From Implementation to Specification: Integrating Textual BON Into EiffelStudio

May 23, 2012
Spring Semester, 2012

Authors: Sune Alkærsig (sual@itu.dk, 011087-3085)
Thomas Hallier Didriksen (thdi@itu.dk, 300989-1977)
Supervisor: Joseph R. Kiniry

IT University
of Copenhagen

Abstract

The Business Object Notation (BON) is a method and notation for writing software specifications. The notation consists of two parts, a graphical and a textual. The purpose of this project is to bridge the gap between the textual BON world and the world of Eiffel. Support for the graphical aspect of BON already exist in the main Eiffel IDE, EiffelStudio, and as such involving the textual aspect seems like a natural step. Support for textual BON in EiffelStudio should make it easier to get an overview over an Eiffel system, and should provide a higher level description of a system than the code itself.

To achieve this, support for textual BON has been implemented directly into EiffelStudio. It allows for seamless switching between Eiffel source code and extracted textual BON. To provide a way to verify the validity of the extracted BON a general-purpose type checker was developed. The idea is that this type checker at some point can be implemented into EiffelStudio and be an aid in BON development in an Eiffel environment.

This has resulted in a tool in EiffelStudio that extract textual BON from a class written in Eiffel. To provide an overview over a system, textual BON for all descendants of the class is also extracted. In addition, a general purpose type checker has been built upon an already existing parser and lexer. This type checker is not yet integrated into EiffelStudio.

Having the option of extracting textual BON in EiffelStudio makes it possible to get an overview of a system using BON. However, this does not mean that this area is fully explored. There are many interesting features to be added, for example fully implement the type checker into EiffelStudio and extraction of a full system into textual BON in EiffelStudio. This project hopefully provides a starting point for hopefully many other projects that can continue the work on integrating textual BON into EiffelStudio and fully bridge the gap between these two worlds.

Contents

List of Figures

Chapter 1

Introduction

When designing systems, most developers face a two-fold challenge of expressing the system and its components, so that it is both understandable for people outside the software engineering world, and at the same time gives both a low- and high-level description of the system for the purpose of development. Many methods have been designed to solve this, one of which is the BON method. BON relies on both graphical and textual elements. When developing Eiffel in EiffelStudio, the user has a graphical BON tool available, which he can quickly open for a graphical overview of inheritance and client relations of a given class. However, there is no tool for inspecting the textual aspect of BON. If one wants a textual representation, he must write it by hand.

1.1 The Business Object Notation

The Business Object Notation (BON) is a specification language and notation for writing software specifications defined by Kim Waldén and Jean-Marc Nerson in their book *Seamless Object-Oriented Architecture: Analysis and Design of Reliable Systems* [WN95]. BON is based on the principles of reusability, seamlessness, and reversibility, along with software contracting. Reusability refers to the idea of reducing the complexity of a system by relying on reusable components, while at the same time keeping the specifications created in BON at an abstraction level that enables other developers to benefit from one's work. Seamlessness is based on the notion of seamless transition between requirement specification, analysis, design, and implementation. Reversibility concerns the idea that for a software specification to have any value, changes in the implementation must always be reflected back on the specification and vice versa.

BON has two parts; a graphical notation and a textual notation. While this project focuses on textual BON, the graphical notation is used to present object structures and hierarchies throughout the report. Below is a simple example of graphical BON:

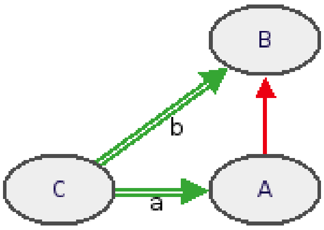


Figure 1.1: Example of the graphical BON notation. Red arrows denote inheritance relations, while green double arrows denote client-supplier relations.

This project is aimed at adding seamlessness and reversibility to EiffelStudio by adding seamless switching between Eiffel source code and textual BON, which by consequence will lead to reversibility.

The grammar for the textual BON notation is defined on pages 352-359 in the aforementioned book [WN95], and this report relies heavily on this grammar and the keywords and ideas that it introduces.

1.2 Reading Guide

This report is primarily addressed to readers with a quite thorough understanding of BON and the general ideas of object-oriented design. Readers with no or only little prior exposure to BON will benefit greatly from keeping a copy of *Seamless Object-Oriented Architecture* for reference while reading[1]. Even readers well versed in the ideas of BON, but not in the syntax of the textual notation will find the previously mentioned definition of the grammar to be helpful.

Aside from BON, the reader is assumed to a have solid understanding of basic programming concepts, specifically object-orientation and imperative programming style. Seeing as the products of this project is written in the Eiffel programming language[2], the following terms should be fully understood by the reader:

- *feature* — corresponds to what is commonly denoted as a *method* in other imperative languages such a Java and C#.
- *attribute* — corresponds to the term *field*.
- *agent* — agents wrap actions into objects. They correspond to the notion of an *anonymous inner class* in Java, a *delegate* in C#, and the idea of a closure in functional programming.
- *attached* — if a feature is attached, it refers to the fact that a reference must be associated with an object (i.e. not refer to *Void*)

[1] As the book is currently out of print, a PDF copy can be obtained from *http://www.bon-method.com/*

[2] For further information, see http://www.eiffel.com/

Outline

Chapter 1 (this chapter) presents a definition of the problem that this project aims to solve alongside basic knowledge for further understanding notions presented later on.

Chapter 2 describes the background for the project as well as many of the design decisions on which the implementations presented in later sections are based.

Chapter 3 unfolds the specifics of how textual BON has been integrated into EiffelStudio.

Chapter 4 discusses the implementation of a general-purpose type checker for textual BON, in particular how the restrictions on the grammar are implemented.

Chapter 5 lays out a number of possibilities for further development of the work initiated by this project.

Chapter 6 contains a conclusion on the findings in the remainder of the report.

1.3 Problem Definition

The purpose of this project is to bridge the gap between textual BON and EiffelStudio by providing a way to statically extract textual BON from Eiffel source code within the EiffelStudio environment. It should be easy to quickly switch from editing Eiffel source code to viewing textual BON. Furthermore, a textual BON type checker must be able to check the well-typedness of the extracted BON and return an appropriate error message to the user when it is not well-typed. It should be implemented in such a way that in the future it can be inserted into EiffelStudio. The final product must:

- Provide a way to view textual BON within the EiffelStudio environment
- Add syntax highlighting for textual BON
- Provide a way to extract textual BON in EiffelStudio
- Offer a type checker for textual BON

To achieve the first part, the EiffelStudio environment will be analyzed and discussed in chapters 2 and 3. The analysis is based on similar functionality already existing in EiffelStudio. In particular, it is studied how EiffelStudio currently provides functionality for viewing Eiffel source code from different perspectives.

For the second part, based on further analysis of EiffelStudio, a syntax highlighting mechanism for textual BON is examined.

Textual BON extraction will be discussed in chapter 3 as an extension of the aforementioned way of displaying textual BON. The goal is for the extractor to provide a meaningful overview over an Eiffel system with textual BON.

Finally, chapter 4 contains a discussion of a textual BON type checker and how it is implemented. Important design decisions and interesting challenges are explained.

1.4 Related Work

Extended BON

The Extended BON Tool Suite (EBON) is a project whose objective is to add semantic properties into the BON standard. Extended BON is developed by Joseph Kiniry. The textual BON parser and lexer used for the work described in this report is based on BON the parser and lexer from EBON, as well the abstract syntax as a meta object graph.

BONc

BONc is a command-line textual BON parser and type checker implemented in Java [KF01]. The tool is developed and maintained by Fintan Fairmichael. BONc provides support for generating graphical representations of a BON specification, namely through generation of informal charts in HTML format and graphical BON diagrams. The standard types defined in the type checker described in this report is based on the built-in types of BONc.

BON IDE

The BON Integrated Development Environment is a tool which puts emphasis on the graphical aspect of BON [Ski10]. It provides tool for visualizing a BON specification as a graphical diagram, and bases its work on the Ecore model of BON.

The BON CASE Tool

The BON CASE Tool is a CASE (Computer-Aided Software Engineering) tool that supports the creation of BON models and the formal reasoning of such [PKOL10]. It supports integration of BON with JML(the Java Modeling Language) in order to be able to utilize the formal techniques and tools of JML. The BON CASE Tool is developed by Paige, Kaminskaya, Ostroff, and Lancaric.

Other Papers

[PO01] formulates a metamodel for BON using PVS(the Prototype Verification System), which expresses the syntactic constraints that all models using BON must obey. Some of the rules defined here have been implemented in the type checker described in this report.

[PO99] describes how the properties of BON make the language a well suited tool for formal specifications and algorithm refinement.

[PO98] shows how a transition from the Z formal notation to BON can be made. Furthermore, Paige and Ostroff show how BON has the expressive power of Z with the added value of an object-oriented approach.

1.5 Acknowledgments

We would like to express a special thanks to our supervisor Joseph Kiniry, for sharing our interest in programming languages, thus making this project possible. Furthermore we would like to thank Emmanuel Stapf for not only allowing us access to the EiffelStudio repository, but also for helping us through our journey of learning Eiffel and the EiffelStudio structure. For her excellent modeling library, EiffelBase2, which is used for the type cheker, we would like to thank Nadia Polikarpova. For his work on the BONc, especially the outline for the standard BON types used in the type checker, Fintan Fairmichael. Eric Bezault is gratefully acknowledged for his work on the GOBO tools. We would also like to thank Søren Engel for many interesting discussions about the Eiffel programming language and EiffelStudio, in our joined voyage of getting familiar with the two. Finally, we would like to thank Christian Harrington and Nicolai Dahl for reviewing and proofreading this report.

Chapter 2

Background and Design

This chapter will explain the background for the project and introduce the reader to key concepts. Moreover, the basic ideas behind the design of the EiffelStudio integration and textual BON type checker will be explained to give an overview over the implementations before discussing the implementations in detail. The purpose of this chapter is to prepare the reader for the later chapters.

2.1 EiffelStudio

EiffelStudio is the main IDE for the Eiffel programming language. It is an open source project mainly developed by the Swiss Federal Institute of Technology Zürich and Eiffel Software. It has many tools to ease the design of systems with Eiffel and is focused on providing an "all-in-one" package for the user, so that no outside software is needed. It aims to cover the entire development cycle, from requirements, specification, and design to quality assurance and maintenance (paraphrased from [**?**]). Worth noting is specification. One way currently in place in EiffelStudio to make specification is through graphical BON and the UML(the Unified Modeling Language). Thus, the idea of integrating textual BON into EiffelStudio fits well into the general mission of the IDE.

2.1.1 The System

EiffelStudio consists of three main sections; Libraries, Framework, and Application (as seen in figure 2.1). This project mainly focuses on the UI subsection of Application, more specifically, the Tools part.

The UI consists of toolbars, tools and windows. The window shows the user the currently viewed context in the desired fashion, which for instance can be the pure source code or a specific perspective on the source code, but also a visual representation of it. The toolbars and tools allow the user to switch what is currently shown in the window without altering the source. When a user wants to change what is shown, he will select the appropriate perspective, and the window will be informed by this by the bar or tool in question. When displaying text, the window uses different formatters to translate the underlying source to the desired perspective on the code. The formatters are representations of the specific views. The specific formatters transform source code by having a

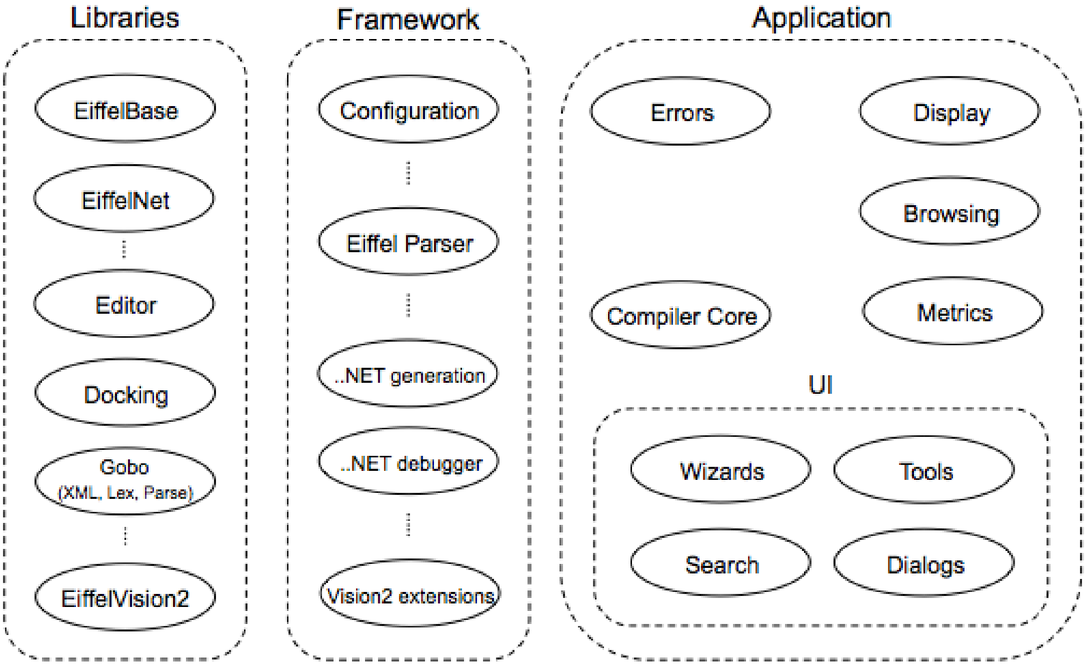


Figure 2.1: Overview of the EiffelStudio structure.

specialized output strategy define the text shown in the window based on an abstract Eiffel syntax. An output strategy is responsible for decorating a textual output based on abstract syntax as it can be seen in figure 2.2. This figure also shows an overview over how this works conceptually, and therefore might differ slightly from the actual implementation. A more precise description can be found in section 3.1.

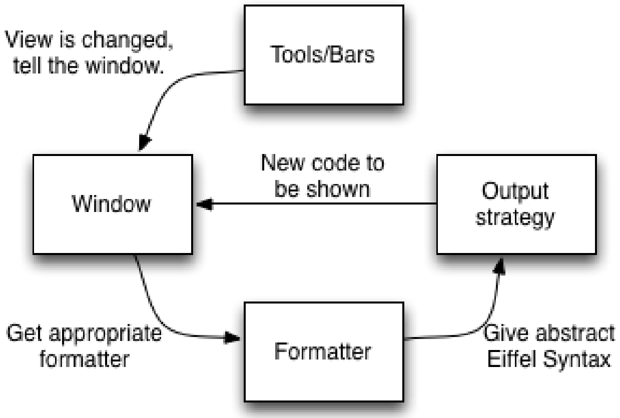


Figure 2.2: Overview of how EiffelStudio changes views.

The Idea of a View

EiffelStudio has multiple ways of showing different perspectives on the same code. The View feature which allows for switching between perspectives, or views. While viewing a piece of Eiffel source code, the user can switch to the interface view, for instance, where he can see the signature, comments, and contracts of all the features of the current class, including inherited ones. This gives the user an overview over the structure of the class. By showing information not normally available, and hiding information that is not relevant to the view. In the case of the interface view, the body of features is hidden to ensure a proper overview and to remove clutter.

The function of the textual BON tool is similar to the existing views in many ways. The tool allows the user to see the code from a textual BON perspective, where the ideas of charts (informal) and components (formal) are utilized when creating an overview over the structure of the code. The exact functionality of the BON tool will be discussed in later sections. Due to the similarity between the currently implemented views and the textual BON tool, adding the new tool as a view makes it feel like an integrated part of EiffelStudio, and not some external tool.

2.2 BON Extraction

The textual BON extractor is a static analysis tool that provides an overview of a program written in Eiffel. Its purpose is to enable the user to benefit from the features of the textual BON notation without leaving an Eiffel development environment.

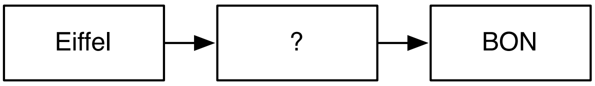


Figure 2.3: BON Extraction Model 1

Conceptually, the objective of the extractor is to bridge the gap between the Eiffel universe and the textual BON universe, as shown in figure 2.3. It does so by allowing the user to switch from an Eiffel-related view to either a formal or an informal textual BON view. When switching view, the user is presented to a textual BON representation of the Eiffel code. To add further overview over the system, the extractor also analyzes the descendants of the chosen class.

2.2.1 Overall Architecture

The textual BON extractor has two main parts. One is the interface to EiffelStudio, the other is an internal representation of the BON language. The two parts are independent and changes can be done in one without it having any effect on the other. With the exception of creation calls from the interface that hook into the internal representation.

For generating textual BON, an internal representation of the textual BON syntax has been created. This representation has a MOG-like structure with classes representing most of the BON elements defined in the grammar, with a few exceptions (See section 2.2.2). Each of these classes contain metadata about the element, and knows how to format itself into textual BON (both formal and informal) based on these data.

From EiffelStudio to BON

To bridge the aforementioned gap between EiffelStudio and the internal BON representation, an interface linking the two parts is needed. This interface needs be able to take input from EiffelStudio, and hand over the abstract Eiffel syntax to the BON modeling section. Furthermore, it also needs to take care of determining whether the current view is displaying informal or formal textual BON.

With the creation of such an interface, a link between Eiffel and BON is established. However, the chain is not complete. A link between the internal representation BON and the abstract Eiffel syntax is still missing.

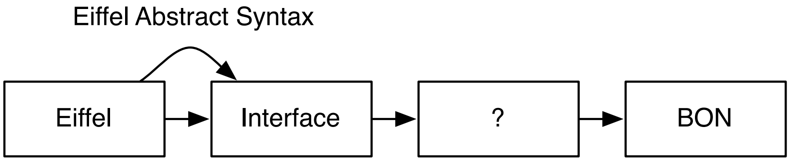


Figure 2.4: BON Extraction Model 2

To create the BON representation, a link (the missing link in figure 2.4) between the abstract syntax and the internal representation is needed. Creating a meta-object that instantiates the modeling classes based on the abstract Eiffel syntax provides this functionality. The meta-object will inspect the abstract Eiffel syntax and instantiate the textual BON objects based on the extracted information.

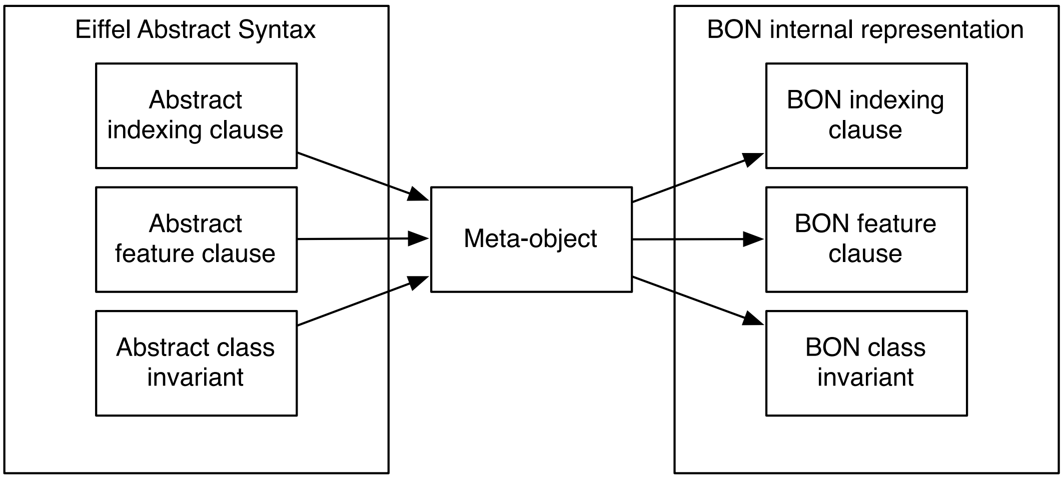


Figure 2.5: The function of the BON meta-object

The purpose of the meta-object is to convert the abstract Eiffel syntax created by EiffelStudio into an internal BON representation. As it can be seen in figure 2.5 the meta object extracts the information out of the abstract Eiffel syntax and turns it into its BON counterparts. The meta-object creates objects for both informal and formal specifications.

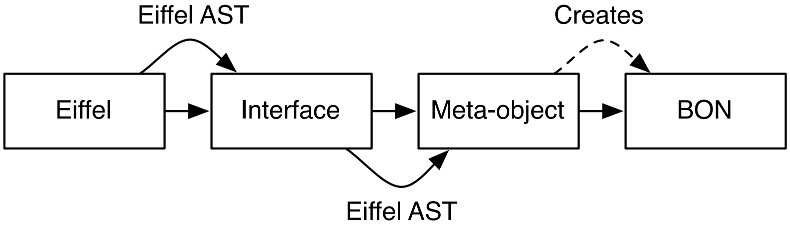


Figure 2.6: BON Extraction Model 3

Since each of the BON objects in the internal representation are able to generate textual BON from the information given by the meta-object, the system is now ready to generate BON.

From BON to EiffelStudio

Next step is for the modeling classes to hand back the formatted textual BON to EiffelStudio for it to be shown to the user. To do so either each step needs a reference back to where the textual BON is needed in order to be displayed, or each step needs to be able to receive the result of the following step and hand it back to its predecessor. In the current implementation it has been decided to pass a reference pointing to a decorating object all the way down the BON object structure, such that it is transparent to the creator of the textual BON that it is writing directly to EiffelStudio.

