

Contract Wizard 3.0: Developing a GUI

Diploma Thesis

Petra Marty

Supervised by Prof. Dr. Bertrand Meyer and Dr. Karine Arnout

Chair of Software Engineering
ETH Zürich

August 2004

Project period April 26, 2004 – August 25, 2004

Student name Petra Marty

Email address petra@computerscience.ch

Supervisor name Dr. Karine Arnout

Email address karine.arnout@inf.ethz.ch

Professor name Prof. Dr. Bertrand Meyer

Email address bertrand.meyer@inf.ethz.ch

Chair of Software Engineering

Department of Computer Science, ETH Zürich



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

ACKNOWLEDGMENTS

First of all I want to thank Dr. Karine Arnout for her supervision, her scientific support and her very helpful comments and suggestions on this diploma thesis. I am grateful to Prof. Dr. Bertrand Meyer who made this master thesis possible.

I thank Dominik Wotruba. He developed the tool I extended and helped me when I had questions regarding his implementation. I would also like to thank all the members and master students of the Chair of Software Engineering at ETH Zürich for their valuable comments concerning the wizard. My thanks especially go to Beat Fluri, Olivier Jeger, Marcel Kessler and Joseph N. Ruskiewicz for their constructive comments and support. Working in this group has been a real pleasure for me.

I also would like to thank Hans Dubach for his great help and support in dealing with administrative questions during my whole study at ETH.

Finally, I am deeply grateful to my parents for their support in every respect.

Zürich, im August 2004

Petra Marty

ABSTRACT

Design by Contract is a crucial concept to build reliable software components. However contracts are still a specificity of the Eiffel language. With the Contract Wizard 3.0 it is possible to add contracts like preconditions, postconditions, and invariants to an arbitrary .NET assembly, even if it was not initially written in Eiffel.

The main goal of this project was to enrich the existing tool with a graphical user interface. It is now possible to add contracts in an easy and intuitive way. The user can also edit the assertions of an already contracted assembly.

TABLE OF CONTENTS

1.	Introduction.....	2
2.	Contract Wizard 2.0: the existing tool.....	3
2.1	How does it work?	3
2.2	Architecture.....	4
2.2.1	Abstract syntax tree	4
2.2.2	Parser	5
2.2.3	Code generator.....	7
3.	Gui design and implementation.....	9
3.1	GUI design patterns	9
3.2	Functionality	11
3.3	Implementation	13
3.3.1	Main window	13
3.3.2	Project dialog.....	15
3.3.3	Assembly dialog	16
3.3.4	Information dialog	22
3.3.5	Progress bar	23
3.3.6	Error dialog.....	25
4.	Contract Wizard 3.0: additional functionality.....	27
4.1	New functionality.....	27
4.1.1	Project file.....	27
4.1.2	Syntax check.....	29
4.1.3	Compilation	32
4.1.4	Backup	33
4.2	Assembly parsing.....	33
4.2.1	.NET interface.....	34
4.2.2	.NET parser.....	37
4.2.3	Eiffel names for .NET types	38
4.2.4	Overloaded names	39
4.2.5	Abstract syntax tree	44
4.3	Code generation	47
4.3.1	Eiffel visitor	47
4.3.2	Ace file	47
4.3.3	XML visitor	49
4.3.4	HTML	49
4.4	Command line interface	51
5.	Advantages and limitations.....	53
5.1	Benefits of using Contract Wizard.....	53
5.2	Limitations	53
5.3	Future Work	58

6.	User Manual	61
6.1	System requirement.....	61
6.2	Installation.....	61
6.3	How to create a .NET assembly	62
6.4	How to use Contract Wizard 3.0.....	64
6.4.1	Adding assembly to GAC.....	64
6.4.2	Gui version	64
6.4.3	Command-line version.....	78
7.	Conclusion	80
A.	Intended results.....	81
	References	83

LIST OF FIGURES

Figure 1: Contract Wizard architecture	3
Figure 2: Internal representation of proxy classes	5
Figure 3: Eiffel code generator	8
Figure 4: Save as dialog in TextPad	10
Figure 5: Main dialog of Contract Wizard	12
Figure 6: Main GUI classes of Contract Wizard	14
Figure 7: Classes corresponding to CW_PROJECT_DIALOG	16
Figure 8: Classes corresponding to the assembly dialog	17
Figure 9: CW_INFORMATION_DIALOG	23
Figure 10: Screenshot of progress dialog	24
Figure 11: CW_ERROR_DIALOG	26
Figure 12: Creating a key pair	63
Figure 13: Deploy assembly into global assembly cache	63
Figure 14: .NET Configuration 1.1, assembly cache	64
Figure 15: Warning dialog when the CONTRACT variable is not set	65
Figure 16: Welcome dialog	65
Figure 17: Project dialog	66
Figure 18: Warning dialog when specified project already exists	67
Figure 19: Warning dialog when working directory could not be created	67
Figure 20: Warning dialog for an invalid assembly	68
Figure 21: Progress dialog while parsing a .NET assembly	68
Figure 22: Assembly dialog with invariant view	70
Figure 23: Warning dialog when you try to add the same invariant twice	71
Figure 24: Assembly dialog with precondition and postcondition view	71
Figure 25: Question dialog if all preconditions shall be removed	72
Figure 26: Warning dialog when user wants to contract an attribute	73
Figure 27: How to show contract view of a type	73
Figure 28: Launched browser with interface view of selected class CW_ACCOUNT	74
Figure 29: Error dialog after a syntax error in one of the checked classes	75
Figure 30: Progress dialog showing compilation	76
Figure 31: Error dialog showing details about compilation error	76
Figure 32: Last dialog after a successful generation of a contracted .NET assembly	77
Figure 33: Question dialog asking to restore the working directory before ending Contract Wizard	77

LIST OF TABLES

Table 1: Main features of CW_MAIN_WINDOW	15
Table 2: Actions on tree items in CW_TREE	18
Table 3: Feature subset of CW_LIST_BOX	19
Table 4: Access features of CW_ASSERTION_DIALOG	20
Table 5: Code snippet from add_condition	22
Table 6: Code snippets for progress dialog	25
Table 7: Creation procedure of CW_ERROR_DIALOG	26
Table 8: Project file	27
Table 9: Creation procedure of CW_PROJECT_HANDLER	28
Table 10: Second set of classes	30
Table 11: get_enumerator function	30
Table 12: Use of CW_GELINT in CW_MAIN_WINDOW	32
Table 13: Use of CW_COMPILER_LAUNCHER in CW_MAIN_WINDOW	32
Table 14: C# classes versus generated Eiffel classes	35
Table 15: C# classes with explicit interface member implementation versus generated Eiffel classes	36
Table 16: Compilation error CW_STRING_WRITER	37
Table 17: Translation of special mscorlib types	38
Table 18: Overloaded .NET methods from System.IO.TextWriter	39
Table 19: Disambiguated Eiffel function names	39
Table 20: Disambiguated .NET constructors	40
Table 21: extend_feature from CW_LIST_ITEM	42
Table 22: Disambiguated Eiffel function names	43
Table 23: Document type definition for generated Eiffel classes	46
Table 24: Class CW_CLR_VERSION	48
Table 25: Feature is_postcondition of CW_XML_VISITOR	49
Table 26: Features put_basic and put_span of CW_HTML_TEXT	50
Table 27: Class interface of IPRINCIPAL	54
Table 28: Relevant features from class CW_WINDOWS_PRINCIPAL	55
Table 29: Part of class interface of ITYPE_LIB_CONVERTER	56
Table 30: Extracts of class CW_CASE_INSENSITIVE_HASH_CODE_PROVIDER	57
Table 31: Compiler error of class CW_SERIALIZATION_INFO_ENUMERATOR	58
Table 32: Suggestion of inheritance structure of CW_TYPE_LIB_CONVERTER	59
Table 33: Factory-like Contract Wizard proxy class	60
Table 34: Output of contract_wizard -help command	79

1. INTRODUCTION

.NET assemblies do not have contracts. To build more reliable software and improve existing components, the Contract Wizard enables a user to interactively add contracts to the classes and routines of a .NET assembly. The tool parses an assembly using the reflection mechanism of .NET, generates Eiffel proxy classes containing the added contracts, and compiles them into a new .NET assembly. The contracted proxy assembly can now be used instead of the original assembly. Besides the Eiffel classes, the wizard also generates an XML file to store the contracts. This enables the client to append contracts to an already contracted assembly.

The purpose of this project is to extend the existing features of the Contract Wizard. The resulting tool – Contract Wizard 3.0 – has a graphical user interface (GUI) to add contract to a .NET assembly. The GUI displays the types of the assembly along with their features in a browser-like tree. The user can add contracts by entering assertions in the provided text fields. It is also possible to edit or remove existing invariants, preconditions, and postconditions. The tool gives feedback to the user by showing the progress while parsing an assembly, generating Eiffel files, or compiling the generated source code. A dialog with an accurate error message informs the user when the syntax check or the compilation did not succeed.

First, this thesis gives a description of the existing tool; then it describes the design and implementation of the graphical user interface. Following the thesis illustrates the new functionalities of the Contract Wizard 3.0 and discusses the benefits and limitations of the tool. A user manual for the Contract Wizard precedes the conclusion.

Note: For simplicity reasons, this report uses words such as “he” and “his” to refer to unspecified persons, instead of using longer constructs such as “he or she” and “his or her”, with no connotation of gender.

2. CONTRACT WIZARD 2.0: THE EXISTING TOOL

This diploma thesis is based on an existing work called Contract Wizard 2.0, implemented at ETH Zürich by Dominik Wotruba [1]. The goal of the current project was to extend the Contract Wizard. Therefore, I briefly describe its functionality and architecture.

2.1 HOW DOES IT WORK?

The Contract Wizard reads a .NET assembly and separately provided contracts. It merges this information and automatically generates Eiffel proxy classes containing the given contracts. Then, the Eiffel classes are compiled into a new contracted .NET assembly. In fact, the new assembly is a proxy to the original one, in the sense that it has the same interface but its implementation just forwards every call to the corresponding method in the original assembly after the contracts have been checked.

After a successful compilation the Contract Wizard generates an XML representation of the contracted Eiffel proxy classes. This enables the client to append contracts to an already contracted assembly and also to edit them. Dominik Wotruba has shown that it is faster to read the content of the proxy classes from the XML file than from the .NET assembly directly [1]. In a repeated use of the same assembly the Contract Wizard uses the XML file to build the internal representation used to generate a proxy classes.

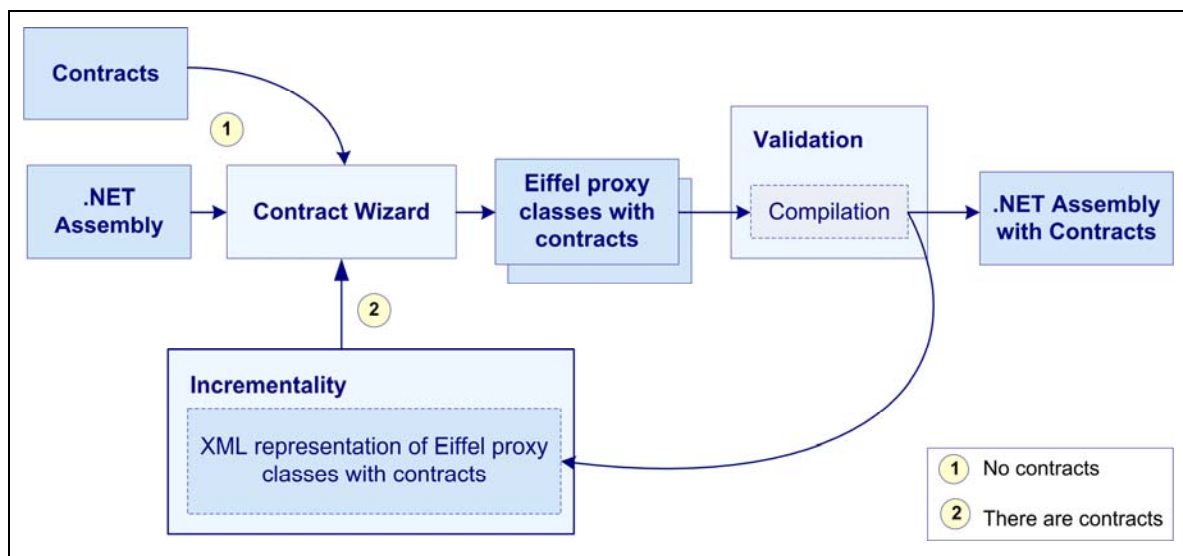


Figure 1: Contract Wizard architecture

2.2 ARCHITECTURE

The most important parts of the Contract Wizard are the parsers (.NET parser, XML parser) and the code generators (Eiffel proxy classes generator, XML generator). In the following sections I will go through their architecture and implementation; I will explain what is needed to understand the improvements I made on these classes (described in chapter 4).

2.2.1 ABSTRACT SYNTAX TREE

The class *CW_CONTROLLER* decides if the wizard parses the .NET assembly or the XML file. For that it looks up whether a XML file exists or not. The .NET parser and the XML parser produce an abstract syntax tree (*AST*) that contains the data to produce Eiffel proxy classes and their XML representation containing the contracts.

The *AST* is a list of elements of type *CW_TYPE*. An instance of *CW_TYPE* has features (*CW_FEATURE*), interfaces (*CW_INTERFACE*) and invariants (*CW_INVARIANT*). A feature can either be a routine (*CW_ROUTINE*) or an attribute (*CW_ATTRIBUTE*). Attributes represent fields; routines represent computations applicable to all instances of a class. A routine may have multiple arguments (*CW_ARGUMENT*). It is further classified into a function (*CW_FUNCTION*) if it returns a result or a procedure (*CW_PROCEDURE*) otherwise. Each feature has a list of preconditions (*CW_PRECONDITION*) and postconditions (*CW_POSTCONDITION*).

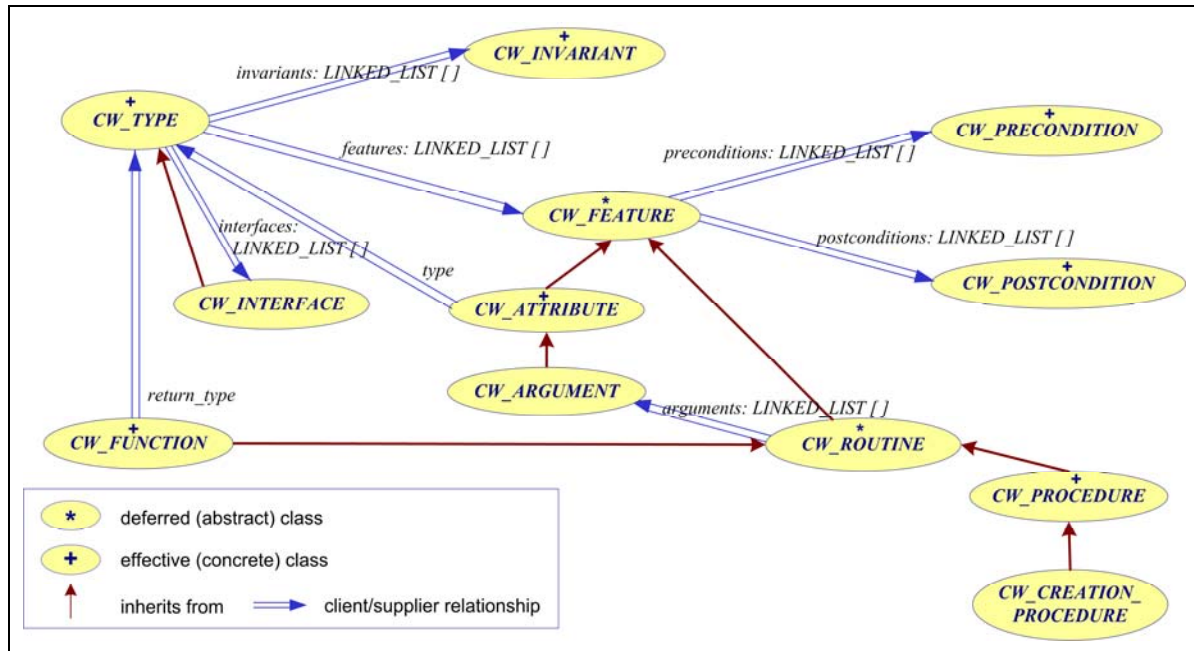


Figure 2: Internal representation of proxy classes

The interfaces are the interfaces that the original .NET type implements or inherits from. Each interface keeps track of feature names which have to be undefined in the proxy class later. For more details, please refer to 0.

The architecture takes into account the future extension of Eiffel where it will be possible to have assertions on attributes [6] [8], which is not possible at the moment [5].

2.2.2 PARSER

As already mentioned there are two parsers: one for parsing a .NET assembly and one for reading an XML representation of the contracts.

.NET PARSER

The .NET parser (*CW_DOTNET_PARSER*) produces the *AST* by retrieving information from the .NET assembly using the reflection mechanism of .NET. It parses the following information required to generate Eiffel proxy classes:

- public types
- public .NET constructors
- public .NET methods
- public .NET fields

For every inspected information a specific type node of the *AST* is instantiated and appended to it (see Figure 2: Internal representation of proxy classes):

- *CW_TYPE* for each public type
- *CW_ATTRIBUTE* for every inspected public field
- *CW_PROCEDURE* respectively *CW_FUNCTION* for each method
- *CW_CREATION_PROCEDURE* for every parsed constructor

To generate the Eiffel proxy classes correctly, the generator has to know more than just the name of an *AST* node. Important information can be: is a type deferred or expanded, is a feature static or deferred, what is the return type of a function? To satisfy these needs every node type holds some specific attributes that the parser sets. For each type (*CW_TYPE*) the parser sets the following queries:

- *is_deferred* (is type deferred?)
- *is_expanded* (is type expanded?)
- *is_array* (is type an array?)
- *is_enum* (is it an enumeration type?; an enumeration type is a distinct type with named constants)
- *is_interface* (does type represent a .NET interface?)

Moreover, the parser attaches to a type all interfaces (*CW_INTERFACE*) which it implements. An attribute is enriched with the following information:

- *is_static* (is it a static field?)
- *is_constant* (does the field represent a constant value?)
- *value* (if the parsed field is a constant, a constant value is assigned)

For each procedure and function, the parser sets the following attributes:

- *is_static* (is method static?; static method are accessed through the class)
- *is_deferred* (is method deferred?; deferred methods are methods declared in a .NET interface type)

Additionally a function has the attributes:

- *return_type* (the function's return type)
- *is_property* (a property is a member that provides access to a characteristic of an object or a class, attribute is set to true if the function is a get accessor)

A helper class *CW_NAME_FORMATTER* translates the .NET names into Eiffel names. The .NET naming style is "CamelCase" and Eiffel's convention is to use lower case and to separate names with an underscore. For example "toString" becomes "to_string". This class *CW_NAME_FORMATTER* also handles the translation of: .NET primitive types, .NET class names that conflict with the EiffelBase library and, .NET names that conflict with keywords of the Eiffel language.

XML PARSER

The XML parser (*CW_XML_PARSER*) reads the XML representation of the Eiffel proxy classes – that may contain contracts – and builds the corresponding *AST*. It uses the Eiffel parser (*XM_EIFFEL_PARSER*) that is part of the Gobo library [3].

The structure of the XML file is defined through a document type definition (*DTD*). It can be used to validate an XML file. (A listing of the *DTD* can be found in section 4.2.5: Abstract syntax tree.)

2.2.3 CODE GENERATOR

The generator uses the *AST* obtained by the parser to create Eiffel proxy classes and their XML representation. Initially there were three generators: an Eiffel generator (*CW_EIFFEL_GENERATOR*), a Lace¹ generator (*CW_LACE_GENERATOR*), and a XML generator (*CW_XML_GENERATOR*). The new distribution has an additional generator called *CW_HTML_GENERATOR* (see section 4.3.4). Each generator class has a specialized visitor that traverses the *AST* and produces code corresponding to the node type of the *AST*.

The class *CW_EIFFEL_GENERATOR* generates the Eiffel classes using an Eiffel visitor (*CW_EIFFEL_VISITOR*). It stores the classes in a directory that the user specified at the beginning of the program execution.

For every type of an *AST* node (e.g. *CW_TYPE*, *CW_PROCEDURE*...) the generated code structure looks different. The visitor simplifies the code generation considerably: it defines a feature for each type of an *AST* node to treat every node individually. For instance, an *AST* node of type *CW_PROCEDURE* is visited through the feature *visit_procedure*. This feature generates Eiffel code specific to a procedure, including preconditions and postconditions. The Eiffel visitor specifies features for the generation of creation procedures, attributes, functions, invariants, and types.

¹ Language for the Assembly of Classes in Eiffel (Ace), the control file of Eiffel projects

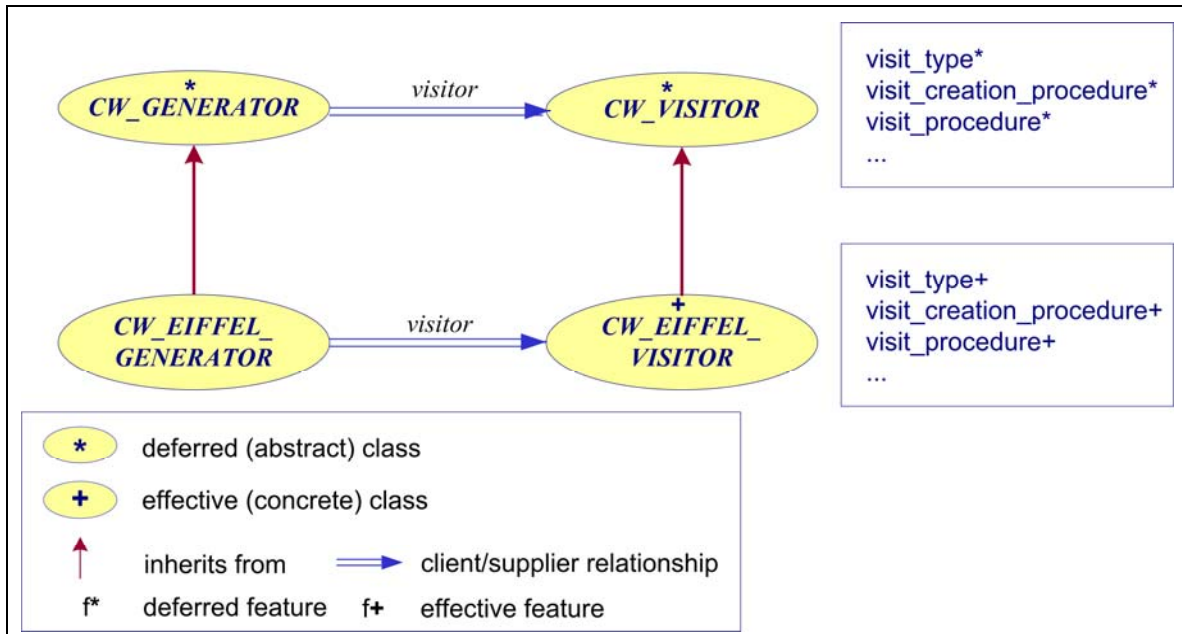


Figure 3: Eiffel code generator

The class *CW_LACE_GENERATOR* generates an Ace file which is needed to compile the obtained Eiffel classes. The class *CW_XML_GENERATOR* takes care of producing the XML file with all types, features, and assertions contained in the *AST*. Here, the generator uses the class *CW_XML_VISITOR* to produce a specific XML representation for each node type.

3. GUI DESIGN AND IMPLEMENTATION

The main part of this thesis was to extend the existing Contract Wizard with a Graphical User Interface (GUI). In this section I describe the design and implementation of the added GUI.

3.1 GUI DESIGN PATTERNS

Before starting with the development of the GUI, I had a look on how to design a user friendly GUI. Several GUI design patterns help to fulfill user demands such as ease of use, flow, visual stimulation, consistency, and ease of navigation. The design patterns can be divided into several categories like presentation, selection, guidance/feedback and navigation, as proposed in [10].

Below I list the patterns which I consider important to develop a GUI for the Contract Wizard II.

GRID LAYOUT

When several information objects are presented and arranged spatially on a limited area, it is important that the user quickly understands the information and takes action depending on that information.

Arranging all objects in a grid using the minimal number of rows and columns improves the time needed to scan the information. It also causes a consistent layout with minimal visual clutter. Objects that are of the same type must be aligned and displayed in the same way.

