>
<td><em>node_extended</em></td>
<td>Graph Node Insertion</td>
<td>n.a.</td>
</tr>
<tr>
<td><em>node_pruned</em></td>
<td>Graph Node Removal</td>
<td>n.a.</td>
</tr>
<tr>
<td><em>is_weakly_connected</em></td>
<td>Graph Weak Connectivity Property</td>
<td>n.a.</td>
</tr>
<tr>
<td><em>is_strongly_connected</em></td>
<td>Graph Strong Connectivity Property</td>
<td>n.a.</td>
</tr>
<tr>
<td><em>is_subgraph</em></td>
<td>Graph Subgraph Property</td>
<td>$x \supseteq y$</td>
</tr>
<tr>
<td><em>is_supergraph</em></td>
<td>Graph Supergraph Property</td>
<td>$x \subseteq y$</td>
</tr>
<tr>
<td><em>is_proper_subgraph</em></td>
<td>Graph Proper Subgraph Property</td>
<td>$x \supset y$</td>
</tr>
<tr>
<td><em>is_proper_supergraph</em></td>
<td>Graph Proper Supergraph Property</td>
<td>$x \subset y$</td>
</tr>
</tbody>
</table>
B.3 MML Generic Operations

<table>
<thead>
<tr>
<th>Feature</th>
<th>Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>filtered</td>
<td>Collection Filtering</td>
<td>$f \mid g$</td>
</tr>
<tr>
<td>mapped</td>
<td>Collection Mapping</td>
<td>$map f c$</td>
</tr>
<tr>
<td>applied</td>
<td>Collection Processing</td>
<td>$apply p c$</td>
</tr>
<tr>
<td>accumulated</td>
<td>Collection Accumulation</td>
<td>$acc o i c$</td>
</tr>
<tr>
<td>there_exists</td>
<td>Existential Quantifier</td>
<td>$\exists x : P(x)$</td>
</tr>
<tr>
<td>for_all</td>
<td>Universal Quantifier</td>
<td>$\forall x : P(x)$</td>
</tr>
<tr>
<td>curried</td>
<td>Function Currying</td>
<td>$curry f$</td>
</tr>
<tr>
<td>composed</td>
<td>Function Composition</td>
<td>$f \circ g$</td>
</tr>
<tr>
<td>iterated</td>
<td>Function Iteration</td>
<td>$iter f x$</td>
</tr>
<tr>
<td>negated</td>
<td>Logical Negation</td>
<td>$\neg x$</td>
</tr>
<tr>
<td>ored</td>
<td>Logical Disjunction</td>
<td>$x \lor y$</td>
</tr>
<tr>
<td>xored</td>
<td>Logical Exclusive Disjunction</td>
<td>$n.a.$</td>
</tr>
<tr>
<td>anded</td>
<td>Logical Conjunction</td>
<td>$x \land y$</td>
</tr>
<tr>
<td>implied</td>
<td>Logical Implication</td>
<td>$x \Rightarrow y$</td>
</tr>
</tbody>
</table>

B.4 MML Conversion Operations

<table>
<thead>
<tr>
<th>Feature</th>
<th>Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>identity_pair</td>
<td>Identity Pair Conversion</td>
<td>n.a.</td>
</tr>
<tr>
<td>identity_relation</td>
<td>Identity Relation Conversion</td>
<td>n.a.</td>
</tr>
<tr>
<td>power_set</td>
<td>Powerset Conversion</td>
<td>n.a.</td>
</tr>
<tr>
<td>random_sequence</td>
<td>Random Sequence Conversion</td>
<td>n.a.</td>
</tr>
<tr>
<td>cartesian_product</td>
<td>Cartesian Product Conversion</td>
<td>n.a.</td>
</tr>
</tbody>
</table>
Appendix C

EiffelBase Class Specifications

C.1 EiffelBase Model Structures

This appendix presents the EIFFELBASE class specifications mentioned in chapter 7.

C.1.1 LINEAR_SUBSET
defered class interface SPECIFICATION_LINEAR_SUBSET [G]

feature -- Access

  index: INTEGER
    -- Current index
  ensure
    Result >= sequence.lower_bound
    Result <= sequence.upper_bound + 1
    Result.is_equal (cursor)

feature -- Comparison

  is_equal (other: like Current): BOOLEAN
    -- Is ‘other’ attached to an object considered equal to current object?
  ensure then
    Result implies sequence.is_equal (other.sequence)
    Result implies set.is_equal (other.set)

  is_subset (other: SPECIFICATION_SUBSET [G]): BOOLEAN
    -- Is current set a subset of ‘other’?
  ensure then
    sequence.is_equal (old.sequence) and cursor.is_equal (old.cursor)

  is_superset (other: SPECIFICATION_SUBSET [G]): BOOLEAN
    -- Is current set a superset of ‘other’?
  ensure then
    sequence.is_equal (old.sequence) and cursor.is_equal (old.cursor)

feature -- Status report

  before: BOOLEAN
    -- Is cursor at left from first item?
  ensure
    Result.is_equal (not sequence.is_defined (cursor))
    cursor < sequence.lower_bound
    is_equal (old Current)
sequence.equals (old sequence)
cursor.is_equal (old cursor)
set.equals (old set)

islast: BOOLEAN
— Is cursor at last item?

ensure
Result.is_equal (cursor.is_equal (sequence.upper_bound))
is_equal (old Current)
sequence.equals (old sequence)
cursor.is_equal (old cursor)
set.equals (old set)
Result implies sequence.is_defined (cursor)

off: BOOLEAN
— Is there no current item?