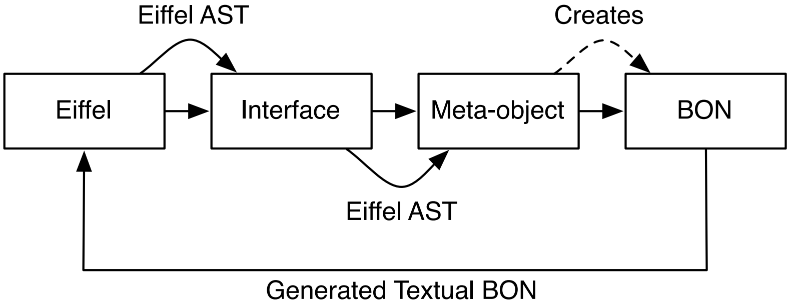


Figure 2.7: BON Extraction Model 4

Alternatives

This way of extracting BON from Eiffel has the feel of a visitor pattern, but is not quite such, as an action is executed for each of the objects in the internal representation (the visitor pattern is explained in section 2.3.2). This raises the question: Why not use a visitor pattern? If the intention was to use the BON extractor for multiple languages, a visitor pattern would have had its benefits. But since it is a part of EiffelStudio, and therefore only targets the Eiffel language, a visitor pattern would only cause a more cluttered structure. Furthermore it would require all BON generation to be located in one class, namely the visitor class. This would lead to a potentially confusing centralized structure, and in terms of extending the BON generator, would make it harder to locate where to extend the code. There would be both objects holding the data, and objects representing the BON structure, while at the same time a central class taking care of handling the information.

An alternative solution would be to only have one centralized class taking care of all the generation. This would require less overhead than the implemented solution, since it would remove the need for a meta-object and the need for the MOG-like representation of the BON grammar. Like the visitor pattern, this would, however, create a disordered structure with all generation happening in a single class, which would make extension difficult.

2.2.2 Deviations from the BON grammar

There are some deviations from the original BON grammar [WN95, pp. 352-359], both in the internal BON representation, but also in the generated textual BON. This section will discuss the most important ones.

The only major difference in the generated BON is the omission of the *part* keyword. In general, there are two uses of *part*. One is that a single class in BON can exist over multiple documents. The extractor will never try to split a class up over more than one document, thus this use of the keyword will not be needed. The other is the ability to have partial classes. This feature is seen in some languages (like C#, [MSD09]), but since Eiffel does not support partial classes there is no need for it in this implementation. Therefore the *part* keyword is not included in the generated BON nor modeled in the internal representation.

Lastly, some elements such as scenario charts and event chart are also not included. Some of these elements were excluded because they only hold semantic value and therefore require the static analysis to gather semantic information from the abstract syntax. Others, such as creation charts, have been excluded, not because they did not feel relevant or useful for this tool, but because they were deemed out of scope. This, however, does not mean that these elements could not be included in future versions of the extractor.

2.3 Type Checker

A type checker is a piece of software that determines if the semantics of a syntax is alright based on a collection of rules defined for the grammar. Contrary to syntactic analysis, in which it is determined whether the syntax itself is well-formed, a type checker checks whether the semantics implied by the syntax violates any of the aforementioned rules. For the textual BON type checker, these rules are defined in appendix B. This subsection describes the syntactic analysis of the textual BON syntax done through lexing and parsing as well as the structure of the type checker, which performs the semantic analysis. In continuation of this, several aspects of the type system are discussed.

2.3.1 Parsing and Lexing

Syntactic analysis, performed by parsing and lexing a textual BON specification, is a necessary prerequisite for performing any semantic analysis on a specification (in this case, type checking). Parsing and lexing results in an abstract representation of the specification, an abstract syntax, which is understood by a type checker. Notably, an abstract syntax and a type checker are quite tightly coupled, as the type checker has to check every corner of the abstract syntax to make sure that no violations of type rules go unnoticed.

The syntactic analysis of the textual BON grammar has been done using the *Gobo* tools *gelex* and *geyacc* [Bez].

The parser and lexer specifications upon which the type checker has been built is based on the textual BON lexer and parser created by Joseph Kiniry for the Extended BON Tool Suite [Kin]. At the outset of this project, the rule part of the parser was already finished (although some minor alterations were made to it to fit the implementation of the type checker), while the semantic actions were partially implemented. The implementation of these actions have now been completed. No alterations were made to the lexer.

The specification of the parser is designed such that it mostly corresponds to the textual BON grammar presented in [WN95]. For each of the elements in the grammar, the parser has a corresponding production rule, making sure that the entire language is parsed. The result of successful parsing is a model object graph, in which each of the syntactic elements of the grammar is represented by an object. In effect, the resulting abstract syntax is a hierarchy of objects represented as a graph, enabling a type checker to check a BON specification by traversing this graph and checking each element (represented by an object) against the rules defined for it. These rules are defined in appendix B. A detailed description of the model object graph is found in section 4.1.

Deviations from the Grammar

The grammar for the parser deviates from the grammar in [WN95] in a few places. Most of these were already present in the EBON grammar file.

A *Result* token was added for use in feature contracts in formal specifications. It represents the resulting value of its enclosing feature, just like its Eiffel counterpart.

In order to be able to express the grammar in a proper type hierarchy, some alterations regarding the formal assertions were made. In particular, the original grammar implies the following relationship [WN95, p. 357]:

$$\mathit{Expression} <: \mathit{Operator\_expression} <: \mathit{Parenthesized} <: \mathit{Expression}$$

As this situation cannot be modeled due to circular inheritance, a *Parenthesized* element is treated as a boolean expression instead of an operator expression. Changes have been made to *Set_expression* accordingly, as it refers to *Operator_expression*.

The class name in a named indirection has been made optional in order to allow for situations such as the one described in [WN95, p. 372], example in figure B.9. Contrary to the BON parser specification form the EBON projects, the indirection list has not been made optional, as this would create ambiguity between a formal generic name and a named indirection; when confronted with a generic indirection, the parser needs to be able to distinguish between these.

2.3.2 Design Decisions

This section discusses the overall design decisions upon which the textual BON type checker is based on, both with regards to the high-level structure of the system and the more low-level implications of specifying the underspecified grammar.

Type Checking Patterns

When discussing the overall structure and design of the type checker, two patterns were considered; the visitor pattern, and a procedural pattern inspired by a more functional approach.

The Visitor Pattern
The visitor pattern is a general-purpose design pattern used to traverse any kind of object structure. The pattern provides a way to add a new feature to a hierarchy of classes without having to implement the feature in the class itself [Mar02]. Instead, the functionality is implemented in a feature in a visitor class, the content of which is unknown to the visited object. The main idea of the pattern is the technique called *dual dispatch*, called as such due to the two polymorphic dispatches happening when executing the visitor:

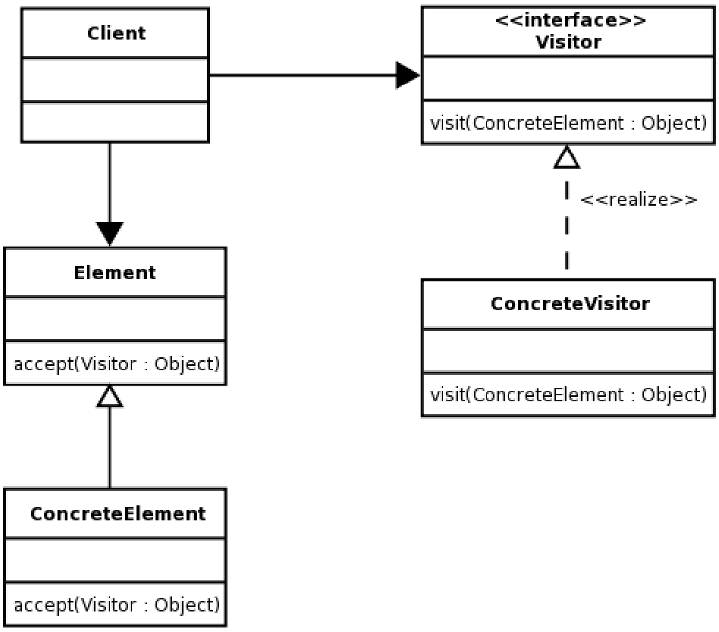


Figure 2.8: Visitor pattern (source: http://en.wikipedia.org/wiki/Visitor_pattern)

1. Each concrete class in the hierarchy must have an *accept* feature that accepts a general (often deferred) visitor type as its argument, as depicted in figure 2.8, and the first dispatch resolves the general visitor type into a concrete visitor,
2. The actual feature called on the concrete visitor is determined (via overloading) by the input type passed to the *visit* feature of the visitor, which is resolved by the second dispatch.

In the above, it is implied that the concrete visitor class has an implemented *visit* feature for each of the types in the hierarchy that is to be visited. Since feature overloading is not supported in Eiffel, the second dispatch would be implemented by a visit feature with a unique name for each of the types in the hierarchy.

The Procedural Approach
The more procedural approach is inspired by the visitor pattern, but differs from it in an important aspect. In this approach, type checking is done by defining a feature of the following form for each BON element:

$$\textit{type\_check\_bon\_element}\text{: } \text{BON\_ELEMENT} \rightarrow \text{BOOLEAN}$$

Each of these features take a BON element as argument and produces a *True* or *False* answer, depending on the semantic correctness of the input. Like the visitor pattern, a type checking feature is implemented for each of the elements in class hierarchy; but unlike the visitor pattern, the feature is called as a function on the BON element, evaluating whether or not the element is well-typed, instead of "asking" the element itself whether it is well-typed.

Choosing a Pattern
In both approaches, all of the type checking code is placed in a single class. The great benefit of the visitor pattern is its generality and object-oriented nature. But this comes at a cost. For the *accept* feature to be sufficiently general, it cannot return a specific type, and consequently the information about the well-typedness of an element must be saved in the state of the visitor. Accordingly, the object-oriented nature of the visitor pattern means that the *accept* feature would have to be implemented in every class of the meta object graph, only to redirect the call back to the visitor.

The functional nature of the procedural approach means that information about the well-typedness of an element propagates through the type checker as results of feature calls. Also, as the type checker is called as a function on an element, and not on the element itself, no additions are made to the classes of the meta object graph. But this also means a loss of generality, which is the main drawback of this approach; no interface is provided for easy traversal of the object graph for reasons other than type checking.

For the structure of the type checker, the procedural approach has been chosen, as the lack of generality seems acceptable when compared to the drawbacks of the visitor pattern. The added generality of the visitor pattern does not justify the extra overhead it would impose on the implementation of the type checker. Also, as the syntax of BON is fixed, meaning that it does not continually evolve (such as the syntax of most programming languages), it makes sense to define the well-typedness of a grammar component as a function of the well-typedness of all its subcomponents.

Phases of the Type Checker

As suggested by Appel and Palsberg in [AP04, section 5.2] for type checking MiniJava, type checking of textual BON also happens in two phases. Just as for MiniJava, this is done because classes and clusters in textual BON are mutually recursive.

The First Phase

During the first phase, the context is built; i.e. the type checker stores information about all defined elements in the AST in appropriate data structures. Informally, the first phase answers the question: “’What do we know?”. An element is considered to be defined when a specification object for it appears in the meta object graph. For instance, a class in a formal specification is defined when a class specification for it exists. If the class is merely mentioned in an inheritance clause of another class or in a cluster, it is not defined.

To avoid the rather substantial amount of symbol tables hinted at in [AP04] (e.g. mapping from classes to features, from features to type, from features to arguments etc.), the aforementioned “appriopriate data structures” are created as a class hierarchy, in which a BON class is represented by a context class with references to features, ancestors, generics, and so forth. These will be explained in greater in detail in section 4.2.

The Second Phase

During the second phase, the relationships between the defined elements are checked. Most of the actual type checking happens in this phase, as only a few rules can be checked without knowledge of the entire context. Specifically, if a type checking feature for an element returns TRUE in the second phase, then the element in question is well-typed. If the opposite is the case, an error message is emitted and FALSE is returned. In practice, not all rules can be checked in the second phase, e.g. checking for circular inheritance, as they depend on the correctness of the relationships between the defined elements. Details about the handling of these rules are explained in section 4.3.1

Type System

Textual BON as defined in [WN95] is quite underspecified in several key points, most likely to preserve flexibility of the notation (i.e. to avoid tight coupling to a single or a few programming languages). This section presents the necessary specifications and delineations made in order to type check the notation.

Names

All names of classes and clusters must be unique. There is no notion of a fully qualified name, and as such, even if static references with cluster qualification are used, it is not possible to have two classes or clusters with identical names.

In the current implementation of the type checker, there is no relation or information sharing between the type contexts of informal and formal specifications, respectively. Consequently, it is possible for a class or cluster in an informal specification to have a name identical to a class or cluster in a formal specification.

Informal Structure

Following the example set forth in BONc [KF01], a class in an informal BON specification must be in exactly one cluster. Similarly, a cluster in an informal specification must appear in a system chart or be a subcluster of a cluster that appears in a system chart.

Formal Structure
Unlike the structure of informal specifications, a class does not have to be specified in a cluster specification in a formal specification (there is no equivalent to system charts for formal specifications). As one might wish to describe a part of the system at a more granular level in a static diagram than one would do in an informal specification, it is not required that the entire system structure is specified for the specification to type check. If static references with cluster qualifications are used, however, the type checker requires that the cluster relations implied by the static references are explicitly defined.

Variance
When type checking features and feature arguments, the legal types of variance has to be implemented in the type checker in order to determine the well-typedness of these elements. Seeing as Eiffel implements covariance for both feature types and feature argument types [Mey01, the Covariance rule], the same is supported in the type checker for feature types and feature argument types in formal specifications of textual BON. This is mainly due to the fact that the primary objective of the textual BON type checker is to type check specifications extracted from Eiffel source code in EiffelStudio.

However, in a future version of the type checker, it might be desirable to be able to customize the legal types of variance during type checking, in order to consistently type check bon specifications extracted from other programming languages than Eiffel. This is left as an option for further development, and is not prohibited by the current design of the type checker.

Inheritance
As specified in [WN95, p. 65], single, multiple, and repeated inheritance are all allowed. Circular inheritance (a class inheriting from itself) is not allowed, though, and is enforced by the type checker. Conformance between generic types is not considered; thus, no conformance between LIST[INTEGER] and LIST[REAL] is considered by the type checker, even though INTEGER inherits from REAL.

However, in practice there is in fact a conformance relation between a generic type C[INTEGER] and C[REAL], assuming that INTEGER conforms to REAL (C could be substituted for any generic type with one type parameter). If C[REAL] is defined as a function f and C[INTEGER] as a function g, the following is known:

$$(1)\ f : \text{REAL} \rightarrow () \qquad (2)\ g : \text{INTEGER} \rightarrow () \qquad (3)\ \text{C[REAL]} <: \text{C[INTEGER]}$$

f is a function that takes a type REAL as argument and g is a function that takes an INTEGER as argument. Now consider a scenario where the integer 7 is passed as argument to g. As the argument type of g is INTEGER, this is allowed. Passing the same integer as an argument to f is also allowed, as 7 trivially translates into 7.0. Next, consider a scenario where 7.5, a real number, but not an integer, is passed to f. This is still allowed, as *f* accepts a type REAL. However, passing 7.5 to *g* is not allowed, as *g* only accepts integers. Accordingly, *f* will accept any argument passed to *g*, but *g* will not accept any argument passed to *f*. As a consequence, one can conclude what is stated in (3), that C[REAL] must be a subtype of C[INTEGER], because one can always pass an INTEGER as type parameter when a REAL is expected, but not the other way around.

Assertions

Due to the non-restrictive nature of the assertion grammar presented in [WN95], some restrictions have to be made in order for the assertion grammar to type check.

Firstly, the original grammar allows for a restriction or a proposition in a quantification to be a constant, a call, or an operator expression of arbitrary type. The type checker enforces these to have boolean type; the definition of boolean type in this respect is defined in section 4.6.6.

Secondly, the grammar defines a set expression to be an operator expression or a call of arbitrary type, or an enumerated set with elements of possibly unrelated types. To type check a set expression, the type checker checks if it is an expression of an enumerable type (no matter if it is an operator expression or a call), or an enumerated set in which all of the elements conform to at least one of the elements in the set. Implementation details of these restrictions are presented in section 4.6.6.

Chapter 3

EiffelStudio Integration

The following chapter will go into detail about how the BON extractor described in the previous chapter is integrated into EiffelStudio. In continuation of this, the implementation of the extractor itself will be thoroughly explained. This chapter can also be used as a knowledge foundation for developers interested in continuing development on the BON extractor or similar projects in EiffelStudio.

3.1 EiffelStudio

In continuation of the UI discussion in the previous section, this section will examine how the actual code behind this UI works. Also it will be analyzed, how EiffelStudio acts to switch to the appropriate view, and how text inside the textual views are generated. All classes in figure 3.2 will be described on a conceptual level to give the reader an idea how views are handled. To get a deeper understanding it is suggested to study the source code itself.

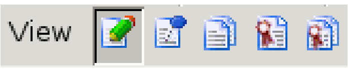


Figure 3.1: View bar in EiffelStudio

In figure 3.1 the toolbar for changing between the standard views of EiffelStudio is shown, from left to right, *Basic text view*, *Clickable view*, *Flat view*, *Contract view*, and *Interface view*. Each of these views are represented by a subclass of EB_CLASS_TEXT_FORMATTER, and as such, to keep in line with EiffelStudio's structure, any new views also should be as such. A formatter is responsible for displaying text in a text area. This concept is illustrated by figure 3.2, where it can be seen that a subclass of the class text formatter has been made for formal textual BON. Accordingly, a similar class has been implemented for informal BON. The changing between these views is handled by a development window, the purpose of which is to be a container for project tools, in this case a textual view. The TEXTUAL_BON_FORMAT_TABLES class is a shared format table that contains metadata about the view selected by the user. The TEXT_FORMATTER_DECORATOR is responsible for decorating text to be shown, and works as a mediator between the output strategy (explained next) and the formatter. This text formatter decorator also works as the link from the internal representation back to EiffelStudio (described by figure 2.7), as it processes text given from the internal BON representation as tokens. The output strategy's (in figure 3.2: TEXTUAL_BON_FORMAL_OUTPUT_STRATEGY) function is to generate some sort of textual output based on an abstract syntax. In the case of the views already in place in EiffelStudio, the output strategy works as a visitor that traverses through the abstract Eiffel syntax and generates decorated text that way. As it was decided not to use a visitor pattern for the textual BON extractor, the output strategy's role is to initialize the meta-object (mentioned in section 2.2) and then start the generation of the textual BON.

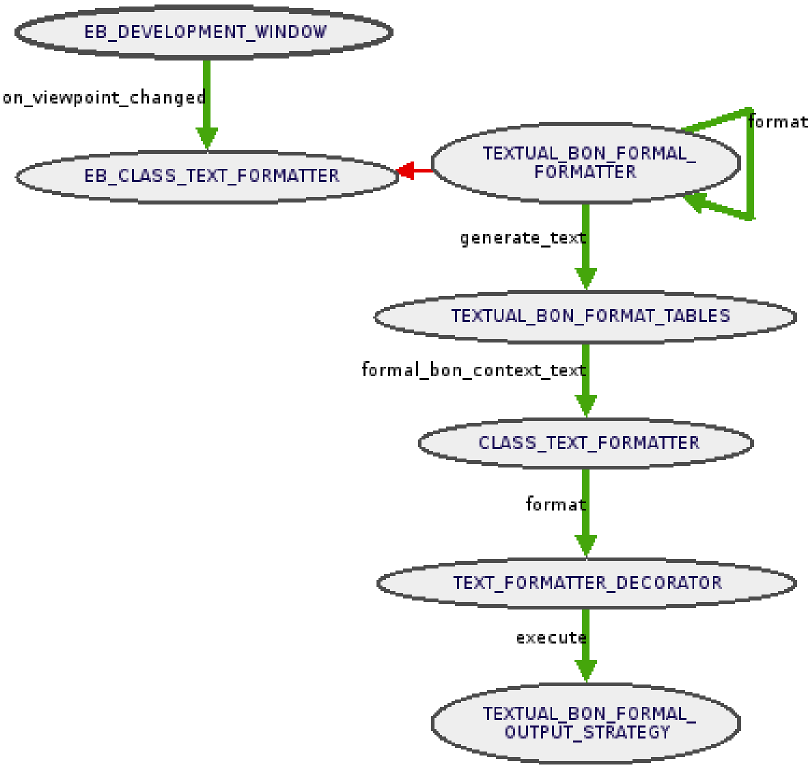


Figure 3.2: Structure of the BON extractor's interaction with EiffelStudio for formal BON.

3.1.1 BON Syntax Highlighting

At the heart of the syntax highlighter for textual BON in EiffelStudio is the scanner class EDITOR_TEXTUAL_BON_SCANNER. Like the textual BON scanner used for type checking, this scanner class is also generated by the *gelex* tool. The syntax highlighting scanner for textual BON is a modified version of an already existing scanner for syntax highlighting made for the Eiffel views in EiffelStudio.

Whenever an element that needs highlighting is encountered, such as a keyword or a symbol, an appropriate subtype of the class EDITOR_TOKEN is instantiated by the scanner. These are the same token classes that EiffelStudio makes use of when highlighting Eiffel syntax. The highlighted text in a view is thus represented as a stream of tokens, each of them appropriately decorated with a color chosen through the preferences of EiffelStudio. For a uniform experience of switching between an Eiffel view and a textual

BON view, the colors for each of the different syntactic elements are the same for BON as they are for Eiffel.

The extracted BON does not rely on this highlighting scanner, however. Instead, whenever a user switches to a textual BON view, the extractor code generates the aforementioned stream of decorated tokens while traversing the internal BON representation, after which these tokens are handed back to EiffelStudio to be displayed. As such, highlighting and extraction is done simultaneously.

The highlighting scanner's raison d'être is thus future use when actually editing the extracted BON. At this point, the scanner has not been integrated into EiffelStudio, but this integration should be possible through the class EB_EDITOR or has subclass thereof.

3.2 BON extraction

Section 2.2 described how the textual BON extraction tool consists of two parts: an internal representation of textual BON and an interface to the Eiffel part of EiffelStudio. This section will describe the structure of these two, and how they are implemented.