CONTEXTUAL MENU

At any point in time, the user needs to know what his possibilities are in order to decide what to do, especially when the number of possibilities is large and not all possibilities are available in the current context.

A good example is the “Edit” submenu from Microsoft Word: the functions in the menu are semantically grouped and only the actions available in the current context are highlighted; all other menu items are disabled.

UNAMBIGUOUS FORMAT

In situations where the user needs to supply the application with structured data, the system designer cannot be sure that the user enters the data in the correct format. The main idea here is to avoid entering incorrect data by making it impossible to enter wrong data. If possible avoid fields

where users can type free text, otherwise explain the syntax with an example or a description of the format. Also advisable is to provide sound defaults for required fields.

SHIELD

A shield is used to prevent the user from accidentally selecting a function that has irreversible effects or requires a lot of resource to reverse. In the extra protection layer the user is asked to confirm his/her intent with the default answer being the safe option. An example is the “Save As” dialog in a text editor when the file already exists at the specified location. Overwriting it would result in loss of the copy.

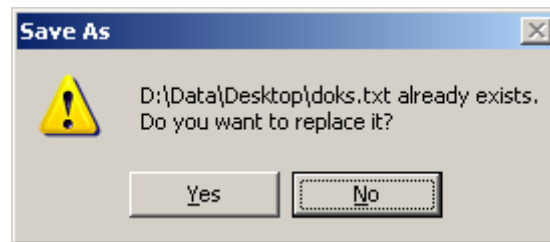


Figure 4: Save as dialog in TextPad

PROGRESS

When a system task takes a long time (typically more than a few seconds) and must be completed before another successive task can start it is useful to deal with a progress bar. In that case the user knows that the operation is being performed and approximately how long the task will take until it is finished. The user may also not be familiar with the complexity of the task and he might decide to interrupt it because it takes too much time. A progress bar with the estimated time until completion is very helpful when you for example download a file from the internet. When there is a too slow connection, the task can be aborted.

WIZARD

A wizard is useful when a user needs to perform a complex task consisting of several subtasks where decisions need to be made in each subtask and the number of subtasks is small (typically between three and ten steps). The wizard guides the user in the right order through the tasks; therefore it is guaranteed that decisions necessary to perform the actual subtask have been taken.

The user receives feedback about the purpose of each task and when each complex task is completed. By using a navigation widget the user can go to the next task. The user is able to revise a decision by navigating back to a previous task. At any point in the sequence it is possible to abort the task.

NAVIGATING SPACES

When the user needs to access an amount of information which cannot be put on the available space, the information can be shown in several spaces. Large amounts of data are usually not unrelated and can be divided into categories that match the user's conceptual model of the data.

This pattern suggests the user to navigate between the spaces. If the number of spaces is small (e.g. less than eight), the navigation areas should be placed at the top using a tab control. When the number of spaces is large, the navigation area should be placed on the left side of the spaces using a tree structure.

3.2 FUNCTIONALITY

As the name suggests, the Contract Wizard is designed as a wizard; it is divided into four subtasks. The first window of the Contract Wizard informs the user about what he can achieve by using the wizard. The following dialog asks the user to open an existing project or to create a new project. A project file always has the extension `cpr`. For a new project, the following information is needed:

- full path to the assembly to be contracted
- working directory where to store generated classes and XML file
- project name

Default values are provided, except for the assembly path. The assembly and the directory can be entered directly in the corresponding text fields or by browsing in a file, respectively directory dialog. A filter in the file dialog makes sure that only files with valid extensions are selected (e.g. for an assembly: files with extension `.dll` or `.exe`). If the specified directory and the `.NET` assembly are valid, the Contract Wizard backs up all files from the project directory, loads the assembly and parses it. A progress dialog shows the advancement.

The next task is the most important one. It is the place where the user can add contracts to the selected `.NET` assembly. In detail the user is able to:

- look at types and features of a `.NET` type through browsing in a tree
- add, edit and remove (all) invariants of a selected `.NET` type
- add, edit and remove (all) preconditions or postconditions of a selected `.NET` method
- look at the Eiffel contract view of each `.NET` type (with updated assertions)
- look at the signature of each feature
- pick-and-drop a feature's signature in the text box where assertions are entered

When a user asks to delete all invariants of a type, a dialog appears and the user must confirm he really wants to execute the operation.

The dialog is divided into several parts. An horizontal box at the top displays the name of the selected assembly and its path. On the left-hand side is a tree – the *assembly tree* – that reflects the public types of the assembly. The public fields and methods of a type are subnodes of the type. A double click on a feature node, a right click on a type node, or a pres on the key “F4” launches a browser which displays the Eiffel contract view (feature signature with pre- and postconditions, type invariants) of the appropriate class/feature. The right-hand side provides two text boxes to enter the assertion: one for the assertion tag, the other for the assertion expression. Depending on the selected node type, the invariants of a type, or the pre- and postconditions of a feature are listed in the widget above the text fields. Figure 5 shows a screenshot of the main dialog, called *assembly view*.

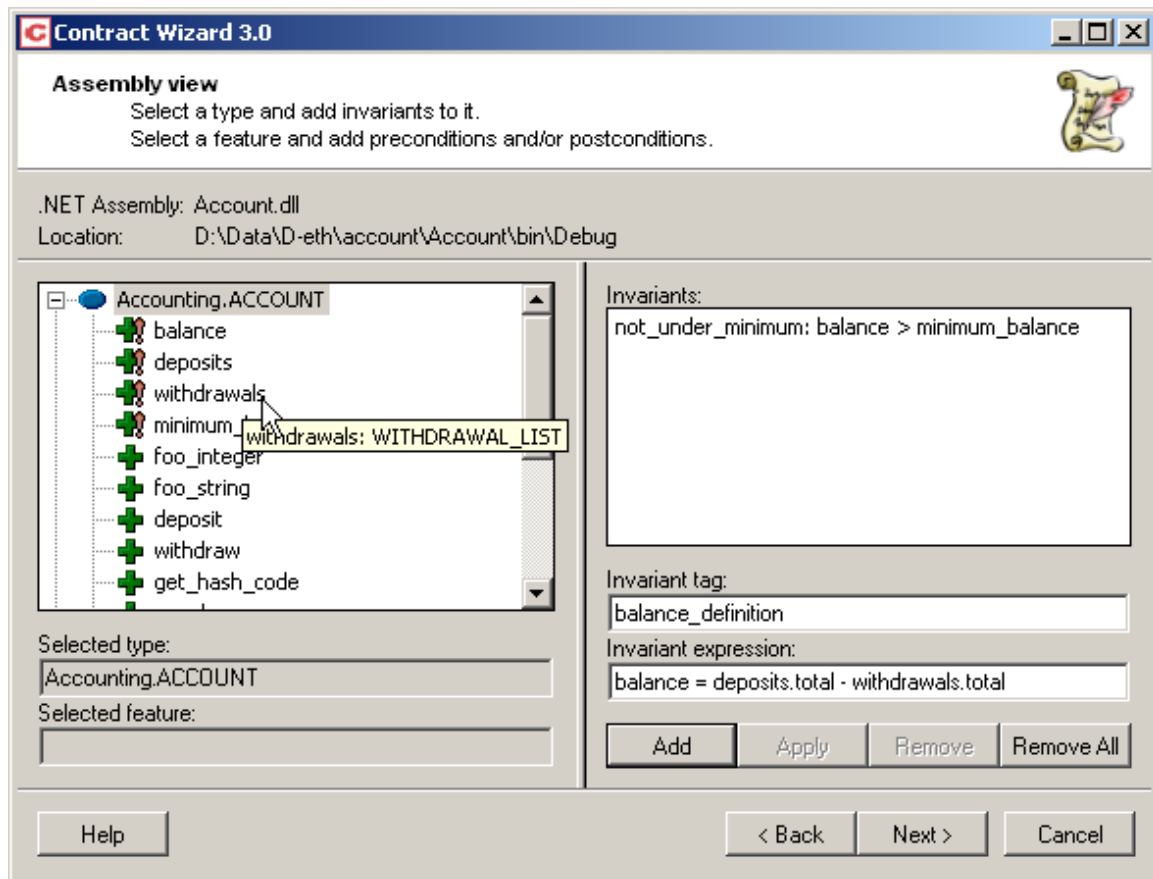


Figure 5: Main dialog of Contract Wizard

Because .NET supports overloading it is possible that a class contains more than one feature with the same name but different signature. To avoid confusion these features are displayed in both *assembly tree* and contract view with their corresponding Eiffel name. Chapter 4.2.4 discusses overloaded .NET members.

A click on button “Next >” launches the code generator to create the Eiffel proxy classes, the Ace file and the XML file. The code generator stores them in the project directory. Then the Contract Wizard checks the syntax of the proxy classes. If not error occurred, the wizard launches

the compiler to create a new, contracted .NET assembly; otherwise a dialog states the exact error. Again, a progress dialog makes sure that the user knows how long each task will still take.

In a last dialog the user is informed where the new assembly has been stored (in the EIFGEN\F_CODE directory relatively to the project directory).

It is always possible to cancel the execution of the Contract Wizard. If so, the wizard restores the original state of the project directory. More on the functionality and how to use the Contract Wizard can be found in chapter 6.

3.3 IMPLEMENTATION

Reusability has been a key concept during the development. Especially when working with GUI it is likely that single components can be reused. An important point in the architecture is that the GUI is independent from the business logic. Initially contracts were added to an assembly through a command line; this functionality is still available. The root class for the command line version is *CONTRACT_WIZARD_TUI*. The GUI is built with the EiffelVision2 library provided by ISE Eiffel.

3.3.1 MAIN WINDOW

The entry point of the wizard is the class *CONTRACT_WIZARD* which inherits from *EV_APPLICATION*. The root class first checks if the environment variable \$CONTRACT, pointing to the contract wizard delivery directory, is set. Afterwards it initializes the wizard and launches the application. *CW_MAIN_WINDOW* inherits from *EV_TITLED_WINDOW* and represents window of the wizard. The main window maintains a list of all dialogs appearing in the wizard and replaces the actual dialog displayed in the window when the user presses on the “Next >>” button. Each dialog inherits from *CW_DIALOG*. This class has an attribute *box* of type *EV_BOX* that a specialized class redefines either to *EV_VERTICAL_BOX* or *EV_HORIZONTAL_BOX* depending on the main layout of the actual task. Moreover the class has the deferred features:

- *initialize_box* (initialize *box* and class attributes)
- *is_complete* (are all components of the class filled out as needed?)
- *build_interface* (build interface for task and extend widgets to *box*).

I will discuss the specializations of *CW_TASK* beneath. Figure 6 shows the architecture of the main GUI.

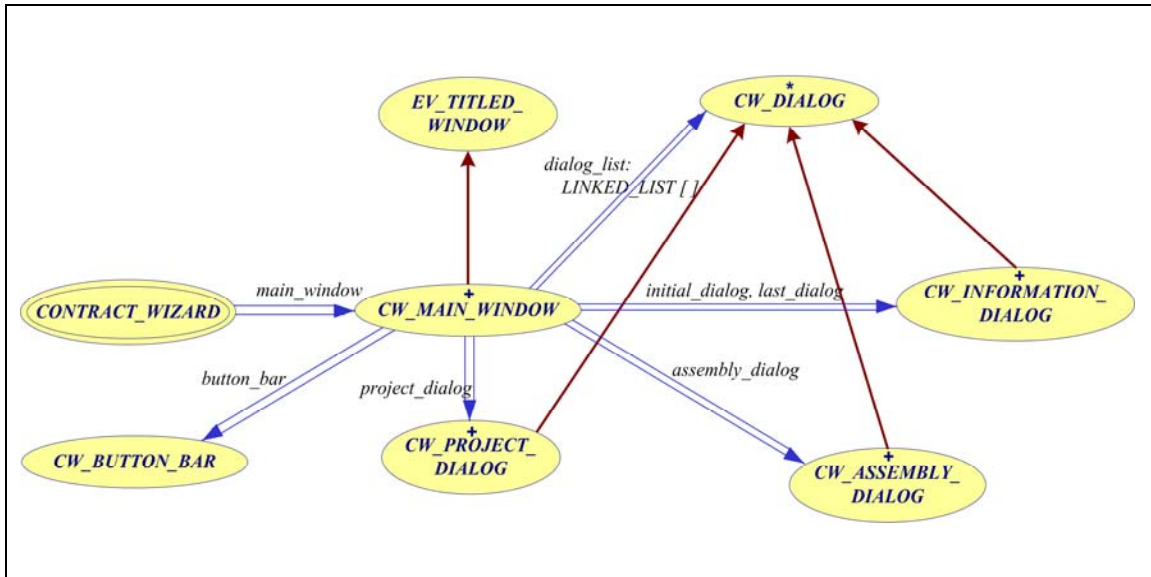


Figure 6: Main GUI classes of Contract Wizard

CW_MAIN_WINDOW also handles navigation between dialogs. For that it has a *button_bar* with “Help”, “<< Back”, “Next >>”, and “Cancel” buttons at the bottom of the window. The main window handles the button actions because a dialog itself knows nothing about its preceding and following dialog. I have chosen this design so that it is easier to have several possible subsequent dialogs to one dialog. Table 1 shows the most important features of class *CW_MAIN_WINDOW* in their contract form.

```

dialog_list: LINKED_LIST[CW_DIALOG]
    -- List with all needed dialog implementations.

is_first_dialog: BOOLEAN
    -- Is first dialog shown?
ensure
    first_dialog_definition: Result = (main_vbox.item = dialog_list.first.box)

is_last_dialog: BOOLEAN
    -- Is last dialog shown?
ensure
    last_dialog_definition: Result = (main_vbox.item = dialog_list.last.box)

go_i_th_dialog (i: INTEGER)
    -- Show `i` th dialog.
require
    i_within_bounds: i > 0 and then i <= dialog_list.count
ensure
    i_th_dialog_active: dialog_list.item = dialog_list.i_th (i)
    i_th_frame_shown: dialog_list.item.box = main_vbox.item
  
```

```

set_button_state
    -- Set button state according to shown dialog.
ensure
    back_button_sensitive: is_first_dialog implies not
        button_bar.back_button.is_sensitive
    next_button_text: is_last_dialog implies
        button_bar.next_button.text.is_equal (finish_button_text)

cancel
    -- User wants to cancel application.

go_next
    -- Show next task.

go_back
    -- Show previous task.

invariant
    initial_dialog_not_void: initial_dialog /= Void
    project_dialog_not_void: project_dialog /= Void
    assembly_dialog_not_void: assembly_dialog /= Void
    last_dialog_not_void: last_dialog /= Void
    consistent_dialog_state: main_vbox.item = dialog_list.item.box
    dialog_list_not_void: dialog_list /= Void

```

Table 1: Main features of CW_MAIN_WINDOW

The feature *go_next* inspects which dialog is currently displayed and executes the corresponding action. For example: it checks existence and validation of the project file, .NET assembly, or working directory; it is responsible for the backup of the working directory; it initiates the parsing of the assembly and the code generation; it checks for syntax errors in the generated files; and it launches the compilation. To fulfill these tasks it uses the helper classes: *CW_PROJECT_HANDLER*, *CW_CONTROLLER*, *CW_SUPPORT* *CW_GELINT*, and *CW_COMPILER_LAUNCHER*. At the end of the feature body, *go_next* calls *go_i_th_dialog* with the index of the next dialog compared to *dialog_list*. When errors occur an error dialog with detailed information appears.

The feature *go_back* simply calls *go_i_th_dialog* with an index one less than the current dialog index. The latter procedure replaces the current dialog in the window with the i-th dialog from *dialog_list*.

3.3.2 PROJECT DIALOG

In the *project dialog* (*CW_PROJECT_DIALOG*) the user can either select an existing project or create a new one. The dialog basically consists of two components, namely *new_project_component* and *exist_project_component*. The component for an existing project

(*CW_EXIST_PROJECT*) asks the user for a path on a project file. The new project component (*CW_NEW_PROJECT*) needs a path pointing to a .NET assembly, a working directory, and a project name. Both components use text fields (*CW_TEXT_FIELD*) for entering text, and path fields (*CW_PATH_FIELD*) for entering text associated with a browse button. A click on this button opens a dialog for selecting a file respectively a directory, depending on the set state. The path field has a feature *set_filter*. It is only possible to browse for files that are specified in the filter, e.g. the filter "**.dll;*.exe*" only allows to select files with a valid assembly extension.

The query *is_complete* originally inherited from *CW_DIALOG* is true when the text fields belonging to a project component are filled out; which component depends on the user's selection to create a new project or to open an existing one.

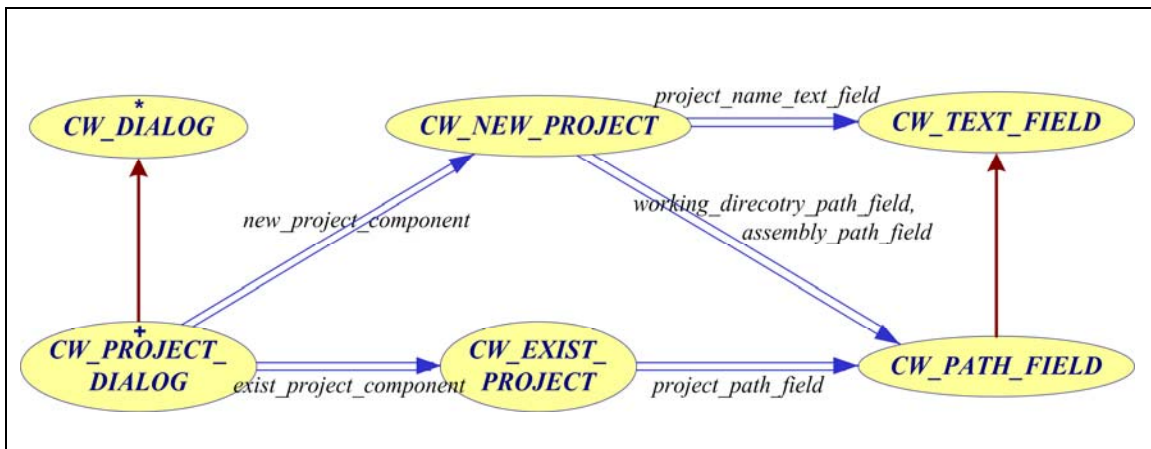


Figure 7: Classes corresponding to *CW_PROJECT_DIALOG*

3.3.3 ASSEMBLY DIALOG

The core functionality of the wizard lies in the *assembly dialog* (*CW_ASSEMBLY_DIALOG*). It enables adding, editing, and deleting contracts of a parsed assembly. Figure 5 shows a screenshot of the dialog. Like in the *project dialog* a taskbar (*CW_TASK_BAR*) is at the top of the dialog. It is a vertical box with a bold label and two normal labels where the user's task of the current dialog is written. Below it is another vertical box, the *assembly_info* (*CW_ASSEMBLY_INFO*). It contains information about the selected assembly. A horizontal split area divides the two main components of the dialog: the *assembly_tree* (*CW_TREE*) that inherits from *EV_TREE* on the left and on its right a component to edit the assertions (*CW_INVARIANT_VIEW* respectively *CW_CONDITION_VIEW*). There are two text fields under the tree: the *type_field* shows the selected type, the *feature_field* shows the selected feature including its signature.

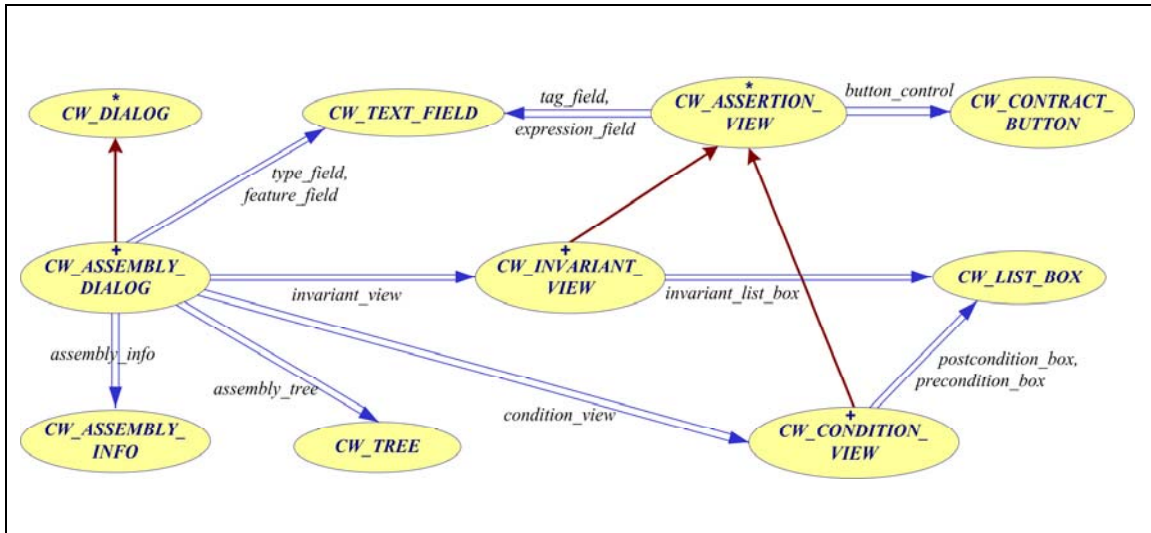


Figure 8: Classes corresponding to the assembly dialog

The *assembly dialog* handles all user inputs concerning assertions and it manipulates the AST directly. The *main_window* registers the AST to the *assembly_dialog* through the procedure *set_ast* right after the parser generated the AST. The feature *set_ast* then calls *assembly_tree.build_tree(ast)* to build the tree representing the types and features of the parsed assembly. The feature *unload_ast* clears the assembly tree, all views and text fields of the *assembly dialog*.

The feature *build_tree* from *CW_TREE* iterates through the types (*CW_TYPE*) of the AST. For every type it creates a tree item *type_tree_item*. The creation procedure of *CW_TYPE_TREE_ITEM* takes an argument of type *CW_TYPE*. For that reason it is always possible to access the type instance of a node, which is particularly useful when a node is selected. The feature *build_tree* extends all public features of a type to every type node. A feature item is of type *CW_FEATURE_TREE_ITEM* and its creation procedure needs a *CW_FEATURE* instead of a *CW_TYPE*. Having an instance of a feature in the feature node allows for example to calculate the signature of a selected feature dynamically. The signature is involved in many ways: a tool tip shows it when the mouse pointer is over a feature's node; it is also used as an argument of *pebble_function*. In Eiffel it is possible to pick a widget and to drop it elsewhere. Data can be assigned to every "picked" widget. In the *assembly tree* the user can right click on a feature node and drop it over the *tag_field* or *expression_field*. As the signature was assigned as pebble it is displayed in the dropped target.

When a tree item is selected it calls all actions registered to it. A right click on a *type_tree_item* shows a context menu to display the Eiffel contract view of the class. A double click on a *feature_tree_item* or a press on the "F4" button also launches the browser with the contract view of the selected node. A right click on a feature node is unhandy because the right mouse button is already assigned to the pebble function. The feature *show_contract_view* determines the type of the selected node, creates an HTML page containing the Eiffel contract view of the selected type, and launches the browser to display the HTML. Table 2 lists a code extract showing how the actions are assigned.

```

select_class_actions: EV_NOTIFY_ACTION_SEQUENCE
    -- Actions when a class node is selected.
select_feature_actions: EV_NOTIFY_ACTION_SEQUENCE
    -- Actions when a feature node is selected.

within build_tree:
type_tree_item.select_actions.extend (agent select_class_actions.call (Void))
feature_tree_item.select_actions.extend (agent select_feature_actions.call (Void))

type_tree_item.pointer_button_press_actions.extend (agent show_menu)
feature_tree_item.pointer_double_press_actions.extend (agent show_contract_view)

```

Table 2: Actions on tree items in CW_TREE

The class *CW_TREE* provides queries whether a tree item is selected: *is_type_selected*, *is_feature_selected*, and *is_node_selected*.

When the user selects a type node in the *assembly tree* the component on the right shows all invariants of the type. The selection of a feature node causes the assertion dialog to show all preconditions and postconditions of the selected feature. The features *set_invariant_view*, *set_condition_view*, and *set_view* (*a_widget: EV_WIDGET*) set the corresponding view. The assertion dialog calls these features when a node is selected.