ensure then
Result.is_equal (not sequence.is_defined (cursor))

valid_index (number: INTEGER): BOOLEAN
— Is ‘number’ a valid index?

ensure
Result.is_equal (number >= sequence.lower_bound − 1)
Result.is_equal (number <= sequence.upper_bound + 1)
is_equal (old Current)
number.is_equal (old number)
sequence.equals (old sequence)
cursor.is_equal (old cursor)
set.equals (old set)

feature — Cursor movement

go_i_th (i: INTEGER)
— Move cursor to ‘i’-th item.

require
valid_index (i)

ensure
cursor.is_equal (i)
i.is_equal (old i)
sequence.equals (old sequence) and set.equals (old set)
i <= sequence.upper_bound implies before
i > sequence.lower_bound implies after
cursor.is_equal (i)

feature — Element change

extend (other: G)
— Ensure that set includes ‘other’.

ensure then
sequence.equals ((old sequence).extended_at (other, (old sequence).upper_bound))
cursor.is_equal (old cursor)

merge (other: SPECIFICATION_SUBSET [G])
— Add all items of ‘other’.

ensure then
sequence.interval (sequence.lower_bound, (old sequence).upper_bound).equals (old sequence)
cursor.is_equal (old cursor)

put_left (other: G)
— Insert ‘other’ before the cursor.

require else
other /= void
not before
C.1. EIFFELBASE MODEL STRUCTURES

ensure then
index = old index + 1

feature -- Removal
prune_all (other: G)
-- Remove all occurrences of 'other'.
ensure then
set.equals ((old set).pruned (other))

wipe_out
-- Remove all items.
ensure then
sequence.is_empty
set.is_empty
cursor > sequence.upper_bound
not sequence.is_defined (cursor)

feature -- Conversion
linear_representation: SPECIFICATION_LINEAR [G]
-- Representation as a linear structure
ensure then
Result.for_all (agent set.is_member (?))
Result.for_all (agent sequence.is_member (?))
sequence.equals (old sequence)
cursor.is_equal (old cursor)
set.equals (old set)

feature -- Duplication
duplicate (number: INTEGER): like Current
-- New structure containing min ('number', 'count') items from current structure
ensure then
Result.sequence.for_all (agent sequence.is_member (?))
Result.sequence.count.is_equal (number)
number <= count implies Result.sequence.count.is_equal (number)
number >= count implies Result.sequence.count.is_equal (count)
sequence.equals (old sequence)
cursor.is_equal (old cursor)

feature -- Element Change
fill (other: SPECIFICATION_CONTAINER [G])
-- Fill with as many items of 'other' as possible.
ensure then
sequence.interval (sequence.lower_bound, (old sequence).upper_bound).equals (old sequence)
other.linear_representation.for_all (agent set.is_member (?))
cursor.is_equal (old cursor)

feature -- Status Report
is_inserted (other: G): BOOLEAN
-- Has 'other' been inserted by the most recent insertion?
ensure then
Result implies set.is_member (other)
Result implies sequence.is_member (other)

invariant
before.is_equal (cursor.is_equal (0))

end -- class SPECIFICATION_LINEAR_SUBSET
C.1.2 BAG

defered class interface
SPECIFICATION_BAG [G]

feature {MML_SPECIFICATION, ANY} -- Model

bag: MML_BAG [G]
    -- The bag model
    ensure
        is_equal (old Current)
        Result /= void

feature -- Access

has (other: G): BOOLEAN
    -- Does structure include ‘other’?
    ensure then
        Result, is_equal (bag, is_member (other))
        bag, is_equal (old bag)

feature -- Measurement

occurrences (other: G): INTEGER
    -- Number of times ‘other’ appears in structure
    require
        other /= void
    ensure
        Result, is_equal (bag, multiplicity (other))
        is_equal (old Current)
        bag, is_equal (old bag)
        other, is_equal (old other)

feature -- Comparison

is_equal (other: like Current): BOOLEAN
    -- Is ‘other’ attached to an object considered equal to current object?
    ensure then
        Result, is_equal (bag, is_equal (other, bag))
        bag, is_equal (old bag)

feature -- Status report

changeable_comparison_criterion: BOOLEAN
    -- May ‘object_comparison’ be changed?
    ensure then
        bag, is_equal (old bag)

is_empty: BOOLEAN
    -- Is there no element?
    ensure then
        Result, is_equal (bag, is_empty)
        bag, is_equal (old bag)

object_comparison: BOOLEAN
    -- Must search operations use ‘equal’?
    ensure then
        bag, is_equal (old bag)

feature -- Element change

extend (other: G)
    -- Add a new occurrence of ‘other’.
    ensure then
        bag, is_equal (old bag, extended (other))
C.1. EIFFELBASE MODEL STRUCTURES

\[ \text{bag.multiplicity}(\text{other}), \text{is}_\text{equal}(\text{old bag.multiplicity}(\text{other})) \]

**feature** -- Removal

\[ \text{prune}(\text{other}: G) \]
--- Remove one occurrence of 'other' if any.
\[ \text{ensure then} \]
\[ \text{bag.is}_\text{equal}(\text{old bag.pruned}(\text{other})) \]

\[ \text{prune_all}(\text{other}: G) \]
--- Remove all occurrences of 'other'.
\[ \text{ensure then} \]
\[ \text{bag.is}_\text{equal}(\text{old bag.domain_element_anti_restricted}(\text{other})) \]

\[ \text{wipe\_out} \]
--- Remove all items.
\[ \text{ensure then} \]
\[ \text{bag.is}_\text{empty} \]