3.2.1 The Interface to EiffelStudio

As mentioned above, to integrate into EiffelStudio the textual BON extractor needs to have subclasses of certain classes. The most interesting ones are the formatter and the output strategy.

The development window from figure 3.2 keeps all the different formatters representing views in a collection. When the window then is told that a button has been clicked on the UI, it runs through this collection and checks which of the formatters have been selected. When it has found the appropriate formatter, the development window will invoke this and start the process towards a generating textual view. When a textual BON formatter is selected (either informal or formal see figure 3.5) it will use the specialized BON format table, rather than the one shared by the other views.

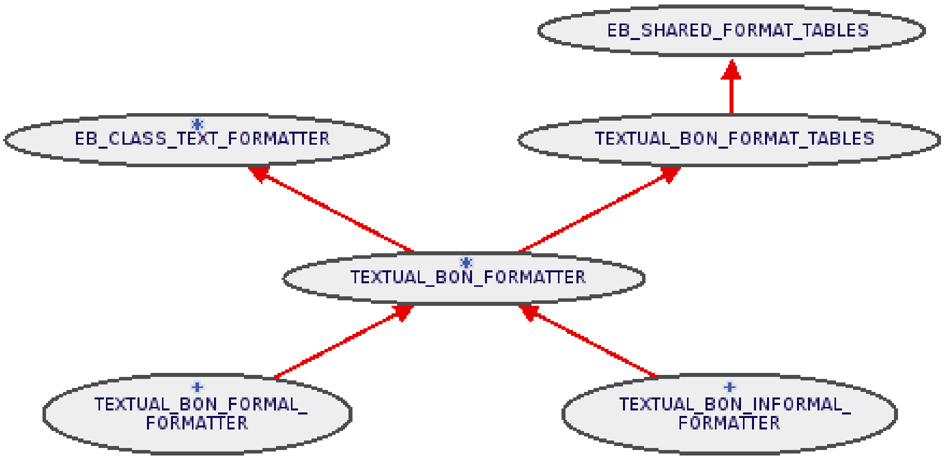

EB_SHARED_FORMAT_TABLES
EB_CLASS_TEXT_FORMATTER
TEXTUAL_BON_FORMAT_TABLES
TEXTUAL_BON_FORMATTER
TEXTUAL_BON_FORMAL_ FORMATTER
TEXTUAL_BON_INFORMAL_ FORMATTER

Figure 3.3: TEXTUAL_BON_FORMATTER inheritance relations.

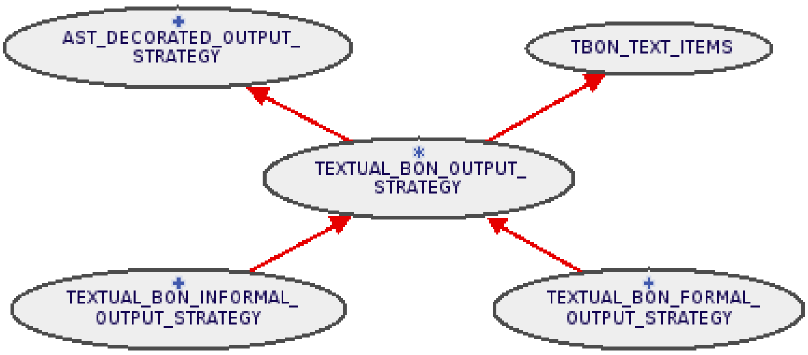


Figure 3.4: TEXTUAL_BON_OUTPUT_STRATEGY inheritance relations.

The output strategies inherits a feature called *process_class_as*. The purpose of this feature is to generate the desired output from a CLASS_AS, which is the abstract eiffel syntax representation of a class. In the implementation of the BON tool, this is done by instantiating the previously mentioned meta-object (See figure 2.6 on page 14) from the CLASS_AS object, which then generates the internal representation. This mechanism will be explained in detail in section 3.2.2.

3.2.2 The Internal BON Representation

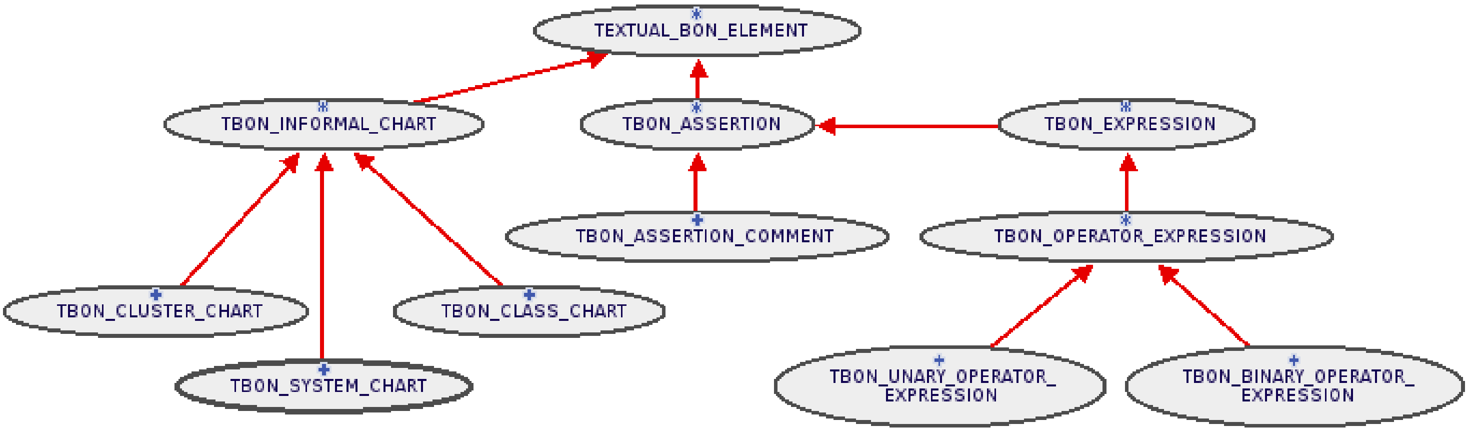


Figure 3.5: An excerpt of the classes created for the internal representation of textual BON in EiffelStudio. TBON is short for textual BON.

A series of classes has been created to represent the elements of the BON grammar. Each of these classes store the information required about the represented grammar element in order to be able to generate textual BON. The presence of mandatory elements of the grammar, such as a class name in a class chart, is ensured through the use of *attached* features and contracts (invariants and pre- and postconditions). A class thatrepresent a grammar element knows how to process itself, and itself only. If it contains other elements, such as a class containing features, the class will have a collection of features, and then delegate the processing of features to the class representing features. Any nested relations are processed this way.

This representation of textual BON is rooted in the TEXTUAL_BON_ELEMENT class. From this class, all classes inherit the features *process_to_formal_textual_bon* and *process_to_informal_textual_bon*. These features format textual BON based on the metadata in the enclosing class. For elements that only exist in either formal or informal BON or have identical representations in the two, these two features have been renamed and joined into the same feature (*process_to_textual_bon*).

Completeness

To give a feeling of working with complete textual BON, other parts than merely the class chart or the class component for the current class is generated. In the informal textual BON view, both a system chart and cluster chart are created. A system chart keeps track of its contained cluster through a collection of clusters, and a cluster chart has a collection of clusters and classes. When a system chart is asked to process itself, it will also invoke the processing feature of its contained clusters, which then does the same for its contained classes and clusters. Analogous to the generation of a system chart and a cluster chart in the extraction of informal BON, a cluster component is generated when extracting formal BON.

Indexing

In order to translate the semantics of the indexing clause in Eiffel properly to informal textual BON, a few alterations are made to it. Certain indexing tag names are identified as explanations and removed from the indexing clause and processed under the *explanation* keyword of class charts in informal textual BON. In the current implementation, the tags identified are *description* and *explanation* (the identificiation is not case-sensitive). This is identified by the feature *is_explanation_string* in TBON_CLASS_CHART. Furthermore an extra entry into the indexing clause of a class chart is added to the informal textual BON. To identify which cluster a class belongs to, a *belongs_to* tag has been added. This is not strictly necessary in this implementation, as one cluster is only ever made. However, if this was to be expanded to scale to a full system, one might want multiple clusters. As such, this tag provides the reader of the specification with a better overview.

Inheritance

To create an overview from the point of view of the currently selected class, specifications for all descendants of the current class is also extracted. This is done by inspecting the *direct_descendants* feature of the compiled class object from EiffelStudio (CLASS_C). These classes are then added to the cluster, and then processed into textual BON.

The Meta-Object

When the output strategy receives the abstract Eiffel syntax, the internal BON representation is created. As previously mentioned, this is done through a meta object. This object is denoted TBON_CLASS. TBON_CLASS takes a CLASS_AS (abstract Eiffel representation of a class) object and instantiates the internal representation of the BON. It does this by inspecting the abstract syntax represented by CLASS_AS and parsing the information needed to the textual BON objects.

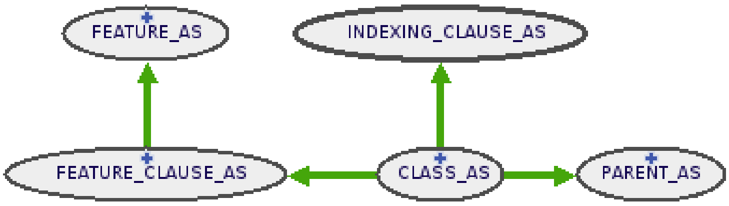


Figure 3.6: An example of the abstract Eiffel syntax.

Figure 3.6 shows an example of a part of the abstract Eiffel syntax tree. In translation to the BON internal representation (seen in figure 3.7) one will notice a few differences or alterations. In the example is shown the representation of informal BON. Since a class chart does not have the notion of feature clauses, but rather the idea of queries and commands, a class chart has a direct reference to a feature, rather than one going through a feature clause. The parent has been replaced by a reference to a class type which represents the notion of a class. Since all informal charts have indexing clauses, the class chart inherits this from the deferred informal chart class.

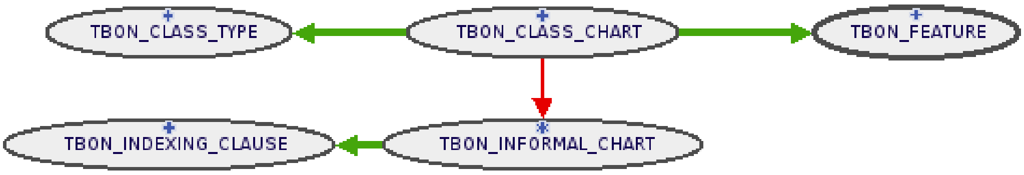


Figure 3.7: Example result of the meta-object extracting information from the abstract Eiffel syntax in figure 3.6.

Chapter 4

General-Purpose Type Checker

To give the reader an in-depth understanding of the textual BON type checker implementation, this chapter will describe how the type checker is structured and how it deals with certain rules. Furthermore, it is explained how known and unresolved types are stored using contexts. For additional study of the type checker see the semi-formal specification of the type rules in appendix B.

4.1 Parsing and Lexing

Before any semantic analysis of a textual BON specification can take place, the lexer and parser has to process it into abstract syntax on which the type checker can operate. Already mentioned in section 2.3.1 is the aspect that the structure of the parser grammar closely resembles the grammar presented in [WN95, pp. 352-359], aside from the previously explained deviations.

After successful syntactic analysis, the parser hands over the parse tree or meta object graph (MOG) to the type checker. The meta object graph represents the syntax of the analyzed specification as a hierarchy of objects.

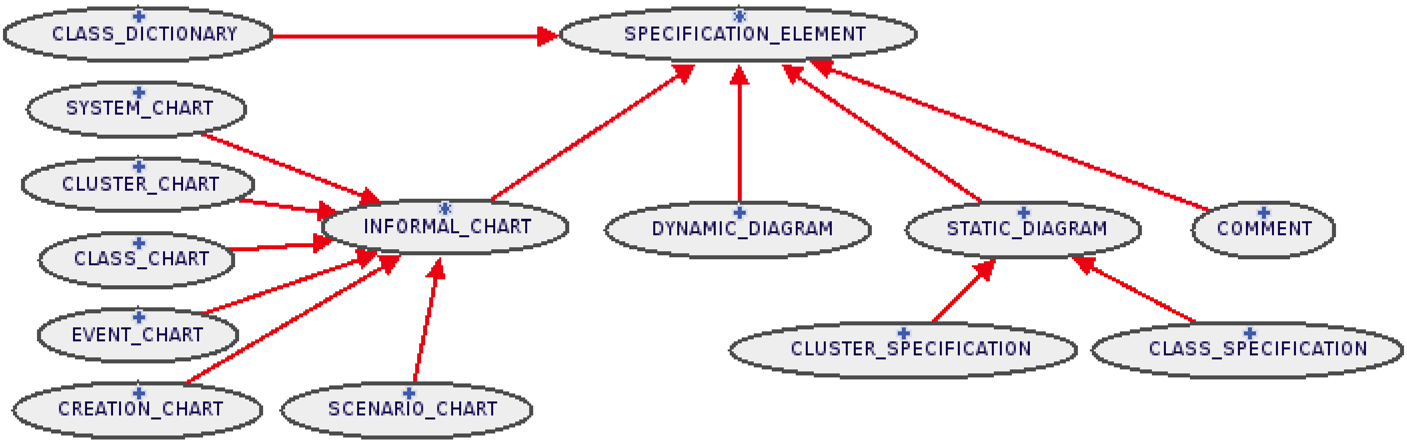


Figure 4.1: First three levels of the inheritance hierarchy of the MOG

To give an impression of the structure, figure 4.1 shows the first three levels of the inheritance hierarchy of the MOG. As one would expect, the actual object graph has a quite a few more levels. Each of these objects represent what in this report is called a *BON element* or component. Most of these are composed of multiple other BON elements, in effect creating a graph of inheritance and client-supplier links between all the meta objects.

Considering the procedural structure of the type checker outlined in section 2.3.2 and the MOG presented here, the connection between the type checker and the abstract syntax should become evident: For each object in the object graph, a corresponding checking feature checking the well-typedness of the BON element it represents is created in the type checker. In particular, this means that the type checker has features such as *check_informal_chart*, *check_class_chart*, and *check_static_diagram*.

Because BON elements are defined in terms of each other, the consequence of this structure is that the checking feature for a given BON element returns *True* only if all the (present) subelements of the element are well-typed. As will be shown in section 4.3.1, though, resolving the correctness of all the subelements when the checking feature for a BON element is called is not always possible. In these cases, auxiliary contexts are used to gather these unresolved BON elements for checking later. Having these unresolved elements do not have any influence on the final verdict of the type checker, which is returned as the result when the feature *check_bon_specification* terminates.

The top-most element of the object graph created by the parser is a type BON_SPECIFICATION, and as such the entry point into the type checker is *check_bon_specification*. When this element is not shown in figure 4.1, it is because it primarily consists of a set of SPECIFICATION_ELEMENT, and therefore it is not a very interesting object in its own right. Thus, all type checking begins and ends in this feature. What happens in between is elaborated upon in section 4.3. Prior to this, however, the translation between the abstract syntax and the type contexts of the type checker is explained.

4.2 Building the Context

The type contexts for type checking of both informal and formal specifications are built using a hierarchy of types. This hierarchy is pictured in 4.2.

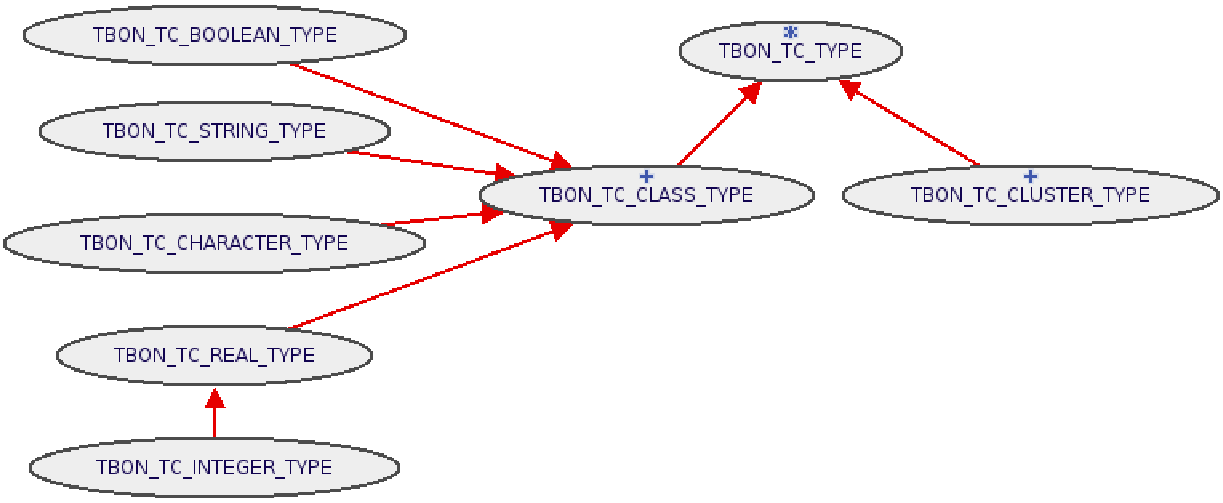


Figure 4.2: Hierarchy of classes used to build the type contexts

A type context, be it informal or formal, is built as a set of instances of TBON_TC_CLASS_TYPE and TBON_TC_CLUSTER_TYPE . Due to the naming restrictions discussed in section 2.3.2, classes and clusters are kept in the same context for convenience and easy detection of name clashes. At any time, the instances in the context represent the defined types currently known at that point. Only four BON elements define types in the context. For informal specifications, these are class charts and cluster charts, while class specifications and cluster specifications define types for formal specifications.

In formal specifications, mapping from classes to features happens through association with a set of TBON_TC_FEATURE. This type is instantiated every time a feature specification is encountered. An instance of TBON_TC_CLASS_TYPE is never associated with an undefined feature, but can be associated with an inconsistent or ill-typed feature. This is elaborated in section 4.3.1. Accordingly, the type of a feature is merely an association to an instance of TBON_TC_CLASS_TYPE.

Analogous to the mapping from classes to features, mapping from classes to generics is done through association with a sequence of TBON_TC_GENERIC. As the order of the type parameters matters, generics cannot be stored in a set. A generic type has both a bounding and an actual class type. The association between these is explained in section 4.6.5.

Types defined in the abstract syntax is primarily added to the context in the first phase of the type checker, whereas most of the relations between the types in the context are established and checked in the second phase. The idea and implementation of the phases is elaborated upon in the next section.

4.3 Going Through the Phases

As presented in section 2.3.2, the type checker operates in two phases; in the first phase, the context is built from the abstract syntax obtained from the parser, and in the second phase, the abstract syntax is checked against the context and the rules of the type system (for a specification of the type system, see appendix B).

The need for implementing a two-phased type checking algorithm arises from 1) the fact that classes in textual BON are mutually recursive and 2) the design decision that the order in which the BON elements from the abstract syntax are traversed and type checked does not matter for the final output of the algorithm. This decision was made to ensure that no logical errors would occur due to incorrect ordering of the BON elements. The phases are implemented using two simple boolean flags, respectively *first_phase* and *second_phase*, indicating the current phase of execution of the type checker. The invariant of the type checking class ensures that the two flags can never have equivalent values.

In the point of entry, the feature *check_bon_specification*, the transition between the two phases is handled by a recursive call, ensuring that the abstract syntax is traversed once for each phase, and that transitional code between the phases is run at the appropriate time. This is realized by performing type checking in the following steps:

1. Type checking begins in the first phase by a call to *check_bon_specification*
2. The abstract syntax is traversed for the first phase
3. Unresolved elements for the first phase are resolved
4. The current phase is switched from first phase to second phase
5. The abstract syntax is traversed (again) for the second phase
6. Unresolved elements for the second phase are resolved
7. Error messages and warnings, if any, are output
8. The correctness (*True*/*False*) of the specification is returned to the caller as *check_bon_specification* terminates

Traversing the abstract syntax more than once means that the type checking features for each of the elements in the abstract syntax must implement the notion of the phases as well (otherwise, the exact same code would be run twice). This notion is implemented by a conditional statement, checking against the current phase of execution.

In cases where a type checking feature is called, but has no checking to do for the current phase, *True* is returned. As the type checker can be characterized as being *optimistic*, a textual BON element is determined to be well-typed until it violates one of the rules pertaining to it (at which point an error message is emitted and *False* is returned). However, there are cases for which the well-typedness of a BON element cannot be determined at the time of the call to the checking feature; for these cases, the element in question is marked as unresolved. The element is then resolved at the end of the current phase (if possible).

4.3.1 Unresolved Elements

Unresolved elements are textual BON elements which are gathered and stored in an appropriate data structure during either the first or the second traversal of the abstract syntax, because their types cannot be resolved at the time they are encountered. The unresolved BON elements are then resolved at the appropriate time by iterating through the gathered elements.

Features

Features of a class specification, along with their arguments, are marked as unresolved in the first phase, as their types cannot be resolved when they are encountered in the first traversal, due to fact that the entire context has not been built yet. Because the type of a feature (and a feature argument) has to covariantly conform to the type of its precursor (see section 2.3.2, Variance), the type of a feature must be known prior to the second phase, in order to be able to check this conformance relation in the second phase. Thus, the features and feature arguments of all classes are resolved by giving them references to types in the context at the end of the first phase, before the second phase is entered.

Generics

The formal generics of a class specification cannot be resolved during the first traversal, as they might refer to unknown types, that are yet to be added to the type context. The resolution of the formal generics cannot be deferred to the second phase, however, as other elements such as features and feature arguments need to be able to refer to instances of the generic classes. Furthermore, it should be possible to check the validity of these instances according to the bounding types of the type parameters in the class specification in the second traversal. Thus, the formal generics of a class specification are resolved in the first phase, after the first traversal, in which the bounding types to which the type parameters refer are coupled with type instances from the context.