The *invariant_view* and the *condition_view*, which displays the preconditions and postconditions, have a deferred ancestor class *CW_ASSERTION_VIEW*. This class comprises two text fields: one to enter an assertion tag (*tag_field*) and the other to enter the assertion's expression (*expression_field*). As already mentioned they are drop targets of a feature node. The initialized feature complies this with *tag_field.enable_dropable* and *expression_field.enable_dropable*.

The class has a button bar of type *CW_CONTRACT_BUTTON*. The instance named *button_control* has four buttons labeled: “Add”, “Apply”, “Remove”, and “Remove All”. The class *CW_ASSERTION_VIEW* defines (partially deferred) features to set its sensitivity state. IF for instance an assertion text field is empty it is not possible to add a new assertion; hence the *add_button* is disabled. Whereas both text fields contain an entry, the feature *set_add_button_state* enables the “Add” button. Additionally it draws a frame around the button to indicate that pressing the “Enter” key has the same effect as a clicking on the button. It uses the feature *enable_default_push_button* from *EV_BUTTON*. Because this feature is not exported I created a new class *CW_BUTTON* that inherits from *EV_BUTTON* and exports the needed features (*is_default_push_button*, *enable_default_push_button*, *disable_default_push_button*) to *CW_ASSERTION_VIEW*.

The *invariant_view* contains a list box of type *CW_LIST_BOX* that enlists the invariants of a selected type. The class *CW_LIST_BOX* provides useful features to extend and replace assertions, to query the selected item, and so on. See Table 3 for a selected overview.

```

selected_action: EV_NOTIFY_ACTION_SEQUENCE
    -- Actions that will be called when user select an item in `list_box`.

selected_index: INTEGER
    -- Index of selected_item in `list_box`.

has_item_text (a_text: STRING): BOOLEAN
    -- Does `list_box` have an item with text `a_text`?
require
    a_text_not_empty: a_text /= Void and not a_text.is_empty

is_selected: BOOLEAN
    -- Is an item in `list_box` selected?

extend_assertion (a_tag, an_expression: STRING)
    -- Add assertion with `a_tag` and `a_directory` at end of `list_box`.
require
    tag_not_empty: a_tag /= Void and not a_tag.is_empty
    expression_not_empty: an_expression /= Void and not an_expression.is_empty
ensure
    assertion_extended: has_item_text (a_tag + colon + space + an_expression)
    one_more: list_box.count = old list_box.count + 1

replace_i_th_assertion (i: INTEGER; a_tag, an_expression: STRING)
    -- Replace assertion at `i`th position.
require
    i_within_bounds: i > 0 and then i <= item_count
    ...
ensure
    same_count: list_box.count = old list_box.count
    has_assertion: has_item_text (a_tag + colon + space + an_expression)

remove_all
    -- Remove all items in `list_box`.
ensure
    all_removed: is_box_empty

remove_selected_item
    -- Remove selected item in `list_box`.
require
    item_selected: selected_index > 0 and then selected_index <= item_count
ensure
    one_less: list_box.count = old list_box.count - 1

select_item (i: INTEGER)
    -- Select item with index `i` in `list_box`.
require
    i_within_bounds: i > 0 and then i <= item_count
ensure
    i_th_item_selected: is_selected and (selected_index = 1)

```

Table 3: Feature subset of CW_LIST_BOX

The *precondition_box* and *postcondition_box* of class *CW_CONDITION_VIEW*, both of type *CW_LIST_BOX*, display the contracts of a selected feature. *CW_CONDITION_VIEW* contains two radio buttons to determine whether the user wants to update a precondition or a postcondition. When a user selects the *precondition_radio* a click on a button in the *button_control* refers to the preconditions; the postconditions are affected otherwise. A client can query the selection of the radio button through *is_precondition_selected* and *is_postcondition_selected*.

As noted above the *assembly dialog* directly manipulates the *AST* by using action features. They can be divided into two blocks: features that handle user interaction on the widgets such as selection of a node in the *assembly tree* or a button press in the button control of an assertion view; and features that update the *AST* like removing an invariant from a type or applying a postcondition to a feature.

The *assembly dialog* has attributes to access specific nodes of the *AST* directly (see Table 4). It updates them along with the user interaction. If for example a user selects the second invariant in the invariant list box, the feature *set_invariant* sets the cursor of the list *selected_invariants* to the according position. That is why the selected invariant is always accessible through *selected_invariants.item*.

```

selected_type: CW_TYPE
    -- Currently selected type in `assembly_tree`.

selected_feature: CW_FEATURE
    -- Currently selected feature in `assembly_tree`.

selected_invariants: LINKED_LIST[CW_INVARIANT]
    -- List of invariants of selected type.

selected_preconditions: LINKED_LIST[CW_PRECONDITION]
    -- List of preconditions of selected feature.

selected_postconditions: LINKED_LIST[CW_POSTCONDITION]
    -- List of postconditions of selected feature.

```

Table 4: Access features of CW_ASSERTION_DIALOG

When a type node is selected the dialog executes the following actions:

- updates selected type: *selected_type := type_tree_item.cw_type*
- sets text in type field: *type_field.set_text(assembly_tree.selected_item.text)*
- clears entry in feature field: *feature_field.remove_text*
- displays the invariant view on its right hand side: *set_invariant_view*

- lists the invariants of the selected type in the invariant list box: *set_invariant_list*

A click on a feature node causes the *assembly dialog* to:

- update selected feature: *selected_feature := feature_tree_item.cw_feature*
- display signature of selected feature in feature field: *set_feature_field*
- update class type of selected feature: *selected_type := type_tree_item.cw_type*
- set name of selected type in type field: *type_field.set_text (assembly_tree.item.text)*
- set selected preconditions: *selected_preconditions := selected_feature.preconditions*
- set selected postconditions:
selected_postconditions := selected_feature.postconditions
- display the condition view on its right hand side: *set_condition_view*
- fill pre- and postcondition boxes with pre- and postconditions of selected feature:
set_condition_list

If the user selects an invariant in the invariant list box, *set_invariant* tracks the cursor in *selected_invariants* list and displays the invariant tag and expression in the corresponding text fields. The features *set_precondition* and *set_postcondition* do the same when the user selects a precondition or a postcondition.

A user can manipulate an assertion through the buttons in the button bar of the assertion view. The *assembly dialog* handles the corresponding button actions. For the invariant view the following features are involved:

- *invariant_view.button_control.add_action.extend (agent add_invariant)*
- *invariant_view.button_control.apply_action.extend (agent apply_invariant)*
- *invariant_view.button_control.remove_action.extend (agent remove_invariant)*
- *invariant_view.button_control.remove_all_action.extend (agent remove_all_invariants)*
- *invariant_view.set_key_actions (agent add_invariant)*

The features behave exactly as their name suggests. They all use the accessors listed in Table 4 which are always kept up to date. The features for the condition view are similar, except for an additional check whether the user selected the precondition or postcondition radio button.

```

add_condition is
    -- Add invariant for selected type.
    require
        tag_not_void: condition_view.tag /= Void
        expression_not_void: condition_view.expression /= Void
    do
        ...
        if condition_view.is_precondition_selected then (1)
            -- Add new precondition
            if not selected_feature.has_precondition (new_precondition) then (2)
                selected_feature.add_precondition (new_precondition) (3)
                -- Add `new_precondition' to `condition_view'.
                condition_view.precondition_box.extend_assertion
                    (a_tag, an_expression) (4)
            else
                display_error_dialog (Precondition_already_exists)
            end
        else
            ...

```

Table 5: Code snippet from add_condition

Table 5 lists a code snippet of the feature *add_condition*. Because all update features follow the same scheme, *add_condition* gives an idea of the implementation of all update features: they update the *AST* (3) and adapt the widget in the assertion view that displays the assertion (4). A new precondition is only inserted when the selected feature does not already contain it (2). Of course this also applies to invariants and postconditions. Handling of pre- and postconditions requires an additional query returning which radio button the user selected (1). Otherwise the wizard does not know if it has to update the pre- or postcondition of a feature.

The *assembly dialog* manages the sensitivity state of the buttons in the button bar. “Add” is only enabled when a node of the tree is selected and the assertion tag and expression fields are filled out. When “Apply” is enabled an assertion from a box has to be selected as well. “Remove” is selectable as soon as the user selected an assertion. “Remove All” is enabled when the assertion box of a selected node is not empty.

3.3.4 INFORMATION DIALOG

The first and the last dialog of the wizard are of type *CW_INFORMATION_DIALOG*. Figure 9 shows first dialog of the Contract Wizard. The *information dialog* shows a nice pixmap on the left hand side. On the right hand side it has three labels: one of them is bold to display the heading, the other labels can be used to display additional information to the user. A client can set the text of these labels through the features *set_title*, *set_text_1* and *set_text_2*.

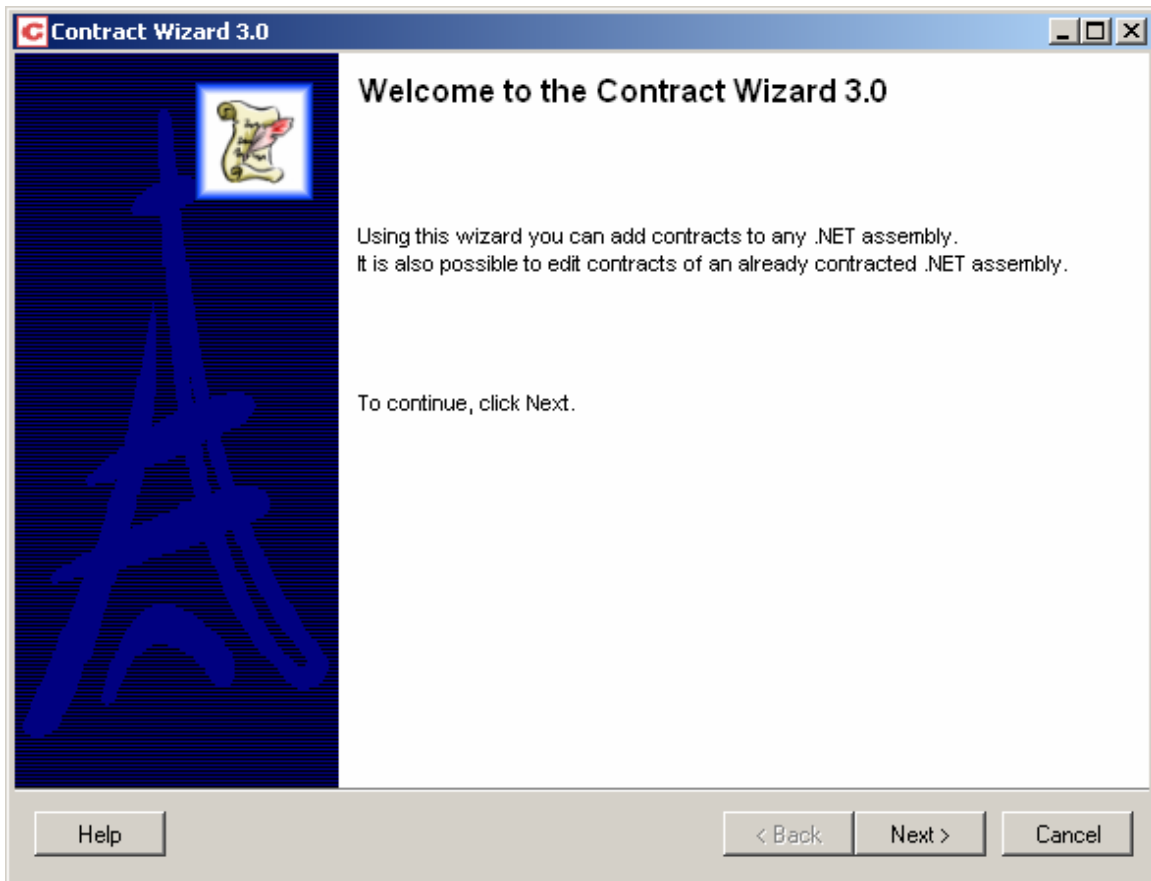


Figure 9: CW_INFORMATION_DIALOG

3.3.5 PROGRESS BAR

The *progress bar* (*CW_PROGRESS_BAR*) is a central element along with the user interaction. It gives the user feedback about a started process and also gives him an idea of how much time the operation will take. Figure 10 shows a screenshot of the *progress dialog*. The wizard uses the *progress dialog* when it:

- parses the .NET assembly
- reads the XML file containing the proxy classes' representation
- backs up the files in the project directory
- generates the Eiffel proxy classes, the Ace and XML files
- checks the syntax of the generated Eiffel files
- compiles the classes to a new .NET assembly
- does finish freezing

- restores files from the project directory

Every class that wants to indicate a progress inherits from the class *CW_GUI_SUPPORT* that provides an access point to the *progress dialog* called *Progress_dialog*. In order that every class accesses the same instance of the dialog, *Progress_dialog* is a once feature. A once feature executes its body only the first time it is called in a system execution. The result obtained by the first call is applicable to all instances of a class.



Figure 10: Screenshot of progress dialog

The *progress dialog* has to know how many steps the process has to update the progress bar accordingly. At the beginning of each process the client of the *progress dialog* sets the number of steps that the process will have (2). It also sets the title of the dialog and the label that indicates what type of file is being processed (3). The check *is_gui_application* (1) is necessary because the command line of the Contract Wizard uses the same classes but then, graphical widgets like the *progress dialog* are not supported. (See Table 6 for a code snippet.)

```

in client class
  if is_gui_application then                                     (1)
    Progress_dialog.set_max_classes (assembly.get_exported_types.count) (2)
    Progress_dialog.set_title (Parsing_xml_text_title)
    Progress_dialog.set_class_label (Parse_xml_class)           (3)
  end
  ...
  Progress_dialog.process (cw_type.eiffel_name)                 (4)

in CW_PROGRESS_DIALOG
  process (a_class_name: STRING) is
    -- Set labels and status of `progress_bar`.
    require
      a_class_name_not_void: a_class_name /= Void
      a_class_name_not_empty: not a_class_name.is_empty
    do
      -- Check if correct progress_vbox is shown.
      if main_vbox.first /= general_progress_vbox then

```



```

        set_general_vbox
    end
    -- Set text labels.
    general_progress_vbox.class_text_label.set_text (a_class_name)      (5)
    general_progress_vbox.class_to_go_text_label.set_text (class_to_go_number.out)
    -- Update progress in `progress_bar`.
    class_to_go_number := class_to_go_number - 1                        (6)
    progress_bar.set_proportion ((max_classes - class_to_go_number) / max_classes)
end

```

Table 6: Code snippets for progress dialog

To update the progress bar *CW_PROGRESS_DIALOG* provides a function *process*. In every step of the process the client class calls *process* together with a string that the dialog displays (4), e.g. the file being parsed or generated. The *process* function sets the label (5) and updates the advancement of the progress bar (6).

The progress dialog for showing the progress of the compilation looks a bit different. It has an extra label to indicate the degree of the compilation and a pixmap on the right shows the degree visually (cf. compilation progress dialog in ISE EiffelStudio). Nonetheless the compilation information has to be shown in the same instance of the progress dialog. Therefore I implemented a vertical box (*CW_COMPILATION_VBOX*) that the dialog replaces with the “normal” progress box (*CW_PROGRESS_BOX*) when it has to process compilation information. The progress bar itself is always the same, only the labels of the dialog are exchanged.

The progress dialog provides a special process function for the compilation input. The compiler forwards its output line by line to the progress bar. A line always has the same structure, i.e.:

```
[ 78% - 131] Degree 1 class CW_ACCOUT
```

The feature *process_compilation* tokenizes the input and sets the label for the degree, cluster or class, and cluster to go or class to go. It advances the progress bar according to the percentage of the compiler output.

3.3.6 ERROR DIALOG

Another important element to give feedback to the user is the error dialog. The wizard uses the standard *EV_WARNING_DIALOG* for errors that can be explained in one or two sentences such as a file does not exist or the .NET assembly is not valid. If the syntax checker finds an error in a class or the compilation produces an error it is appropriate to show a detailed error description to the user. The class *CW_ERROR_DIALOG* as shown in Figure 11 undertakes this task.

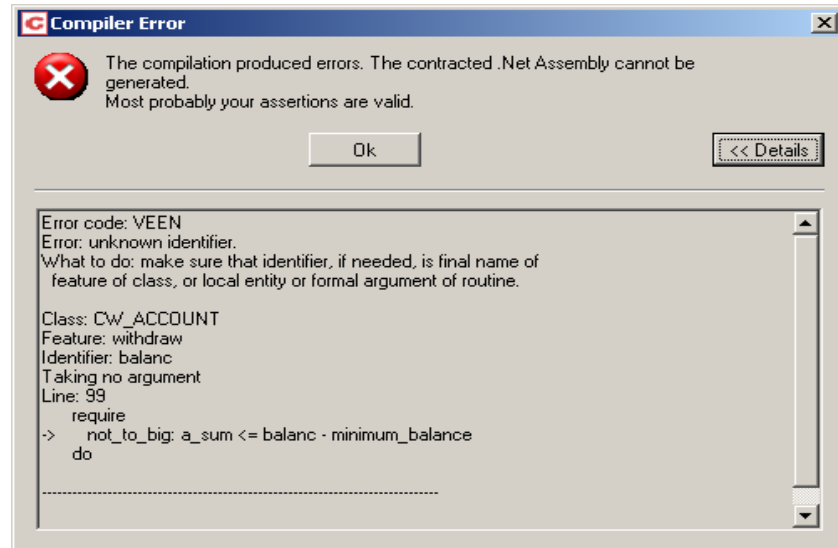


Figure 11: CW_ERROR_DIALOG

When the wizard shows the *error dialog* it first lists a general error description to the user. The user can click on the “Details >>” button to find out more about the error. The feature *detail_button_selected* from *CW_ERROR_DIALOG* extends the error dialog with the detailed error description and removes it after a repeated click. Table 7 shows the creation procedure of *CW_ERROR_DIALOG*. The function *split_string* divides *an_error_text* in multiple lines of length *Const_error_text_length* (1).

```

make_with_error_text (an_error_text, a_detail_text: STRING) is
  -- Create `Current' and assign error messages.
  require
    error_text_not_empty: an_error_text /= Void and not an_error_text.is_empty
    detail_text_not_void: a_detail_text /= Void
  do
    default_create
    set_error_text (split_string (an_error_text, Const_error_text_length))    (1)
    set_detail_text (a_detail_text)
  ensure
    error_text_assigned: error_text.is_equal (split_string (an_error_text,
      Const_error_text_length))
    detail_text_assigned: detail_text.is_equal (a_detail_text)

```

Table 7: Creation procedure of CW_ERROR_DIALOG

4. CONTRACT WIZARD 3.0: ADDITIONAL FUNCTIONALITY

In this diploma project I did not only extend the Contract Wizard with a graphical user interface but also with new functionality. I partly added new classes and partly improved existing classes. In this chapter I will describe the changes I made concerning the business logic of the Contract Wizard.

4.1 NEW FUNCTIONALITY

In this section I will discuss new functionality that affects the wizard's application flow.

4.1.1 PROJECT FILE

For the Contract Wizard I introduced the notion of project. To contract a new .NET assembly the user has to create a new project. Given the right input parameters, provided through the GUI or command line, the class *CW_PROJECT_HANDLER* takes care of creating a new project file. The file stores the project name, working directory, assembly location, and assembly name. The *project handler* stores the file in the working directory. It has the same name as the project and the extension *.cpr*; an abbreviation for *contract project*. Table 8 shows the content of a sample project file.

Project name: Account Working directory: D:\eth\DA\test\ Assembly location: D:\eth\DA\example\Accounting_Example\Account.dll Assembly name: Account.dll
--

Table 8: Project file

Besides creating a project file the *project handler* is also responsible for opening an existing project. It receives the path on the project file as argument in its creation procedure and then the feature *read_project_file* reads the content from the file and assigns it to public attributes of *CW_PROJECT_HANDLER*.

An important job of the project handler is to load the assembly from the assembly path. The method `load_from` takes as argument the path where the assembly is located and returns the loaded assembly **(2)**. The attribute `assembly` is of type `ASSEMBLY`, a class from the .NET Framework. While loading the assembly it is possible that the system throws an exception, for example `FileNotFoundException` (assembly file is not found) or `BadImageFormatException` (assembly file is not a valid assembly). Therefore it is important that the feature calling `load_from` catches the exception in a rescue clause and sets a flag if the assembly is not loaded **(3)**. Table 9 shows the creation procedure of `CW_PROJECT_HANDLER` that creates a new project file and loads the assembly.

```

make (an_assembly_name, an_assembly_path, a_working_directory, a_project_name:
  STRING) is
  -- Initialize Current with values for a new project.
  require
    -- ...Strings not empty...
  local
    is_erroneous : BOOLEAN
  do
    if not is_erroneous then
      assembly_name := an_assembly_name           (1)
      assembly_path := an_assembly_path
      working_directory := a_working_directory
      project_name := a_project_name

      create_project_file
      -- Load assembly
      assembly := assembly.load_from (assembly_path) (2)
      is_assembly_loaded := True
    end
  ensure
    assembly_name_assigned: assembly_name.is_equal (an_assembly_name)
    assembly_path_assigned: assembly_path.is_equal (an_assembly_path)
    working_directory_assigned: working_directory.is_equal (a_working_directory)
    project_name_assigned: project_name.is_equal (a_project_name)
    assembly_not_void: is_assembly_loaded implies assembly /= Void
  rescue
    is_erroneous := True
    is_assembly_loaded := False
  retry
end

```

Table 9: Creation procedure of CW_PROJECT_HANDLER

Other classes (`CW_MAIN_WINDOW`, `CW_CONTROLLER`...) use the project handler to access information about the loaded project or the assembly. Besides the attributes that are directly assigned in the creation procedure **(1)** the project handler provides the functions:

- *assembly_directory*: *STRING*: directory of assembly without assembly name
- *assembly_extension*: *STRING*: extension of current assembly ("exe" or "dll")
- *assembly_name_no_extension*: *STRING*: name of assembly without extension
- *project_id*: *STRING*: pseudo project id of current project

The class *CW_MAIN_WINDOW* uses *project_id* to determine whether users selected the same project as they did before. (Scenario: user was in *assembly dialog*, goes back to *project dialog*, and again forward to *assembly dialog*). The window class maintains two project ids: *project_id* and *previous_project_id*. When they are equal the user has not changed the project settings and the wizard shows exactly the same *assembly dialog* as before. To save time and resources the wizard only parses the .NET assembly and builds the *assembly tree* afresh when another project is defined.

4.1.2 SYNTAX CHECK

Before the wizard compiles the generated Eiffel classes we wanted to check them for syntax errors. There were many approaches until we came up with the final solution. I will discuss them briefly.

Our first idea was to use Gobo Eiffel Lint (*gelint*) [3]. *Gelint* is a tool which is able to analyze Eiffel source code and report validity errors. It uses the tools Gobo Eiffel Lex (*gelex*) and Gobo Eiffel Yacc (*geyacc*). Regrettably *gelint* only works on classic Eiffel, not on Eiffel for .NET. The problem is that the ISE Eiffel compiler relies on some XML files to make the correspondence between .NET and Eiffel for .NET classes; and this format is not published.

The second approach was to implement an Eiffel assertion parser. I took the grammar description from the Gobo Eiffel Parse Library and swept everything out that is not needed to parse an assertion. The start condition was no more `%start Class_declarations` but `%start Assertions`. Then I used *geyacc* to generate an Eiffel assertion parser. Whenever the user added a new assertion (invariant, pre- or postcondition) the *assembly dialog* checks it using the assertion parser. A drawback of this approach is that the assertion grammar description of the *yacc* file has to be adapted manually when the Eiffel syntax changes so it does not come automatically with a new *gobo* release. However the assertion checker still exists and can be rebound in the wizard easily.

I tried to come up with an algorithm that checks for validity of the assertion expression. I made a list of identifiers occurring in the assertion and checked if they match a feature name of the current class or a parameter of the feature in case of a pre- or postcondition. This approach was too restrictive: an identifier is also valid when it relies on a parent feature but then it got rejected.

The next idea was to generate a second set of classes that only contains classic Eiffel code and then analyze the classes with *gelint*. The Contract Wizard generates wrapper classes as listed

in Table 10 on the left hand side. I created a new Eiffel visitor class (*CW_EIFFEL_TEST_VISITOR*) based on the existing Eiffel visitor which generates Eiffel classes as listed in Table 10 in the right column. The wrapper features just have a "do end" as body and no *dotnet_proxy* feature.