**feature** -- Element Change

\[ \text{put}(\text{other}: G) \]
--- Ensure that structure includes 'other'.
\[ \text{ensure then} \]
\[ \text{bag.is}_\text{equal}(\text{old bag.extended}(\text{other})) \]
\[ \text{bag.multiplicity}(\text{other}), \text{is}_\text{equal}(\text{old bag.multiplicity}(\text{other})) \]

**feature** -- Status Report

\[ \text{extendible}: \text{BOOLEAN} \]
--- May new items be added?)
\[ \text{ensure then} \]
\[ \text{bag.is}_\text{equal}(\text{old bag}) \]

\[ \text{is}_\text{inserted}(\text{other}: G): \text{BOOLEAN} \]
--- Has 'other' been inserted by the most recent insertion?
\[ \text{ensure then} \]
\[ \text{bag.is}_\text{equal}(\text{old bag}) \]

\[ \text{prunable}: \text{BOOLEAN} \]
--- May items be removed?
\[ \text{ensure then} \]
\[ \text{bag.is}_\text{equal}(\text{old bag}) \]

**feature** -- Status Setting

\[ \text{compare\_objects} \]
--- Ensure that future search operations will use 'equal'
\[ \text{ensure then} \]
\[ \text{bag.is}_\text{equal}(\text{old bag}) \]

\[ \text{compare\_references} \]
--- Ensure that future search operations will use '='
\[ \text{ensure then} \]
\[ \text{bag.is}_\text{equal}(\text{old bag}) \]

**invariant**

\[ \text{linear\_representation}, \text{for\_all}(\text{agent bag.is}_\text{member}(?)) \]
\[ \text{linear\_representation.is}_\text{empty}, \text{is}_\text{equal}(\text{bag.is}_\text{empty}) \]

**end** -- class \textit{SPECIFICATION\_BAG}
C.1.3 DYNAMIC_LIST

defered class interface

SPECIFICATION_DYNAMIC_LIST [G]

feature -- Removal

remove
  -- Remove current item.
  require else
  prunable
  writable
  sequence.is_defined (cursor)
  ensure then
  sequence.is_equal (old sequence.pruned_at (cursor))
  sequence.is_empty implies cursor.is_equal (sequence.upper_bound + 1)
  indexable.cardinality.is_equal (old indexable.cardinality - 1)
  cursor.is_equal (old cursor)

wipe_out
  -- Remove all items.
  ensure then
  cursor.is_equal (sequence.lower_bound - 1)

end -- class SPECIFICATION_DYNAMIC_LIST
C.1.4 STACK

defered class interface

SPECIFICATION_STACK [G]

feature —— Removal

remove —— Remove current item.

ensure then
cursor.is_equal (old cursor - 1)

feature —— Conversion

linear_representation: SPECIFICATION_LINEAR [G]

— — Representation as a linear structure

ensure then
Result.sequence.model.is_equal (sequence)
sequence.is_equal (old sequence)
cursor.is_equal (old cursor)
bag.is_equal (old bag)

feature —— Element change

append (other: SPECIFICATION_SEQUENCE [G])

— — Append a copy of ‘other’.

ensure then
cursor.is_equal (old cursor + other.sequence.model.count)

extend (other: G)

— — Push ‘other’ onto top.

ensure then
sequence.is_equal (old sequence.appended (other))
cursor.is_equal (old cursor + 1)

replace (other: G)

— — Replace top item by ‘other’.

ensure then
sequence.is_equal (old sequence.front.appended (other))

invariant

not sequence.is_empty implies cursor.is_equal (sequence.upper_bound)

end —— class SPECIFICATION_STACK
C.1.5 QUEUE

defered class interface
SPECIFICATION_QUEUE [G]

feature -- Removal

remove
-- Remove current item.

ensure then
not sequence.is_empty implies cursor.is_equal (old cursor)

feature -- Conversion

linear_representation: SPECIFICATION_LINEAR [G]
-- Representation as a linear structure

ensure then
Result.sequence_model.is_equal (sequence)
sequence.is_equal (old sequence)
cursor.is_equal (old cursor)
bag.is_equal (old bag)

feature -- Element change

append (other: SPECIFICATION_SEQUENCE [G])
-- Append a copy of 'other'.

ensure then
cursor.is_equal (old cursor)

force (other: G)
-- Add 'other' as newest item.

require
other_notVoid: other /= void

ensure
sequence.is_equal (old sequence, appended (other))
bag.is_equal (old bag, extended (other))
other.is_equal (old other)
cursor.is_equal (old cursor)

put (other: G)
-- Ensure that structure includes 'other'.

ensure then
sequence.is_equal (old sequence, appended (other))
cursor.is_equal (old cursor)

extend (other: G)
-- Add a new occurrence of 'other'.

ensure then
sequence.is_equal (old sequence, appended (other))
cursor.is_equal (old cursor)

invariant

not sequence.is_empty implies cursor.is_equal (sequence.lower_bound)
sequence.is_empty implies cursor.is_equal (sequence.lower_bound - 1)

end -- class SPECIFICATION_QUEUE
C.2 EiffelBase Class Hierarchy

The class specifications are only listed in their short form to save space. The advantages of mathematical models and the gained completeness of specification are easily recognizable even from the examples in this appendix. Since the classes of the underlying data structure class hierarchy contain a lot of duplicated code, we decided not to include the referenced EIFFELBASE library classes into this document.