Inheritance Relations

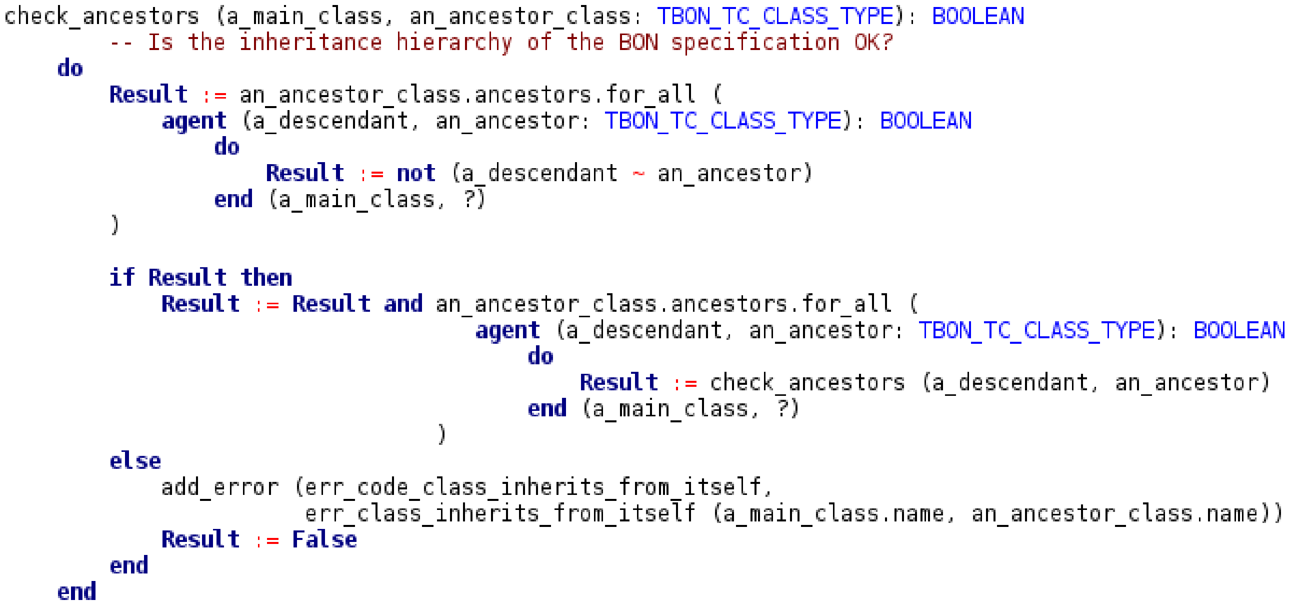
Inheritance relations are checked at the end of the second phase. Such relations cannot be resolved prior to this point, for the reason that not all relations between a class and its ancestors in the context can be expected to be known earlier than after the second traversal.

Static References

Static references used in client relations cannot be resolved in either the first or the second traversal of the abstract syntax, as the entire cluster structure has to be known before a static reference can be determined to be well-typed. The relations between a cluster and its classes and subclusters is not known until the second phase, and consequently, static references must be checked at the end of the second phase, when all of these relations have been established in the context.

Dynamic References

Comparable to the situation for static references, all the defined object groups and the relations between object groups and objects and other object groups must be known before a dynamic reference can be type checked. As these relations are explored in the



```
check_ancestors (a_main_class, an_ancestor_class: TBON_TC_CLASS_TYPE): BOOLEAN
        -- Is the inheritance hierarchy of the BON specification OK?
    do
        Result := an_ancestor_class.ancestors.for_all (
            agent (a_descendant, an_ancestor: TBON_TC_CLASS_TYPE): BOOLEAN
                do
                    Result := not (a_descendant ~ an_ancestor)
                end (a_main_class, ?)
        )

        if Result then
            Result := Result and an_ancestor_class.ancestors.for_all (
                                    agent (a_descendant, an_ancestor: TBON_TC_CLASS_TYPE): BOOLEAN
                                        do
                                            Result := check_ancestors (a_descendant, an_ancestor)
                                        end (a_main_class, ?)
                                 )
        else
            add_error (err_code_class_inherits_from_itself,
                       err_class_inherits_from_itself (a_main_class.name, an_ancestor_class.name))
            Result := False
        end
    end
```

Figure 4.3: Eiffel implementation of the *check_ancestors* algorithm

second phase, a dynamic reference cannot be type checked before the end of the second phase.

4.4 Inheritance

An important part of type checking textual BON is checking the inheritance hierarchy. Apart from checking whether an ancestor class actually exists, it is also important to check for circular inheritance. A class cannot inherit from itself, even through other classes. This means that the type checker has to check not only the direct ancestors of the class in question, but also all other ancestors and in the inheritance hierarchy. This is done with a recursive algorithm presented in figure 4.3.

In the first iteration, *a_main_class* and *an_ancestor_class* are both the root class. First, the direct ancestors of the *an_ancestor_class* is checked for the occurrence of *a_main_class*. If an occurrence is found, an error message is emitted and the algorithm terminates. If not, pass the algorithm will continue to check all ancestors of *an_ancestor_class*. This is done by recursively calling the feature with each ancestor of *an_ancestor_class*. It is important to note that *a_main_class* does not change through out the recursive calls — it is always checking in relation to the same class.

Notably, *Result* is expressed as a conjunction of itself and the result of further recursive calls. This does not change how the algorithm works, due to the conditional statement ensuring that *Result* always is *True* at that point. It is done to express that the value of *Result* does not only depend on the returned value of the expression, but also on the previous evaluation of the direct ancestors. This means that the conditional statement could have been excluded, but as it stops the algorithm from further unnecessary iterations when a type error has been found, it is not. Lastly, error handling is done by the error handling section of the type checker, which will be described further in section 4.8.

4.5 Informal BON

One could wonder why type checking of informal textual BON is interesting, since many parts of it does not involve types, such as queries and commands, and all logical expressions are expressed through the semantics of strings. There are, however, some interesting elements to inspect to ensure the integrity of the system.

Creation charts are checked by making sure that all creators and created classes exist in the context. Furthermore it will also warn the user of possible duplicate entries in the creation chart. This was decided to not be an error as it is not outright faulty type usage, but rather unnecessary. As such, the user could have a reason for keeping these duplicate entries for semantic reasons, and thusly raising a type error, because of this seems pedantic. Similar, scenario charts, and event charts are also checked for duplicate entries by name. Again this is expressed a warning instead of an error.

4.5.1 Structure

Most importantly, any cluster must be in exactly one system chart or in exactly one cluster chart. The type checker ensures this by checking that the cluster is not in more than one chart, and that it is mentioned in at least one chart. If *cluster occurrences* is defined as the number of times a cluster is mentioned in another cluster chart or in a system chart this can be expressed as such:

$$\neg(\textit{cluster occurrences} > 1) \wedge \textit{cluster occurrences} \neq 0 \ \equiv \textit{cluster occurrences} = 1$$

The first step($\neg(\textit{cluster occurrences} > 1)$) is done by the cluster having a reference to its parent (cluster) chart, and a flag indicating if a cluster chart is in a system chart called *is_in_system_chart*. When a cluster chart then tries to assign this reference, if it is already assigned to another chart, or the *is_in_system_chart* is set to *true*, the cluster must be mentioned in more than one system or cluster chart, and thus there is a type error. This also ensures that a cluster cannot be mentioned twice in the same chart. Thereafter, it is checked if all clusters are in a chart ($\textit{cluster occurrences} \neq 0$). Checking all clusters in the systems *is_in_system_chart* flags and references to their enclosing cluster charts does this. If any clusters reference is void, and the *is_in_system_chart* flag is set to *false*, then no system or cluster chart mentions it. If both these checks pass then the cluster is mentioned in exactly one cluster ($\textit{cluster occurrences} = 1$). The only thing left to check is that the cluster is not a subcluster of itself.

Similar to clusters, classes must be in exactly in one cluster. This is checked in the same way as the as it is done for clusters, by letting a class have a reference to its enclosing cluster.

4.6 Formal BON

4.6.1 Structure

Contrary to the structure of the informal specifications described in section 4.5.1, the structure imposed on formal specifications is less restrictive. In fact, there are only two restrictions, which will be elaborated upon in the following paragraphs. The reason for this, as already pointed out in section 2.3.2, is that formal specifications may be used to to give detailed descriptions of a system on the micro-scale, at which point the user probably does not want to recreate the entire system structure.

The first restriction is that a class or cluster can be declared as being a part of at most one cluster (zero is allowed). To ensure consistency of the formal specification, checking the cluster structure happens in the same way as for the informal structure: by emitting a type error if the *parent* attribute of a cluster (TBON_TC_CLUSTER_TYPE) or the *cluster* attribute of a class (TBON_TC_CLASS_TYPE) is assigned to more than once. Also, as for informal specifications, a cluster must not be a subcluster of itself.

The second restriction is that any cluster structure implied by a static reference must be explicitly defined. This is most easily explained with an example.

```
static_diagram STATIC_RELATIONS
component
      COLLECTIONS.CONTAINERS.LIST inherit SEQUENCE
end
```

Figure 4.4: Example of static references in use

In figure 4.4, an example of static references used in an inheritance relation is shown. For this example to be well-typed, the second restriction says that somewhere in the formal specification, the class LIST must be defined in a cluster CONTAINERS, which in turn must be defined as a subcluster of a cluster COLLECTIONS. Interestingly, the static reference on the right, SEQUENCE, implies no structure of the system, and could be defined anywhere in the system structure. It would have to be defined somewhere, though.

4.6.2 Static Relations

In continuation of the above discussion of static references, static relations constitute a valuable tool for emphasizing or make explicit important relations of the system. As the properties of static references (of which static relations are comprised) have already been attended to, these will not be addressed here. Instead, the specifics of inheritance and client relations, respectively, will be elaborated upon.

Inheritance Relations

An important thing to note about inheritance relations is that they do not define any previously unknown relations in the system. Their purpose is to emphasize and give more detail to already existing relations. Consequently, any inheritance relation stated in a static diagram must also be expressed in the **inherit** clause of the class specification of the child class. If it is not stated here, then the parent class must, be an ancestor to the child class through explicitly stated links. An example of such a relation is seen in figure 4.4.

Pertaining to inheritance relations is also an optional multiplicity marker, with which the user can underline the number of times the child class inherits from the parent class. As it does not make sense to state an inheritance relation in which the child class inherits less than once from the parent class, this marker is checked to always be greater than zero.

Previously mentioned in section 4.3.1 is the aspect that inheritance relations are marked as unresolved, and checked at the end of the second phase of type checking. Thus, provided that the static references of the inheritance relation is well-typed, determining if the relation is well-typed is a matter of checking that the child class exists in the context, and that the parent class is indeed an ancestor to the class through direct inheritance links. This is checked by a call to the feature *conforms_to* on the child class with the parent class as input. This feature is discussed in section 4.6.3.

Client Relations

A client relation between two classes tells a different and possibly more complex story than an inheritance relation. Because BON is a specification language with no knowledge of the actual implementations of the body of a feature, there may very well exist a client relation between to classes in the actual implementation of a system that is not made explicit (e.g. through feature types or feature arguments) in the textual BON specification. Accordingly, the well-typedness of a client relation is not determined on the basis of an explicit client-supplier relation in the specification. In fact, the type checker does not check this aspect at all.

Other aspects of a client relation can and should be checked, though. As a consequence of the above, all of these aspects pertain to the client class, as nothing can be said about the supplier, besides that it must be defined.

```
static_diagram CLIENT_RELATIONS
component
   CITIZEN client { drive, cruise, pick_up } CAR "Example 1"

   class CAR_OWNER[E] inherit CITIZEN
   CAR_OWNER client { make: E } CAR "Example 2"

   GARAGE_NUMBER inherit INTEGER
   CELEBRITY client { cars: MAP[GARAGE_NUMBER, ...] } CAR "Example 3"
end
```

Figure 4.5: Examples of client relations

In particular, the feature names potentially mentioned as client entities in the client relation in question must exist in the interface of the client class. The interface is the set of all the features, including inherited ones, that can be called on a class. Example 1 in figure 4.5 shows how features can be used in a client relation. All the three features mentioned in this example must be present in the specification of class CITIZEN, which is not shown here, for the client relation to be well-typed. Accordingly, any formal generic names mentioned as a client entity must exist in the client class as well. This is implemented by a single call to the feature *has_generic_name* of TBON_TC_CLASS_TYPE, which iterates through the generics of the class. A formal generic name E is used as a client entity in example 2 in figure 4.5. As shown, this formal generic name is present in the class specification of CAR_OWNER.

If named indirections are specified as client entities, things become a little more involved. Naturally, every celebrity needs to keep track of his or her cars, even when they are distributed over multiple garages. Example 3 in figure 4.5 shows how this could be expressed. For this relation to be well-typed, a class MAP must exist, and it must have two type parameters. Furthermore, GARAGE_NUMBER must conform (covariantly) to the bounding type of the first of these parameters. And because the ellipses refer to the supplier type of the relation [WN95, p. 371], the supplier type must conform to the bounding type of the second parameter. These conformance checks are implemented by calls to the feature *conforms_to*, the semantics of which are explained next.

4.6.3 Variance

As described previously in section 2.3.2, the type checker implements covariant conformance for feature types and feature argument types. Furthermore, as will be seen shortly when generics are discussed, this also holds for instantiation of generic classes with respect to bounding types of generic parameters.

```
conforms_to (other): BOOLEAN
 do
   Result := name of Current is equal to ANY
   if not Result and name of other is not NONE then
      if name of Current is equal to name of other then
        if Current.generic_count = other.generic_count then
             if Current.has_actual_type and other.has_actual_type
                Result := Current.instance_equality (other)
             elseif Current.has_actual_type xor other.has_actual_type then
                Result := False
             else
                Result := Current.generics = other.generics
             end
        end
      else
        Result := exists an ancestor a' of Current such that a'.conforms_to(other)
      end
   end
 end
```

Figure 4.6: Pseudocode for *conforms_to*

The conformance check is implemented in the *conforms_to* feature of TBON_TC_CLASS_TYPE, the pseudocode for which is shown in figure 4.6. Seeing that no other types of variance is implemented anywhere in the type checker, all conformance checks happens through a call to this feature. Because all classes conform to ANY, including ANY itself, the algorithm first checks whether the input is ANY. If so, nothing remains to be done. Likewise, if the input is NONE, the algorithm terminates with the result being *False*.

In all other cases, if the names of the current class and the input are equal, and the number of generics are equal, the algorithms proceeds. If the number of generics are not equal, the two classes can never be equal (even though their names are), because the definition of type parameters is an integral part of the class specification.

Next (on line 7), the algorithms checks whether both the current class and the other class has actual types. The semantics of a call to *has_actual_type* on a class are as follows:

$$\exists\ g \in \mathit{Current.generics} \mid g\text{: } \text{TBON\_TC\_GENERIC} \bullet g.\mathit{actual\_type} \neq \mathit{Void}$$

Thus, if at least one of the generics of the current class has an actual type, the same must hold for the other class. Actual types will be explained in-depth in section 4.6.5. In this case, instance equality must hold between the two classes. Instance equality is defined as a check that checks whether the actual types of two generic classes are identical. Because, as defined in section 2.3.2, no conformance relation is considered between two instances of the same generic type with different concrete type arguments, instance equality between two classes only holds if the concrete types are identical.

If only one of the classes have actual types (line 9), they can never conform to each other, as the class with no actual types would represent a class specification, while the other would represent an instance of that class. Accordingly, if neither the current class nor the other class have any actual types, they are both class specifications, and as such, they are considered to be equal if their names are equal and their generics (concerning formal generic names and bounding types of these) are equal (line 12).

In the end (line 16), if the other class is not ANY or NONE, and not equal by name to the current class, the algorithms call itself recursively on all the ancestors of the current class, until the entire inheritance hierarchy has been traversed.

As a consequence of the above, one need only to change the logic of *conforms_to* in order to implement contravariance for feature types, for instance. It should be noted, however, that the current implementation implements covariance for both feature types and feature argument types. Should the variance for these two be unequal in the future, one would need change the feature calls in the type checker accordingly.

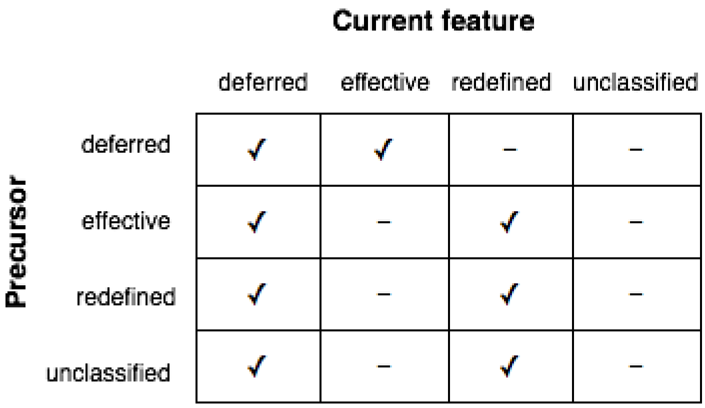
Instance equality is implemented in the feature *is_instance_equal* in TBON_ TC_CLASS_TYPE. To implement a conformance relation between two instances of the same generic class, one would only need to change the logic of this feature (provided that the implementation of *conforms_to* remains unchanged). Hence, the logic for creating a relation between LIST[INTEGER] and LIST[REAL] could be placed here.

4.6.4 Features

At the heart of a class specification are its features, the relations of which are discussed in this section. Features are represented in the type checker by the class TBON_TC_FEATURE. As mentioned in section 4.2, a class in the type context is associated with a set of instances of these. In turn, a feature is associated with its arguments through a list of instances of TBON_TC_FEATURE_ARGUMENT.

Feature Status

Whenever a feature exists in a class specification in which ancestors are specified, it can potentially have a precursor. Luckily, because features repeatedly effect, undefine, and redefine each other all the way down the inheritance hierarchy, only the nearest precursor feature needs to be considered when determining whether a feature specification is well-typed. The status of a feature is implemented by simple boolean flags in TBON_TC_FEATURE; and as specified in the grammar in [WN95], only one (or none) of these can be set at any given time.



		Current feature			
		deferred	effective	redefined	unclassified
Precursor	deferred	✓	✓	–	–
	effective	✓	–	✓	–
	redefined	✓	–	✓	–
	unclassified	✓	–	✓	–

Figure 4.7: Allowed status relations between a feature and its precursor

Figure 4.7 shows the allowed relations between the status of the precursor and the status of the current feature (i.e. the feature whose well-typedness is being determined). Notably, what it shows is that it is always possible to undefine an inherited feature by creating a *deferred* version of it in the current class, regardless of the status of the precursor. This becomes relevant in any situation in which one wants to ensure that all concrete descendants of the current class redefines (or, more precisely, effects) the undefined feature.

As for the *effective* status, the restrictions for it may seem overly constraining at first, when considering that the original specification of it states that "[the effective marker] may also be used just to emphasize that the feature will have an implementation," [WN95, p. 40]. However, figure 4.7 only states that *if* a precursor exists for an *effective* feature, then it must be *deferred*. If no precursor exists, one may use it freely.

The *redefined* status is defined as one would expect from the definition in [WN95, p. 40]. The only noteworthy aspect is that if a precursor is *deferred*, the current feature must be marked as *effective*, as conceptually, one cannot redefine something that has yet to be defined.

For unclassified features, the definition is straightforward: an unclassified feature defined in the current class can never have a precursor. An unclassified feature inherited from an ancestor can be redefined and undefined as expected, though.

Aggregation

As specified in [PO01], a feature cannot introduce aggregation with its enclosing class. Allowing this would lead to an infinitely recursive definition of the class in question, making the class specification inconsistent. Also, such a specification would never be able to run as intended when converted into actual code, as a compile-time or run-time error would most likely occur. Checking this aspect is quite simple: whenever aggregation is used, the type checker makes sure that the feature type is not equal to the enclosing class.

Prefix and Infix Features

For regular features, having two features with identical names in the same class (either directly defined in the class or inherited) is not allowed; this is not true for prefix and infix features. For instance, it should be possible to define the operator "+" as both a unary and a binary operator on an object, e.g. an integer. The really interesting aspects of prefix and infix features arises when they are used, though, which will be described in section 4.6.6.

4.6.5 Generics

In textual BON, formal specifications can involve generics in several different ways:

- Class specifications, e.g. LIST[E]
- References to formal generic names of enclosing class,
  e.g. *first_element*: E
- Generic class specifications instantiated with concrete types,
  e.g. *string_list*: LIST[STRING]
- Generic class specifications instantiated with formal generic names of enclosing class, e.g. *element_list*: LIST[E]

Note that the first and the last example both refer to LIST[E], but in very different respects: the former is a specification of a new class LIST with one (unbounded) type parameter E, while the latter is an instantiation of the same class, in which it is assumed that the enclosing class of the feature *element_list* has a (possibly bounded) type parameter E. How each of these types of generics are handled by the type checker is explored in this section.

Generics in Class Specifications

As mentioned in section 4.2, classes in the context are associated with type parameters through a sequence of instances of TBON_TC_GENERIC. The class TBON_TC_GENERIC has two notable attributes: *bounding_type* and *actual_type*. The attribute *bounding_type* represents the type bound of the formal generic name of the type parameter. Accordingly, the *actual_type* represents the concrete type associated with the formal generic name.

```
static_diagram
component
      class NUMBER_LIST [E -> REAL]
end
```

Figure 4.8: Example of a generic class specification with type bounds

In the type context, no generic class specifications have any actual types associated with any of its type parameters; these are only present in instances of these, which will be discussed next. To exemplify, consider the case presented in figure 4.8. Here, the class NUMBER_LIST would exist in the type context, and have a single type parameter with a formal generic name E and a *bounding_type* attribute pointing to the type REAL (which would also exist in the type context in order for the specification to be well-typed). The *actual_type* attribute of the type parameter would point to *Void*, as this example is not an instantiation of the type in question, but a specification.