<pre> class CW_A feature foo is do dotnet_proxy.foo end feature -- Implementation dotnet_proxy: DOTNET_A end </pre>	<pre> class CW_A feature foo is do end end </pre>
---	---

Table 10: Second set of classes

These classes are Eiffel classes. Thus, they can be parsed by *gelint* to determine whether assertions are valid or not. The generation of the classes worked fine but they still had references to .NET classes (argument type of procedures or functions, and return type of functions). Table 11 shows an example: *IENUMERATOR* relies on a type from the .NET Framework and thus *gelint* cannot handle it.

<pre> frozen get_enumerator: IENUMERATOR is do end </pre>

Table 11: get_enumerator function

We then had the following idea: get the list of referenced assemblies from the assembly given as input to the Contract Wizard. The referenced assemblies are needed to compile the given assembly and are listed in the "assembly" clause of the Ace file. Generate the wrapper classes corresponding to the classes in those referenced assemblies. So the work that the Contract Wizard does for only one assembly would be done to all referenced assemblies. Every wrapper class only has "do end" bodies and no *dotnet_proxy* attribute. To refer to the example in Table 11: among others the wizard generates a wrapper class for *IENUMERATOR*. Save those wrapper classes in a temporary directory. When checked with *gelint* whether contracts are valid we include that directory in the Ace file and remove the "assembly" clause. If not errors occurred, we regenerate the new assembly with contracts as the Contract Wizard always did before (with "assembly" clause in the Ace file and no inclusion of that temporary directory). We considered incrementality for performance reasons.

I started to implement this idea and noticed that the Contract Wizard does not yet handle overloaded .NET methods. In .NET languages (C#, C++) it is possible to overload a method with different signatures. Eiffel does not support feature overloading and therefore the feature names of overloaded methods have to be adapted (see section 4.2.4).

After this odyssey – where I learnt quite a few things – we came up with the final solution: test the generated classes on syntax errors and then start the Eiffel compiler to verify the source code.

CW_GELINT

The class *CW_GELINT*, based on *gelint*, takes over the syntax check. The root feature reads an Ace file as input and looks through the system clusters to map the Eiffel class names with the corresponding file names. I generated the Ace file using the class *CW_LACE_TEST_GENERATOR*. The difference with the standard Ace file generated by the Contract Wizard is that it has no “assembly” clause and no attributes which set .NET properties.

When the parsing of the Ace file produces no errors, it returns a universe (*ET_UNIVERSE*) which is used to parse the classes. Foremost the creation procedure sets the compiler to the ISE Eiffel compiler and redirects the error reporting to a file (*error_file_out*). This forces the parser to write errors in a file from where they can be retrieved later on. In the next step the features calls *parse_root_class*. It is very important to call *error_file_out.close* after the parsing otherwise the file will remain empty even if syntax errors occurred.

The creation procedure of *CW_GELINT* receives along with a string of the directory (*directory*) where the Ace file and the classes to be parsed are stored, a class list (*class_list*) with the name of all generated Eiffel classes. The procedure *parse_root_class* iterates through all classes of the universe but only parses classes in the root cluster. It is not necessary to check the classes in other clusters because they are library classes (base, base_net) and have a correct syntax. When the cluster equals the root cluster the feature iterates through *class_list* and parses all classes in it. This means that only the generated classes are parsed.

The feature *parse_all* in *ET_UNIVERSE* parses all classes of the current cluster. I could not use *parse_all* because it parses all classes from the current directory. It is possible that you have classes in that directory that do not correspond to the actual project (e. g. when you created another project in the same directory before). Instead of *parse_all* I use the *parse_file* feature. It takes as arguments a file, its name, and a cluster. It parses all classes in the file within the given cluster. If a class has a syntax error the flag *has_syntax_error* is set to true.

If *has_syntax_error* is true the private feature *generate_syntax_error* reads the file containing the errors line by line and appends each line to *syntax_error*.

With this features a client of *CW_GELINT* can verify the generated classes and retrieve syntax errors in a comfortable way. Table 12 shows a code snippet showing how *CW_MAIN_WINDOW* uses *CW_GELINT* to test the generated classes on syntax errors.

```

-- Check generated files on syntax errors.
create gelint.make (project_handler.working_directory, controller.class_list)
if gelint.has_syntax_error then
    Progress_dialog.hide
    create error_dialog.make_with_error_text (Syntax_error_text, gelint.syntax_error)
    error_dialog.set_title (Syntax_error_title)
    error_dialog.show_modal_to_window (Current)
    failure := True
else
    -- Launch compilation and check for compilation errors.

```

Table 12: Use of CW_GELINT in CW_MAIN_WINDOW

4.1.3 COMPILATION

The class *CW_COMPILER_LAUNCHER* handles the compilation of the Eiffel source. The class already existed but it now has new functionalities: it finalizes the code instead of only freezing it and it executes “finish_freezing” to compile the C code and to link part of the finalization. Table 13 shows how *CW_MAIN_WINDOWS* uses *CW_COMPILER_LAUNCHER* to compile the Eiffel classes to a .NET assembly. The code snippet follows right after the code extract of Table 12 above.

```

-- Launch compilation and check for compilation errors.
create compiler_launcher.make (project_handler.assembly_name_no_extension,
    project_handler.assembly_extension, project_handler.working_directory) (1)
compiler_launcher.compile (2)
compiler_launcher.wait_for_exit
if compiler_launcher.is_successful_compiled then
    -- Execute finish freezing.
    compiler_launcher.finish_freezing (3)
    compiler_launcher.wait_for_exit
else
    Progress_dialog.hide
    create error_dialog.make_with_error_text (Verification_error_text
        compiler_launcher.error_message) (4)
    error_dialog.set_title (Verification_error_title)
    error_dialog.show_modal_to_window (Current)
    failure := True
end

```

Table 13: Use of CW_COMPILER_LAUNCHER in CW_MAIN_WINDOW

CW_MAIN_WINDOW first initializes an instance of *CW_COMPILER_LAUNCHER* (*compiler_launcher*) with the necessary arguments (1). *CW_COMPILER_LAUNCHER* uses the class *SYSTEM_DLL_PROCESS* to launch the Eiffel compiler. The creation procedure *make* sets the working directory of the compilation and the needed parameters to redirect its output. After the creation of *compiler_launcher* the main window initiates the compilation (2). *Compile* sets the compile command depending on the existence of a compiled project and launches the compilation process. Moreover it reads the compiler output and forwards it to the *progress dialog* which displays it to the user.

After a successful compilation the main window calls *finish_freezing* (3). This feature changes the working directory to “EIFGEN\F_Code” where it executes “finish_freezing”. If the compilation produces errors the wizard retrieves the error message (*error_message*) from the compiler launcher and shows an error dialog with the detailed error description (4). The compiler launcher sets *error_message* by gathering all strings appearing after “Error Code” from the compiler output.

4.1.4 BACKUP

The wizard backs up all files from the project directory after having parsed the .NET assembly. The files are temporarily stored in the directory “cw_tmp” relatively to the contract delivery directory (\$CONTRACT). If the wizard has already generated Eiffel files and the user cancels the application, the wizard restores the files to the project directory – if the user wants to. The Contract Wizard already had a backup functionality but this version parsed the .NET assembly twice and regenerated the Eiffel files from the backup *AST* in case of failure.

The procedures *backup_directory* and *restore_to_directory* from the class *CW_SUPPORT* do the backup and the restore. They both take a path as argument: *backup_directory* uses it as source directory for the backup, *restore_to_directory* as destination directory for the restore. The second directory is always “\$CONTRACT\cw_tmp”. After having launched a *progress dialog* they call *copy_directory*. This procedure copies all files from the source directory to the destination directory and shows its progress in the *progress dialog*.

4.2 ASSEMBLY PARSING

When I tried to compile the classes that the wizard generated, I realized that the source was not always compilable. Possible errors were: no support of overloaded .NET methods, undeclared methods inherited from an explicit interface member implementation, or insufficient accuracy in parsing the signature of a .NET method. In this section I will discuss the changes that affected the parsing of the .NET assembly.

4.2.1 .NET INTERFACE

An interface specifies the members that must be supplied by classes implementing the interface. The interface itself does not provide implementations for the members that it defines. The Contract Wizard class corresponding to an interface is a deferred class since Eiffel does not support the notion of interface. The semantic of the class stays the same: a .NET class *X* that implements an interface *I* is represented by an effected Eiffel class *CW_X* that inherits from *I*. Table 14 compares an interface *I*, an abstract class *X* that implements the interface *I*, and a class *Y* that extends class *X* directly with the Eiffel classes that the Contract Wizard generated from those C# classes.

.NET C#	Contract Wizard
<pre> public interface I { int index(); } </pre> <hr/> <pre> public abstract class X: I { public int index() { return 1; } public string foo() { return "foo"; } } </pre> <hr/> <pre> public class Y: X { public string bar() { return "bar"; } } </pre>	<pre> deferred class CW_I feature -- Access index: INTEGER is -- dotnet_name: "I.index (): Int32" deferred end end </pre> <hr/> <pre> deferred class CW_X inherit I ... feature -- Access index: INTEGER is -- dotnet_name: "X.index (): Int32" do Result := x_ref.index end foo: SYSTEM_STRING is -- dotnet_name: "X.foo (): String" do Result := x_ref.foo end feature {NONE} -- Implementation x_ref: X -- Reference to class X end </pre> <hr/> <pre> class CW_Y inherit I ... feature -- Access index: INTEGER is </pre>

<pre> } } </pre>	<pre> -- dotnet_name: Y.index (): Int32" do Result := y_ref.index end foo: SYSTEM_STRING is -- dotnet_name: "Y.foo (): String" do Result := y_ref.foo end bar: SYSTEM_STRING is -- dotnet_name: "Y.foo (): String" do Result := y_ref.foo end feature {NONE} -- Implementation y_ref: Y -- Reference to class Y end </pre>
------------------	--

Table 14: C# classes versus generated Eiffel classes

What you can see is that *CW_X* and *CW_Y* both inherit from the interface *I*, although *Y* does not explicitly declare to implement *I*. A C# class inherits all interface implementations provided by its base classes; since *Y* extends *X*, *Y* inherits the interface implementation of *X*, namely *index*. When a class *Y* extends another class *X*, *Y* inherits all methods from *X*; for example the method *foo* in the above example. In C# all this works implicitly. The Contract Wizard code generator does this explicitly: it lists all (indirectly) inherited interfaces in the inherit clause and generates code for all inherited features. In exchange, the code generator does not list the super class *X* in the inheritance clause of *CW_Y* because it already writes the inherited feature *foo* from class *X* directly in the code.

The structure of the generated Eiffel proxy classes comes from the .NET parser; or rather how the .NET parser (*CW_DOTNET_PARSER*) uses the reflection mechanism of .NET. A call *type.get_methods* returns all public methods from a type including the inherited method implementations. A call *type.get_interfaces* returns all the interfaces implemented or inherited by the current type.

The problem arises with explicit interface member implementations [11]. An explicit interface member declaration is a method or property declaration that references a fully qualified interface member name. For an annotated example see Table 15 on the left hand side, it is an extract from *mcorlib*, the base library of the .NET Framework [13]. *IDisposable.Dispose* is an explicit interface member implementation of the class *TextWriter*.

It is not possible to access an explicit interface member implementations by its fully qualified name in a method invocation. They can only be accessed through the interface instance

by its member name. They are often used when the name of an interface member is not appropriate for the implementing class.

.NET C#	Contract Wizard
<pre> interface <i>IDisposable</i> { public void <i>Dispose()</i>; } ----- class <i>TextWriter: IDisposable</i> { <i>void IDisposable.Dispose()</i> {...} } ----- class <i>StringWriter: TextWriter</i> { ... } </pre>	<pre> deferred class <i>CW_IDISPOSABLE</i> feature -- Access <i>dispose is</i> --dotnet_name: "IDisposable.Dispose ()" deferred end end ----- deferred class <i>CW_TEXT_WRITER inherit</i> <i>IDISPOSABLE</i> ... feature {<i>NONE</i>} -- Implementation <i>dispose is</i> (1) -- dotnet_name: "TextWriter.Dispose ()" do end end ----- class <i>CW_STRING_WRITER inherit</i> <i>IDISPOSABLE</i> ... feature {<i>NONE</i>} -- Implementation <i>dispose is</i> (2) -- dotnet_name: "StringWriter.Dispose()" do end end </pre>

Table 15: C# classes with explicit interface member implementation versus generated Eiffel classes

Because explicit interface member implementations are not accessible through class instances, they are excluded from the public interface of a class. Therefore the .NET parser does not obtain them through the call `type.get_methods`. However, every class that inherits from an interface and is effective must effect all features declared in the interface class. The Contract Wizard did not consider that. Therefore, the generated classes were not compilable. In the above example `CW_TEXT_WRITER` and `CW_STRING_WRITER` inherit from `IDISPOSABLE` and did

not list *dispose* as a proper feature. As for *CW_TEXT_WRITER* this has no consequences since it is deferred, the compiler throws an error when *CW_STRING_WRITER* did not declare *dispose* (see Table 16).

<p>Error code: VCCH(1) Error: class has deferred feature(s), but is not declared as deferred. What to do: make feature(s) effective, or include `deferred' before `class' in Class_header.</p> <p>Class: <i>CW_STRING_WRITER</i> inherit <i>IDISPOSABLE</i> Deferred feature: <i>dispose</i> From: <i>IDISPOSABLE</i></p>
--

Table 16: Compilation error CW_STRING_WRITER

The solution is shown in Table 15: every class that inherits from an interface lists all explicit interface member implementations, but does not export them. Additionally, the interface member implementations have an empty body. It is not possible to call the feature on the reference object as *string_writer_ref.dispose* since *dispose* is not declared as public in the class *StringWriter* and therefore not available to client classes. This also means that it is impossible to add contracts to those features because nothing is propagated to the actual class. The following subsection discusses the changes affecting the .NET parser.

4.2.2 .NET PARSER

The *parse_type* feature from *CW_DOTNET_PARSER* inspects now – besides public constructors, public fields, and public methods – explicit interface member implementations. The query *has_interface_methods* checks if the type being parsed has an interface with at least one public method. (This is true for *TextWriter* and *StringWriter*: they have an interface *IDISPOSABLE* with the public method *dispose*.) If *has_interface_methods* returns true the parser calls the procedure *inspect_interface_methods*.

The feature *inspect_interface_methods* loops through all interfaces that comply with the Common Language Specification (CLS) of the actual type. For every interface it inspects all its methods. If the type does not already have a method with the same return type and arguments as the inspected method, the parser appends the method to the *AST* as a private feature. Whether the feature is appended as procedure or as function depends on the feature's return type: if the return type is void, the procedure appends it as a procedure, otherwise as a function. The appended features conform to the explicit interface member implementations.

The way I chose to handle the explicit interface member implementations seems at first glance extraordinary. Indeed it was not the first approach. I first had the idea to get all private methods of a type and check whether their method names contain a dot, for example *System.IDisposable.Dispose*. (Explicit interface members are always declared with their fully

qualified interface member name and therefore the declaration contains a dot.) Every method fulfilling the condition was added as a private feature. The problem of this approach was that the parser could not obtain every interface method. Still there was no *dispose* feature listed in the class *CW_STRING_WRITER*. The method *dispose* is not an explicit private interface in this class because it inherits *dispose* from *CW_TEXT_WRITER*.

4.2.3 EIFFEL NAMES FOR .NET TYPES

The Contract Wizard analyzes a .NET assembly and produces Eiffel classes for every type the assembly defines. Eiffel names are derived from .NET type names by taking the substring after the rightmost dot in the full class name. Then the formatter converts the string into an Eiffel compliant name by making it upper case and separating it in embedded words by underscore. For example *System.Collection.ArrayList* becomes *ARRAY_LIST*.

If the basic derivation produces a name which conflicts with a class name in the EiffelBase Library the name formatter (*CW_NAME_FORMATTER*) disambiguates it. *System.String* conflicts with *STRING* and becomes *SYSTEM_STRING*. The query *eiffel_formatted_defined_type_name* from *CW_NAME_FORMATTER* returns a non-conflicting Eiffel name. A conflict also applies to the base types; the feature *eiffel_formatted_primitive_type_name* handles their name conflicts.

When the parsed assembly is mscorlib [13] a few more name conversions are necessary to stay conformant with the class names which the ISE Emitter produces. The class names and their outcome are listed in Table 17. The new feature *eiffel_formatted_special_name* carries out the translation.

Convert	→	<i>SYSTEM_CONVERT</i>
Mutex	→	<i>SYSTEM_MUTEX</i>
Thread	→	<i>SYSTEM_THREAD</i>
Zone	→	<i>SYSTEM_ZONE</i>
DESCUNION	→	<i>DESCUNION_IN_ELEMDISC</i>
SpecialFolder	→	<i>SPECIAL_FOLDER_IN_ELEMDISC</i>

Table 17: Translation of special mscorlib types

The mscorlib classes *Descunion* and *SpecialFolder* are inner classes of *System.Runtime.InteropServices.Elemdesc* respectively *System.Environment*. Since the Contract Wizard does not parse inner classes yet, their translation is a little hack.

4.2.4 OVERLOADED NAMES

OVERLOADED .NET MEMBER NAMES

The .NET object model allows overloading function names. This means that a .NET type can support multiple functions with the same name but different argument types. Since the Eiffel model prohibits overloading, any overloaded routine must be disambiguated. Table 18 shows the signature of a few overloaded methods in their .NET form. They are all declared in the class *System.IO.TextWriter*.

```
WriteLine (System.String)
WriteLine (System.Object)
WriteLine (System.String, System.Object)
WriteLine (System.Char[])
WriteLine (System.Char[], System.Int32, System.Int32)
```

Table 18: Overloaded .NET methods from System.IO.TextWriter

Table 19 shows the overloaded .NET methods together with the disambiguated Eiffel function name. To obtain a unique name the algorithm appends as much signature type names as necessary after the name of the function [9]. The type names are separated by underscore.

<pre>WriteLine (System.String) write_line_string (value: SYSTEM_STRING)</pre>	(1)
<pre>WriteLine (System.Object) write_line_object (value: SYSTEM_OBJECT)</pre>	(2)
<pre>WriteLine (System.String, System.Object) write_line_string_object (format: SYSTEM_STRING; arg_0: SYSTEM_OBJECT)</pre>	
<pre>WriteLine (System.Char[]) write_line_character_array (buffer: NATIVE_ARRAY [CHARACTER])</pre>	(3)
<pre>WriteLine (System.Char[], System.Int32, System.Int32) write_line_character_array_integer (buffer: NATIVE_ARRAY [CHARACTER]; index: INTEGER; count: INTEGER)</pre>	

Table 19: Disambiguated Eiffel function names

To make routine names better readable the algorithm omits the *system_* prefix from defined types **(1)** and **(2)**. If the type of a signature is an array of type *X*, the name appended to the basic feature name is *x_array* **(3)**.

CONSTRUCTORS IN .NET

Like creation procedures in Eiffel, .NET constructors are used to initialize new instances of types. The .NET object model allows overloading constructors. Like an overloaded method, an overloaded constructor differs in the set of argument types.

The constructor name obtained from the .NET type is always *.ctor*. If a class only defines one constructor the corresponding Eiffel creation procedure name is *make* (as it was already implemented in *CW_NAME_FORMATTER*). However, if there are overloaded versions of the constructor, these versions need to be transformed to be compilable in Eiffel.

The algorithm translates an overloaded constructor by starting with *make_from_* and then appending the argument names separated with the conjunction *_and_*. Table 20 lists the constructors from the class *System.Collections.BitArray* with their according Eiffel creation procedure declaration.

<i>void .ctor (length: Int32)</i> <i>make_from_length (a_length: INTEGER)</i>	(1)
<i>void .ctor (length: Int32, defaultValue: Boolean)</i> <i>make_from_length_and_default_value (a_length: INTEGER; a_default_value: BOOLEAN)</i>	(2)
<i>void .ctor (bytes: Byte[])</i> <i>make_from_bytes (a_bytes: NATIVE_ARRAY [INTEGER_8])</i>	
<i>void .ctor (values: Boolean[])</i> <i>make_from_values (a_values: NATIVE_ARRAY [BOOLEAN])</i>	(3)
<i>void .ctor (values: Int32[])</i> <i>make_from_values_1 (a_values: NATIVE_ARRAY [INTEGER])</i>	(4)
<i>void .ctor (bits: BitArray)</i> <i>make_from_bits (a_bits: BIT_ARRAY)</i>	

Table 20: Disambiguated .NET constructors

When a constructor has one argument, the algorithm appends the argument name **(1)**. If there are more arguments, the Eiffel name results from the argument names connected with *_and_* **(2)**. It may occur that more than one constructor has the same argument names but different

argument types [(3) and (4)]. In that case digits are appended to the procedure name to remove remaining ambiguity.

IMPLEMENTATION

The classes used to implement the overloaded member names are *CW_FEATURE_LIST* and *CW_FEATURE_ITEM*. *CW_FEATURE_LIST* is a linked list of *CW_FEATURE_ITEM*. *CW_FEATURE_ITEM* itself is a linked list of *CW_FEATURE*. The feature list contains all features of a parsed type, stored in the feature items. All features of one feature item have the same (.NET) name. In the example of Table 19 all features named *WriteLine* are maintained in the same feature item. Other features from type *TextWriter* such as *Write* or *Flush* are stored in another feature item. That is why the feature list has as many feature items as the type has different member names: for every .NET member there exists one feature. This data structure is used to construct overloaded member names.

For a client it is easy to get the overloaded names: it first needs to pass all features of a type to the feature list by calling *feature_list.extend_feature(a_feature)* for each feature and then to initiate the overloaded name generation by calling *feature_list.set_overloaded_feature_names*. Afterwards the client can access the overloaded names of a feature through *a_feature.overloaded_name*.

Actually, the class *CW_NAME_GENERATOR* does exactly the described pattern and it is even simpler to get the overloaded name of a feature: initialize the *name_generator* with the *AST* obtained by the parser and call *name_generator.generate_overloaded_names*. This feature loops through all types of the *AST* and their features. Thereafter every *CW_FEATURE* in the *AST* is assigned with an overloaded name. The feature *parse_dotnet* from *CW_CONTROLLER* uses the name generator after having obtained the *AST* from the .NET parser.

The features *extend_feature* and *set_overloaded_feature_names* from *CW_FEATURE_LIST* do the actual task of transferring overloaded method names into Eiffel conforming names. The procedure *extend_feature* attaches the received feature *a_feature* to the right feature item. If the feature list already has a feature item with the name of *a_feature* it extends the feature to the existing list item. Otherwise it creates a new list item and extends the feature to that. See Table 21 for a code illustration.

```

extend_feature (a_feature: CW_FEATURE) is
    -- Extend Current with new feature item or extend existing feature item.
    require
        a_feature_not_void: a_feature /= Void
        a_feature_name_not_empty: not a_feature.eiffel_name.is_empty
    local
        feature_item: CW_FEATURE_ITEM
    do
        -- Does `Current' contains item with `a_name'?
        if not has_feature_name (a_feature.eiffel_name) then
            create feature_item.make_with_name (a_feature.eiffel_name)
            extend (feature_item)
        end
        go_name (a_feature.eiffel_name)
        item.extend (a_feature)
    ensure
        has_item: has_feature (a_feature)
    end
end

```

Table 21: extend_feature from CW_LIST_ITEM

The feature *set_overloaded_feature_names* iterates through all feature items of *CW_FEATURE_LIST*. If the feature item contains overloaded names (that means the feature item contains more than one feature) it calls *item.set_overloaded_names* on it.

As discussed in the subsection above the feature *set_overloaded_names* from *CW_FEATURE_ITEM* implements the algorithm to assign a unique feature name to every feature. All features in the feature item have the same .NET name. Informally spoken the algorithm appends as much signature type names as necessary to the feature name. To do so it inspects every feature from the current feature item. If the feature is an attribute, the feature name stays the same. An attribute has by definition no arguments and therefore no argument type that can be appended.

When the feature is not an attribute, it is a routine. The feature *set_overloaded_names* iterates through the arguments of the routine and appends their type name to the overloaded name. This is done until either an argument type differs from the other arguments types compared to all other features in the list item or all arguments of the current routine have been inspected. This check is done by *is_argument_distinguishable (a_routine, arguments.index)*. Table 22 contains an example of overloaded feature names with the same original name. In every box the original Eiffel name is displayed above the overloaded feature name.