We advise the interested reader to consult the MML code base, which provides the following clusters related to model specification:

- specification/hierarchy: The EIFFELBASE class hierarchy specifications
- specification/structures: The EIFFELBASE data structure specifications
Appendix D

Mathematical Definitions

D.1 Fundamental Logic

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \land y$</td>
<td>Logical Conjunction</td>
<td>22</td>
</tr>
<tr>
<td>$x \lor y$</td>
<td>Logical Disjunction</td>
<td>22</td>
</tr>
<tr>
<td>$\neg x$</td>
<td>Logical Negation</td>
<td>22</td>
</tr>
<tr>
<td>$x \implies y$</td>
<td>Logical Implication</td>
<td>22</td>
</tr>
<tr>
<td>$\forall x : X \bullet P(x)$</td>
<td>Universal Quantifier</td>
<td>23</td>
</tr>
<tr>
<td>$\exists x : X \bullet P(x)$</td>
<td>Existential Quantifier</td>
<td>23</td>
</tr>
</tbody>
</table>

D.2 Sets

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>${a_1, a_2, \ldots, a_n}$</td>
<td>Manifest Set</td>
<td>23</td>
</tr>
<tr>
<td>$x = y$</td>
<td>Set Equality</td>
<td>23</td>
</tr>
<tr>
<td>$x \neq y$</td>
<td>Set Inequality</td>
<td>23</td>
</tr>
<tr>
<td>$x \in X$</td>
<td>Set Membership</td>
<td>23</td>
</tr>
<tr>
<td>$x \notin X$</td>
<td>Set Non-Membership</td>
<td>23</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>Empty Set</td>
<td>24</td>
</tr>
<tr>
<td>$\mathbb{P}_1X$</td>
<td>Non-Empty Subset</td>
<td>24</td>
</tr>
<tr>
<td>$X \subseteq Y$</td>
<td>Subset</td>
<td>24</td>
</tr>
<tr>
<td>$X \subset Y$</td>
<td>Proper Subset</td>
<td>24</td>
</tr>
<tr>
<td>$X \cup Y$</td>
<td>Set Union</td>
<td>24</td>
</tr>
<tr>
<td>$X \cap Y$</td>
<td>Set Intersection</td>
<td>24</td>
</tr>
<tr>
<td>$X \setminus Y$</td>
<td>Set Difference</td>
<td>25</td>
</tr>
</tbody>
</table>
D.3 Tuples

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(a_1, a_2, \ldots, a_n)$</td>
<td>Manifest Tuple</td>
<td>25</td>
</tr>
<tr>
<td>$(x, y)$</td>
<td>Ordered Pair</td>
<td>25</td>
</tr>
<tr>
<td>$\text{first}$</td>
<td>First Element</td>
<td>26</td>
</tr>
<tr>
<td>$\text{second}$</td>
<td>Second Element</td>
<td>26</td>
</tr>
</tbody>
</table>

D.4 Relations

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X \leftrightarrow Y$</td>
<td>Binary Relation</td>
<td>27</td>
</tr>
<tr>
<td>$x \mapsto y$</td>
<td>Maplet</td>
<td>27</td>
</tr>
<tr>
<td>$id X$</td>
<td>Identity Relation</td>
<td>27</td>
</tr>
<tr>
<td>$\text{dom } X$</td>
<td>Domain Projection</td>
<td>27</td>
</tr>
<tr>
<td>$\text{ran } X$</td>
<td>Range Projection</td>
<td>27</td>
</tr>
<tr>
<td>$X \circ Y$</td>
<td>Relational Composition</td>
<td>28</td>
</tr>
<tr>
<td>$X \circ Y$</td>
<td>Domain Restriction</td>
<td>28</td>
</tr>
<tr>
<td>$X \triangleright Y$</td>
<td>Range Restriction</td>
<td>28</td>
</tr>
<tr>
<td>$X \triangleleft Y$</td>
<td>Domain Anti-Restriction</td>
<td>29</td>
</tr>
<tr>
<td>$X \trianglerighteq Y$</td>
<td>Range Anti-Restriction</td>
<td>29</td>
</tr>
<tr>
<td>$X^{-1}$</td>
<td>Inverse Relation</td>
<td>29</td>
</tr>
<tr>
<td>$X [Y]$</td>
<td>Relational Image</td>
<td>30</td>
</tr>
<tr>
<td>$R^*$</td>
<td>Transitive Closure</td>
<td>30</td>
</tr>
<tr>
<td>$R^1$</td>
<td>Reflexive-Transitive Closure</td>
<td>30</td>
</tr>
</tbody>
</table>

D.5 Functions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f : X \rightarrow Y$</td>
<td>Partial Function</td>
<td>31</td>
</tr>
<tr>
<td>$f : X \rightarrow Y$</td>
<td>Total Function</td>
<td>31</td>
</tr>
<tr>
<td>$f : X \rightarrow Y$</td>
<td>Finite Function</td>
<td>31</td>
</tr>
<tr>
<td>$f \cap g$</td>
<td>Functional Intersection</td>
<td>32</td>
</tr>
<tr>
<td>$f \sqcup g$</td>
<td>Functional Union</td>
<td>32</td>
</tr>
</tbody>
</table>
D.6 Sequences