As a result of this, and to differentiate between them, there is no equivalence relation between a class specification (without any actual types) and an instance of this class (with actual types), as they in practice are two different types. They are not unrelated, however, as will be explained next.

Instantiation of Generic Types

Because a generic class specification only specifies a family of concrete classes that have a common base type, it has to be instantiated with concrete types to be of any practical use. One could say that a generic class specification merely acts as a template for such concrete instances (in some languages, such as C++, generic classes are explicitly denoted *templates* [Str97]), and therefore it is not possible to refer directly to a generic class specification in places where a concrete type is expected. In textual BON, these places include feature types, feature argument types, formal assertions, and interestingly, type bounds in generic class specifications. In order to instantiate a generic type, a mechanism that ambiguates between the generic class specification and an instantiation of it is needed, while still ensuring that the instantiated type has all the same attributes (e.g. features) of the generic class specification.

All types eligible for participation in the instantiation of a generic type are found in the type context after the first phase. Whenever an instantiation of such a type occurs, for instance as a feature type, the type checker [...]:

1. Creates a deep copy of the generic class

2. Registers the copy in the generic class as an instance of that class
3. Adds the concrete type parameters as actual types to the generics of the copied class

In practice, the first two steps happen as a single call to the *new_instance* feature of the generic class, the specification of which is found in the type context. As such, the individual instantiations of a generic class are not added directly to the type context. Instead, each generic class keeps track of all the instantiations that have been made of it. The benefit of this approach is that whenever the generic class is updated, i.e. an ancestor or a feature is added in the second phase, the generic class can update all of the instantiations of it accordingly. Consequently, when an instance is registered as an instance (step 2 in the above enumeration), one can think of it as an observer observing the generic class, which then notifies/updates the instantiations whenever a change occurs.

In step 1 above, it is noted that a *deep* copy is created. This is an important aspect because, as previously explained, type parameters of a class are implemented by associating a type with a sequence of instances of TBON_TC_GENERIC. If only a shallow copy was created, the copy would refer to the exact same type parameters as the original. This would make it impossible to add individual concrete actual types to each of these parameters for each instance without affecting the original generic specification.

References to Generic Types

The instantiation of generic classes explained above happens every time a reference to a concrete type is expected and a generic type is provided in the abstract syntax.

```
static_diagram
component
      class MAP [K -> INTEGER, V]
      class MAP_CLIENT
         feature
            string_map: MAP [INTEGER, STRING]
         end
end
```

Figure 4.9: Example of a reference to a generic type

In the example in figure 4.9, an instantiation of the generic type MAP is shown. At the time the type checker resolves the feature *string_map* (at the end of the first phase), the generic class specification (with no actual types, although the first type parameter has a type bound of type INTEGER) of the type MAP exists in the type context. To resolve the feature, the resolving code recognizes that MAP has type parameters, and therefore creates a new instance of MAP to which the feature can refer (steps 1 and 2 in the previous section). Furthermore, the concrete type INTEGER is added as actual type to the first parameter of the new instance, and STRING is added as actual type to the second parameter. As such, the feature *string_map* now refers to a separate instance of MAP, which has all the features of the generic class from which it originates plus the added actual types.

In order for this new instance to be well-typed, the type of the first concrete parameter type must conform covariantly to the bounding type of the first parameter of the class specification, K (just as defined for feature types and feature argument types in section 2.3.2). Textual BON has no construct for differentiating between different types of variance for type parameters, which is known from languages such as Java and C#, and enforcing covariant conformance is thus a design decision on the part of the type checker. This decision mainly based upon the fact that the Eiffel language also enforces covariant conformance of type parameters [Mey01, Constrained genericity]. Future versions could include a switch to fine-tune this aspect of type checking.

As described in section 4.6.3, this conformance check also happens through the feature *conforms_to*. In the presented example, the concrete type trivially conforms to the bounding type, as they are both INTEGER. Some non-trivial situations, however, require a little more consideration.

Solutions to Non-trivial Generics Situations

The first non-trivial situation considered is presented in figure 4.10.

```
static_diagram
component
   class SEQUENCE [G -> REAL]
   class CLIENT_CLASS [H]
      feature
         seq: SEQUENCE [H]
      end
end
```

Figure 4.10: Situation 1) Example of non-conforming type parameter bounds

In this example, the feature *seq* instantiates the class SEQUENCE with the type parameter H of the enclosing class CLIENT_CLASS. The type parameter of SEQUENCE is bounded by REAL, while the parameter H has no bound; so how can it be determined if H is a legal parameter, which makes the instantiation well-typed? Intuitively, one might say that if the type parameter has no bounding type, nothing at all can be said about it – it could be anything. And by following that intuition, the answer is given: if H has no explicit bounding type, it is implicitly bounded by ANY, which is the base class of all classes. This mimics the behaviour found in Eiffel, in which unconstrained type parameters are also considered to be bounded by ANY [Mey01, p. 77].

Now, is the example then well-typed? No, because the concrete type parameter of an instantiation must conform to the bounding type of the type parameter in the class specification, and as the only thing known about H is that it has type ANY, the example is not well-typed, because ANY does not conform to REAL.

The second situation contain similar, but different, subtleties.

```
static_diagram
component
   class SEQUENCE[G]
   class A [E -> SEQUENCE[REAL]]
   class B [M -> SEQUENCE[INTEGER], N -> A[M]]
end
```

Figure 4.11: Situation 2) Specification of a generic class with generic type bounds

Firstly, the differences between the three occurrences of the class SEQUENCE should be noted (also note that in this example, the type parameter of SEQUENCE has no bounding type). Secondly, the example shows a type parameter N that is bounded by a generic class A instantiated with the first parameter of class B, M. Instantiating a bounding type with another type parameter is permitted, as long the type parameter used in the instantiation is declared at the time of its use (i.e. the type parameter appears to the left of the type parameter that uses it in its specification).

What happens in this case is that M is considered to be a sequence of INTEGER, while the type parameter E of A is bounded by a sequence of REAL. At the time of the instantiation of class A for the bound of type parameter N, the bounding type of M has to conform to the bounding type of A in order for the instantiation to be well-typed. But, as already defined in section 2.3.2, the type checker does not consider a sequence of REAL to conform to a sequence of INTEGER, even though INTEGER might be a subtype of REAL. Consequently, to make this situation well-typed, one would have to change the bounding type of either type parameter E or M, such that they become equal.

4.6.6 Formal Assertions

With formal assertions, an vast amount of expressive power is added to the textual BON language. They make detailed specifications and contracts possible, adding extra formality to formal specifications. Interestingly, this part of the language is very underspecified, making it impossible to type check without imposing quite an amount of restrictions on it. While some of these have already been presented in section 2.3.2, other are presented for the first time here, as they need to be accompanied by detailed explanations.

Variable Context and Scope

For keeping track of the variables introduced in a quantification, a variable context is introduced. This context is implemented as a hash table mapping a variable name to an instance of TBON_TC_CLASS_TYPE.

For the reason that the grammar in [WN95] specifies that variables can only be introduced in member or type range expressions in a quantification, discussing variable scope is only relevant for these. The implemented scoping rules for quantifications defines all variables in the same assertion clause to have the same scope. Hence, two boolean expressions in two distinct assertion clauses have distinct scopes.

```
 1| static_diagram
 2| component
 3|  class A
 4|   feature
 5|     list_one: LIST[INTEGER]
 6|     list_two: LIST[INTEGER]
 7|   invariant
 8|     for_all i: INTEGER; i member_of list_one it_holds
 9|       exists i: INTEGER; i member_of list_two it_holds i = i;
10|         -- The above introduces ambiguity;
11|     exists i: INTEGER; i member_of list_one it_holds i = 7;
12|         -- This is fine
13| end
```

Figure 4.12: Example of consequence of nested scoping

The consequence of this definition is that no local scope exists for individual quantification expressions within the same assertion clause. Allowing local scope would mean that the first assertion clause (lines 8-9) in the invariant in figure 4.12 would be well-typed. However, the proposition of the existential quantification of the first assertion clause leaves ambiguity between the origin of the two occurrences of the variable *i*. Do they both represent the inner *i*, in which case the proposition is a tautology? Or does one of them represent the outer *i*? Disambiguation is impossible. Although it could have been decided to implement hiding, in which the second definition of *i* would hide the first definition, this is deemed unnecessary and thus disallowed. There is rarely a reason for insisting on having identical variable names in a nested quantification, especially if the consequence is less expressive power due to the loss of access to hidden variables. Also, as seen on line 11 in the example, it is quite possible to use the same variable again in one of the other assertion clauses of the invariant.

Operator Expressions

Operator expressions are expressions involving unary or binary operators. Although the grammar in [WN95] specifies than an operator can also be an arbitrary parenthesized boolean expression, this specification has been reworked as described in section 2.3.1 in order for the grammar to type check.

As previously mentioned in the Features subsection, unary/prefix and binary/infix operators are implemented as instances of TBON_TC_FEATURE with either the *is_prefix* or *is_infix* flag set (but never both). In consequence, for any operator expression in a formal assertion to be well-typed, a prefix or infix feature for the types involved in the expression must be defined in the system. This is true even for default types (described in section 4.9), for which all operators are implicitly defined in the type context. For unary operators, this (quite obviously) means that the operator must be defined as a prefix feature in the class specification of the operand. For binary expressions, it means that the operator must be defined as an infix feature in the class specification of the *left* operand, and the feature must take a single argument of the type of the *right* operand. This is illustrated in the following example:

```
 1| static_diagram
 2| component
 3|    class A
 4|       feature
 5|         infix "+": INTEGER
 6|           -> other: B
 7|          b: B
 8|       invariant
 9|         (Current + b) > 0; -- OK;
10|         (b + Current) <= 10; -- Type error!;
11|    end
12|    class B
13| end
```

Figure 4.13: Definition and use of an operator expression

The example in figure 4.13 shows that because the infix feature is defined in class A, and takes a B as its argument, the first expression on line 9 is well-typed. The second expression on line 10 is not, however. For this expression to be well-typed, a similar feature taking an A as its argument would have to be specified in class B. While not implementing symmetry for binary operators may seem restrictive at first, consider that the expression could involve any binary operator. If, for instance, a user were to implement the *member_of* operator for a custom set type, having both "*var member_of* SET_TYPE" and "SET_TYPE *member_of var*" being legitimate expressions would most likely be undesired.

Checking that a binary operator expression is well-typed is implemented by finding the corresponding infix feature in the interface of the class of the left operand and checking that the type of the right operand conforms to the type of the feature argument (by a call to *conforms_to*). For unary expressions, the prefix feature corresponding to the operator must merely be present in the interface of the type of the operand.

Calls

Determining which calls are allowed to be made in an assertion clause depends on where the assertion clause is placed. The first call in a call chain in a feature contract can be to any feature in the interface of the enclosing class (aside from the current feature itself) or to any of the arguments of the current feature. If in a class invariant, only the features in the interface of the current class can be called as the first call in the call chain. These are the rules for calls without a parenthezised qualifier. Is a parenthesized qualifier present, only the features in the interface of the type of the parenthesized qualifier expression can be invoked as the first call.

Beyond the first call in the call chain, the general rule is that a feature that is called as call *n* in the call chain must be in the interface of the type of call *n-1*. This is clarified by the following example:

```
 1| static_diagram
 2| component
 3|  class A
 4|    feature
 5|       infix "+": B
 6|          -> other: B
 7|          require
 8|             other /= Void
 9|          ensure
10|            Result.id = 10
11|          end
12|        b: B
13|    invariant
14|      b /= Void;
15|      (Current + b).id > 5
16|    end
17|
18|  class B
19|    feature
20|       id: INTEGER
21|    end
22| end
```

Figure 4.14: Using calls in assertions

Figure 4.14 shows several interesting aspects. First of all, it shows that feature arguments should be accessible from pre- and postconditions of a feature and that attributes of the type of a feature should be accessible through the *Result* keyword. A more subtle point is shown on line 15: because the "+" operator called with *Current* as left operand and *b* as right operand returns a type B, the feature *id* of B is available for comparison in the invariant of A. In fact, disregarding equality checks, the only feature that can be called on the parenthesized qualifier on line 15 is *id*, as this is the only feature of B.

Type checking calls is done by resolving the type of the call or the parenthesized qualifier through a call to the feature *type_of_expression* in the type checker, the details of which are explained later in this section. When the type has been resolved, it is checked that only allowed features are called on the resolved type. Does the call involve features with arguments, the concrete types given as actual arguments must conform to the types specified in the specification of the feature in question (checked through *conforms_to*).

Besides the uses that have already been shown, calls can also participate in set expressions, at which point an extra layer of checking is added to them. This, amongst other issues of type checking set expressions, is discussed next.

Set Expressions

Set expressions are introduced in the grammar in member range expressions of quantifications, an example of which is shown in figure 4.12 as the right operand of the *member_of* operator. Basically, two types of sets can appear in set expressions in textual BON: enumerated sets and enumerable types.

Enumerated sets Enumerated sets are explicitly defined sets in which all the elements are explicitly listed. An example of such is shown below.

{ 7.5, 10, 117, 3.14159 }

To check the type correctness of such a set, the type checker enforces that the all the types represented by the elements of the set must conform to the same common type. This common type has to be the type of an element in the set. To clarify, all of the elements in the example set above must conform to the same type, REAL in this instance, assuming that numbers with a decimal component are represented by REAL. Further assuming that INTEGER is a subtype of REAL, the rule holds for the example, as the first and the last element are of type REAL (which trivially conforms to itself), and the second and third element are of type INTEGER. Thus, all the elements conform to REAL, and the enumerated set is well-typed.

This is only half the story, though, because as already mentioned, set expressions are only used in member range expressions. As such, the type of the enumerated set must conform to the type of the variable defined as the left operand of the *member_of* operator – the user should not be able to iterate through a set with a variable whose type does not match the types of the elements in the set. Also, to avoid placing unnecessary burdens on the user, the left operand of the member range expression will automatically be assigned the type of the enumerated set, if it does not have a type prior to that point.

Calls and operator expressions As calls and operator expressions used in set expressions are handled in almost the exact same way in the type checker, they will both be described here. If a set expression is not an explicitly defined enumerated set, it must be a reference to an expression (either a call or an operator expression) whose type is enumerable. An example of this is shown in figure 4.12, in which both *list_one* and *list_two* are references to the type LIST[INTEGER], which is considered to be enumerable as it is a part of the standard types defined in the type checker (see section 4.9 for details).

But what does it take for a type to be considered enumerable? Any type, even standard types defined by the type checker, is considered to be enumerable if it is defined as a subtype of the type ENUMERABLE, which is defined by default along with the other standard types. Hence, if a user wants to define a custom type as being enumerable, it should merely inherit from this class. The type checker does not emit a type error if one tries to use a type in a set expression which does not conform to ENUMERABLE, though, as this mechanism is intended to be a help, not a nuisance. Instead, a friendly warning is given, bringing to the user's attention that the semantics of the member range expression might not be as expected.

The type of the variable in a member range expression when using enumerable types should be the same as or an ancestor of the type of the actual type of the first argument of the enumerable type (if the enumerable type is generic). For example, if the enumerable type is SET[STRING], the type of the variable should be STRING. If an enumerable type is used for which the type of the elements cannot be determined, a warning is emitted. For instance, it might be apparent to a human that the type STRING_LIST is a list of strings, but the type checker will have no idea that this type is composed of elements of type STRING. Thus, a warning is given, as it cannot be determined for certain that a type rule has been violated.

Resolving the Type of an Expression

In all of the preceding sections on formal assertions, it has been assumed that the type of an expression is always available. While this is correct, no magic has been involved — every time, the feature *type_of_expression* has been invoked.

The types of all manifest constants naturally map to the standard value types defined in BON [WN95, p. 51]: BOOLEAN, CHARACTER, INTEGER, REAL, and STRING. These are also defined as standard types in the type checker, as described in section 4.9. Also, because a quantification is a logical expression, it always has type BOOLEAN.

Because both calls and operator expressions (by virtue of prefix and infix features) are modeled as instances of TBON_TC_FEATURE, the type of one of these is resolved by calling the *type* attribute of its corresponding TBON_TC_FEATURE instance, naturally assuming that the call or operator expression is well-typed.

For the *Result* keyword, the type is always the type of the current feature. The current feature is known at any time when checking formal assertions, except when there is no current feature, i.e. when checking invariants. If *Result* is used anywhere but in a postcondition of a feature, a type error occurs. The situation for the keyword *Current* is analogous to the situation for *Result*: its type is always the type of the enclosing class, which is always known by the type checker. Contrary to the *Result* keyword, *Current* can be used wherever a type is expected. The type of a *Void* constant should be self-explanatory.

Quantifications and Boolean Types

The previous section briefly mentioned that a quantification always has boolean type. However, looking only at the grammar in [WN95], restrictions and proposition of a quantification could have any type, seeing that calls, operator expressions, and constants of arbitrary type are all allowed for these. As such, as also stated in section 2.3.2, the type checker restricts restrictions and propositions to be one of the following:

- A boolean constant; *True* or *False*
- Another quantification
- An operator expression with BOOLEAN type, e.g. a comparison
- A call with BOOLEAN type

Considering that the above items are the only elements that can have boolean type in a formal assertion, this restriction does not remove any expressive power from quantifications. Rather, they ensure that a quantification makes sense. In the actual implementation, what happens is that *type_of_expression* is called on the expression of both the restriction (if present) and the proposition of a given quantification. The type checker then makes sure that the resolved type is BOOLEAN.

4.7 Dynamic Diagrams

A dynamic diagram contains a set of a scenario descriptions, object groups, object stacks, objects, and message relations. In this section, the representation (the context) and the checking of these will be explained

The dynamic object representation in the context is rooted the deferred TBON_TC_DYNAMIC_OBJECT class. All known dynamic diagram components are subtypes of this class. Not all the aforementioned kinds of dynamic diagrams are represented directly by a class in the context though. Since an object and an object_stack are syntactically equal, except for the keyword itself, they do not need separate representations. Consequently, they are both represented by the TBON_TC_OBJECT class. Lastly, message relations are not represented in a designated object, since they only consist of dynamic references. With this simple structure of message relations, checking them only requires checking the dynamic references, and as such a dedicated object for this element is unnecessary. Scenario descriptions and object groups are both represented directly as they appear in the grammar.

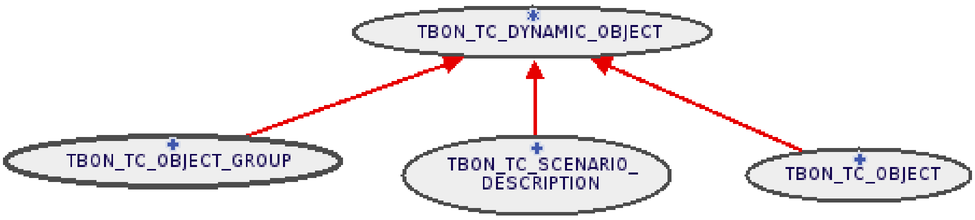


Figure 4.15: Inheritance structure for dynamic diagrams.

When checking dynamic diagrams, because of the object based nature of the diagrams, as opposed to the class based nature of the other kinds of specifications, two new contexts are introduced. The object context keeps track of the objects in a dynamic diagram. Objects mentioned by message relations must be defined as objects or object stacks. The object context is used for checking this. Similarly, any scenario mentioned by a message relation must be defined by a scenario description. When encountering new scenarios, the type checker adds them to the context, in which it can then check if they are present when mentioned in message relations.

Even though some elements are not represented by the a designated class in the context, they still need to be type checked as separate entities. In order for a scenario description to be well-typed, its name must be unique. Having more than one scenario with the same name would ruin the semantic value of describing a scenario. Labeled actions of scenarios are represented by strings, however integer ranges are supported by the type checker (as seen in [WN95, pp. 379-380]). When parsing strings of the structure INTEGER "–" INTEGER, the type checker will split up the range and create multiple scenarios, one for each number in the range. The example in figure 4.16 would therefore result in nine different scenarios. Everything other than integer ranges are just parsed directly as strings. Only the scenario labels (e.g. "1-3") are interesting for the type checker as the description of a scenario can have any string value.

```
dynamic_diagram Start_car
component
   scenario "Scenario 1: Start the car"
   action
      "1-3"     "Put key in keyhole"
      "4-5"     "Turn key"
      "7-9"     "Drive"
    end
end
```

Figure 4.16: Example of use of integer ranges in scenario descriptions.

Similar to scenarios, object groups must have unique names, or be nameless. Furthermore all its contained elements (dynamic components) must also be well-typed, in order for the object group to be well-typed.

Due to their similarity, objects and object stacks are checked in the same way, however not by the same feature. Had they been in the same feature it would have meant complications when expanding the system, or changing the semantics of object stack. Having separate features also allows for cleaner code, as different error messages can be emitted. Well-typed objects and object stacks must have unique names.

A message relation consists of two dynamic references and a scenario, which both must be well-typed. Both dynamic references must be as an object or object stack, in the object context. Message labels must be identical with one of the labeled actions in the dynamic context in order to be well-typed. The example seen in [WN95, p. 380] is therefore not considered well-typed by the type checker.

```
 1| dynamic_diagram open_fridge
 2| component
 3|    scenario "Scenario 2: Open fridge"
 4|    action
 5|       "1"       "Grab handle"
 6|       "2-3"     "Pull door"
 7|       "4"       "Look at the food"
 8|     end
 9|     object FRIDGE.door_handle
10|     object HAND
11|     object EYES
12|     object FRIDGE.content
13|     HAND calls FRIDGE.door_handle "1"
14|     HAND calls FRIDGE.door_handle "2-3"
15|     EYES calls FRIDGE.content "4: Is there any bacon?"
16| end
```

Figure 4.17: Example of a scenario chart that is not well-typed.