First the algorithm inspects the feature *write (value: SYSTEM_STRING)* and iterates through all its arguments. Although there is another feature with the same argument type at the first position (3), the algorithm stops after having appended *_string* to the feature name. This is because the routine only has one argument and nothing more can be appended. This has no further impact since it is sure that there is no other feature with exactly the same signature. (Otherwise the original .NET assembly would not be compilable because overloaded methods

must have different signatures.) So, if a feature has the same argument type as another feature at position i , it is sure that they differ in another argument type.

<pre>write_line (value: SYSTEM_STRING) write_line_string (value: SYSTEM_STRING)</pre>	(1)
<pre>write_line (value: SYSTEM_OBJECT) write_line_object (value: SYSTEM_OBJECT)</pre>	(2)
<pre>write_line (format: SYSTEM_STRING; arg_0: SYSTEM_OBJECT) write_line_string_object (format: SYSTEM_STRING; arg_0: SYSTEM_OBJECT)</pre>	(3)
<pre>write_line (buffer: NATIVE_ARRAY [CHARACTER]) write_line_character_array (buffer: NATIVE_ARRAY [CHARACTER])</pre>	(4)
<pre>write_line (buffer: NATIVE_ARRAY [CHARACTER]; index: INTEGER; count: INTEGER) write_line_character_array_integer (buffer: NATIVE_ARRAY [CHARACTER]; index: INTEGER; count: INTEGER)</pre>	(5)

Table 22: Disambiguated Eiffel function names

The algorithm only appends the type name of the first two arguments to the last feature of Table 22 although the feature has three arguments. The third argument is not needed because the second argument type *INTEGER* differs from all other arguments at position two.

Before appending an underscore and the name of the type, *set_overloaded_names* checks whether the argument type is a defined type or not. If it is a defined type, the name of the type is appended without the *system_* prefix (as it is also done in Eiffel for .NET). Array types are appended with *x_array* where *x* is the array's type; for example *NATIVE_ARRAY [CHARACTER]* becomes *character_array*.

The concept of name overloading for .NET constructors is the same: first all creation procedures of a type are registered to *CW_CREATION_PROCEDURE_LIST*, then a call of the form *creation_procedure_list.set_overloaded_creation_procedure_names* causes the creation procedure list to assign an overloaded name to every creation procedure it contains. This feature sets the overloaded name by iterating through all arguments of a creation procedure and appending every argument name of the procedure separated by an underscore. (See also algorithm description of subsection "Constructors in .NET".) Only procedures with the same argument names but different types need special treatment. An example can be found in Table 20, the procedures **(3)** and **(4)**. The feature *is_multi_overloaded* returns true when another creation procedure in the list already has the same overloaded name. Then the function *multi_overloaded_name* takes the temporary overloaded name as an argument and returns a distinct name for the creation procedure. That for *multi_overloaded_name* appends a unique index to the procedure name. The index is a global class attribute of *CW_CREATION_PROCEDURE_LIST*. After the algorithm appended the index to a procedure

name it increases the index. Like this it is guaranteed that the same index is not appended again to another procedure of the current procedure list.

4.2.5 ABSTRACT SYNTAX TREE

To master the more powerful .NET parser it was necessary to adapt the type definition from the abstract syntax tree (*AST*).

As already mentioned in the previous section, the parser adds an explicit interface member implementation as a private feature to the current type. I extended the class *CW_FEATURE* with the attribute *is_public*. When *is_public* returns false the feature is treated as private. In the GUI application only public types are listed in the assembly tree.

An important feature is the attribute *overloaded_name* that returns the overloaded Eiffel name (name of Eiffel feature when more than one feature in the class has the same name). The attribute is attached to *CW_FEATURE*. The classes *CW_FEATURE_LIST* and *CW_CREATION_PROCEDURE_LIST* handle the assignment of *overloaded_name* to every feature in the *AST*.

The class *CW_TYPE* has a new attribute *is_by_reference*. If a method parameter in the original .NET method is specified with the **out** or **ref** keyword, the type of the parameter is a reference type and *is_by_reference* is set to true. A method can consist of the following kinds of formal parameters: value parameters, reference parameters, output parameters, and parameter arrays.

DOCUMENT TYPE DEFINITION

The XML generator stores the structure of the *AST* in an XML file. In an iterative use the wizard parsed the XML file instead of the .NET assembly to get the internal representation of the Eiffel classes. The structure of the XML file is defined by a document type definition (*DTD*). This *DTD* defines the legal building blocks of an XML document. Various tools verify the structure of an XML document against a *DTD*. Table 23 shows the *DTD*, changes are marked with a yellow border.

```
<?xml version="1.0" encoding="utf-8"?>
<!ELEMENT cw_ast (cw_type+)>
<!ELEMENT cw_type (interfaces?, cw_creation_procedures?, cw_attributes?,
cw_procedures?, cw_functions?, cw_invariants?)*>
<!ELEMENT cw_creation_procedures (cw_creation_procedure)*>
<!ELEMENT cw_attributes (cw_attribute)*>
<!ELEMENT cw_procedures (cw_procedure)*>
<!ELEMENT cw_functions (cw_function)*>
<!ELEMENT cw_preconditions (cw_precondition)*>
<!ELEMENT cw_postconditions (cw_postcondition)*>
```

```

<!ELEMENT cw_invariants (cw_invariant)*>
<!ELEMENT interfaces (interface)*>
<!ELEMENT cw_arguments (cw_argument)*>
<!ELEMENT cw_type
  dotnet_name CDATA #REQUIRED
  eiffel_name CDATA #REQUIRED
  namespace CDATA #REQUIRED
  is_deferred (yes | no) #REQUIRED
  is_expanded (yes | no) #REQUIRED
  is_enum (yes | no) #REQUIRED
  is_interface (yes | no) #REQUIRED
>
<!ELEMENT cw_creation_procedure (cw_arguments?, cw_preconditions?,
cw_postconditions?)*>
<!ELEMENT cw_creation_procedure
  dotnet_name CDATA #REQUIRED
  eiffel_name CDATA #REQUIRED
  overloaded_name CDATA #REQUIRED
>
<!ELEMENT cw_attribute (type+, cw_preconditions?, cw_postconditions?)*>
<!ELEMENT cw_attribute
  dotnet_name CDATA #REQUIRED
  eiffel_name CDATA #REQUIRED
  overloaded_name CDATA #REQUIRED
  is_constant (yes | no) #REQUIRED
  value CDATA #REQUIRED
  is_static (yes | no) #REQUIRED
  is_public (yes | no) #REQUIRED
>
<!ELEMENT cw_procedure (cw_arguments?, cw_preconditions?, cw_postconditions?)*>
<!ELEMENT cw_procedure
  dotnet_name CDATA #REQUIRED
  eiffel_name CDATA #REQUIRED
  overloaded_name CDATA #REQUIRED
  is_deferred (yes | no) #REQUIRED
  is_static (yes | no) #REQUIRED
  is_public (yes | no) #REQUIRED
>
<!ELEMENT cw_function (cw_arguments?, type+, cw_preconditions?, cw_postconditions?)*>
<!ATTLIST cw_function
  dotnet_name CDATA #REQUIRED
  eiffel_name CDATA #REQUIRED
  overloaded_name CDATA #REQUIRED
  is_deferred (yes | no) #REQUIRED
  is_static (yes | no) #REQUIRED

```

<pre> is_public (yes no) #REQUIRED is_property (yes no) #REQUIRED > <!ELEMENT cw_argument (type+)> <!ATTLIST cw_argument dotnet_name CDATA #REQUIRED eiffel_name CDATA #REQUIRED > <!ELEMENT cw_invariant EMPTY> <!ATTLIST cw_invariant tag CDATA #IMPLIED expression CDATA #REQUIRED > <!ELEMENT cw_precondition EMPTY> <!ATTLIST cw_precondition tag CDATA #IMPLIED expression CDATA #REQUIRED > <!ELEMENT cw_postcondition EMPTY> <!ATTLIST cw_postcondition tag CDATA #IMPLIED expression CDATA #REQUIRED > <!ELEMENT interface EMPTY> <!ATTLIST interface dotnet_name CDATA #REQUIRED eiffel_name CDATA #REQUIRED to_string (yes no) #REQUIRED equals (yes no) #REQUIRED get_hash_code (yes no) #REQUIRED > <!ELEMENT type EMPTY> <!ATTLIST type dotnet_name CDATA #REQUIRED eiffel_name CDATA #REQUIRED is_array (yes no) #REQUIRED is_by_ref (yes no) #REQUIRED </pre>

Table 23: Document type definition for generated Eiffel classes

4.3 CODE GENERATION

4.3.1 EIFFEL VISITOR

With the new functionality of the .NET parser (explicit interface member implementations, support of overloaded names) it was necessary to adapt the existing *CW_EIFFEL_VISITOR*. See section 2.2.3 for a short description of the Eiffel visitor. An *AST* node (*CW_TYPE*, *CW_FUNCTION*...) has a flag whether it was already visited through a visitor or not. (After the first visit, *is_visited* is true.) The Eiffel visitor is registered to every feature node type twice. In the non-visited state the features are either extended to *creation_procedure_list* (*CW_CREATION_PROCEDURE_LIST*) or *feature_list* (*CW_FEATURE_LIST*). Later, the visitor uses these two lists to generate an additional comment for overloaded features.

In the actual visit where the visitor generates code for every node type, the visitor distinguishes between public and private features. The code of the feature body – the part between **do** and **end** – is only generated when the feature is public, otherwise the body remains empty. A private feature is an explicit interface member implementation of a type (see section 4.2.1). The generated source code of these features is not extended to the list containing procedures or features, but to the list *interface_features*. Finally, the source code that *interface_features* contains is written after the *{NONE}* clause of the current class text.

To append the name of a feature to the class text, the visitor now uses *overloaded_name* instead of *eiffel_name*. Every feature node has an attribute *overloaded_name*, which is a unique name of the feature. When the original .NET method is not overloaded, *overloaded_name* is equal to *eiffel_name*. For all overloaded features *generate_overloaded_comment* comments the feature with additional information of the form -- (+2 overloads). This comment means that the class has two other features with the same Eiffel name. Now the *creation_procedure_list* and *feature_list* come into play since every feature is registered to one of them. A call of the form *feature_list.overloaded_count (a_feature.eiffel_name)* returns the number of features that have the same Eiffel name as *a_feature*. It works similarly with the creation procedures. Due to *creation_procedure_list* the code generation for the **create** clause implemented in *generate_creation_text* is also simplified.

When *is_by_reference* of an argument type is true, the feature *generate_feature_type* sets type to the actual generic parameter of *TYPED_POINTER [G]*. The features *generate_preconditions* and *generate_postconditions* generate the feature text corresponding to **require** and alternatively **ensure** clause. When a feature appears in the redefinition clause of a class, **require** is changed to **require else** and **ensure** to **ensure then**.

4.3.2 ACE FILE

In Eiffel for .NET you can specify whether the Eiffel compiler should generate an exe (executable) or a dll (Dynamic Link Library) assembly. The associated option in the Ace file is

`msil_generation_type`. The Ace file is a central control file for Eiffel projects where - among other things - the compilation options of the project is described.

The option `msil_generation_type ("exe")` forces the compiler to create an executable; the declaration `msil_generation_type ("dll")` enforces the creation of a dynamic link library. The class *CW_LACE_GENERATOR* creates an Ace file that the compiler uses to create a .NET assembly. It sets the option in the Ace file according to the extension of the input assembly. If the original assembly file has the extension "dll" the compiler will generate a dll assembly.

Another option in the Ace file is `msil_clr_version (x)`. If the parameter `x` is set to "v1.0.3705", the compiler uses the libraries from .NET Framework version 1.0, if it is set to "v1.1.4322", the compiler uses the libraries from .NET Framework version 1.1. The parameter `msil_clr_version (x)` is also important: if the compiler tries to use the libraries from a version that is not installed, it fails to generate the contracted .NET assembly.

The lace generator sets the CLR² version of the Ace file to the same version number as used to develop the contract wizard. If a developer takes the Ace file `contract_wizard_1_1.ace` to create a new project with EiffelStudio, the lace generator adds the line `msil_clr_version ("v1.1.4322")` to the generated Ace file.

The trick to set the CLR version dynamically is to have a class with the same name in two different files. I have implemented the class *CW_CLR_VERSION* in the class files `cw_clr_version_1_0.e` and `cw_clr_version_1_1.e`. The constant *Msil_clr_version* returns the right version according to the Ace file name. An exclude clause in the Contract Wizard Ace file makes sure that EiffelStudio only creates a new Contract Wizard project with one of both class files. When the lace generator adds the CLR version to the Ace file it gets the same version as used in EiffelStudio. Table 24 shows a listing of class *CW_CLR_VERSION* with the parameter set for version 1.1 of the .NET Framework.

```
class CW_CLR_VERSION
feature -- Access
    Msil_clr_version: STRING is "msil_clr_version (%"v1.1.4322%)"
    -- Msil_clr_version ("v1.1.4322")
end
```

Table 24: Class CW_CLR_VERSION

When I tested the Contract Wizard on other .NET libraries, I noticed that the Ace generator does not create a list of all referenced assemblies in Ace file; it just appended a reference to the `microsoft` statically. However, the compiler needs a list of all assemblies referenced by the input

² Common Language Runtime

assembly in the “assembly” clause of the Ace file. The feature *generate_referenced_assembly_list* of class *CW_LACE_GENERATOR* appends all referenced assemblies with their name, version number, and public key token to the “assembly” clause in the Ace file. Furthermore it sets a prefix for most assemblies to disambiguate class names.

4.3.3 XML VISITOR

The XML generator uses the XML visitor (*CW_XML_VISITOR*) to generate the XML file holding the Eiffel class representation. The new attributes in the AST (*overloaded_name*, *is_public*, *is_by_reference*) caused changes in the XML visitor. I adapted the visitor in a way that it appends the new attributes at the according position together with their values.

The feature *generate_contracts* creates the XML tags with the contracts of a class. It checks for every assertion whether it is an invariant (*is_invariant*), a precondition (*is_precondition*), or a postcondition (*is_postcondition*). However, *is_postcondition* (*an_assertion*) always returned false even if *an_assertion* was a postcondition. Thus the XML file did not contain the postconditions of a feature. The mistake was a local variable with a wrong type: *a_postcondition* was declared as *CW_PRECONDITION*. Table 25 shows the corrected feature.

```

is_postcondition (an_assertion: CW_ASSERTION): BOOLEAN is
    -- Is 'an_assertion' a postcondition ?
    require
        assertion_not_void: an_assertion /= Void
    local
        a_postcondition: CW_POSTCONDITION
    do
        a_postcondition ?= an_assertion
        Result := (a_postcondition /= Void)
    end

```

Table 25: Feature *is_postcondition* of *CW_XML_VISITOR*

4.3.4 HTML

The user can access the interface of any type by selecting the corresponding type node in the *assembly tree*. The wizard launches a web browser and displays the interface view of the selected type including its contracts. The text layout is the same as in the HTML documentation generated by ISE EiffelStudio [2].

It is important that the Eiffel interface is displayed in a nice layout with different colors. However, the EiffelVision2 library provides no facility to render text. After considering other

possibilities, we decided to have a platform independent solution: generating an HTML file for every selected type and launching a browser to display them.

The HTML generator (*CW_HTML_GENERATOR*) handles the creation of the HTML files. The creation procedure *make* creates an instance of *CW_HTML*, namely *html_page*. *CW_HTML* represents the content of an HTML file, divided into title, head, and body. A client can set these attributes by calling *set_title (a_string)*, *set_head_attributes (a_string)*, *append_head_attributes (a_string)*, and *set_body_content (a_string)* without worrying about HTML tags. A call like *html_page.out* returns the content of the whole HTML file with correct title, head, and body tags. After the creation of *html_page*, *make* appends a reference to the style sheet used to render HTML. The style sheet is located in the subdirectory `doc\css` relative to `$CONTRACT`.

Like the Eiffel visitor, the HTML visitor (*CW_HTML_VISITOR*) takes over the part of generating the HTML source. *CW_HTML_VISITOR* inherits from *CW_VISITOR* and redefines *type_text* and *feature_text* to *CW_HTML_TEXT*. (The attributes *type_text* and *feature_text* hold the text being generated by the visitor.) *CW_HTML_TEXT* inherits from *STRING* and provides features to append new lines, tabs, text, or text together with a class identifier from a style sheet. The client does not need to take care of HTML tags since *CW_HTML_TEXT* manages them. See Table 26 for a code listing.

```

put_basic (s: STRING) is
    -- Append `s' to `html_text'.
    require
        s_not_empty: s /= Void and then not s.is_empty
    do
        append (s)
    ensure
        s_appended: has_substring (s)
        text_bigger: count > old count
    end

put_span (a_text: STRING; a_class_ident: STRING) is
    -- Put `a_text' in span with `a_class_ident' attribute.
    require
        a_text_not_void: a_text /= Void
        a_text_not_empty: not a_text.is_empty
        a_class_ident_not_void: a_class_ident /= Void
        a_class_ident_not_empty: not a_class_ident.is_empty
    do
        put_basic ("<SPAN CLASS=")
        put_basic (Inverted_comma)
        put_basic (a_class_ident)
        put_basic (Inverted_comma)
        put_basic (">")
        put_basic (a_text)
        put_basic ("</SPAN>")
    end

```

Table 26: Features *put_basic* and *put_span* of *CW_HTML_TEXT*

The features from the HTML visitor are the same as the features from the Eiffel visitor but the text is appended in a different way. For example to add the text “class interface” the visitor calls *type_text.put_span (Class_keyword + Space + Interface_keyword, E_keyword)*. The text “class interface” is now linked to the class “.ekeyword” defined in the stylesheet and therefore displayed as an Eiffel keyword. Always when the text has to be formatted in a special way, the visitor calls *put_span* with the text and the corresponding identifier of the style sheet class; otherwise it calls *put_basic*. The HTML visitor does not generate code for the non-public features. They are not listed in the *assembly tree* and the user cannot add contracts to them.

The HTML page is generated dynamically after the user decided to display the interface view of a class. Thus, the invariants, preconditions, and postconditions are always up-to-date and match with the contracts listed in the *invariant view* and *contract view* of the *assembly dialog*.

Note: The HTML generator stores the HTML files in the subdirectory *cw_html* relatively to the Contract Wizard delivery directory (\$CONTRACT).

4.4 COMMAND LINE INTERFACE

The Contract Wizard also provides a command line interface (TUI, short for text user interface) to add contracts to a .NET assembly. The input options and the application flow have changed slightly compared to the initial version developed by Dominik Wotruba. Refer to chapter 6.4.3 for a guided tour about how to use the command line of the Contract Wizard. The following section describes the modifications in the respective class files.

The class *CONTRACT_WIZARD_TUI* (originally called *APPLICATION*) handles the user input from the command line. Some changes in the class *CONTRACT_WIZARD_TUI* were necessary because the GUI and the TUI use the same class files; other changes are optimizations.

I added the notion of project to the Contract Wizard. Therefore the text interface creates a project file from its input. It takes the path to the assembly file (*an_assembly_path*) and checks if the file exists. If this is not the case it terminates the application, otherwise the TUI extracts the name of the assembly (*an_assembly_name*) from the assembly path. Afterwards it gets the name of the project (*a_project_name*) by taking the assembly path without the extension .dll or .exe. The “wizard” only continues if the specified working directory exists. Now the TUI has enough information to create a project file, respectively a project handler (*CW_PROJECT_HANDLER*). The TUI needs no longer to care about loading the assembly from the assembly path; this job is done by the project handler.

If the user wants to remove all contracts from an assembly, the TUI deletes the XML file from the project directory and launches the compiler to generate an assembly without contracts. The old version did not start the compilation thus the actual contracted .NET assembly remained unchanged. The TUI does not provide an option to edit existing contracts anymore. It comes from the location where the generated XML file is stored. Originally the “wizard” stored it in a subdirectory of the global assembly cache (GAC) and it was not accessible to the user outside the

TUI environment. Now the XML file is stored in the project directory and the user can edit its content at any time.

The feature *add_contracts* reads the contracts from a file and uses the contract handler (*CW_CONTRACT_HANDLER*) to add the contracts to the corresponding classes and features. Before the contract handler adds an invariant to a class, it checks whether the class already contains the same invariant. It also inspects the existing preconditions of a feature before calling *a_feature.add_precondition (a_precondition)*; the same applies to postconditions.

After the creation of the Ace file and Eiffel proxy classes the TUI starts the compilation (finalizing and “finish freezing”). The resulting assembly has the same file extension as the original assembly. The controller creates the XML file containing the representation of the Eiffel classes after a successful compilation. In the old wizard version, the controller generated the Eiffel proxy classes a second time although they were not changed.

5. ADVANTAGES AND LIMITATIONS

5.1 BENEFITS OF USING CONTRACT WIZARD

Except Eiffel for .NET, the .NET languages such as C#, VB.NET, C++, etc. do not support contracts. It is because the CLI (Common Language Infrastructure) does not natively support Design by Contract. The CLI is the core of Microsoft .NET technology; it is a runtime environment which specifies a common type system and an intermediate language. It supports managed execution of components written in multiple high level languages that all compile to a common intermediate language (IL).

Thanks to the Contract Wizard it is possible to add Eiffel-like contracts to any .NET assembly, whatever language it was originally written in. The programmer uses the wizard to explore the classes and their features, interactively deciding to add contract elements – preconditions, postconditions, and class invariants. The wizard uses this input to generate a proxy assembly that implements the contracts and call the non-contracted original component. Once the new assembly is generated, the programmer can simply use it instead of the original assembly and has the benefits of the specified contracts.

The Contract Wizard provides incrementality: it is possible to add contracts to an already contracted assembly. It is also possible to edit and remove existing contracts. Among the Eiffel proxy classes, the tool generates an XML file to store the contracts, hence the incrementality.

5.2 LIMITATIONS

However, adding contracts to an assembly a posteriori is not as good as putting the contracts right from the start. It is still better to have native support for contracts and write them right from the beginning when developing software. The runtime assertion monitor raises an exception when an assertion is violated. An assertion violation result from a bug that probably would not have been detected without native support of contracts. Therefore, contracts help in writing correct software.

Contracts also facilitate the task of software documentation. By just reading the contracts of a class, a developer quickly gets an overview of its obligations and benefits (especially useful when “reading” libraries). Above all, it is difficult to specify contracts for an assembly a posteriori. The best contracts cross someone’s mind while developing the software. If they cannot be written directly into the source code, the probability is big that the contracts get lost.

OVERLOADED .NET MEMBERS

The Contract Wizard is more powerful than before, but it still has limitations. The applied algorithm to resolve overloaded method names works fine apart from some special cases. An unsuccessful example is the class *System.Security.Principal.WindowsPrincipal* from the core .NET library mscorlib. It is an effective class and inherits from the interface *IPrincipal*. This means that the according Eiffel proxy class *CW_WINDOWS_PRINCIPAL* has to provide an implementation for all deferred features declared in the interface *IPRINCIPAL*. Table 27 shows the interface view³ of *IPRINCIPAL*.

```

deferred class interface IPRINCIPAL

feature -- Query

    deferred is_in_role (a_role: SYSTEM_STRING): BOOLEAN
        -- dotnet_name: "IPrincipal.IsInRole (role: String): Boolean"

    deferred identity: IIDENTITY
        -- dotnet_name: "IPrincipal.get_Identity (): IIdentity"

end

```

Table 27: Class interface of IPRINCIPAL

A problem arises because the class *WindowsPrincipal* effects the feature *is_in_role* and additionally overloads this feature. As a consequence, the Contract Wizard renames the overloaded features in the resulting class *CW_WINDOWS_PRINCIPAL* to get disambiguated names. But now, the class does not compile because it no more provides an implementation for the deferred feature *is_in_role* inherited from *IPRINCIPAL*. The overloading algorithm appends to every overloaded feature name the type of the argument and therefore, no feature name matches *is_in_role*. The compiler states: “Error: class has deferred feature(s), but is not declared as deferred.” Table 28 lists the relevant code of *CW_WINDOWS_PRINCIPAL* generated by the Contract Wizard.

```

class CW_WINDOWS_PRINCIPAL inherit
    IPRINCIPAL

feature -- Query

    is_in_role_string (a_role: SYSTEM_STRING): BOOLEAN is
        -- dotnet_name: "WindowsPrincipal.IsInRole (role: String): Boolean"
        -- (+2 overloads)

    do

```

³ An interface view is not a valid Eiffel class; it is a form of software documentation.

```

Result := windows_principal_ref.is_in_role (a_role)
end

is_in_role_windows_built_in_role (a_role: WINDOWS_BUILT_IN_ROLE): BOOLEAN is
-- dotnet_name:
-- "WindowsPrincipal.IsInRole (role: WindowsBuiltInRole): Boolean"
-- (+2 overloads)
do
Result := windows_principal_ref.is_in_role (a_role)
end

is_in_role_integer (a_rid: INTEGER): BOOLEAN is
-- dotnet_name: "WindowsPrincipal.IsInRole (role: Int32): Boolean"
-- (+2 overloads)
do
Result := windows_principal_ref.is_in_role (a_rid)
end

```

Table 28: Relevant features from class CW_WINDOWS_PRINCIPAL

A similar error occurs in the class *System.Type* which inherits from *System.Reflection.ICustomAttributeProvider*. The interface *ICustomAttributeProvider* has an overloaded feature *GetCustomAttributes*. In Eiffel for .NET the corresponding feature names are translated into *get_custom_attributes* and *get_custom_attributes_type*. (The Eiffel emitter⁴ only adapts the name of one feature; the other feature name stays the same.) This means that all Contract Wizard proxy classes inheriting from *ICUSTOM_ATTRIBUTE_PROVIDER* must list features with exactly the same name. The Contract Wizard does not know how the features are named in the inherited Eiffel for .NET class. It applies the standard renaming algorithm and translates the feature names into *get_custom_attributes_boolean* and *get_custom_attriubtes_type*.