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨x₁, x₂, ..., xₙ⟩</td>
<td>Manifest Sequence</td>
<td>33</td>
</tr>
<tr>
<td>seq X</td>
<td>Finite Sequence</td>
<td>33</td>
</tr>
<tr>
<td>seq₁ X</td>
<td>Non-Empty Finite Sequence</td>
<td>33</td>
</tr>
<tr>
<td>rank X</td>
<td>Finite Injective Sequence</td>
<td>33</td>
</tr>
<tr>
<td>X ▷◁ Y</td>
<td>Sequence Concatenation</td>
<td>34</td>
</tr>
<tr>
<td>first</td>
<td>First Sequence Element</td>
<td>34</td>
</tr>
<tr>
<td>last</td>
<td>Last Sequence Element</td>
<td>34</td>
</tr>
<tr>
<td>front</td>
<td>Sequence Front</td>
<td>34</td>
</tr>
<tr>
<td>tail</td>
<td>Sequence Tail</td>
<td>34</td>
</tr>
<tr>
<td>X⁻¹</td>
<td>Reverse Sequence</td>
<td>35</td>
</tr>
<tr>
<td>X ↾ Y</td>
<td>Sequence Filtering</td>
<td>35</td>
</tr>
<tr>
<td>X ↦ Y</td>
<td>Sequence Disjointness</td>
<td>36</td>
</tr>
<tr>
<td>X ⊎ Y</td>
<td>Sequence Partitioning</td>
<td>36</td>
</tr>
</tbody>
</table>

D.7 Multisets

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x₁, x₂, ..., xₙ]</td>
<td>Manifest Bag</td>
<td>37</td>
</tr>
<tr>
<td>bag X</td>
<td>Finite Multiset</td>
<td>37</td>
</tr>
<tr>
<td>bag₁ X</td>
<td>Non-Empty Finite Multiset</td>
<td>37</td>
</tr>
<tr>
<td>x ∈ X</td>
<td>Multiset Membership</td>
<td>38</td>
</tr>
<tr>
<td>x ∉ X</td>
<td>Multiset Non-Membership</td>
<td>38</td>
</tr>
<tr>
<td>mult Y x</td>
<td>Multiset Multiplicity</td>
<td>38</td>
</tr>
</tbody>
</table>

D.8 Functionals

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>f ⨿ g</td>
<td>Functional Composition</td>
<td>39</td>
</tr>
<tr>
<td>curry f</td>
<td>Function Currying</td>
<td>39</td>
</tr>
<tr>
<td>map</td>
<td>Map Functional</td>
<td>40</td>
</tr>
<tr>
<td>filter</td>
<td>Filter Functional</td>
<td>41</td>
</tr>
<tr>
<td>accumulate</td>
<td>Accumulate Functional</td>
<td>41</td>
</tr>
<tr>
<td>forall</td>
<td>Forall Functional</td>
<td>41</td>
</tr>
</tbody>
</table>
Index

abstract
  contract, 43
  data type, 15, 56
  machine, 9
  syntax tree, 106
abstraction
  function, 14, 14, 16, 43, 44, 69, 81
  invariant, 14, 16, 44, 69, 81
  mathematical abstraction, 14, 15, 15, 19, 57, 80
accumulate, 41, 67
ACE file, 68, 109
active
  data structure, 84
ADT, 15, 47, 56
agent
  inline agent, 52, 72, 76, 78
algebra
  relation algebra, 9, 55, 57, 58, 113
ancestor, 80, 81
  class, 58, 65
  common ancestor, 80
anonymous
  type, 62
antecedent, 101
ANY, 49, 62–64, 68, 72, 74, 76, 80
argument
  actual argument, 100
ARRAY, 86, 87
ARRAYED_LIST, 86
ARRAYED_SET, 87, 91
assertion, 10
  case differentiation assertion, 76
  checking, 94
  contract assertion, 13
  definition assertion, 47, 47, 51, 80, 81, 89, 92, 96, 100
  flattened assertion, 100, 101
  immutable argument assertion, 17, 80
immutable model assertion, 17, 17
invariant assertion, 81
model assertion, 13, 14, 14–17, 46, 47, 80, 93, 95, 96, 99–101, 103, 105–107
pure feature assertion, 17, 17, 80, 81
regular assertion, 13, 13, 14, 20, 80, 95–97, 99–101, 103
shared assertion, 93, 96
assignment
  attempt, 74, 75
AST, 106, 107
attribute, 14
  class attribute, 16
axiom, 93, 96
B method, 7, 9
BAG, 82, 83, 87, 90, 91, 96
bag, see multiset, 57, 111
  containment, 65
bijective
  function, 31
BILINEAR, 85, 87
binary
  operator, 48
BOOLEAN, 47
Boolean
  operator, 53
BOX, 84
calculus
  predicate calculus, 23
call
  CAT call, 97
  feature call, 56
cardinality, 22, 23, 24
Cartesian product, 15, 16, 27, 57, 59
case
  differentiation, 49, 93
cast
type cast, 75
CHAIN, 87, 88, 90
CHARACTER, 10, 47
class
  ancestor class, 58, 65
  attribute, 16
  auxiliary class, 64, 67
  base class, 71
  compositor class, 62
  deferred class, 59, 61, 84
  facet class, 64, 66, 84, 91
  generic class, 14
  implementation class, 71, 94
  interface class, 71, 94
  internal class, 66
  invariant, 81, 93, 96
  marker class, 64
  model class, 15, 44, 45, 47, 50, 64, 66, 69, 70, 97, 101, 103, 104
  proxy class, 94
  specification class, 94
clean
  feature, 43, 46, 46, 48
closure
  reflexive transitive closure, 30, 65
  sequence closure, 36, 40, 65
  transitive closure, 30, 37, 60, 65
code
  generation, 9
COLLECTION, 85, 87, 88
command, 17, 86
comment
  feature comment, 81, 94
  header comment, 81
compatibility
  mathematical compatibility, 58, 67, 72
complete
  specification, 51
COMPLETE_GRAPH, 93
completeness
  graph completeness, 66
  specification completeness, 47, 101
composed, 62
composite operation, 96
composition
  function composition, 39, 40, 65, 67
  relation composition, 28, 39, 62, 65
compositor
  class, 62
concatenation
  sequence concatenation, 34
conjunction
  logical conjunction, 22
connectivity
  graph connectivity, 66
  strong connectivity, 19
consequent, 101
CONTAINER, 83, 84
containment
  bag containment, 65
  graph containment, 66
  sequence containment, 66
  set containment, 65
contract, 19
  abstract contract, 43
  correctness, 99
  flattened contract, 96
  flattener, 103, 104
  flattening, 19, 99, 101, 110
  form, 99, 100
  model contract, 13, 14, 15, 17, 43–46, 51, 72, 90, 93, 94, 99–101, 103, 104, 110
  regular contract, 14, 101
  specification, 17
correctness
  contract correctness, 99, 100
creation
  procedure, 48
curry, 39
currying, 39, 40
  function currying, 67
CURSOR, 88
cursor
  internal cursor, 15, 16, 88
CURSOR_STRUCTURE, 88
decomposition
  sequence decomposition, 34, 66
  system decomposition, 56, 57
deep
  equality, 49
defered
  feature, 46
definition
by comprehension, 26
by extension, 26
degree, 106
descendant
direct descendant, 80, 81
Design by Contract™, 7, 10, 13, 94, 111
difference
set difference, 25
differentiation
case differentiation, 49, 93
DIRECTED_GRAPH, 19
discrete
function, 55
disjointness
sequence disjointness, 36
disjunction
logical disjunction, 22
dispenser, 82
do_all, 83
documentation, 94
domain, 21, 27, 27, 57, 73
restriction, 28, 65
DYNAMIC_CHAIN, 84, 88, 92
DYNAMIC_LIST, 82, 83, 88, 96
EiffelBase, 19, 41, 47, 49, 53, 68, 79–81, 83–88, 90–97, 109, 112, 125, 133
EiffelStudio, 68, 72, 83, 105, 109
equality, 23
depth equality, 49, 75
mathematical equality, 58, 63–65, 72
model equality, 18
object equality, 16–18, 20, 49, 63, 65, 72, 75, 76
reference equality, 48, 49, 53, 56, 63, 71, 75, 77
set equality, 23
shallow equality, 49, 75
equivalence
relation, 70
eroof
validity error, 49
evaluation
partial evaluation, 39
exception, 14
exists, 42
expanded