The example in figure 4.17 is not well-typed for two reasons. In line 14 the scenario mentioned is labeled “2-3”. While this is identical to the string in line 6, the the type checker has parsed this into two different scenarios with labels “2” and “3”. To reference these labels one would therefore have to use strings “2” and “3” and not “2-3”. Other ways of expressing sets such as "2, 3" or "[2; 3]" are not parsed, and will be considered as manifest strings. The other reason can be found in line 15. While it is suggested by Waldén and Nerson that this structure is allowed it has not been included in this implementation.

4.8 Error Handling

Whenever a type error occurs during type checking, an error is emitted by the type checker. Such an error consists of two elements, an error code and an error message. Accordingly, the error system of the type checker is composed of two classes: TBON_TC_ERROR and TBON_TC_TEXT_ITEMS. TBON_TC_ERROR is the main error class, which provides a creation procedure for creating new errors and lists the error codes, while TBON_TC_TEXT_ITEMS contains the error messages. Error codes are mainly needed for the purpose of testing, which should not rely on checking the value of a manifest string. The errors are accumulated as a list of TBON_TC_ERROR instances, where an error is added by a call to *add_error*, e.g.:

add_error(err_code_class_does_not_exist, err_class_does_not_exist (class_name))

The error messages are output at the end of the second phase. However, one type rule violation might prevent the execution of a later check. Instead, it stops checking and prints the accumulated error messages. This means that the list of errors returned by each execution of the type checker is not necessarily comprehensive, as solving one error might lead to another. The reason for this is that solving one type error may allow the type checker to check an aspect of the specification that was not previously checkable.

The type checker succeeds silently, meaning that printing error messages is the only way the type checker communicates that the specification analyzed was not well-typed. Therefore, if no errors occur/are printed, the analyzed specification is well-typed. A well-typed specification can have have elements for which the type checker cannot be sure that a type error has not been violated, though. At these occasions, warnings are emitted. Warnings bring to the user's attention that some aspects of the specification might not be as expected. For instance, a warning is emitted if a type that is not marked as ENUMERABLE as used in a set expression. While this is not a type error per se, it cannot be determined whether it is not a type error. At other occasions, e.g. if two scenarios with identical names are found in a scenario chart, the warning functions as a reminder that something might be wrong.

4.9 Standard Types

To ease the creation of well-typed specifications, an array of standard types are defined by default by the type checker. The consequence of type checking is that all classes used in a specification must be defined in either a class chart or a class specification in order for the specification to type check. But this also means that standard types such as the ones listed below must be defined *every* time a new specification is made – otherwise, the type checker will complain about undefined types. Hence, the motivation for defining these types is that it enables the user to focus on his own specification, without having to worry about pleasing the type checker by defining standard types which for any specific system are not very interesting. The following standard types are defined by the type checker:

- ANY
- NONE
- BOOLEAN
- CHARACTER
- INTEGER
- REAL
- STRING
- `deferred` ENUMERABLE
- `deferred` CONTAINER[E]
- LIST[E]
- SET[E]
- TABLE[K, V]
- ARRAY[E]
- TUPLE[G, H]

A textual BON specification of these is found in appendix C. The implication of defining these types, though, is that the user *cannot* define types with these names himself, even if he so desired, as this would lead to a situation where a type is defined more than once (which leads to a type error). While this might be considered restrictive, this rule exists to ensure integrity of the type context. For instance, if the type BOOLEAN is overwritten by the user, the type checker has no way of knowing whether that type still represents a *True* or *False* value. Consequently, all use of boolean values in assertions would lead to undefined behaviour. This is true for the other standard values as well. Also, it is always possible to inherit from one of the standard types, in effect extending its behaviour (although it still has to be under another name).

As discussed when set expressions were explained, all enumerable types inherit from type ENUMERABLE. Of the other standard types, LIST, SET, and ARRAY inherit from this class. A type TABLE has a set of keys and a set of values, and these are individually enumerable as well.

In practice, the standard types behave just as if the specification in appendix C was parsed and type checked at the beginning of every execution of the type checker. Doing this in each execution would incur unnecessary overhead, though, and therefore these are added directly to the type context at the beginning of an execution of the type checker. This can be seen in the feature *initialize_contexts* in the type checker.

Chapter 5

Further Development

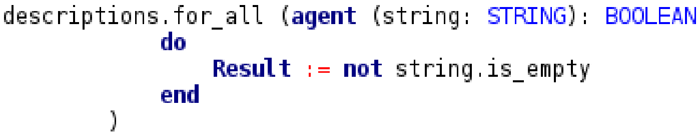
As mentioned in the introduction, the purpose of this project is to provide a foundation for more textual BON related work in EiffelStudio. This chapter will go over some suggestions for future projects that could be in continuation of this project. This, however, is but a segment of all possible project, as such, the purpose of this chapter is to give other developers an idea of what could be done.

5.1 BON Extractor

If the BON extractor is to be developed further, there are a number of interesting features that could be added.

Currently, the BON extraction treats all types the same way. Every type is just mirrored into the generated BON without further analysis. An interesting addition could be to analyze the inheritance of classes to see if they were of a certain kind. For example, if a class inherits from the SET class, it is known to have the attributes of a set. The extractor could then transform the type in question into one of the predefined standard type, with the same attributes as a set. This, however, should not be forced upon the user, but rather be implemented as a feature that could be disabled at will.

In continuation of the above, if the extractor was able to recognize certain standard types it would also be able to take use of the quantifications available in the textual BON language. When a feature call such as *for_all* is encountered in the Eiffel source code the extractor could translate this into textual BON using the *for_all* keyword. For instance, the quantification in figure 5.1 could be translated into textual BON in figure 5.2.



```
descriptions.for_all (agent (string: STRING): BOOLEAN
            do
                Result := not string.is_empty
            end
        )
```

Figure 5.1: An example of an Eiffel quantification.

for_all *str*: STRING; *str* **member_of** *descriptions* **it_holds** **not** *string.is_empty*

Figure 5.2: How figure 5.1 could be translated in textual BON.

```
Invariant                Constraint
     f /= Void                 "Current must have an f"
```

This could cause issues when agents are more advanced that in this example. In an assertion, not everything from the agent is interesting to analyze, and as such not everything should be included. The extractor could analyze what the *Result* keyword relies upon and interpret that. This could easily lead to even more currently unknown complications, which is why this feature is not included in this project.

Another interesting feature to add could be the ability to switch to textual BON view when viewing a cluster. In EiffelStudio the user can select a cluster and see metadata about it. From this view an option to switch to both formal and informal textual BON should be available. This could extract the textual BON from the classes contained in the cluster, and would add to the extended overview of the code base that the BON view provides.

There is no way of expressing void safety directly in BON, only through invariants, and pre- and post-conditions. Therefore the *attached* keyword does not translate to textual BON in the current implementation. A possible extension could be to let the project settings from Eiffel be expressed in the extracted textual BON by generating constraints for attached features. An attached feature *f* could then translate into formal and informal BON in the following way:

Last but not least, the textual BON tool could be more interactive. The user could be able to add other classes from a project to a BON view. Furthermore, similar to the diagram tool already in place in EiffelStudio the BON tool could also work as a CASE tool. The user could add classes through the BON view have them automatically translate into Eiffel source code.

Related to the BON extractor it is obvious to also consider a tool working the opposite way. Generating Eiffel from textual BON would allow the user to specify a full system in textual BON with all features and contracts, generate the related Eiffel and instantly have a full skeleton of the system. This would enable seamless interaction between specification and development.

5.2 Type Checker

While the type checker on its own is quite complete, it could be improved in a number of places.

First of all, it could be integrated into EiffelStudio. An ideal solution could be that whenever a piece of Eiffel source code is viewed in either the informal BON view or the formal BON view, the "Compile" button in the project bar of the EiffelStudio interface is replaced with a "Type check" button, running the type checker on the specification in the view. Alternatively, such a button could be placed alongside the "Compile" button, such that Eiffel compilation is still available from inside the BON view. This would require adding a corresponding menu item to the project bar, and hooking the type checking functionality into the command executed by the button.

The current textual BON specification generated by the extraction tool does not type check due to some bugs mentioned in the next section. Additionally, another fundamental aspect of extraction constitutes an obstacle for type checking. Whenever a specification is extracted, either informal or formal, the ancestors of the main class (if any) are listed in the *inherit* clause of either the class chart or the class specification. But in the current implementation, only the specifications of the *descendants* of the main class are extracted by the tool. As such, the main class lists ancestor classes whose specification are not found in the extracted BON. Consequently, because the type checker requires that all listed ancestors are also defined in a corresponding class chart or class specification, it will complain that the ancestor classes were specified as ancestors, but never defined.

Defining the ancestors in the extracted specification is a recursive problem. Thus, if an ancestor of the main class is defined, this ancestor will also list ancestors, which in turn will list even more ancestors. As a result, all ancestors would have to be included until the root class of the inheritance hierarchy, ANY, is reached.

Defining all the ancestors of a class in a system as extensive as EiffelStudio is clearly not a viable solution. Accordingly, a possible place for further development would be to make the type checker compatible with the limitations of the extraction tool. This could be done by either creating a new version of the type checker based on the existing version, or by adding an EiffelStudio flag to the type checker, which for instance could make sure that ancestors of the main class are not required to be defined. In either case, there should be a clear distinction between when a type checker is checking the grammar defined in [WN95], and when it is checking an extracted specification in EiffelStudio. In the current implementation, the former has been made in order to provide a stepping stone for further development. In other words, the implemented type checker is not tailored to EiffelStudio, but supports further development in this direction.

5.3 Bugs and Missing Features

5.3.1 BON Extractor

The BON extraction tool has a few flaws. The most notable one has to do with feature calls. At the time of writing feature calls are not translated properly into textual BON from Eiffel. Only the first feature of the call chain is extracted, not any following features, nor arguments. The missing information is put in a comment if the contract has no tag. Else, a tag is put in the comment instead of the missing information. A call x on feature y with a parameter z it would translate as such:

$$y.x(z) \rightarrow y. \text{ -- } y.x(z)$$

Furthermore, generics of a classes don't keep their formal name throughout the class. In a class TUPLE [G, H], the two will be represented by a G followed by an index. In the example of the tuple, the first formal generic name, G will be represented as G#1 and second formal generic name, H as G#2. This is still consistent and readable, but is not ideal as it can be confusing with many generics and more importantly, it does not type check.

When analyzing inheritance the consistency of comments seem to fail. The further down into the inheritance hierarchy the tool digs, the more the comments seem to vanish. To accommodate for this features without comments are extracted by name in informal BON. When changing the point of origin (e.g., the current class) the comment structure seems to change.

All of these bugs most likely ground in a lack of understanding the abstract Eiffel syntax, which could be solved by studying this further.

Lastly, anchored types are translated directly into textual BON. This is not purposely done, however a future extension of the type checker and the BON language could be to include anchored types.

5.3.2 Type Checker

A few features are missing from the implementation of the type checker.

Primarily, notational tuning as defined on page 359 in [WN95] has not been implemented in the type checker or in the parser. However, it is still possible to use the keywords and constructs of notational tuning in a specification handed over to the type checker without getting any errors. The parser does nothing but recognize the presence of these constructs (no meta object is generated), while the type checker does not check it at all. Accordingly, notational tuning can be used, but support for it is not implemented.

In terms of type checking, a few features have not been implemented. Firstly, renaming of features in formal specifications are not checked. Unlike for notational tuning, renaming is implemented in the parser, and as such, the type checker has all the prerequisites for implementing a check for renaming. Secondly, selective export status is not checked for calls in formal assertions. As a consequence, all features of a type are always available in assertions, even though the specification states an explicit selective export status. Like for renaming, support for implementing a check for this is present, but it has not been done. The reason for omitting these checks in the initial version of the type checker is lack of time and a realization that checking other parts of the grammar would provide support for a wider range of specifications.

Bugs

Currently, the type checker has some known bugs. From running the test suite, the following bugs have been detected:

- Using calls with binary operators can lead to a feature call on Void target exception. This is most likely due to a bug in the way the type of a call is determined.

- Having indirection lists for a client entity in a client relation currently leads to a parse error.

- Detection of duplicate elements, such as when two classes with the same name are defined simultaneously, does not currently work as expected, as the parser accumulates these elements in sets. As such, duplicate elements are never sent to the type checker, because they are discarded by the parser. This could quite easily be fixed by accumulating the elements in lists, though.

- Detection of duplicate feature names as a result of inheritance is currently not done. This is only an issue when a class C inherits a feature *f* from two parents, e.g. A and B, but C does not itself have a feature *f*

- Defining a specification in which a cluster is a subcluster of itself leads to an infinite loop. This almost certainly is a bug in the features *check_informal_structure* and *check_formal_structure*.

Chapter 6

Conclusion

In this report it has been shown how textual BON can be integrated into EiffelStudio. For displaying textual BON in EiffelStudio, two additional views were implemented in the EiffelStudio GUI: an informal BON view for displaying informal specifications and a formal BON view for displaying formal specifications, respectively. These allow the user to easily obtain a BON specification from his or her Eiffel source code at any given time during implementation. As such, the textual BON tool supports the notion of reversibility, one of the main ideas of BON, since the extracted specification is always up-to-date with the latest changes in the source code. The extractor was examined in chapter 3

To achieve the goal of providing an overview of a system, the textual BON extractor also analyzes and extract the descendants of the class in scope. This overview allows the user to inspect multiple classes within an inheritance hierarchy, without changing view. The extracted BON is fairly complete, however with a few omissions (see appendix D) and bugs (see section 5.3.1). Overall it fulfills its purpose of giving an overview over a system.

A general-purpose type checker for textual BON based on the grammar presented in Walden and Nerson's book was implemented. The type checker is based on a procedural pattern, in which the well-typedness of each of the elements/components of the grammar is evaluated by a designated feature. The type checking algorithm features a two-phased strategy, which ensures consistent type checking. Due to the underspecified nature of the notation, it was explained which rules were added to make it type check. A comprehensive list of these rules can be found in appendix B.

With these additions to the Eiffel and BON worlds, a stepping stone for bridging the gap between these two has been laid. Other projects can, with basis in this one, help to further bridge this gap and let these two worlds work together.

Bibliography

[AP04] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java.* Cambridge University Press, 2nd edition, 2004.

[Bez] Eric Bezault. *Gobo: Eiffel Tools and Libraries*. http://www.gobosoft.com/.

[Eif10] Eiffel. *EiffelStudio Integrated Development Environment*. http://eiffelstudio.origo.ethz.ch/, 2006 - 2010.

[KF01] Joseph Kiniry and Fintan Fairmichael. *BONc*. http://kindsoftware.com/products/opensource/BONc/, 2001.

[Kin] Joseph Kiniry. *The Extended* BON *Tool Suite*. http://ebon.sourceforge.net/.

[Mar02] Robert C. Martin. *The Visitor Family of Design Patterns.* Rough chapter from The Principles, Patterns, and Practices of Agile Software Development. http://www.objectmentor.com/resources/articles/visitor.pdf, 2002.

[Mey01] Bertrand Meyer. *An Eiffel Tutorial*. http://archive.eiffel.com/doc/online/eiffel50/intro/language/tutorial.pdf, 2001.

[MSD09] MSDN. *Partial Class Definitions (C# Programming Guide)*. http://msdn.microsoft.com/en-us/library/wa80x488(v=vs.80).aspx, 2009.

[PKOL10] Richard Paige, Liliya Kaminskaya, Jonathan Ostroff, and Jason Lancaric. *BON-CASE: An Extensible CASE Tool for Formal Specification and Reasoning*. http://www.jot.fm/issues/issue_2002_08/article5/, 2010.

[PO98] Richard F. Paige and Jonathan S. Ostroff. *From Z to BON/Eiffel.* 1998.

[PO99] Richard F. Paige and Jonathan S. Ostroff. *Developing BON as an Industrial-Strength Formal Method.* 1999.

[PO01] Richard F. Paige and Jonathan S. Ostroff. *Metamodelling and Conformance Checking with PVS.* 2001.

[Ski10] Ralph Skinner. *An Integrated Development Environment for Business Object Notation*. http://kindsoftware.com/documents/reports/Skinner10.pdf, 2010.

[Str97] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3rd edition, 1997.

[WN95] Kim Waldén and Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture. Analysis and Design of Reliable Systems.* Prentice Hall, 1st edition, 1995.

Appendix A

Running the Code

A.1 BON Extractor

The textual BON extractor is a part of EiffelStudio, more specifically, the Eiffel Verification Environment, or EVE for short. Therefore, to run the extractor one needs to installEVE. An instruction for this can be found at the EVE website `http://eve.origo.ethz.ch/` or on the attached CD (*EVE Setup.html*). It is important to note that the latest version of EVE requires a new version of the Eiffel compiler (EiffelStudio 7.1). This can be found at `http://dev.eiffel.com/Downloads` in the beta area.

When in EVE the textual BON views should show up along side the others as seen in figure A.1.

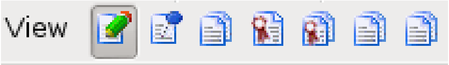
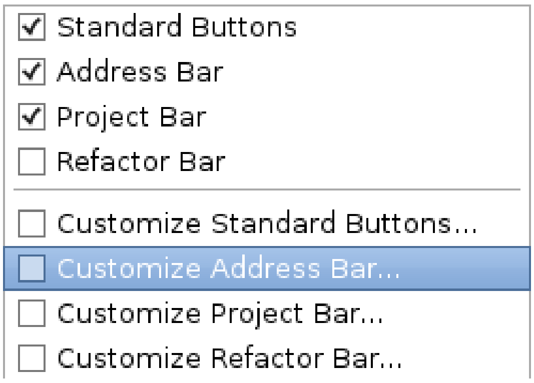


Figure A.1: EiffelStudio picture 1

Should they not, right click a tool bar to and select ”Customize Address Toolbar” as seen in figure A.2 .



☑ Standard Buttons
☑ Address Bar
☑ Project Bar
☐ Refactor Bar
☐ Customize Standard Buttons...
☐ Customize Address Bar...
☐ Customize Project Bar...
☐ Customize Refactor Bar...

Figure A.2: EiffelStudio picture 2

A window similar to the one in figure A.3 should pop up. In this window select the BON views and add them to the toolbar. Optionally, move them to the other views.

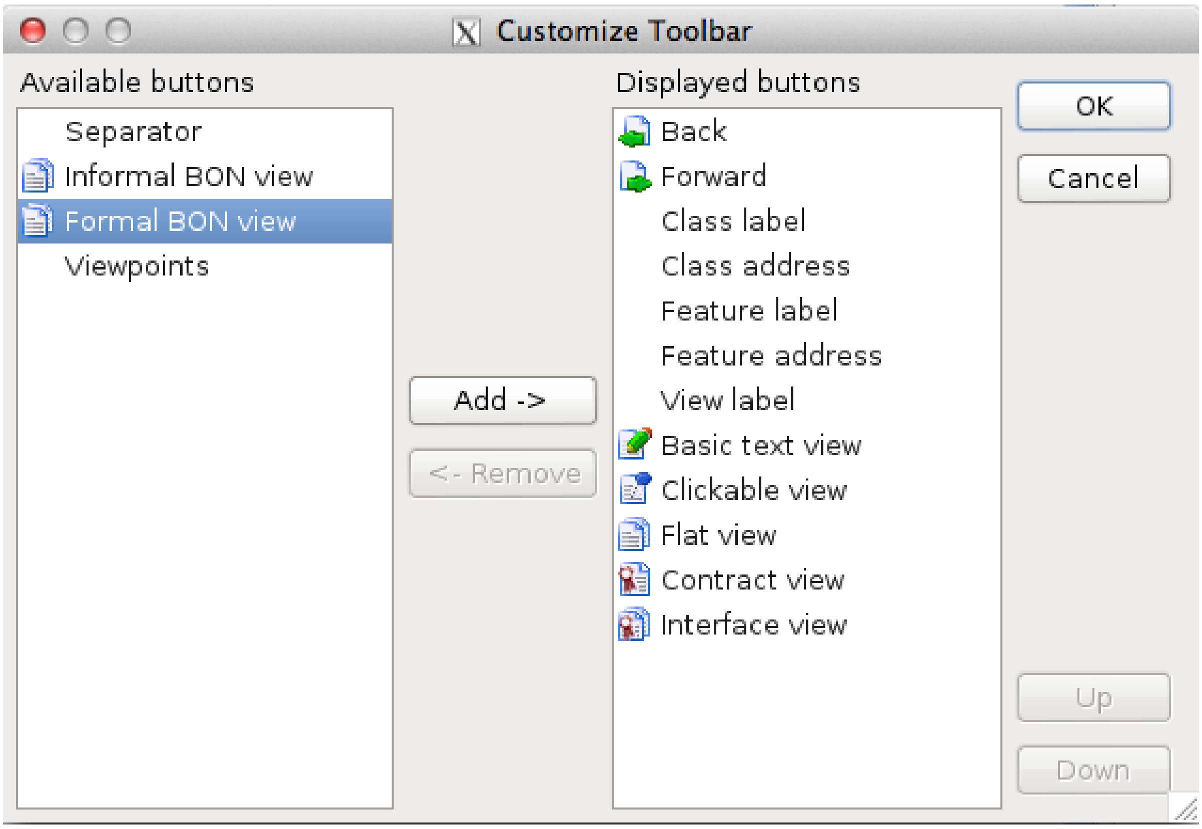


Figure A.3: EiffelStudio picture 3

A.2 Type Checker

The type checker have two dependencies: The GOBO library and the EiffelBase2 library. Furthermore the following environment must be set

1. ISE_PRECOMP - Set by EiffelStudio.
2. ISE_LIBRARY - Set by EiffelStudio.
3. REBON - The location of the rEBON project.
4. EIFFEL_BASE_2 - The location of the EiffelBase2 library.