This particular problem could be solved when the generated Eiffel proxy classes do not inherit from the abstract classes produced by the Eiffel Emitter (e.g. *ICUSTOM_ATTRIBUTE_PROVIDER*), but from the according class generated by the Contract Wizard (e.g. *CW_ICUSTOM_ATTRIBTE_PROVIDER*). Because the class *CW_ICUSTOM_ATTRIBUTE_PROVIDER* contains two features with the same name, it adapts the name of both.

There exist classes where it is not understandable how the Eiffel Emitter generates the names for overloaded features in wrapper classes. (The algorithm used by the Emitter is not publicly available.) Table 29 shows an example from the deferred class *ITYPE_LIB_CONVERTER*. The Contract Wizard does not generate the same name for the overloaded feature *convert_type_lib_to_assembly* in the descendant Eiffel proxy class *CW_TYPE_LIB_CONVERTER* and therefore the compiler prints an error message. The first

⁴ The tool Eiffel Emitter integrated in EiffelStudio analyzes a .NET Framework assembly and produces XML files of the assembly classes. The compiler uses its internal recipe to compile all this into a .NET assembly.

feature name from *ITYPE_LIB_CONVERTER* remained untouched. On the other hand, the Emitter appended all type names except for the last type name to the second feature name. It would be sufficient to add fewer types name to obtain a unique feature name. In such cases the Contract Wizard has no chance to get exactly the same feature names for the overloaded feature *convert_type_lib_to_assembly* in the proxy classes that inherit from *ITYPE_LIB_CONVERTER*.

```

deferred class interface ITYPE_LIB_CONVERTER

feature -- Query

    deferred convert_type_lib_to_assembly (type_lib: SYSTEM_OBJECT;
        asm_file_name: SYSTEM_STRING; flags: INTEGER;
        notify_sink: ITYPE_LIB_IMPORTER_NOTIFY_SINK;
        public_key: NATIVE_ARRAY; key_pair: STRONG_NAME_KEY_PAIR;
        unsafe_interfaces: BOOLEAN): ASSEMBLY_BUILDER
        -- dotnet_name: "IPrincipal.IsInRole (role: String): Boolean"

    deferred convert_type_lib_to_assembly_object_string_type_lib_importer_flags_
        itype_lib_importer_notify_sink_integer_8_array_strong_name_key_pair_string
        (type_lib: SYSTEM_OBJECT; asm_file_name: SYSTEM_STRING;
        flags: TYPE_LIB_IMPORTER_FLAGS;
        notify_sink: ITYPE_LIB_IMPORTER_NOTIFY_SINK;
        public_key: NATIVE_ARRAY; key_pair: STRONG_NAME_KEY_PAIR;
        asm_namespace: SYSTEM_STRING; asm_version: VERSION):
        ASSEMBLY_BUILDER
        -- dotnet_name: "IPrincipal.IsInRole (role: String): Boolean"

```

Table 29: Part of class interface of *ITYPE_LIB_CONVERTER*

A problem occurs when a .NET interface overloads a method from class *System.Object* and another class provides an implementation for this interface. The interface *System.Collections.IHashCodeProvider* defines the function *GetHashCode: int32 (object)*. Every interface inherits from the class *System.Object* which has a function *GetHashCode: int32()*. This means that the feature *GetHashCode: int32 (object)* from class *IHashCodeProvider* is overloaded. In the corresponding Contract Wizard proxy class, the feature is renamed into *get_hash_code_object* (see Table 30). The Eiffel proxy class *CW_CASE_INSENSITIVE_HASH_CODE_PROVIDER* inherits from *IHASH_CODE_PROVIDER* (1) and implements the feature *get_hash_code_object* with the correct overloaded name (4).

By default, every Contract Wizard class inherits from *SYSTEM_OBJECT*. If the descendant class of *SYSTEM_OBJECT* is effective, it redefines the features *get_hash_code*, *equals*, and *to_string* (3) and implements them (5). When a class inherits from an interface, it has to undefine the features *equals*, *to_string*, and *get_hash_code* to avoid ambiguous feature names. (These features are already inherited from *SYSTEM_OBJECT* (3).) A special case is when the interface defines one of these features explicitly. An example can be found in the interface *IMEMBERSHIP_CONDITION*: it defines the features *equals* and *to_string*; therefore the descendant class only has to undefine *get_hash_code* since the other two features are already defined by the interface.

In the example from Table 30 *CW_CASE_INSENSITIVE_HASH_CODE_PROVIDER* inherits from the interface *IHASH_CODE_PROVIDER* and only lists *equals* and *to_string* in the undefine clause. The error can be found in the .NET parser: it states that the class *IHASH_CODE_PROVIDER* defines the feature *get_hash_code*, although this is not the case. Indeed, it defines a feature with the same name, but not the same signature. The bug can be fixed by changing the parser to inspect not only the name of feature, but also its signature. Then, the parser would notice that the class *IHASH_CODE_PROVIDER* does not provide a definition for *get_hash_code* (with the same signature as in *SYSTEM_OBJECT*), *get_hash_code* would appear in the undefine clause, and the class would compile.

```

class CW_CASE_INSENSITIVE_HASH_CODE_PROVIDER inherit
  IHASH_CODE_PROVIDER (1)
  undefine (2)
    equals,
    to_string -- , get_hash_code is missing
  end

  SYSTEM_OBJECT
  redefine (3)
    get_hash_code,
    equals,
    to_string
  end

  feature -- Query

  get_hash_code_object (a_obj: SYSTEM_OBJECT): INTEGER (4)
    -- dotnet_name:
    -- "CaseInsensitiveHashCodeProvider.GetHashCode (obj: Object): Int32"
    -- (+1 overloads)

  get_hash_code: INTEGER (5)
    -- dotnet_name:
    -- "CaseInsensitiveHashCodeProvider.GetHashCode (): Int32"
    -- (+1 overloads)
  ...

```

Table 30: Extracts of class CW_CASE_INSENSITIVE_HASH_CODE_PROVIDER

The same problem occurs in all classes inheriting from *IFORMATTABLE*: it overloads the feature *to_string*.

The interface *System.Collections.IEnumerator* forces its descendants to provide an implementation for the property *Current* returning an instance of *System.Object*. The corresponding Eiffel for .NET class *IENUMERATOR* calls the property *current_* (“*Current*” is an Eiffel keyword and has to be distinguished.) The class

System.Runtime.Serialization.SerializationInfoEnumerator respectively its corresponding Contract Wizard class *CW_SERIALIZATION_INFO_ENUMERATOR* implement *Current*. However, they do not return an object of type *SYSTEM_OBJECT* but of type *SERIALIZATION_ENTRY*. Normally, this is a legal redeclaration since *SERIALIZATION_ENTRY* is a subtype of *SYSTEM_OBJECT*. The problem is that *SERIALIZATION_ENTRY* is expanded and the compiler states a non conforming signature of the redeclaration.

```

Error code: VDRD(2)
Type error: redeclaration has non-conforming signature.
What to do: make sure that redeclaration uses signature (number and types of
arguments and result) conforming to that of the original.

Class: CW_SERIALIZATION_INFO_ENUMERATOR
Redefined feature: current_: expanded CW_SERIALIZATION_ENTRY
From: CW_SERIALIZATION_INFO_ENUMERATOR
Precursor: current_: SYSTEM_OBJECT From: IENUMERATOR

```

Table 31: Compiler error of class CW_SERIALIZATION_INFO_ENUMERATOR

As a temporary solution, I list the deferred signature of the property (*current_*: *SYSTEM_OBJECT*) as a private feature in the {*NONE*} clause of the class *CW_SERIALIZATION_INFO_ENUMERATOR* without giving an implementation. (A call of the according proxy feature would not make sense because *serialization_info_enumerator_ref.current_* would call a deferred feature.) After manually deleting the feature *current_*: *CW_SERIALIZATION_ENTRY* the compilation succeeds since *CW_SERIALIZATION_INFO_ENUMERATOR* provides a (pseudo) implementation for the property *Current*.

5.3 FUTURE WORK

I did as much as possible to eliminate the limitations in the generated Eiffel classes, but I did not have enough time to solve all problems. To improve the wizard it is necessary to analyze precisely all classes that do not compile and to adapt the classes *CW_DOTNET_PARSER* or *CW_EIFFEL_VISITOR* accordingly. However, some classes will be hard to adapt, as for example *CW_TYPE_LIB_CONVERTER* (see discussion above).

When a .NET class *X* implements an interface *I*, the corresponding Contract Wizard class *CW_X* implements the same interface *I*. It is not always straightforward to provide an implementation of all features defined in interface *I* in a descendant class *CW_X* (see discussion of section 5.2). It is worthwhile to analyze how the Contract Wizard proxy classes behave when they inherit from an interfaces *CW_I* (1) generated by the Contract Wizard, as shown in Table 32.

```

class CW_TYPE_LIB_CONVERTER inherit
  CW_ITYPE_LIB_CONVERTER (1)
  undefine
    get_hash_code,
    equals,
    to_string
  end

  CW_SYSTEM_OBJECT
  redefine
    get_hash_code,
    equals,
    to_string
  end

```

Table 32: Suggestion of inheritance structure of CW_TYPE_LIB_CONVERTER

In that case all overloaded features are renamed with the same algorithm and one does not have to bother about the Eiffel Emitter handles overloaded features. As a consequence, the Contract Wizard has to generate Eiffel proxy classes for all assemblies referenced by the assembly given as input to the Contract Wizard (i.e. the assembly to which the user wants to add contracts). The referenced assemblies are needed to compile the given assembly; they are listed in the “assembly” clause of the Ace file. For performance reasons, we could imagine a few optimizations:

Generate the Eiffel proxy classes corresponding to the referenced assemblies as soon as the user selects an assembly to add contracts to, and not when he starts adding contracts. (Because these classes corresponding to the .NET classes will always be the same, we do not add contracts to them.)

We could add incrementality: always generate those references classes at the same place on the disk, e.g. \$CONTRACT\assemblies and check whether a given assembly has already been generated before generating new proxy classes again.

Another alternative is to also give the user the possibility of adding contracts to those referenced assemblies.

An interesting approach is to generate factory-like classes to strengthen Design by Contract. Instead of returning Eiffel wrapper classes generated by the Emitter tool, we return (contracted) classes generated by the Contract Wizard. Table 33 gives an idea of the approach.

```
class CW_ACCOUNT
feature -- Query

  deposits: CW_DEPOSIT_LIST is
    -- dotnet_name: "Account.Deposits: DepositList"
  do
    create Result.make (account_ref.deposits)
  end
```

Table 33: Factory-like Contract Wizard proxy class

6. USER MANUAL

6.1 SYSTEM REQUIREMENT

Currently, the only operating system that Contract Wizard supports is Windows, basically because of the Microsoft .NET Framework [13]. However, with Mono [15] it should be possible to use the Contract Wizard with slight changes under Linux. Mono is an open source development platform based on the .NET Framework and includes Microsoft .NET compatibility libraries.

The graphical user interface itself is platform independent as it is built with the EiffelVision2 library. Also the file paths are set in a way that it does not matter on which platform you run the Contract Wizard. I only tested the application under Windows XP, but it should also run under Windows 2000.

I used version 5.4 of EiffelStudio to develop the Contract Wizard. You should use (at least) the same version because it supports a transparent way to call overloaded .NET routines from Eiffel [2].

6.2 INSTALLATION

First you have to make sure that you have the **.NET Framework** installed. You can check if you have already installed it by clicking *Start* on your Windows desktop, selecting *Control Panel*, and then double-clicking the *Add or Remove Programs* icon. When that window appears, scroll through the list of applications. If you see Microsoft .NET Framework 1.1 listed, the latest version is already installed and you do not need to install it again. If you see Microsoft .NET Framework v1.0.3705 you have installed the older version 1.0. If you see neither of them you have to install the .NET Framework Redistributable 1.1 [12]. It includes everything you need to run applications developed using the .NET Framework.

Now you are ready to install this package. Please follow the instructions below:

1. Download the file “ContractWizard.zip”. It contains:
 - the source code of Contract Wizard (for both the GUI and command line versions) in the “src” directory
 - executables for the GUI and command line versions in the “bin” directory
 - this user manual in the “doc” directory

- the document type definition of a valid XML structure in the “dtd” directory
2. Unzip its content to a directory of your choice and set the CONTRACT environment variable to point to that directory.
 3. Set the PATH environment variable to the bin directory where your ISE Eiffel compiler and wel_hook.dll is located. On a Windows machine the path looks similar to “\$ISE_EIFFEL\studio\spec\windows\bin“. If the PATH variable already has values, append the Eiffel bin path with a semicolon.

It is necessary to set the path on the wel_hook.dll because the Contract Wizard uses it when you pick-and-drop a node from the *assembly tree* (see section 6.4.2).

Before you create a new project in EiffelStudio check which version of the .NET Framework you have installed (see 6.1 above). If you have version 1.0 select the control file “contract_wizard_1_0.ace”, for version 1.1 select “contract_wizard_1_1.ace”. I did the development under version 1.0 but it also works for version 1.1.

6.3 HOW TO CREATE A .NET ASSEMBLY

If you want to contract a .NET assembly first you must have an assembly: either you take an existing one or you create one by yourself. In the following example I show how to create a .NET assembly using Visual Studio .NET [14] or Eiffel ENViSioN! [2]. Of course you can also use another development environment.

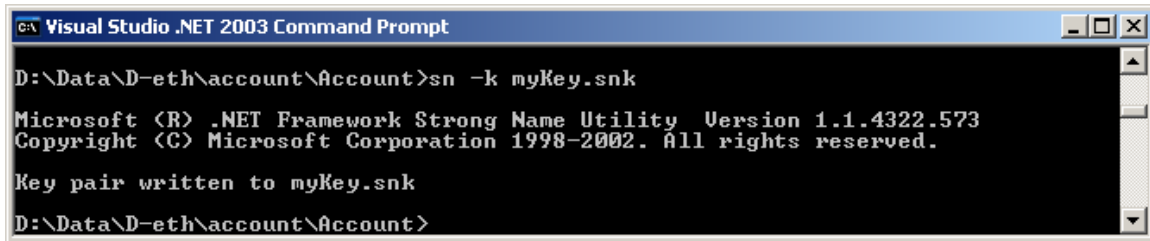
To create a .NET assembly launch Visual Studio .NET and select the button “New Project”.

Decide what kind of programming language you want to use and select the corresponding project type (Visual Basic Projects, Visual C# Projects, Visual J# Projects, Visual C++ Projects, or Eiffel Projects). In the template section on the right hand side select the item named “Class Library”. Give your project a name and specify a location where it shall be stored.

Now you can implement your classes for the assembly. You can also add existing files to the project. When you are finished, build your project. Your assembly file is now in the directory /bin/Debug relative to your project directory. It has the same name as your project with the extension .dll, e.g. Account.dll.

The next step is to deploy the assembly in the global assembly cache (*GAC*). The *GAC* stores assemblies designated to be shared by several applications on the computer. The assembly has to be in the *GAC* because the Eiffel compiler, launched by the Contract Wizard, needs to access your assembly. To put an assembly in the *GAC* it must have a strong name.

To sign an assembly with a strong name, you need a public/private key pair. This public and private cryptographic key pair is used during compilation to create a strong-named assembly. You can create a key pair using the Strong Name tool (sn.exe). Key pair files usually have an .snk extension. The example in Figure 12 creates a key pair called myKey.snk.



```
CA Visual Studio .NET 2003 Command Prompt
D:\Data\D-eth\account\Account>sn -k myKey.snk
Microsoft (R) .NET Framework Strong Name Utility Version 1.1.4322.573
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.
Key pair written to myKey.snk
D:\Data\D-eth\account\Account>
```

Figure 12: Creating a key pair

Once you have created the key pair, you must put the file where the strong name signing tools can find it. When using command-line compilers, you can simply copy the key to the current directory containing your code modules. If you are using Visual Studio .NET you have to understand where the development environment looks for the key file. For example, the C# compiler looks for the key file in the directory containing the binary. As you have the key file in the project directory, set the file attribute in `AssemblyInfo.cs` as follows:

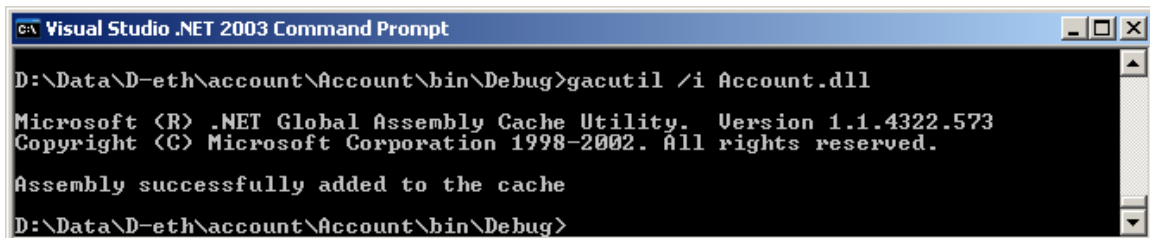
```
[assembly: AssemblyKeyFile("../..\myKey.snk")]
```

When you created the new project, Visual Studio .NET automatically created the file `AssemblyInfo.cs`. There you can also specify more information about the assembly such as the version of the assembly or its description. To indicate that your assembly is CLS (Common Language Specification) compliant add the following line to the `AssemblyInfo` file.

```
[assembly: System.CLSCompliant(true)]
```

The assembly must be CLS compliant; otherwise the Contract Wizard cannot generate Eiffel proxy classes of your code.

Rebuild your project. The assembly now has a strong name (public key token is assigned) and can be installed in the GAC. There are several ways to deploy an assembly into the global assembly cache, for example you can use a developer tool called the Global Assembly Cache tool (`gacutil.exe`), provided by the .NET Framework SDK. Figure 13 shows how you can do it using `gacutil`.



```
CA Visual Studio .NET 2003 Command Prompt
D:\Data\D-eth\account\Account\bin\Debug>gacutil /i Account.dll
Microsoft (R) .NET Global Assembly Cache Utility. Version 1.1.4322.573
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.
Assembly successfully added to the cache
D:\Data\D-eth\account\Account\bin\Debug>
```

Figure 13: Deploy assembly into global assembly cache

Now your assembly is ready to be used by the Contract Wizard!

6.4 HOW TO USE CONTRACT WIZARD 3.0

6.4.1 ADDING ASSEMBLY TO GAC

Before you start adding contracts to a .NET assembly, make sure that the assembly is installed in the global assembly cache (GAC); otherwise the Contract Wizard cannot compile your assembly. If you are using Windows you can use the program “Microsoft .NET Framework Configuration” or “Microsoft .NET Framework Configuration 1.1” available from “Control Panel\Administrative Tools” (see Figure 14). (Which version depends on the .NET Framework version you are using.) There you can view a list of assemblies in the GAC. You can add an assembly to the cache by selecting it from the appearing “Add an Assembly” dialog.

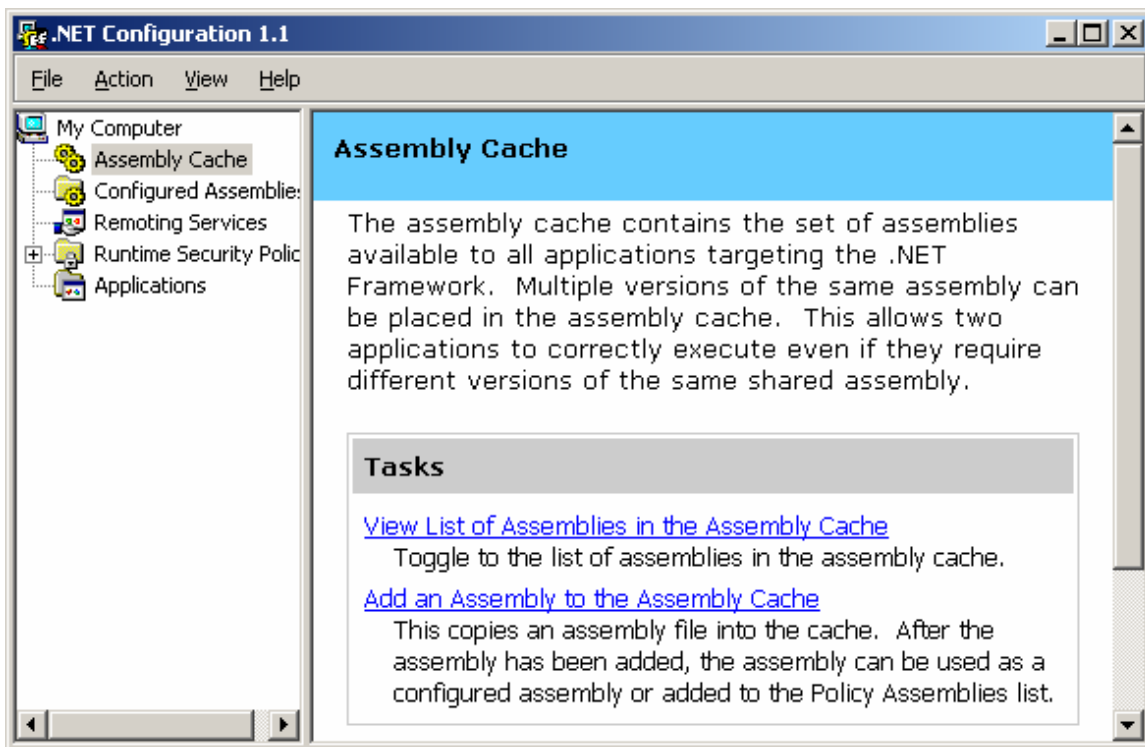


Figure 14: .NET Configuration 1.1, assembly cache

6.4.2 GUI VERSION

When you launch the application the Contract Wizard first checks whether you have defined the environment variable CONTRACT. If you have forgotten to set it, the wizard asks you to do so. Figure 15 shows the dialog that the wizard launches when you missed to set the CONTRACT variable.



Figure 15: Warning dialog when the CONTRACT variable is not set

After a successful check the wizard shows a welcome dialog (see Figure 16). It tells you what you can do with the wizard. At the bottom it has a button bar with four buttons that always stay the same during the wizard's execution. A press on the "Help" button launches your default web browser and shows this user guide. The "< Back" button is disabled because it is the first dialog of the wizard. If you click on "Cancel" the wizard window disappears and the execution is canceled. You must press the "Next >" button to proceed to the next dialog.