inheritance, 58
export
status, 97
expression
Boolean expression, 10
model expression, 51
old expression, 52, 75
pure expression, 14, 16, 43, 46, 46, 47, 52
regular expression, 54, 76
ternary Boolean expression, 76
extendibility, 7
external
feature, 44, 48, 50, 51
facet, 56, 56, 57, 60, 65, 66, 80
class, 64, 66
convertible facet, 57, 65
extendible facet, 57, 65
filterable facet, 57, 65
mappable facet, 57, 65
processable facet, 58, 65, 66
quantifiable facet, 58, 65, 66
reduceable facet, 58
typeable facet, 58, 65, 66
feature
call, 56
clean feature, 46, 46, 48
comment, 81, 94
deferred feature, 46, 76, 110
external feature, 44, 48, 50, 51
hiding, 97
impure feature, 67
model feature, 14, 14–18, 20, 43–45, 47, 49–51, 64, 68–70, 80, 81, 87, 93, 94, 101, 103, 105–107
once feature, 44, 48, 50, 51
operator feature, 72
pure feature, 17, 43, 44, 45, 45, 46, 48, 50, 51, 67, 87, 110
purity, 44, 45, 48
wrapper feature, 72, 78
field
model field, 10
filter, 41
filtering
sequence filtering, 35
finite
INDEX

function, 31
relation, 26
sequence, 33, 33
first
pair first, 25, 28
sequence first, 35
flattening
contract flattening, 19, 99, 101, 104, 110
feature flattening, 109
for_all, 83
forall, 41
form
contract form, 94, 99, 100
flat form, 16, 51, 80, 94, 99–101, 104–106, 110
flattened contract form, 94, 97, 100, 103, 104, 106–108, 110, 112
short form, 133
front
sequence front, 35
FUNCTION, 85
function, 31, 57, 78, 111
abstraction function, 14, 14, 16, 43, 44, 69, 81
acyclic function, 36
bijective function, 31
composition, 39, 40, 65, 67
continuous function, 71, 78
currying, 39, 40, 67
discrete function, 55, 58, 71
finite function, 31, 58, 71, 78
higher-order function, 38, 40
identity function, 40, 67
injective function, 31
intersection, 32, 65
inversion, 33, 65
iteration, 40, 67
mapping function, 57, 66, 85
once function, 48
partial function, 31
pure function, 17
surjective function, 31
total function, 31
union, 32, 38, 65
functional
accumulate, 41, 67
exists, 42
filter, 41
forall, 41
iterate, 40
map, 40
reduce, 41
garbage collection, 10
generation
   code generation, 9
genericity, 10
GRAPH, 15
graph, 57, 58, 60
   containment, 66
directed graph, 66, 69
   strongly-connected graph, 19
H.B. Curry, 39
hiding
   feature hiding, 97
hierarchy
   inheritance hierarchy, 96
identity
   function, 40, 67
   object identity, 56
   relation, 27
IFL_ANCESTOR_BUILDER, 105
IFL_ASSERTION_FLATTENER, 107
IFL_CLASS, 105
IFL_CONTRACT, 105
IFL_ERROR_HANDLER, 105
IFL_FACTORY, 105
IFL_FEATURE_CHECKER, 105, 106
IFL_FEATURE_COLLECTOR, 106
IFL_FEATURE_FLATTENER, 105
IFL_IMPLEMENTATION_CHECKER, 105, 106
IFL IMPLICATION.Builder, 107
IFL_INVARIANTS, 105
IFL_LANGUAGE_VERIFIER, 106
IFL_POSTCONDITIONS, 105
IFL_PRECONDITIONS, 105
IFL_UNIVERSE, 105
IFLVALIDITY_ERROR, 105
image
   relation image, 30
immutability, 47
immutable
   object, 45, 45, 46, 56, 71
implementation
  model implementation, 16, 61, 67
implication, 101
  logical implication, 22, 101
impure
  feature, 67
INDEXABLE, 88, 91
information hiding, 59
inheritance
  expanded inheritance, 58
  implementation inheritance, 10
  multiple inheritance, 10
  relation, 56
injective
  function, 31
  sequence, 33
INTEGER, 47, 75
INTEGER_INTERVAL, 84, 85, 88, 89, 91, 97
INTEGER_REF, 75
interface
  model interface, 59–61
intersection
  function intersection, 32, 65
  generalized intersection, 24, 58, 65
  powerset intersection, 58
  set intersection, 24, 56, 65
invariant, 10, 81, 99, 100
  abstraction invariant, 14, 16, 44, 69, 81
  class invariant, 14, 15, 81, 93, 96
  gluing invariant, 15, 15, 18–20, 69, 81
  model invariant, 19, 81
  regular invariant, 81
inverse
  relation, 29
inversion
  function inversion, 65
  relation inversion, 65
iterate, 40
iteration, 67
  function iteration, 40, 67
  relation iteration, 30
iterator
  external iterator, 84
  internal iterator, 84
Java, 7, 10
JavaDoc, 10