To run the type checker, a proper Eiffel environment must be setup. Be sure to have the latest version of the Eiffel compiler and all dependencies. The project can both be compiled through EiffelStudio and command-line. Remember to finalize!

To compile from command line first navigate to the ecf file (rEBON\src\type_checker\). From here run the following:

```
ec -config tcb.ecf -clean -finalize -target tcb
```

When the project is compiled a ”tcb” executable will appear in rEBON\src\type_checker\EIFGENS\tcb\W_code\. The tcb binary takes one argument; a bon file[1]. Thus, to run it simply type (for example for class_chart.bon):

```
tcb class_chart.bon
```

If the type checker has a precondition violation in *check_bon_specification* it means that there has been a parse error. If the type checker finds any errors it will print the appropriate error message. If the type checker does not find any errors it will say nothing. Anything other than the above suggest that something unexpected has happened, possibly a bug. Feel free to report it!

[1] It will accept more than one bon specification file, but this is untested. Best guess is that it will type check the last file only.

Appendix B

Specification of the Implemented Type System

No rules have been implemented for components that are not mentioned

Contexts:

Informal_type_context

Formal_type_context

Variable_context

Class dictionary

$\exists$ Dictionary_entry $\in$ Class_dictionary

Dictionary entry

For a given class *cla*:

$\exists\ c \in \text{Informal\_type\_context}\ (c.\mathit{Class\_name} = \mathit{cla}.\mathit{Class\_name})$

For a given cluster *clu*:

$\exists\ c \in \text{Informal\_type\_context}\ (c.\mathit{Cluster\_name} = \mathit{clu}.\mathit{Cluster\_name})$

System chart

For a given system chart *sys*:

$\forall\ c \in \mathit{sys}\ (\exists\ \mathit{clu} \in \text{Informal\_type\_context}\ (c.\mathit{Cluster\_name} = \mathit{clu}.\mathit{Cluster\_name}))$

Cluster entry

For a given cluster entry *ce*:

$\exists\ c \in \text{Informal\_type\_context}\ (c.\mathit{Cluster\_name} = \mathit{ce}.\mathit{Cluster\_name})$

Cluster chart

For a given Cluster chart *cc*

$\exists!\ c \in \text{Informal\_type\_context} \bullet c.\mathit{Cluster\_name} = \mathit{cc}.\mathit{Cluster\_name}$

$\neg\ \exists\ \text{c} \in \text{cc.Cluster\_entries} \bullet c.\mathit{Cluster\_name} = \mathit{cc}.\mathit{Cluster\_name}$

If n is all nested clusters of *cc* then:

$\neg\ \exists\ \text{c} \in \text{n.Cluster\_entries} \bullet c.\mathit{Cluster\_name} = \mathit{cc}.\mathit{Cluster\_name}$

Class entry

For a given class entry *ce*

$\exists!\ c \in \text{Informal\_type\_context} \bullet c.\mathit{Class\_name} = \mathit{ce}.\mathit{Class\_name}$

Class chart

For a given class chart *cc*

$\exists!\ c \in \text{Informal\_type\_context} \bullet c.\mathit{Class\_name} = \mathit{cc}.\mathit{Class\_name}$

$\forall\ a \in \text{cc.Ancestors} \bullet a.\mathit{Class\_name} \neq \mathit{cc}.\mathit{Class\_name}$

$\forall\ a \in \text{cc.Ancestors} \bullet \exists\ c \in \text{Informal\_type\_context} \bullet a.\mathit{Class\_name} = c.\mathit{Class\_name}$

Structure

If a given class is *cla*

$\exists!\ \mathit{clu} \in \text{Informal\_type\_context} \bullet \exists!\ c \in \mathit{clu} \bullet c.\mathit{Class\_name} = \mathit{cla}.\mathit{Class\_name}$

If a given cluster is *clu*

$\exists!\ \mathit{clu} \in \text{Informal\_type\_context} \bullet \exists!\ c \in \mathit{clu} \bullet c.\mathit{Cluster\_name} = \mathit{clu}.\mathit{Cluster\_name}$

Event chart

Event entry

For an event entry *ee*

$\forall\ e \in ee.Class\_name\_list \bullet \exists\ c \in$ Informal_type_context $\bullet\ c.Class\_name = e.Class\_name)$

Event chart

Event entry

Warning for duplicate scenarios

Creation chart

Creation entry

For a creation entry *ce*

$\exists\ c \in$ Informal_type_context $\bullet\ c.Class\_name = ce.Class\_name$

$\forall\ e \in ce.Class\_name\_list \bullet \exists\ c \in$ Informal_type_context $\bullet\ c.Class\_name = e.Class\_name)$

Static_diagram

Static_block

Cluster

For a given cluster component named *clu*:

$\exists$! c $\in$ Formal_type_context (c.name = clu.Cluster_name)

Class

For a given class component named *class*:

$\exists$ f $\in$ features | f is deferred $\bullet$ class is deferred

$\exists$! c $\in$ Formal_type_context (c.name = class.Class_name)

Formal_generics

For a given Formal_generic *gen*:

$\exists$! g $\in$ Formal_generics of *class* (g.Formal_generic_name = gen.Formal_generic_name)

gen has Class_type $\Rightarrow \exists$ c $\in$ Formal_type_context (c.name = gen.Class_type.Class_name)

gen has Class_type $\Rightarrow$ number of actual generics of Class_type must be equal to number of Formal_generics in corresponding defined class in Formal_type_context

Actual_generics

The i'th Type, *t*, in the list of Actual_generics in Class_type of *gen* must conform to the bounding Class_type, *u* of the i'th Formal_generic in the corresponding defined class in Formal_type_context.

If *t* is a Formal_generic_name:

The Class_type of the Formal_generic corresponding to *t* must conform to u. If no Class_type is present for *t*, it is considered to be ANY.

t must be defined. *t* is defined if its definition appears before *gen* (left-to-right)

If *t* is a Class_type:

t must conform covariantly to *u*

Static_relation

Static ref

For a static reference *sta* with class *cl* and Cluster_prefix *cp*:

$\exists\ s \in$ Formal_type_context $(s.Class\_name = sta.Class\_name)$

If *cp* has *n* clusters, *cl* must be in cluster *n*, cluster *n* must be in cluster *n-1*, cluster *n-1* must be in cluster *n-2* etc.

Inheritance_relation

For a relation between Child *chi* and Parent *par*:

chi conforms to *par*

Multiplicity > 0

Client_relation

Client_entity

Feature_name

For a client relation with client class *c* feature name client entity *fn*:

$\exists$ f $\in$ features of $c \bullet f.Feature\_name = fn$

Indirection_feature_list

For a given Indirection_feature_list *ifl* with client class *cl*

$\forall f \in$ ifl $\bullet\ \exists$ *fe* $\in$ features of *cl* $\bullet$ *f* = *fe.Feature_name*

Generic_indirection

For a given Generic_indirection *gi* with client class *cl*:

If *gi* is Formal_generic_name:

$\exists$ fgn $\in$ formal generic names of *cl* $\bullet$ fgn = gi

If *gi* is Named_indirection:

$\exists\ c \in$ Formal_type_context $\bullet$ (*c.Class_name* = *gi.Class_name*)

gi has Indirection_list $\land$ *gi* has Class_name $\Rightarrow$ the number of type arguments in the corresponding defined type in the Formal_type_context must be equal to the number of Indirection_elements in the Indirection_list of *gi*

gi has Indirection_list $\land$ *gi* has Class_name $\Rightarrow$ The type of the *i*'th Indirection_element in the Indirection_list must conform to the type bounding Class_type of the *i*'th type parameter in class corresponding to the Class_name of *gi.*

Class_interface

$\forall$ p $\in$ Parent_class_list $\bullet$ p $\in$ Formal_type_context

Feature_clause

Selective_export

$\forall$ *c in* Class_name_list $\bullet$ ($\exists$ t $\in$ Formal_type_context *bullet* c = t.name)

Feature_specification

Status:

For a given Feature *feat* with precursor pre:

pre is *deferred* $\Rightarrow$ *feat* status is *deferred* or *effective*
pre is *effective* $\Rightarrow$ *feat* status is *deferred* or *redefined*
pre is *redefined* $\Rightarrow$ *feat* status is *deferred* or *redefined*
pre is unclassified $\Rightarrow$ *feat* status is *deferred* or *redefined*

Type

For a given Feature *feat* with Type *t*:

t $\in$ Formal_type_context

If *feat* has precursor *pre*:

Type of *feat* must conform covariantly to type of *pre*

Type_mark

For a given Feature *feat* with Type *t* and Type_mark *tm* in class *cl*:

tm is aggregation $\Rightarrow$ $t \neq cl$

Feature_argument

For a given Feature *feat* in class *cl*

$\forall$ *arg* $\in$ arguments of *feat* $\bullet\ \exists\,!$ *ark* $\in$ arguments of *feat* (arg.Identifier = ark.Identifier)
$\forall$ *arg* $\in$ arguments of *feat* $\bullet$ *arg.Identifier* $\neq$ "Result"
$\forall$ *arg* $\in$ arguments of *feat* $\bullet$ *arg.Identifier* $\neq$ "Current"
$\forall$ *arg* $\in$ arguments of *feat* $\bullet$ *arg.Identifier* $\neq$ "Void"
$\forall$ *arg* $\in$ arguments of *feat* $\bullet$ *arg.Identifier* $\neq$ *cl.Class_name*

If *feat* has precursor *pre*:

The number of arguments of *feat* must be equal to the number of arguments of *pre*

The Type of the i'th argument of *feat* must conform to the Type of the i'th argument of *pre*

Feature_name

For a given Feature_name *f* of Feature *feat* in class cl:

If *f* is Identifier:

$\exists\,!$ *fn* $\in$ Feature names of *cl* (*fn* = *f*) (where Feature names also include inherited names)

If *f* is prefix:

$\exists\,!$ *fn in* Feature_names of *cl* (*fn* = *f* and *fn* is prefix) *f* is prefix *Rightarrow feat* has exactly zero arguments

If *f* is infix:

∃ ! *fn in* Feature_names of *cl* (*fn* = *f* and *fn* is infix)

f is infix ⇒ *feat* has exactly one argument

Assertion

Quantification

Expression of Restriction must have boolean type

Expression of Proposition must have boolean type

Identifier_list

∀ *id* ∈ *Identifier_list* • ∃ ! *i in Identifier_list* (id = i)

Scope rule:

∀ *var* ∈ Variable_context • ∃ ! *s in* Variable_context (variable name of *var* = variable name of *s*)

Member_range

∀ *id* ∈ *Identifier_list* • id.Type conforms to type of set in Set_expression

Set_expression

Enumerated_set

For a given Enumerated_set *es*:

∀ element *in es* • element ≠ *Void*

All elements must conform to one reference element in the set:

∃ element *in es* • ∀ e *in es* • *e* conforms to *element*

Call or Operator_expression

For a given Call or Operator_expression *ex* with type *tex* appearing in a Set_expression

Warning if not

tex conforms to ENUMERABLE

Type_range

For a given Type *t* of a Type_range:

t ∈ Formal_type_context

Call

If parenthesized qualifier *paq* is specified with call chain *ch*:

First call of *ch* must be a feature in the interface of the type of *paq*

If no parenthesized qualifier is specified with call chain *ch*:

If in invariant in enclosing class *ec*:

First call of *ch* must be a feature in the interface of *ec*, but not an infix or a prefix feature.

If in pre- or postcondition of a feature *feat* in enclosing class *ec*:

First call of *ch* must be an argument to *feat* or another feature in the interface of *ec*, but not *feat* itself and not an infix or a prefix feature.

For non-first calls in a call chain *ch*:

Identifier in call *n* of *ch* (*n* >1) must exist in the interface of the type of call *n-1* in *ch*.

Unqualified_call

Actual_arguments

For a given feature *feat* in enclosing class *ec*:

The number of arguments of the call must be equal to the number of arguments in the specification of *feat*

The type of the *i*'th argument in the list of Actual_arguments to *feat* must conform to the type of the *i*'th argument in the definition of *feat*

Operator_expression

Unary_expression

For a given Unary_expression with operator *op* and expression *ex*:

∃ *pf* in Features in Type of *ex* • *pf.Feature_name* = *op*∧ *pf* is prefix feature

Binary_expression

For a given Binary_expression with operator *op* and left expression *lex* and right expression *rex*:

∃ *if* in Features in Type of *lex* • *if.Feature_name* = *op*∧ *if* is infix feature∧ *if.Feature_arguments.count* = 1∧ Type of *rex* conforms to type of first argument of *if*

Dynamic_diagram

Scenario_description

For a given Scenario description *sce*:

$\exists\ !\ s \in$ Dynamic_object_context $(s.\mathit{Scenario\_name} = sce.\mathit{Scenario\_name})$

Labeled_actions

For a given Labeled action *lab*:

$\exists\ !\ l \in$ Dynamic_object_context $(l.\mathit{Action\_label} = lab.\mathit{Action\_label})$

Object_group

For a given Object group *obg*:

$\exists\ !\ o \in$ Dynamic_object_context $(o.\mathit{Group\_name} = obg.\mathit{Group\_name})$

Object_stack

For a given Object stack *obs*:

$\exists\ !\ o \in$ Dynamic_object_context $(o.\mathit{Object\_name} = obs.\mathit{Object\_name})$

Object

For a given Object *obj*

$\exists\ !\ o \in$ Dynamic_object_context $(o.\mathit{Object\_name} = obj.\mathit{Object\_name})$

Message relation

Dynamic reference

For a given Dynamic reference *dyn*:

$\exists\ o \in$ Dynamic_object_context $(o.\mathit{Object\_name} = dyn.\mathit{Object\_name} \vee o.\mathit{Group\_name} = dyn.\mathit{Group\_name})$

dyn has Group_prefix $\Rightarrow \exists\ og \in$ Dynamic_object_context $((dyn.\mathit{Group\_prefix}.\mathit{Group\_name} = og.\mathit{Group\_name}) \wedge \exists\ o \in og\ (o.\mathit{Object\_name} = dyn.\mathit{Object\_name}))$

Message label

For a given Message label *ml*:

$\exists\ m \in$ Dynamic_object_context $(m.\mathit{Manifest\_string} = ml.\mathit{Manifest\_string} \wedge m$ is an Action_label)

Appendix C

Specification of Standard Types

```
system_chart STANDARD_TYPES
explanation "The standard types of the BON type checker"
cluster STANDARD_VALUE_TYPES
    description "Standard value types that are defined by default."
cluster DATA_STRUCTURES
    description "Standard data structures defined by default."
end

cluster_chart STANDARD_VALUE_TYPES
class BOOLEAN
    description "A boolean type."
class CHARACTER
    description "A character type."
class INTEGER
    description "An integer type."
class REAL
    description "A real type."
class STRING
    description "A string type."
end

class_chart BOOLEAN
end

class_chart CHARACTER
end

class_chart INTEGER
inherit REAL
end

class_chart REAL
end

class_chart STRING
```

```
end

cluster_chart DATA_STRUCTURES
class ENUMERABLE
    description "An enumerable type."
class CONTAINER
    description "A type containing other elements."
class LIST
    description "A list of elements."
class SET
    description "A set of elements."
class TABLE
    description "A table mapping from one type to another."
class ARRAY
    description "An array of elements."
class TUPLE
    description "A tuple of elements."
end

class_chart ENUMERABLE
end

class_chart CONTAINER
query
    "How many elements do this container contain?"
end

class_chart LIST
inherit ENUMERABLE, CONTAINER
query
    "Which element does this list have at position i?"
end

class_chart SET
inherit ENUMERABLE, CONTAINER
query
    "What is one of the elements of this set?"
end

class_chart TABLE
query
    "What are the keys of this table?",
    "What are the values of this table?"
end

class_chart ARRAY
inherit ENUMERABLE, CONTAINER
query
    "Which element does this array have at position i?"
```

```
end

class_chart TUPLE
end

static_diagram STANDARD_LIBRARIES
component
    cluster STANDARD_VALUE_TYPES
        component
            class BOOLEAN
            class CHARACTER
            class INTEGER
                inherit REAL
                feature
                    as_real: REAL
                end
            class REAL
            class STRING
        end

    cluster DATA_STRUCTURES
        component
            deferred class ENUMERABLE

            deferred class CONTAINER[E]
                feature
                    count: INTEGER
                end

            class LIST[E]
                inherit
                    CONTAINER[E];
                    ENUMERABLE
                feature
                    i_th: E
                        -> index: INTEGER
                end

            class SET[E]
                inherit
                    CONTAINER[E];
                    ENUMERABLE
                feature
                    any_item: E
                end

            class TABLE[K,V]
                feature
                    keys: SET[K]
                    values: SET[V]
                end

            class ARRAY[E]
                inherit
                    CONTAINER[E];
                    ENUMERABLE
                feature
                    i_th: E
                        -> index: INTEGER
                end

            class TUPLE[G, H]
        end
end
```

Appendix D

Components Not Extracted

Some elements are not extracted by the textual BON extractor. Below is a list of omitted elements. Any subelement of an excluded are also excluded. The elements are shown as they appear in [WN95, pp. 352–359].

Event_chart
In textual BON an event charts purpose is to express important events. As extracting semantics from source code is out of this projects scope, it has been omitted.

Scenario_chart
Similar to event charts, scenario charts purpose is to express semantics.

Creation_chart
Creation charts could have been included, but it would have been very involving as a lot of further analysis would have been required. This is a possibility for a future project.

Static_relations
Similar to creation charts, static relations could have been included, but as with creation charts, it's something for further development.

Dynamic_diagram
Since dynamic diagrams, as the name implies, describe a dynamic context, our static analysis would not be able to make sensible dynamic diagrams.

Class_dictionary
Class dictionaries could have been included to give further overview over a system, however it was deemed unnecessary due to its similarity to informal charts. Thus, there were more interesting parts of the project to dedicate time to.

Part The reason for parts omission was discussed in section 2.2.2.

Appendix E

Textual BON Extraction Example

In this appendix is an example of textual BON extracted from Eiffel. The class in question is the MML_MODEL class from the EiffelBase2 library. Both informal and formal BONare included. The formal BON specification starts on line 280 of the extracted textual BON. Eiffel and textual BON can also be found on the attached CD. Lastly, small ␣'s seemed to have sneaked their way in during the LaTeX formatting. These are not present in the extracted BON and should be replaced with spaces.

E.1 Eiffel Source Code

```
note
        description: "Mathematical␣models."
        author: "Nadia␣polikarpova."
        date: "$Date$"
        revision: "$Revision$"

deferred class
        MML_MODEL

feature -- Comparison
        is_model_equal alias "|=|" (other: MML_MODEL): BOOLEAN
                        -- Is this model mathematically equal to ‘other’?
                deferred
                end

        is_model_non_equal alias "|/=|" (other: MML_MODEL): BOOLEAN
                        -- Is this model mathematically equal to ‘other’?
                do
                        Result := not is_model_equal (other)
                end

        frozen model_equals (v1, v2: ANY): BOOLEAN
                        -- Are ‘v1’ and ‘v2’ mathematically equal?
                        -- If they are models use model equality, otherwise reference equality.
                do
                        if attached {MML_MODEL} v1 as m1 and attached {MML_MODEL} v2 as m2 then
                                Result := m1 |=| m2
                        else
                                Result := v1 = v2
                        end
                end
end
```