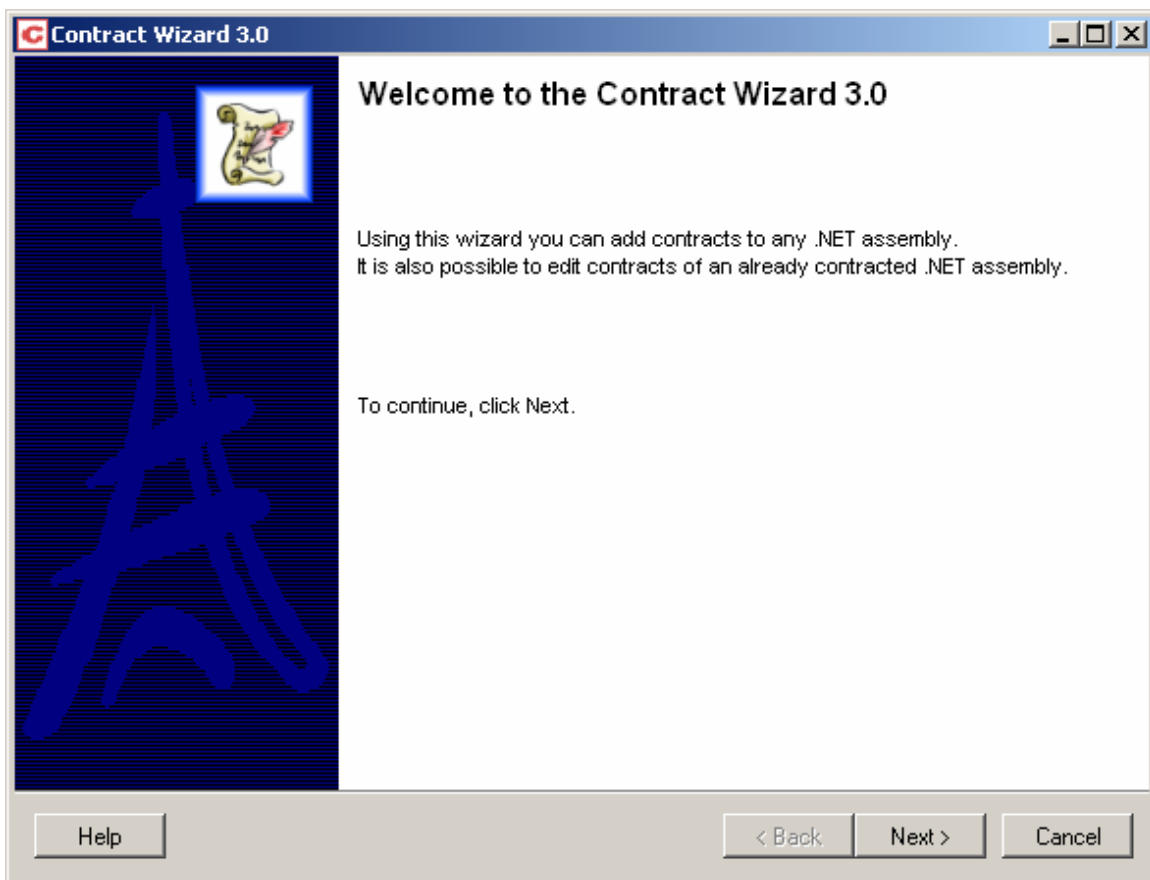


Figure 16: Welcome dialog

PROJECT DIALOG

The following dialog asks you to open an existing project or to create a new project (see Figure 17). By default the section where you can open an **existing project** is enabled. You need to provide the path to a project file with the extension **.cpr** (contract **project**). You can either type it in the text field or use the “Browse” button, which opens an “open file” dialog. It is only possible to select a project file with the correct project extension.

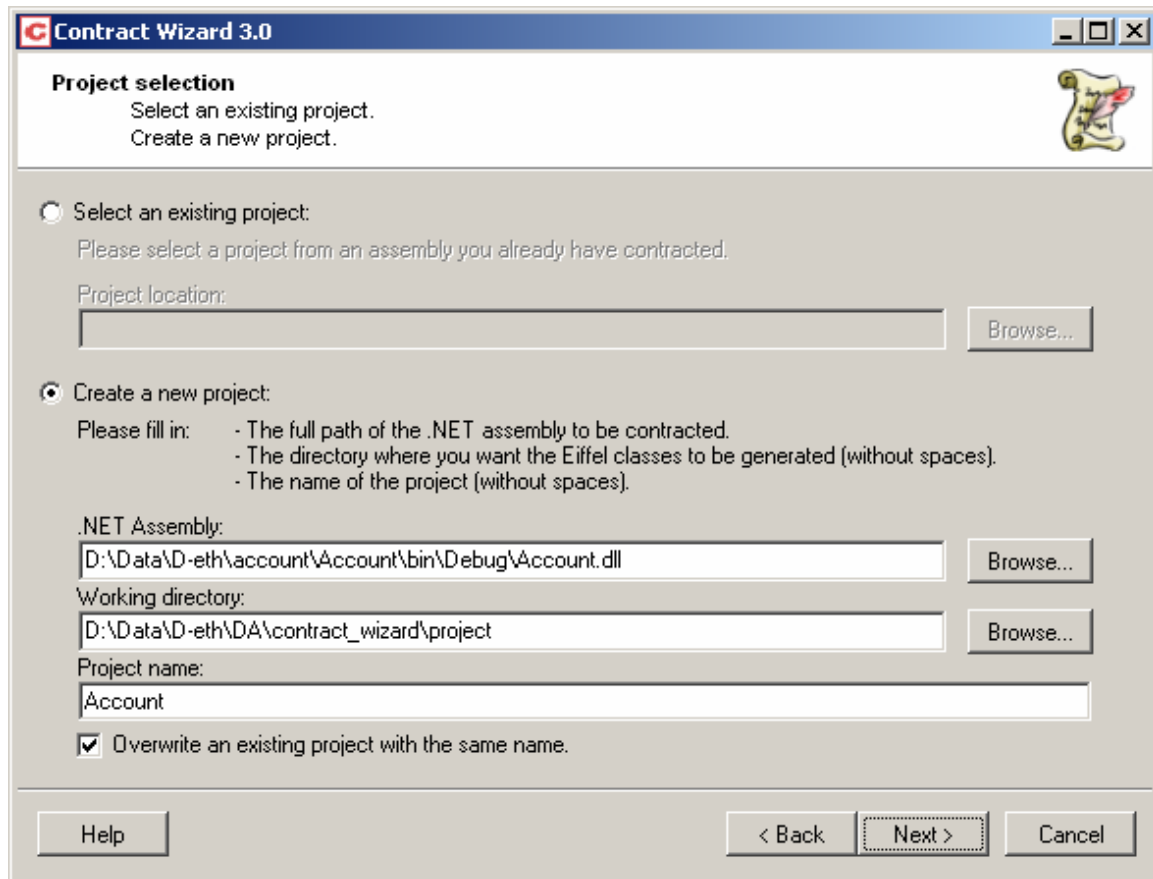


Figure 17: Project dialog

If you wish to create a **new project**, select the radio button with the label “Create a new project”. This section gets activated and you can choose the parameters of the new project. In the first text field you have to enter the full path to a .NET assembly. A click on the “Browse” button launches a file dialog where you can specify an assembly file in a more comfortable way. An assembly file has the extension **.dll** (dynamic **link library**) or **.exe** (**executable**). The filter in the dialog is set so that you can only select an assembly file with a valid extension.

By default the working directory is “\$CONTRACT\project”. The wizard stores the generated files in this directory. You can change it either by typing in the path to the desired directory or by choosing the directory from a dialog launched by pressing the “Browse” button.

When you select an assembly and the field for the project name is empty, the wizard creates a project name for you. It has the same name as the assembly. Of course you can change this name just by replacing it with another one.

The check box named “Overwrite a project with the same name” is by default selected. When you deselect it and it has already a project file with the same name in the specified directory the wizard brings up a dialog as shown in Figure 18.



Figure 18: Warning dialog when specified project already exists

To go to the next dialog of the wizard, you must press the “Next >” button. The wizard performs some validity checks before going on to the next dialog. When you decided to open an existing project you must have specified the path to the project file, otherwise a dialog appears and tells you to fill out all needed information. The same occurs if you select “Create a new project” and forget to fill in a text field from this section. If the quoted working directory does not exist the wizard tries to create it. Anyhow a problem can occur; for example if the drive letter is not valid as “X:\contract_wizard” on my hard disk. Figure 19 shows the warning dialog that the wizard shows after a directory creation error.

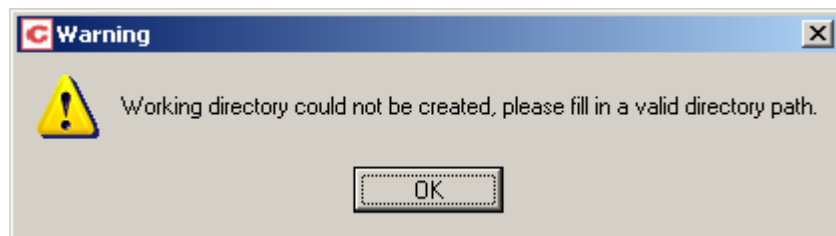


Figure 19: Warning dialog when working directory could not be created

The next check affects the assembly: first the wizard verifies that the specified file exists and that it has a valid assembly extension. If that is not the case the wizard tells you that the assembly does not exist. Then the wizard tries to load the specified assembly. The loading can cause an error when the assembly cannot be read correctly (which is the case for system assemblies). Figure 20 shows the according warning dialog.

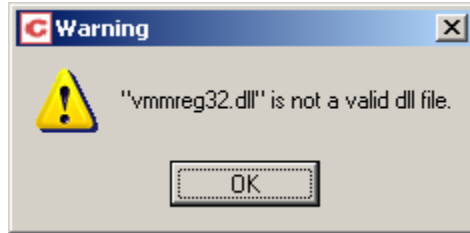


Figure 20: Warning dialog for an invalid assembly

In a last step the wizard inspects the assembly for CLS compliance. To fully interact with other objects regardless of the language they were implemented in, objects (or assemblies) must expose to callers only those features that are common to all the languages they must interoperate with. For this reason, the Common Language Specification (CLS), which is a set of basic language features needed by many applications, has been defined. If the assembly exposes features that are not CLS compliant, the wizard is not able to generate Eiffel code. Hence, a dialog informs you when the assembly does not meet this criterion.

If you indent to open an existing project, the wizard checks if you specified a valid input for the project file: the file must exist and must have the correct extension (.cpr). Otherwise you are informed through a warning dialog.

When all entered information is valid the wizard parses your specified .NET assembly in case you created a new project, or it reads the XML file containing the structure and contracts of the Eiffel classes. The latter is the case if you have already added contracts to the assembly given as input. A dialog with a progress bar appears showing you the progress of the parsing (see Figure 21). After the parsing the wizard backs up all files from your specified working directory. It stores them in the directory "cw_tmp" relatively to the contract delivery directory \$CONTRACT. In case of an unexpected error the wizard can restore the files. The backup process is also indicated through the *progress dialog*. While the progress dialog is shown the "Next >" and "< Back" button are disabled.

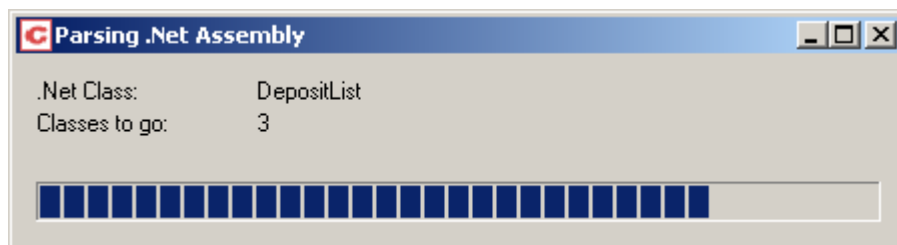





Figure 21: Progress dialog while parsing a .NET assembly

ASSEMBLY DIALOG

Now the wizard shows the next dialog (see Figure 22). In this dialog you can add contracts to the parsed .NET assembly. On top of the dialog is scanty information what you can do in the actual task. A click on “Help” opens a web browser with detailed instructions how to find one’s way in the *assembly dialog*. Right below the short instruction, you see which assembly you have selected and where it is located.

On the left hand side of the window is a tree representing the assembly with its types and their features. At the beginning the tree is collapsed and you only see the types of the .NET assembly. Every type is marked with a blue oval icon (●). A deferred type additionally has a star on the left hand side of the oval (★). You can expand the tree by clicking on the plus sign or by double clicking the class icon respectively the class name. The leaves of each type are its public features, tagged with the ISE Eiffel feature icons: attributes with , deferred features with , and the remaining features with an  symbol.

When you select a type in the assembly tree, the wizard displays the selected type name including its namespace in the text field labeled “Selected type”. A click on a feature causes the wizard to set this field with the name of the type to which the feature belongs. Furthermore you can see the feature’s signature right below in the field “Selected feature”. You can see the signature of a feature by retaining the mouse cursor a short while over a feature, as shown in Figure 22.

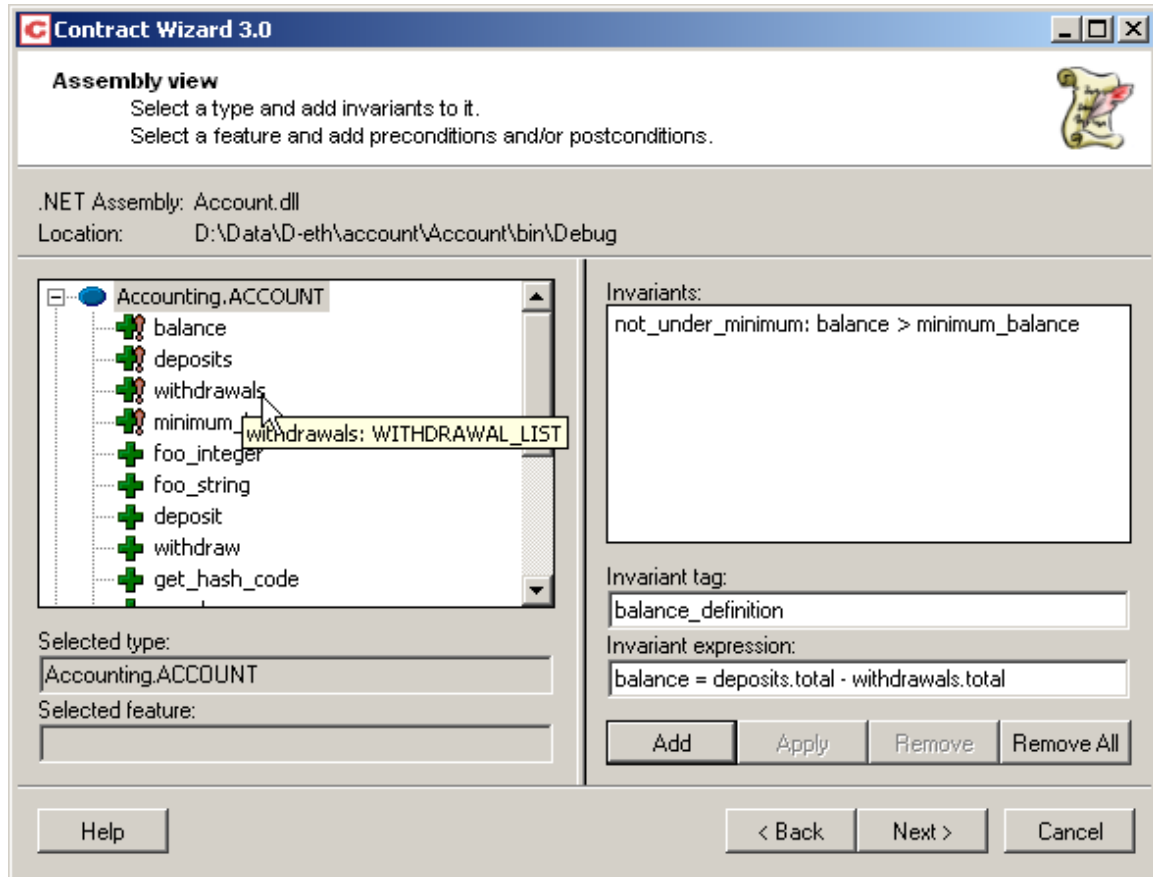


Figure 22: Assembly dialog with invariant view

In the case the selected type already has some invariants the wizard lists them in the box on the right hand side. The text fields “Invariant tag” and “Invariant expression” are there to type in the invariants of the actual type. As soon as both text fields have a value the “Add” button is enabled and you can press it to add your new invariant. Pressing the Enter key has the same effect; indicated through a border around the button. You can change an added invariant by selecting it in the invariant box and changing its tag and/or expression as you wish. When no text field is empty you can click on “Apply” and the selected invariant is updated accordingly. The wizard enables the “Remove” button when you have selected an invariant which you intend to remove. You can click on “Remove all” once the selected type contains invariants. If you do so you are asked if you really want to remove all invariants of the specified type. The wizard does not remove the invariants unless you click the “OK” button in the shown dialog.

For an invariant expression you can use all features of the selected type. See Figure 22 above for an example. Class *ACCOUNT* has an attribute *withdrawals* which is of type *WITHDRAWAL_LIST*, that again has a public feature *total*. The same applies to *deposits*; hence *balance = deposits.total - withdrawals.total* is a valid invariant expression.

It is only possible to add or update an invariant when the actual type does not already have the same tag or expression; otherwise a warning dialog appears (see Figure 23).

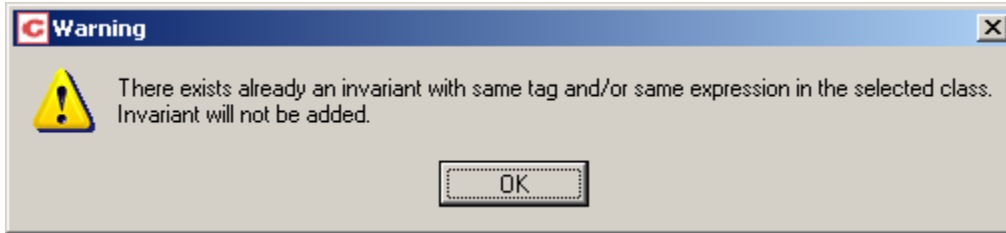


Figure 23: Warning dialog when you try to add the same invariant twice

The wizard helps you add a new contract through pick-and-drop. When you select a feature with the right mouse button, the mouse pointer changes to a crossed feature icon (✖). Then you can move the mouse pointer over the tag or expression field and the cursor alters to a normal feature icon (+). This indicates you that the text fields accept the mouse pointer. Now you can right-click again and the signature of the selected feature appears in the text field.

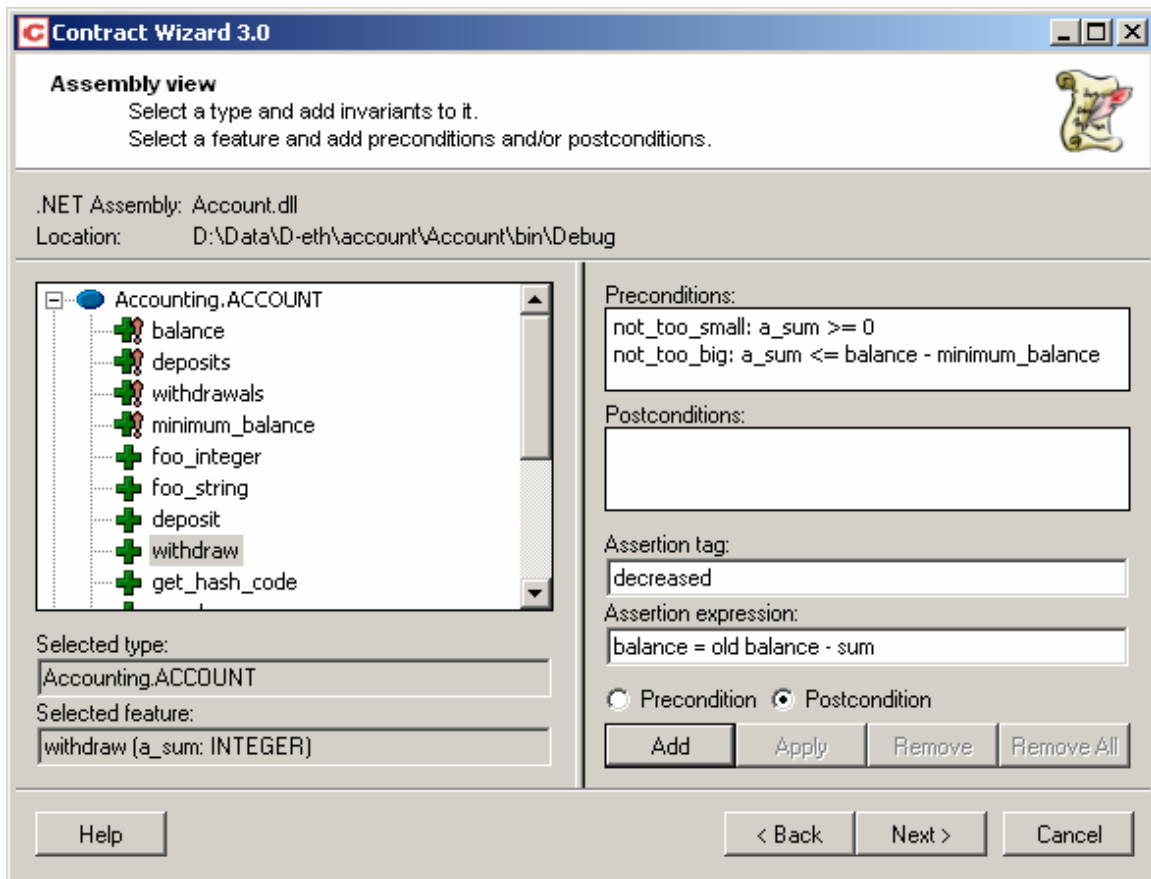


Figure 24: Assembly dialog with precondition and postcondition view

When a feature is selected in the tree, the box on the right changes to two boxes where the preconditions and postconditions of the selected feature are shown, as you can see in Figure 24. A click on a type node triggers the wizard to change back to the assertion view.

The addition of pre- and postconditions to a selected feature works similar to adding an invariant to a type. Additionally you have to specify whether you want to add a precondition or postcondition by selecting one of the radio buttons below the assertion text fields. A selection of the “Add”, “Apply”, “Remove”, or “Remove All” button always applies to the current selection of “Precondition” and “Postcondition”. When you have selected “Precondition” and then you click on “Add” or press the Enter key, the assertion is added to the preconditions of the selected feature. If you want to add a postcondition, simply select the radio button “Postcondition”.

The buttons are only enabled when it makes sense to select them. You cannot add a precondition if you did not fill in the assertion or the expression field; hence the “Add” button is disabled until both text fields have a value. To determine a button’s sensitivity state the wizard also takes into account which radio button you have selected. If you have selected a precondition in the precondition box, the “Remove” button is enabled. Now you change the radio button to “Postconditions” but you have no postconditions specified for the selected feature. Therefore the wizard disables the “Remove” button since there is no postcondition that can be removed. A click on “Remove all” causes the wizard to show a dialog asking you whether you want to remove all pre- or postconditions (depending on the selection of the radio button). It protects you from unintentional deletion of all conditions. The shield is shown in Figure 25. You can change an existing precondition when you select it in the “Preconditions” box, update its assertion or expression accordingly, and press on “Apply”. Make sure you have “Precondition” selected in the radio box otherwise you cannot use the “Apply” button. A postcondition update is analogous.

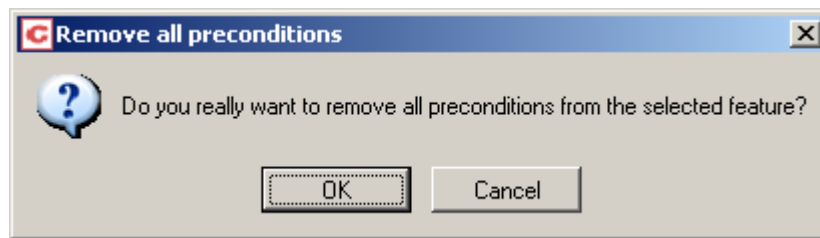


Figure 25: Question dialog if all preconditions shall be removed

A valid expression for a pre- or postcondition consists of the feature names of the selected type and/or the arguments of the selected feature. When you use a feature name make sure you have the arguments set correctly regarding the argument count and the type of each argument. Use the tool tip functionality or the pick-and-drop mechanism to help you.

The wizard adapts .NET feature names to Eiffel conformant feature names. .NET features with the same name but different signatures are translated to Eiffel names by appending the argument type to the feature name. For example, *foo_integer* and *foo_string* displayed in the assembly tree in Figure 24 originally have the same feature name, namely *foo*. If you involve an overloaded feature name in an assertion expression, use the feature name as shown in the tree and not the original feature name.