JIVE, 10
Larch, 10
last
  sequence last, 35
LINEAR, 84, 85, 87, 89
LINEAR_SUBSET, 82, 83, 89–91, 96
linked
  list, 14
LINKED_SET, 91
LINKED_LIST, 14–16, 18, 19
LIST, 90
list, 55
  linked list, 14–16, 18
  sequential list, 15
machine
  abstract machine, 9
map, 40
maplet, 27, 56
mapping, see function
  finite mapping, 31
  function, 57
Melting Ice Technology™, 106
membership
  multiset membership, 38
  set membership, 23
message
  passing, 56
mismatch
  impedance mismatch, 55
MML_ANY, 63, 65, 67, 72
MML_BAG, 58, 65
MML_COMPOSITION, 74
MML_CONVERTIBLE, 66
MML_DEFAULT_FUNCTION, 71
MML_EXTENDIBLE, 66
MML_FILTERABLE, 66
MML_FUNCTION, 58, 65, 71, 73, 78
MML_FUNCTIONALS, 67, 73
MML_GRAPH, 58, 66, 78
MML_MAPPABLE, 66
MML_PAIR, 58, 65, 69, 73
MML_POWERSET, 58, 65, 74
MML_PROCESSABLE, 67
MML_QUANTIFIABLE, 67, 74
MML_REDUCEABLE, 67
MML_REDUCEABLE_SET, 60
MML_RELATION, 58, 62, 65, 66, 69, 73, 74
MML_SEQUENCE, 58, 66, 73
MML_SET, 58, 63, 65, 69, 73, 74
MML_SPECIFICATION, 64, 68, 69, 106
MML_TYPEABLE, 67
MML_ANY, 44
MML_FUNCTION, 55
MML_PAIR, 55
MML_SET, 55
MML_SPECIFICATION, 45, 50

model
assertion, 14, 14–17, 46, 47, 80, 93, 95, 99–101, 103, 105–107
class, 15, 44, 45, 47, 50, 66, 69, 70, 97, 101, 103, 104
contract, 14, 15, 17, 43–46, 51, 72, 93, 94, 99–101, 104, 110
equality, 18
expression, 51
feature, 14, 14–18, 20, 43–45, 47, 49–51, 64, 68–70, 80, 81, 87, 93, 94, 101, 103, 105–107
field, 10
implementation, 16, 61, 67
interface, 59–61
invariant, 19, 81
object, 69
selection, 15
multiplicity, 38
multiset, 37, 57, 58, 65, 78, 82, 111
member, 38
multiplicity, 38

negation
logical negation, 22
network, 57, 60, 78
NONE, 49, 72
NUMERIC, 60

object
equality, 49
identity, 56
immutable object, 45, 45, 46, 71
model object, 69
object-oriented
programming, 56

obligation
proof obligation, 19
old
expression, 52
once
feature, 44, 48, 50, 51
function, 48
operation
composite operation, 93, 96
operator
'and then', 51
'or else', 51
binary operator, 48, 60
Boolean operator, 49, 51, 53, 67
non-strict, 51
infix operator, 78
non-strict operator, 49
prefix operator, 78
reference operator, 51
unary operator, 48

order
total order, 84
overloading, 71
model overloading, 10
operator overloading, 10
pair, 56, 65, 78
first, 25, 26, 28
ordered pair, 25, 25, 26, 28, 58
second, 25, 26, 28

parameter
actual generic parameter, 71
formal generic parameter, 62, 74
parser, 106