E.2 Extracted BON

```
system_chart SYSTEM_OF_MML_MODEL
explanation
        "A␣description␣of␣your␣system."
cluster CLUSTER_OF_MML_MODEL
        "A␣description␣of␣your␣cluster."
end

cluster_chart CLUSTER_OF_MML_MODEL
explanation
        "A␣description␣of␣your␣cluster."
class MML_MODEL
        "Mathematical␣models."
class MML_RELATION
        "Finite␣relations."
class MML_SEQUENCE
        "Finite␣sequence."
class MML_BAG
        "Finite␣bags."
class MML_SET
        "Finite␣sets."
class MML_INTERVAL
        "Closed␣integer␣intervals."
class MML_MAP
        "Finite␣maps."
end

class_chart MML_MODEL
indexing
        author: "Nadia␣polikarpova.";
        date: "$Date$";
        revision: "$Revision$";
        belongs_to: "CLUSTER_OF_MML_MODEL"
explanation
        "Mathematical␣models."
query
        "␣Is␣this␣model␣mathematically␣equal␣to␣‘other’?",
        "␣Is␣this␣model␣mathematically␣equal␣to␣‘other’?",
        "␣Are␣‘v1’␣and␣‘v2’␣mathematically␣equal?\
␣␣␣␣␣␣␣␣\␣␣If␣they␣are␣models␣use␣model␣equality,␣otherwise␣reference␣equality."
end

class_chart MML_RELATION
indexing
        author: "Nadia␣Polikarpova";
        date: "$Date$";
        revision: "$Revision$";
        belongs_to: "CLUSTER_OF_MML_MODEL"
explanation
        "Finite␣relations."
inherit
        MML_MODEL
query
        "has",
        "is_empty",
        "domain",
        "range",
        "image_of",
        "image",
        "count",
        "is_model_equal",
        "extended",
        "removed",
        "restricted",
        "inverse",
        "union",
        "intersection",
        "difference",
        "sym_difference",
        "lefts",
        "rights",
        "meq_left",
        "meq_right"
command
        "␣Is␣this␣model␣mathematically␣equal␣to␣‘other’?",
        "␣Are␣‘v1’␣and␣‘v2’␣mathematically␣equal?\
␣␣␣␣␣␣␣␣\␣␣If␣they␣are␣models␣use␣model␣equality,␣otherwise␣reference␣equality.",
        "make_from_arrays"
constraint
        "lefts_exists",
        "rights_exists",
        "same_lower",
        "same_upper",
        "start_from_one"
end

class_chart MML_SEQUENCE
indexing
        author: "Nadia␣Polikarpova";
        date: "$Date$";
        revision: "$Revision$";
        belongs_to: "CLUSTER_OF_MML_MODEL"
explanation
        "Finite␣sequence."
inherit
        MML_MODEL
query
        "has",
        "is_empty",
        "is_constant",
        "item",
        "domain",
        "range",
        "to_bag",
        "count",
        "occurrences",
        "is_model_equal",
        "is_prefix_of",
        "first",
        "last",
        "but_first",
        "but_last",
        "front",
        "tail",
        "interval",
        "removed_at",
        "restricted",
        "removed",
        "extended",
        "extended_at",
        "prepended",
        "concatenation",
        "replaced_at",
        "inverse",
        "array",
        "meq"
command
        "default_create",
        "␣Are␣‘v1’␣and␣‘v2’␣mathematically␣equal?\
␣␣␣␣␣␣␣␣\␣␣If␣they␣are␣models␣use␣model␣equality,␣otherwise␣reference␣equality.",
        "make_from_array"
constraint
        "array_exists",
        "starts_from_one"
end

class_chart MML_BAG
indexing
```

```
        author: "Nadia␣Polikarpova";
        date: "$Date$";
        revision: "$Revision$";
        belongs_to: "CLUSTER_OF_MML_MODEL"
explanation
        "Finite␣bags."
inherit
        MML_MODEL
query
        "has",
        "is_empty",
        "is_constant",
        "domain",
        "occurrences",
        "count",
        "is_model_equal",
        "extended",
        "extended_multiple",
        "removed",
        "removed_multiple",
        "removed_all",
        "restricted",
        "union",
        "difference",
        "keys",
        "values",
        "meq"
command
        "␣Are␣‘v1’␣and␣‘v2’␣mathematically␣equal?\
␣␣␣␣␣␣␣␣\␣␣If␣they␣are␣models␣use␣model␣equality,␣otherwise␣reference␣equality.",
        "singleton",
        "multiple",
        "make_from_arrays"
constraint
        "keys_exists",
        "values_exists",
        "same_lower",
        "same_upper",
        "start_at_one"
end

class_chart MML_SET
indexing
        author: "Nadia␣Polikarpova";
        date: "$Date$";
        revision: "$Revision$";
        belongs_to: "CLUSTER_OF_MML_MODEL"
explanation
        "Finite␣sets."
inherit
        MML_MODEL
query
        "has",
        "is_empty",
        "for_all",
        "exists",
        "any_item",
        "extremum",
        "filtered",
        "count",
        "is_model_equal",
        "is_subset_of",
        "is_superset_of",
        "disjoint",
        "extended",
        "removed",
        "union",
        "intersection",
        "difference",
        "sym_difference",
        "array",
        "meq"
command
        "default_create",
        "singleton",
        "make_from_array"
constraint
        "array_exists",
        "starts_from_one"
end

class_chart MML_INTERVAL
indexing
        author: "Nadia␣Polikarpova";
        date: "$Date$";
        revision: "$Revision$";
        belongs_to: "CLUSTER_OF_MML_MODEL"
explanation
        "Closed␣integer␣intervals."
inherit
        MML_SET
query
        "lower",
        "upper"
command
        "from_range",
        "from_tuple"
end

class_chart MML_MAP
indexing
        author: "Nadia␣Polikarpova";
        date: "$Date$";
        revision: "$Revision$";
        belongs_to: "CLUSTER_OF_MML_MODEL"
explanation
        "Finite␣maps."
inherit
        MML_MODEL
query
        "has",
        "is_empty",
        "is_constant",
        "item",
        "domain",
        "range",
        "image",
        "sequence_image",
        "to_bag",
        "count",
        "is_model_equal",
        "updated",
        "removed",
        "restricted",
        "override",
        "inverse",
        "keys",
        "values",
        "meq_key",
        "meq_value"
command
        "␣Is␣this␣model␣mathematically␣equal␣to␣‘other’?",
        "␣Are␣‘v1’␣and␣‘v2’␣mathematically␣equal?\
␣␣␣␣␣␣␣␣\␣␣If␣they␣are␣models␣use␣model␣equality,␣otherwise␣reference␣equality.",
        "make_from_arrays"
constraint
        "keys_exists",
        "values_exists",
        "same_lower",
        "same_upper",
        "start_from_one"
end

static_diagram SYSTEM_OF_MML_MODEL -- This is a comment.
component
        cluster CLUSTER_OF_MML_MODEL
        component
                class MML_MODEL
                indexing
                        description: "Mathematical␣models.";
                        author: "Nadia␣polikarpova.";
                        date: "$Date$";
                        revision: "$Revision$";
                        belongs_to: "CLUSTER_OF_MML_MODEL"
                feature --  Comparison
                        deferred is_model_equal: BOOLEAN
                                        --  Is this model mathematically equal to ‘other’?
                                -> other: MML_MODEL


                        is_model_non_equal: BOOLEAN
                                        --  Is this model mathematically equal to ‘other’?
                                -> other: MML_MODEL


                        model_equals: BOOLEAN
                                        --  Are ‘v1’ and ‘v2’ mathematically equal?
                                        --  If they are models use model equality,
                                        --  otherwise reference equality.
                                -> v1: ANY
                                -> v2: ANY

                end
                class MML_RELATION [G, H]
                indexing
                        description: "Finite␣relations.";
                        author: "Nadia␣Polikarpova";
                        date: "$Date$";
                        revision: "$Revision$";
                        belongs_to: "CLUSTER_OF_MML_MODEL"
                inherit
                        MML_MODEL

                feature {NONE}
                        default_create
                                        --  Is this model mathematically equal to ‘other’?


                        singleton
                                        --  Are ‘v1’ and ‘v2’ mathematically equal?
                                        --  If they are models use model equality,
                                        --  otherwise reference equality.
                                -> x: G#1
                                -> y: G#2

                feature
                        has: BOOLEAN
                                -> x: G#1
                                -> y: G#2

                        is_empty: BOOLEAN

                feature
                        domain: MML_SET [G#1]

                        range: MML_SET [G#2]

                        image_of: MML_SET [G#2]
                                -> x: G#1

                        image: MML_SET [G#2]
                                -> subdomain: MML_SET [G#1]
                                require
                                        subdomain /= Void -- subdomain_exists
                                end
                feature
                        count: INTEGER

                feature
                        is_model_equal: BOOLEAN
                                -> other: MML_MODEL

                feature
                        extended: MML_RELATION [G#1, G#2]
                                -> x: G#1
                                -> y: G#2

                        removed: MML_RELATION [G#1, G#2]
                                -> x: G#1
                                -> y: G#2

                        restricted: MML_RELATION [G#1, G#2]
                                -> subdomain: MML_SET [G#1]
                                require
                                        subdomain /= Void -- subdomain_exists
                                end
                        inverse: MML_RELATION [G#2, G#1]

                        union: MML_RELATION [G#1, G#2]
                                -> other: MML_RELATION [G#1, G#2]
                                require
                                        other /= Void -- other_exists
                                end
                        intersection: MML_RELATION [G#1, G#2]
                                -> other: MML_RELATION [G#1, G#2]
                                require
                                        other /= Void -- other_exists
                                end
                        difference: MML_RELATION [G#1, G#2]
                                -> other: MML_RELATION [G#1, G#2]
                                require
                                        other /= Void -- other_exists
                                end
                        sym_difference: MML_RELATION [G#1, G#2]
                                -> other: MML_RELATION [G#1, G#2]
                                require
                                        other /= Void -- other_exists
                                end
                feature {MML_MODEL}
                        lefts: V_ARRAY [G#1]

                        rights: V_ARRAY [G#2]

                        make_from_arrays
                                -> ls: V_ARRAY [G#1]
                                -> rs: V_ARRAY [G#2]
                                require
                                        ls /= Void -- ls_exists
                                        rs /= Void -- rs_exists
                                        ls. = rs. -- same_lower
                                        ls. = rs. -- same_upper
                                        ls. = 1 -- start_from_one
                                end
                        meq_left: BOOLEAN
                                -> v1: G#1
                                -> v2: G#1

                        meq_right: BOOLEAN
                                -> v1: G#2
                                -> v2: G#2
```

```

invariant
        lefts /= Void -- lefts_exists
        rights /= Void -- rights_exists
        lefts. = rights. -- same_lower
        lefts. = rights. -- same_upper
        lefts. = 1 -- start_from_one
end
class MML_SEQUENCE [G]
indexing
        description: "Finite␣sequence.";
        author: "Nadia␣Polikarpova";
        date: "$Date$";
        revision: "$Revision$";
        belongs_to: "CLUSTER_OF_MML_MODEL"
inherit
        MML_MODEL

feature {NONE}
        default_create

        singleton
                        --  Are 'v1' and 'v2' mathematically equal?
                        --  If they are models use model equality,
                        --  otherwise reference equality.
                -> x: G#1

feature
        has: BOOLEAN
                -> x: G#1

        is_empty: BOOLEAN

        is_constant: BOOLEAN
                -> c: G#1

feature
        item: G#1
                -> i: INTEGER
                require
                        bracket_error -- in_domain
                end
feature
        domain: MML_INTERVAL

        range: MML_SET [G#1]

        to_bag: MML_BAG [G#1]

feature
        count: INTEGER

        occurrences: INTEGER
                -> x: G#1

feature
        is_model_equal: BOOLEAN
                -> other: MML_MODEL

        is_prefix_of: BOOLEAN
                -> other: MML_SEQUENCE [G#1]
                require
                        other /= Void -- other_exists
                end
feature
        first: G#1
                require
                        not is_empty -- non_empty
                end
        last: G#1
```

```
                require
                    not is_empty -- non_empty
                end
            but_first: MML_SEQUENCE [G#1]
                require
                    not is_empty -- not_empty
                end
            but_last: MML_SEQUENCE [G#1]
                require
                    not is_empty -- not_empty
                end
            front: MML_SEQUENCE [G#1]
                -> upper: INTEGER

            tail: MML_SEQUENCE [G#1]
                -> lower: INTEGER

            interval: MML_SEQUENCE [G#1]
                -> lower: INTEGER
                -> upper: INTEGER

            removed_at: MML_SEQUENCE [G#1]
                -> i: INTEGER
                require
                    bracket_error -- in_domain
                end
            restricted: MML_SEQUENCE [G#1]
                -> subdomain: MML_SET [INTEGER]
                require
                    subdomain /= Void -- subdomain_exists
                end
            removed: MML_SEQUENCE [G#1]
                -> subdomain: MML_SET [INTEGER]
                require
                    subdomain /= Void -- subdomain_exists
                end
        feature
            extended: MML_SEQUENCE [G#1]
                -> x: G#1

            extended_at: MML_SEQUENCE [G#1]
                -> i: INTEGER
                -> x: G#1
                require
                    1 <= i or i <= count + 1 -- valid_position
                end
            prepended: MML_SEQUENCE [G#1]
                -> x: G#1

            concatenation: MML_SEQUENCE [G#1]
                -> other: MML_SEQUENCE [G#1]
                require
                    other /= Void -- other_exists
                end
            replaced_at: MML_SEQUENCE [G#1]
                -> i: INTEGER
                -> x: G#1
                require
                    bracket_error -- in_domain
                end
            inverse: MML_RELATION [G#1, INTEGER]

        feature {MML_MODEL}
            array: V_ARRAY [G#1]

            make_from_array
                -> a: V_ARRAY [G#1]
                require
                    a /= Void -- a_exists
                    a. = 1 -- starts_from_one
```

```
                                end
                        meq: BOOLEAN
                                -> v1: G#1
                                -> v2: G#1

                invariant
                        array /= Void -- array_exists
                        array. = 1 -- starts_from_one
                end
                class MML_BAG [G]
                indexing
                        description: "Finite␣bags.";
                        author: "Nadia␣Polikarpova";
                        date: "$Date$";
                        revision: "$Revision$";
                        belongs_to: "CLUSTER_OF_MML_MODEL"
                inherit
                        MML_MODEL

                feature {NONE}
                        default_create
                                        --  Are 'v1' and 'v2' mathematically equal?
                                        --  If they are models use model equality,
                                        --  otherwise reference equality.


                        singleton
                                -> x: G#1

                        multiple
                                -> x: G#1
                                -> n: INTEGER
                                require
                                        n >= 0 -- n_positive
                                end
                feature
                        has: BOOLEAN
                                -> x: G#1

                        is_empty: BOOLEAN

                        is_constant: BOOLEAN
                                -> c: INTEGER

                feature
                        domain: MML_SET [G#1]

                feature
                        occurrences: INTEGER
                                -> x: G#1

                        count: INTEGER

                feature
                        is_model_equal: BOOLEAN
                                -> other: MML_MODEL

                feature
                        extended: MML_BAG [G#1]
                                -> x: G#1

                        extended_multiple: MML_BAG [G#1]
                                -> x: G#1
                                -> n: INTEGER
                                require
                                        n >= 0 -- n_non_negative
                                end
                        removed: MML_BAG [G#1]
                                -> x: G#1

```

```
        removed_multiple: MML_BAG [G#1]
                -> x: G#1
                -> n: INTEGER
                require
                        n >= 0 -- n_non_negative
                end
        removed_all: MML_BAG [G#1]
                -> x: G#1

        restricted: MML_BAG [G#1]
                -> subdomain: MML_SET [G#1]
                require
                        subdomain /= Void -- subdomain_exists
                end
        union: MML_BAG [G#1]
                -> other: MML_BAG [G#1]
                require
                        other /= Void -- other_exists
                end
        difference: MML_BAG [G#1]
                -> other: MML_BAG [G#1]
                require
                        other /= Void -- other_exists
                end
feature {MML_MODEL}
        keys: V_ARRAY [G#1]

        values: V_ARRAY [INTEGER]

        make_from_arrays
                -> ks: V_ARRAY [G#1]
                -> vs: V_ARRAY [INTEGER]
                -> n: INTEGER
                require
                        ks /= Void -- ks_exists
                        vs /= Void -- vs_exists
                        ks. = vs. -- same_lower
                        ks. = vs. -- same_upper
                        ks. = 1 -- start_at_one
                        ks. -- ks_has_no_duplicates
                        vs. -- vs_positive
                end
        meq: BOOLEAN
                -> v1: G#1
                -> v2: G#1

invariant
        keys /= Void -- keys_exists
        values /= Void -- values_exists
        keys. = values. -- same_lower
        keys. = values. -- same_upper
        keys. = 1 -- start_at_one
end
class MML_SET [G]
indexing
        description: "Finite␣sets.";
        author: "Nadia␣Polikarpova";
        date: "$Date$";
        revision: "$Revision$";
        belongs_to: "CLUSTER_OF_MML_MODEL"
inherit
        MML_MODEL

feature {NONE}
        default_create
                ensure
                        is_empty -- empty
                end
        singleton
                -> x: G#1
                ensure
                    count = 1 -- one_element
                    has -- has_x
                end
        feature
            has: BOOLEAN
                -> x: G#1

            is_empty: BOOLEAN
                ensure
                    Result = (count = 0) -- count_zero
                end
            for_all: BOOLEAN
                -> test: PREDICATE [ANY, TUPLE [G#1]]
                require
                    test /= Void -- test_exists
                    test. = 1 -- test_has_one_arg
                end
            exists: BOOLEAN
                -> test: PREDICATE [ANY, TUPLE [G#1]]
                require
                    test /= Void -- test_exists
                    test. = 1 -- test_has_one_arg
                end
        feature
            any_item: G#1
                require
                    not is_empty -- not_empty
                ensure
                    has -- element
                end
            extremum: G#1
                -> order: PREDICATE [ANY, TUPLE [G#1, G#1]]
                require
                    not is_empty -- not_empty
                    order /= Void -- order_exists
                    order. = 2 -- order_has_one_arg
                ensure
                    has -- element
                    for_all -- extremum
                end
        feature
            filtered: MML_SET [G#1]
                -> test: PREDICATE [ANY, TUPLE [G#1]]
                require
                    test /= Void -- test_exists
                    test. = 1 -- test_has_one_arg
                ensure
                    Result <= Current -- subset
                    Result -- satisfies_test
                    not Current - Result -- maximum
                end
        feature
            count: INTEGER

        feature
            is_model_equal: BOOLEAN
                -> other: MML_MODEL

            is_subset_of: BOOLEAN
                -> other: MML_SET [G#1]
                require
                    other /= Void -- other_exists
                ensure
                    Result = for_all -- other_has_all
                end
            is_superset_of: BOOLEAN
                -> other: MML_SET [G#1]
                require
                    other /= Void -- other_exists
                ensure
                    Result = (other <= Current) -- other_is_subset
                end
            disjoint: BOOLEAN
                -> other: MML_SET [G#1]
                require
                    other /= Void -- other_exists
                ensure
                    Result = not other. -- no_common_elements
                end
        feature
            extended: MML_SET [G#1]
                -> x: G#1
                ensure
                    Result ^ (Current +) -- singleton_union
                end
            removed: MML_SET [G#1]
                -> x: G#1
                ensure
                    Result ^ (Current -) -- singleton_difference
                end
            union: MML_SET [G#1]
                -> other: MML_SET [G#1]
                require
                    other /= Void -- other_exists
                ensure
                    Current <= Result -- contains_current
                    other <= Result -- contains_other
                    Result -- minimal
                end
            intersection: MML_SET [G#1]
                -> other: MML_SET [G#1]
                require
                    other /= Void -- other_exists
                ensure
                    Result <= Current -- contained_in_current
                    Result <= other -- contained_in_other
                    for_all -- maximal
                end
            difference: MML_SET [G#1]
                -> other: MML_SET [G#1]
                require
                    other /= Void -- other_exists
                ensure
                    Result <= Current -- contained_in_current
                    Result -- disjoint_from_other
                    for_all -- maximal
                end
            sym_difference: MML_SET [G#1]
                -> other: MML_SET [G#1]
                require
                    other /= Void -- other_exists
                ensure
                    Result ^ ((Current + other) - (Current * other)) -- definition
                end
        feature {MML_MODEL}
            array: V_ARRAY [G#1]

            make_from_array
                -> a: V_ARRAY [G#1]
                require
                    a /= Void -- a_exists
                    a. = 1 -- starts_from_one
                    a. -- no_duplicates
                end
            meq: BOOLEAN
                -> v1: G#1
                -> v2: G#1

        invariant
```

```
        array /= Void -- array_exists
        array. = 1 -- starts_from_one
end
class MML_INTERVAL
indexing
        description: "Closed␣integer␣intervals.";
        author: "Nadia␣Polikarpova";
        date: "$Date$";
        revision: "$Revision$";
        belongs_to: "CLUSTER_OF_MML_MODEL"
inherit
        MML_SET

feature {NONE}
        from_range
                -> l: INTEGER
                -> u: INTEGER

        from_tuple
                -> t: TUPLE [l: INTEGER; u: INTEGER]
                require
                        t /= Void -- t_exists
                end
feature
        lower: INTEGER
                require
                        not is_empty -- not_empty
                end
        upper: INTEGER
                require
                        not is_empty -- not_empty
                end
end
class MML_MAP [K, V]
indexing
        description: "Finite␣maps.";
        author: "Nadia␣Polikarpova";
        date: "$Date$";
        revision: "$Revision$";
        belongs_to: "CLUSTER_OF_MML_MODEL"
inherit
        MML_MODEL

feature {NONE}
        default_create
                        --  Is this model mathematically equal to 'other'?


        singleton
                        --  Are 'v1' and 'v2' mathematically equal?
                        --  If they are models use model equality,
                        --  otherwise reference equality.
                -> k: G#1
                -> x: G#2

feature
        has: BOOLEAN
                -> x: G#2

        is_empty: BOOLEAN

        is_constant: BOOLEAN
                -> c: G#2

feature
        item: G#2
                -> k: G#1
                require
                        bracket_error -- in_domain
                end
```

```
    feature
        domain: MML_SET [G#1]

        range: MML_SET [G#2]

        image: MML_SET [G#2]
            -> subdomain: MML_SET [G#1]
            require
                subdomain /= Void -- subdomain_exists
            end
        sequence_image: MML_SEQUENCE [G#2]
            -> s: MML_SEQUENCE [G#1]
            require
                s /= Void -- sequence_exists
            end
        to_bag: MML_BAG [G#2]

    feature
        count: INTEGER

    feature
        is_model_equal: BOOLEAN
            -> other: MML_MODEL

    feature
        updated: MML_MAP [G#1, G#2]
            -> k: G#1
            -> x: G#2

        removed: MML_MAP [G#1, G#2]
            -> k: G#1

        restricted: MML_MAP [G#1, G#2]
            -> subdomain: MML_SET [G#1]
            require
                subdomain /= Void -- subdomain_exists
            end
        override: MML_MAP [G#1, G#2]
            -> other: MML_MAP [G#1, G#2]
            require
                other /= Void -- other_exists
            end
        inverse: MML_RELATION [G#2, G#1]

    feature {MML_MODEL}
        keys: V_ARRAY [G#1]

        values: V_ARRAY [G#2]

        make_from_arrays
            -> ks: V_ARRAY [G#1]
            -> vs: V_ARRAY [G#2]
            require
                ks /= Void -- ks_exists
                vs /= Void -- vs_exists
                ks. = vs. -- same_lower
                ks. = vs. -- same_upper
                ks. = 1 -- start_from_one
                ks. -- ks_has_no_duplicates
            end
        meq_key: BOOLEAN
            -> k1: G#1
            -> k2: G#1

        meq_value: BOOLEAN
            -> v1: G#2
            -> v2: G#2

    invariant
        keys /= Void -- keys_exists
```

```
                        values /= Void -- values_exists
                        keys. = values. -- same_lower
                        keys. = values. -- same_upper
                        keys. = 1 -- start_from_one
                end
        end
end
```