In the current version of Eiffel it is not possible to add a precondition or a postcondition to an attribute. This will be supported in the next version. A dialog pops up (see Figure 26) and informs you in the case you have selected an attribute in the tree and wanted to add an assertion to

it. The wizard also shows a dialog if you try to add the same precondition to a feature twice (of course the same is valid for postconditions).

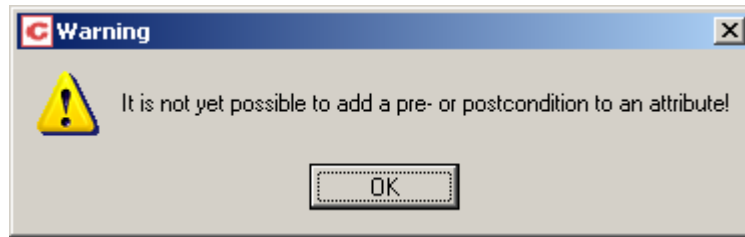


Figure 26: Warning dialog when user wants to contract an attribute

INTERFACE VIEW OF A CLASS

Sometimes it helps to look at an overview of all features of a type altogether. For that case the wizard provides a contract view for every type in the tree. You can force the wizard to display it in your default web browser by selecting a type and pressing the right mouse button. A little menu appears where you can select “Show Contract View F4” (see Figure 27). A double click on a feature node also launches the browser with the contract view of the type belonging to the clicked feature. As indicated in the context menu a press on key “F4” also launches the contract view with the currently selected tree item.

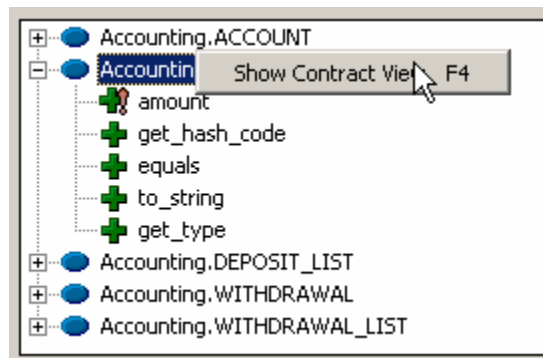


Figure 27: How to show contract view of a type

```

note: "Automatically generated by the Contract Wizard."
dotnet_name: "Accounting.Account"

class interface
    CW_ACCOUNT

create
    make

feature (NONE) -- Initialization

    make (a_initial_amount: INTEGER)
        -- Account..ctor (InitialAmount: Int32)

feature -- Access

    balance: INTEGER
        -- Account.Balance: Int32

    deposits: DEPOSIT_LIST
        -- Account.Deposits: DepositList

    withdrawals: WITHDRAWAL_LIST
        -- Account.Withdrawals: WithdrawalList

    minimum_balance: INTEGER
        -- Account.MinimumBalance: Int32

feature -- Commands

    foo_integer (an_index: INTEGER)
        -- Account.foo (anIndex: Int32)
        -- (+1 overloads)

    foo_string (a_text: SYSTEM_STRING)
        -- Account.foo (aText: String)
        -- (+1 overloads)

    deposit (a_sum: INTEGER)
        -- Account.Deposit (Sum: Int32)

    withdraw (a_sum: INTEGER)
        -- Account.Withdraw (Sum: Int32)

        require
            not_too_small: a_sum >= 0
            not_too_big: a_sum <= balance - minimum_balance

        ensure
            decreased: balance = old balance - a_sum

invariant

    not_under_minimum: balance > minimum_balance
    balance_definition: balance = deposits.total - withdrawals.total

```

Figure 28: Launched browser with interface view of selected class CW_ACCOUNT

Figure 28 shows an extract of the class interface of *CW_ACCOUNT*. What you can see are all features you can add contracts to including creation procedures; although they are exported to *NONE*. The contracts of the class (invariants, pre- and postconditions) are visible as well – always up to date. The comment on every feature tells you its .NET name. If a feature is overloaded it has an additional comment saying how many other features originally had the same name.

Note: every generated HTML page is stored in the directory “cw_html” relatively to your contract delivery directory.

FINISH EXECUTION

When you are finished with adding contracts to the assembly and want to generate a new contracted assembly click on the “Next >” button at the bottom of the wizard. Now the wizard starts generating the Eiffel class files, an Ace file (needed to compile the Eiffel files), and a XML file containing the representation of the Eiffel files including the contracts. A dialog with a progress bar appears indicating you the state of the file generation.

After the file generation the wizard checks the generated Eiffel class files for syntax errors. An example of syntax error is an assignment in an assertion expression. The *progress dialog* shows you the current file being verified and how many files still have to be controlled. When the wizard finds a syntax error in one of the classes, it informs you through a dialog (see Figure 29). To find out more about the error you can press on the “Details >>” button and a more precise error description is appended to the dialog. You are now invited to correct the error or to cancel the application.

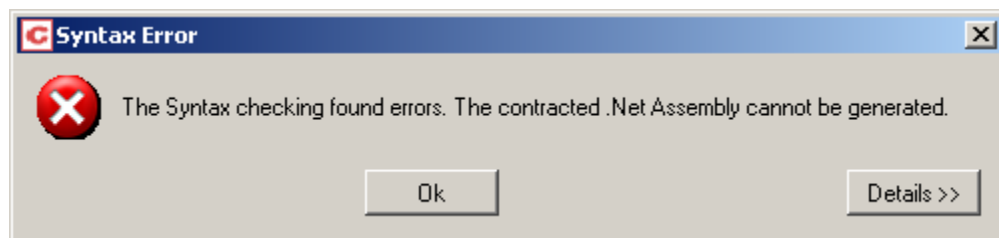


Figure 29: Error dialog after a syntax error in one of the checked classes

If no syntax error occurs, the wizard starts compiling the generated set of classes. The progress bar shows the advancement of the compilation. It looks very similar to the progress dialog of ISE EiffelStudio so you are already used to it (see Figure 30). The thermometer on the right changes from red to blue when the wizard starts finalizing the classes. After a successful compilation the wizard calls “finish freezing” to compile the C code and to link part of the finalization.

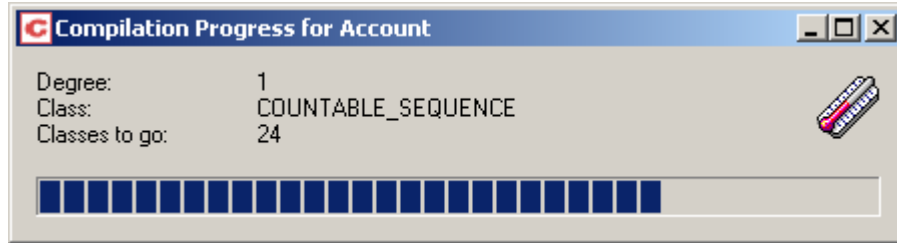


Figure 30: Progress dialog showing compilation

However, it is possible that the compilation does not succeed. Figure 31 shows the error dialog that the wizard displays in such a case. Pressing the “Details >>” button shows you the specific error description from the compiler. In the example below an unknown identifier (*balanc* instead of *balance*) caused the error.

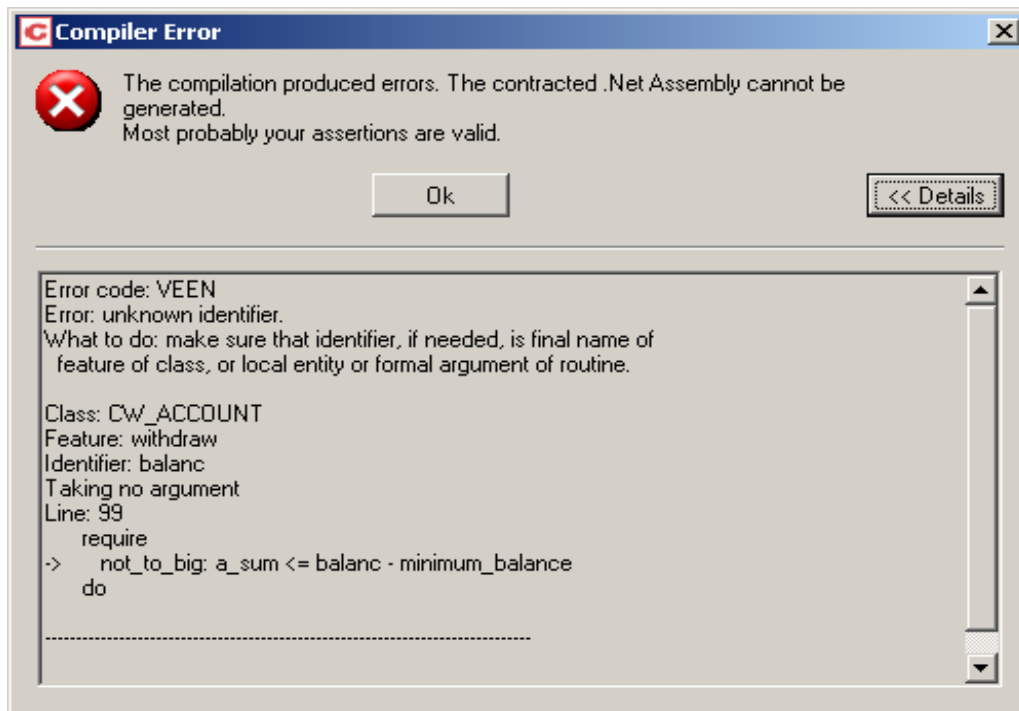


Figure 31: Error dialog showing details about compilation error.

If compilation finishes successfully the wizard passes to its last dialog, as shown in Figure 32. This dialog informs you where you can find the newly generated and contracted .NET assembly. (The assembly is in the subdirectory “EIFGEN\F_Code” relative to the selected working directory.) A click on “Finish” ends the application.

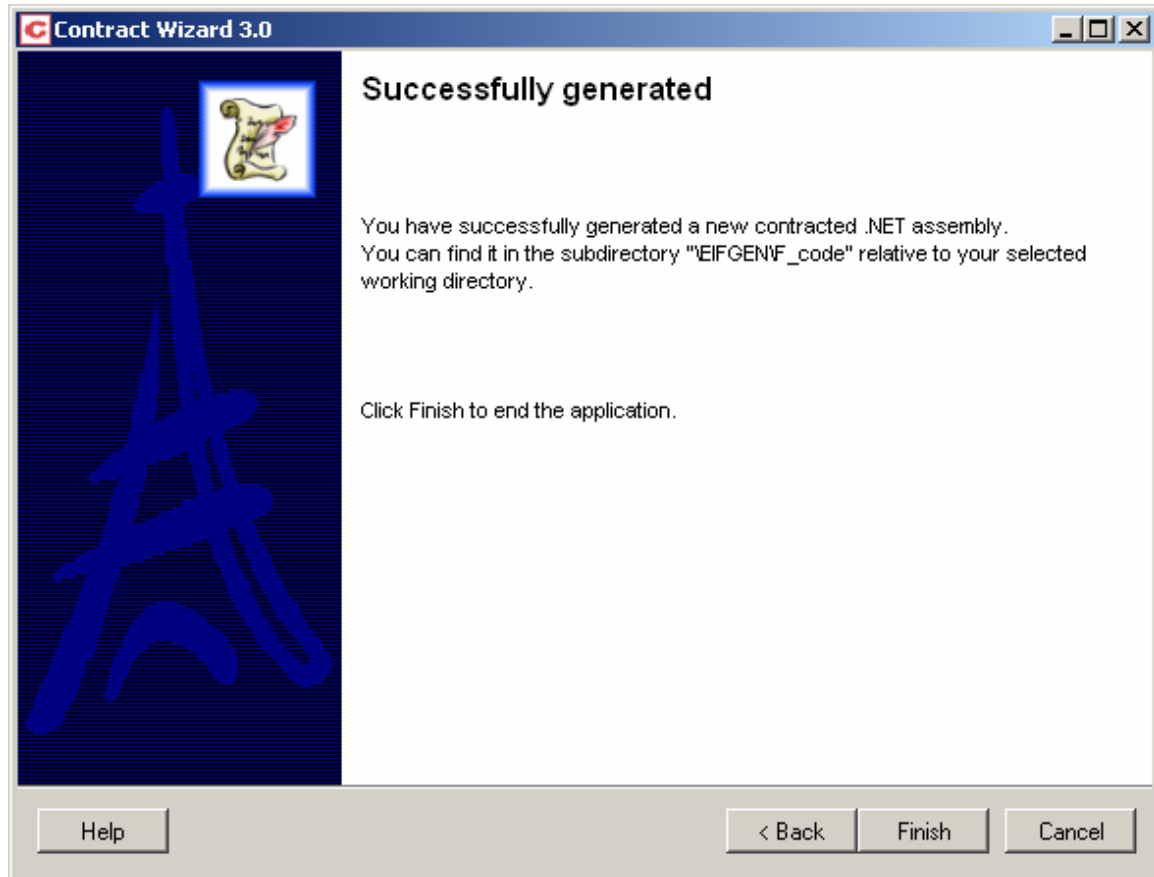


Figure 32: Last dialog after a successful generation of a contracted .NET assembly

At any time it is possible to abort the Contract Wizard application by clicking on “Cancel”. If you do so, a question dialog appears and asks you whether you want to quit the wizard. A click on “OK” terminates the application. If the wizard has already generated new Eiffel class files the dialog asks you if you want to restore the working directory to its original state before ending the application (See Figure 33). Clicking on “Yes” restores the working directory and your changes made on the contracts are lost. A press on “No” ends the application without a restore.

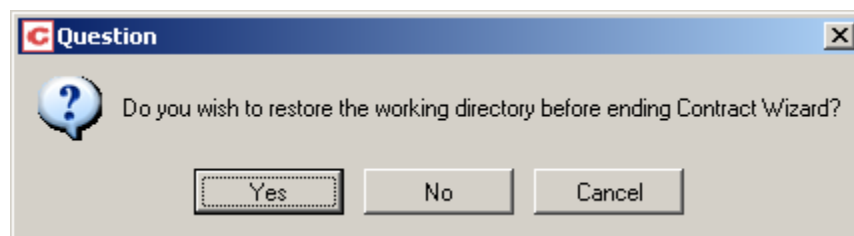


Figure 33: Question dialog asking to restore the working directory before ending Contract Wizard

6.4.3 COMMAND-LINE VERSION

The command line version of the Contract Wizard takes several arguments, depending on what the user wants to do. The first of these arguments is the command. It can be one of the following: “-create_from_assembly”, “-remove_contracts”, “-version”, or “-help”. The number and semantics of the rest of the arguments depends on the command. We will describe each of them.

If you want to create Eiffel proxy classes from an assembly and compile them into a new assembly, you must use the “-create_from_assembly” command. This command must be followed by `Full_assembly_path` and `Target_directory`.

`Full_assembly_path` is the path to the assembly file, for example: `$CONTRACT\examples\Accounting_Example\C#_sources_for_Account.dll`, where `$CONTRACT` must be replaced by the directory path where you unzipped the `ContractWizard.zip` file.

`Target_directory` specifies the directory where the Contract Wizard stores the generated files. It can be any existing directory, but its directory path must not contain spaces as in “C:\Documents and Settings”.

If you want to append contracts to a .NET assembly then you also use the command “-create_from_assembly” together with a second command “-append_contracts”. After “-append_contracts” follows the file name that contains the contracts (`File_name`).

The file containing the contracts must have the following format: an identifier whether you want to add a precondition (`pre`), a postcondition (`post`) or an invariant (`inv`); the name of the class that the contract should be added; the assertion tag; and the assertion expression. For a precondition or a postcondition you must also specify the name of the feature to which you want to add the assertion; in the file the feature name follows the name of the class. The values have to be separated by a comma. If you want to add multiple assertions at once, write every contract on a new line. Here is an example of a file:

```
pre, ACCOUNT, withdraw, not_too_big, a_sum <= balance-minimum_balance
inv, ACCOUNT, not_under_minimum, balance >= minimum_balance
```

You can find the file `c12.csv` with this example in the directory `$CONTRACT\examples\Accounting_Example\C#_sources_for_Account.dll`. (The extension `.csv` stands for comma separated values.) In this directory you will also find an assembly `Account.dll` together with its source code written in C#.

If you want to add the above contracts to the assembly `Account.dll`, type the following command in your command line utility:

```
$CONTRACT\src\EIFGEN\W_code>contract_wizard -create_from_assembly
$CONTRACT\examples\Accounting_Example\C#_sources_for_Account.dll\Account.dll
D:\Data\test -append_contracts
$CONTRACT\examples\Accounting_Example\C#_sources_for_Account.dll\c12.csv
```

When you use as second command “-put_contracts” instead of “-append_contracts” the Contract Wizard removes all existing contracts and adds new contracts specified in the file (File_name).

The command “-remove_contracts” needs the full assembly path (Full_assembly_path) and the source directory where the generated XML file is stored (Source_directory). Full_assembly_path is the path to the assembly; it is the same as in the “-create_from_assembly” command. The source directory is your original target directory. The command removes the XML file storing all contracts for the assembly specified in Full_assembly_path. After a successful deletion it creates Eiffel proxy classes of the assembly and compiles them into a new .NET assembly. This assembly contains no contracts.

The “-version” and “-help” commands take no arguments. The first command prints the current version of the Contract Wizard (Contract Wizard 3.0). The help command prints a message informing you about the correct usage of the command line facility. Table 12 shows the output of this command.

```
$CONTRACT\src\EIFGEN\W_code>contract_wizard -help

Usage:
contract_wizard [-help | -version |
  -remove_contracts Full_assembly_path Source_directory |
  -create_from_assembly Full_assembly_path Target_directory |
  -create_from_assembly Full_assembly_path Target_directory
  -append_contracts File_name |
  -create_from_assembly Full_assembly_path Target_directory
  -put_contracts File_name ]
```

Table 34: Output of contract_wizard -help command

If you want to edit contracts of an assembly open the XML file which the Contract Wizard created of your assembly; its name is cw_xx.xml, whereas xx stands for the name of the contracted assembly. For instance, the name of the XML file is cw_account.xml if the assembly’s name was Account.dll. While editing contracts make sure the XML structure remains valid. The structure is defined in the file contract_wizard.dtd located in the “dtd” directory. When you finished updating the contracts, start the Contract Wizard and run the command “-create_from_assembly Full_assembly_path Target_directory”. The wizard then creates a new assembly according to the contracts specified in the XML file.

7. CONCLUSION

The goal of this project was to extend the Contract Wizard 2.0 with a graphical user interface (GUI). The Contract Wizard parses a .NET assembly, shows the types and the members exposed by each type in an intuitive way, generates Eiffel proxy classes containing the contracts, and finally compiles the Eiffel source to a new assembly. If the added contracts contain errors, the wizard informs the user with an accurate error message. The resulting assembly contains the contracts the user defined and can be used instead of the original assembly. Due to incrementality it is possible to append the contracts to an already contracted assembly. The GUI provides help to add contracts to the assembly in multiple ways. It also makes available a dynamically created interface view of any .NET type.

The use of the GUI is documented with a detailed user manual showing with an example how to create a .NET assembly and explaining how to use the Contract Wizard. The Contract Wizard also provides a command line interface.

The implemented algorithm to handle overloaded features works fine for “simple” .NET assemblies (i.e. Account.dll and Circle.dll delivered with this project). Its limitation mainly comes from classes that inherit from interfaces which define overloaded features. I discussed the restriction and showed a possible solution.

A lot of interesting work can be done on the Contract Wizard:

- Eliminate drawbacks for overloaded names.
- Implement support for inner classes.
- Test the wizard on many assemblies provided with the .NET Framework such as mscorlib, System.Windows.Forms.dll, etc.
- Transform the Contract Wizard 3.0 into a web service to allow any programmers to contribute contracts to .NET components.

A. INTENDED RESULTS

The main goal of this project was to develop a graphical user interface (GUI) for the Contract Wizard 2.0. Before I started implementing the GUI, I had a closer look at the existing tool and defined the expected functionality of the GUI.

INTENDED INTERACTIONS OF CONTRACT WIZARD 3.0

- Load an assembly which has not been contracted
- Load an already contracted assembly
- Choose a working directory for storing generated classes
- Define invariants
 - Add invariant to a class
 - Remove invariant of a class
 - Remove all invariants of a class
 - Edit invariants of a class
- Define preconditions and postconditions
 - Add preconditions and postcondition to a feature
 - Remove precondition and postcondition of a feature
 - Remove all preconditions and postconditions of a feature
 - Edit preconditions and postcondition of a feature

INTENDED FEATURES OF CONTRACT WIZARD GUI

- Assertion check
 - Syntactically correct assertion tag
 - Syntactically and semantically correct assertion expression
- Class view
 - Look at interface of assembly (possibly also contract view from generated proxy classes)
 - Show signature of features
- Help

- Whenever possible auto-completion while typing contracts
- If needed help texts
- Output to user
 - Show degree of compilation (progress bar)
 - Show degree of parsing classes / XML files
 - Warning if wrong directory, compile error, wrong assertion format...
 - Ask user before he wants to exit the wizard before ending application
- Assembly
 - Select an assembly through browsing
 - The extension of the assembly given as input (i.e. whether it is a .dll or a .exe) is checked when analyzing the input. Contracted assembly has same extension
- Performance
 - The Eiffel classes should be finalized instead of frozen, which means that the resulting “new assembly” is in the “EIFGEN\F-Code” directory
 - Do not call *generate_lace* and *generate_eiffel* twice
 - If possible do not parse the whole AST twice for backup purpose
- Directory
 - Easy selection of a directory through browsing
 - Creating non existing directory
 - No exception when a directory does not exist

Except for the auto-completion mechanism while typing contracts, everything else has been reached. At the beginning of the project it was not planned to implement support for overloaded .NET members. (We have not been aware of this limitation.) The Contract Wizard 3.0 can handle overloaded .NET members in most cases.

REFERENCES

- [1] Karine Arnout and Raphaël Simon: *The .NET Contract Wizard: Adding Design by Contract to languages other than Eiffel*, IEEE Computer Society, TOOLS 39, Santa Barbara, USA, April 2004, p 14-23
- [2] Eiffel Software, <http://www.eiffel.com>, consulted in April 2004.
- [3] Gobo Eiffel, <http://www.gobosoft.com/eiffel/gobo>, consulted in July 2004.
- [4] Bertrand Meyer: *Applying 'Design by Contract'*. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986. Published in IEEE Computer, Vol. 25, No. 10, October 1992, p40-51. Also published as *'Design by Contract' in Advances in Object-Oriented Software Engineering*, Eds. D. Mandrioli and B. Meyer, Prentice Hall, 1991, p 1-50. Available from www.inf.ethz.ch/personal/meyer/publications/computer/contract.pdf, consulted in August 2004.
- [5] Bertrand Meyer: *Eiffel: The Language*, Prentice Hall, 1992.
- [6] Bertrand Meyer: *Eiffel: The Language, 3rd edition*, Prentice Hall (in preparation) Available from <http://www.inf.ethz.ch/personal/meyer/#Progress>, consulted in May 2004.
- [7] Bertrand Meyer: *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997.
- [8] Bertrand Meyer: *The start of an Eiffel standard*, in Journal of Object Technology, Vol.1, No.2, July-August 2002, p 95-99. Available from www.jot.fm/issues/issue_2002_07/column8, consulted in August 2004
- [9] Raphaël Simon, Emmanuel Stapf and Bertrand Meyer: Full Eiffel on the .NET Framework, Juli 2002. Available from http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/pdc_eiffel.asp, consulted in July 2004.
- [10] Martijn van Welie: *Patterns in Interaction Design*, <http://www.welie.com/patterns/gui/>, consulted in May 2004.
- [11] Microsoft C# Language Specification: *Explicit interface member implementations*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csspec/html/vclrfcsharpspec_13_4_1.asp, consulted in July 2004.
- [12] Microsoft .NET Framework 1.1 Redistributable, <http://www.microsoft.com/downloads/details.aspx?FamilyId=262D25E3-F589-4842-8157-034D1E7CF3A3&displaylang=en>, consulted in August 2004
- [13] Microsoft .NET library API, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/netstart/html/cpframeworkref_start.asp?frame=true, consulted in Mai 2004.
- [14] Microsoft Visual Studio, <http://msdn.microsoft.com/vstudio/>, consulted in August 2004
- [15] Mono, <http://www.mono-project.com>, consulted in August 2004
- [16] Dominik Wotruba, *Contract Wizard II*, http://se.inf.ethz.ch/projects/dominik_wotruba/diplom/index.html