partial
evaluation, 39
partition
equivalence partition, 15, 16
sequence partition, 66
partitioning
sequence partitioning, 36
polymorphism, 97
postcondition, 10, 16–19, 45, 47, 49, 51, 52, 69, 75, 80, 81, 89, 91, 92, 96, 99–101
powerset, 22, 27, 57, 58, 65
intersection, 58, 65
union, 58, 65
precondition, 10, 16, 45, 80, 81, 85, 87–89, 99, 100
predicate, 93
  unary predicate, 41, 42, 57, 66, 67
procedure
  creation procedure, 48, 51
  once procedure, 48
product
  Cartesian product, 15, 16, 27, 57
programming
  functional programming, 72
  object-oriented programming, 7, 56
  refinement-based programming, 7
proof
  correctness proof, 19
  obligation, 19, 99, 103, 104, 106, 108
property
  characteristic property, 26
proxy
  class, 94
pure
  expression, 14, 16, 43, 46, 47, 52
  feature, 17, 43, 44, 45, 46, 48, 50, 51, 67, 110
  function, 17
  query, 17, 44, 50
purity
  feature purity, 45, 48
quantifier, 10, 67
  existential quantifier, 10, 23, 23, 42, 58, 67, 89
  universal quantifier, 10, 23, 23, 42, 45, 58, 89
query, 17
  pure query, 14, 17, 44, 50
QUEUE, 82, 83, 90, 96
range, 21, 27, 27, 57, 73
  restriction, 28, 65
ranking, 33
REAL, 47, 75
reduce, 41
refactoring, 72
reference, 56
  equality, 48, 53
refinement, 9
regular
assertion, 95, 99–101, 103
expression, 54, 76
relation, 26, 57, 78, 111
  algebra, 9, 55, 57, 58, 113
  binary relation, 26, 27
  composition, 39, 62
  domain, 27, 27
  equality relation, 17, 70, 84
  finite relation, 26
  functional relation, 31, 56
  identity relation, 27
  image, 30
  inheritance relation, 56
  inverse relation, 29
  inversion, 65
  iteration, 30
  mathematical compatibility relation, 58
  mathematical equality relation, 58
  range, 27, 27
  reflexive transitive closure, 30
  transitive closure, 30
restriction
  domain restriction, 28, 65
  range restriction, 28, 65
reusability, 7
reuse, 7
reversal
  sequence reversal, 35, 66
rule
  external query rule, 44
  model argument rule, 44
  model class inheritance rule, 45
  model feature declaration rule, 45
  model implementation rule, 44
  model type rule, 44
  once query rule, 44
  pure query rule, 44
second
  pair second, 25, 28
selection
  model selection, 15
SEQUENCE, 87, 90
sequence, 14, 33, 57, 58, 78, 111
  closure, 36, 65
  concatenation, 34
containment, 66
decomposition, 34
disjointness, 36
filtering, 35
finite sequence, 33, 33
first, 35
front, 35
injective sequence, 33, 33
last, 35
partition, 66
partitioning, 36
random sequence, 65
reversal, 35, 66
tail, 35

SET, 84–86, 88, 89, 91
set, 23, 55, 57, 65, 78, 82, 111
Booleans, 22
containment, 65
difference, 25
empty set, 24
infinite set, 26
Integers, 22
intersection, 24, 65
membership, 65
powerset, 57
proper subset, 24
Reals, 22
subset, 24
theory, 9, 55, 57–59, 113
union, 24, 65

shallow
equality, 49

shared
assertion, 93

side-effect, 51
abstract side-effect, 48
concrete side-effect, 48

specification, 9
complete specification, 51
completeness, 47, 101
contract specification, 17
high-level contract specification, 19
minimal contract specification, 19
state change specification, 18

STACK, 82, 83, 96, 97

status
export status, 97

step
refinement step, 9
stripping
type stripping, 62, 71
structure
active data structure, 84
subexpression
common subexpression, 93

SUBSET, 45, 93
subset, 24, 24
substitution, 9
surjective
surjective function, 31
symbol
terminal symbol, 115

synonym, 96, 97

synopsis, 109
system
decomposition, 56, 57, 66

TABLE, 90, 91
tail
sequence tail, 35
tautology, 17
theory
set theory, 9, 55, 57–59, 113
there_exists, 83

TRAVERSABLE, 83, 85, 91, 97

TRAVERSABLE_SUBSET, 83, 90, 91, 97
tree, 78
abstract syntax tree, 106
TUPLE, 75, 77, 85, 97
tuple, 25, 55, 57, 65, 111
type
abstract data type, 15, 56
anonymous type, 62, 74
base type, 14, 44
basic type, 47, 72
Boolean type, 46
expanded type, 74
information, 65
numeric type, 44, 47, 60
reference type, 17, 44, 47, 74
static type, 71, 74
stripping, 62, 71
unified type system, 10
typing
strong typing, 62
unary
   operator, 48
   predicate, 66, 67
union
   function union, 32, 38, 65
   generalized union, 24, 58, 65
   powerset union, 58
   set union, 24, 56, 65
universal
   quantifier, 67
valid
   expression-valid, 103
   feature-valid, 103
   model-valid, 50, 51, 103, 107
validity error, 49
   IFLVEB-1, 51
   IFLVEB-2, 51
   IFLVEC, 51
   IFLVEE-1, 51
   IFLVEE-2, 51
   IFLVEF-1, 51
   IFLVEF-2, 51
   IFLVEF-3, 51
   IFLVMA, 50
   IFLVMC, 50
   IFLVME, 50
   IFLVMF, 50
   IFLVMF-17, 50
   IFLVMF-2, 50
   IFLVMF-3, 50
   IFLVMT, 50
   IFLVSD, 51
   IFLVSM, 51
   VAPE, 45
variant, 10
VDM, 10

Z specification language, 7, 9
Bibliography


[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.


